Irregularity Mitigation and Portability Abstractions for Accelerated Sparse Matrix Factorization

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
eines Doktor-Ingenieur (Dr.-Ing.)

vorgelegt von

Daniel Thürck, M.Sc.

geboren in Gelnhausen.

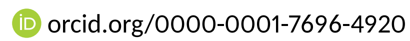
Referenten: Prof. Dr. rer. nat. Kristian Kersting
Technische Universität Darmstadt

Prof. Dr. rer. nat. Matthias Bollhöfer
Technische Universität Braunschweig

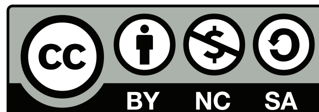
Prof. Dr.-Ing. Michael Goesele
Facebook Reality Labs Research, Redmond, USA

Tag der Einreichung: 24.12.2020
Tag der Disputation: 25.03.2021

Darmstadt, 2021

Thürck, Daniel: Irregularity Mitigation and Portability Abstractions for Accelerated Sparse Matrix Factorization

orcid.org/0000-0001-7696-4920
Darmstadt, Technische Universität Darmstadt
Jahr der Veröffentlichung auf TUprints: 2021
Tag der mündlichen Prüfung: 25.03.2021

Veröffentlicht unter CC BY-NC-SA 4.0 International
https://creativecommons.org/licenses/



Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation selbstständig und ausschließlich mit den angegeben Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die elektronische Fassung der Arbeit stimmt mit der schriftlichen Fassung überein.

Darmstadt, den 24.12.2020

____

Daniel Thürck

Abstract

In this thesis, we investigate new ways to mitigate the inherent irregularity in sparse matrix factorizations and decompose the resulting computation into simple kernels which are portable across a diverse set of compute accelerator architectures through our novel compiler borG. Be it weather prediction, climate models, personalized medicine, genetic analysis and autonomous driving: some of today's central challenges require processing of vast amounts of data, feeding large-scale simulations or AI models. As the scale of these problems outpaces the processing power and available storage capacity, it becomes crucial to exploit their inherent sparsity. Such sparse topologies, i.e., graph topologies where most of the nodes are not directly connected, are often the source for sparse linear systems of equations whose solution poses a major computational challenge.

At the same time, we are witnessing a shift in terms of hardware in the high-performance computing field: as hardware designers try to avoid the quadratically increasing energy consumption for higher clock frequencies, compute setups increase parallelism and specialization instead. Notably, most of the accelerators in use today are optimized for massive parallelism on regular structures and dense data structures.

Processing sparse workloads efficiently on novel, heterogeneous architectures presents a challenge that demands systemic solutions. In this thesis, we investigate strategies and systems focusing on an important building block for computational sciences: sparse numerical (matrix) factorizations. Factorizations exhibit irregularity in two aspects. First, the sparse data structures complicate workload distribution on accelerators geared towards regular grids. Second, numerically mandated pivoting introduces irregularity into the control flow. This leads to expensive synchronization points and requires expensive re-building of data structures.

We propose two building blocks that help mitigate these problems for accelerators. First, a generalization of sparse factorizations to block-sparse matrices, leading to the use of batched, heavily templated compute kernels. Such kernels are relatively simple and can be tuned for the accelerator architecture in question. Second, we propose a data structure for block-sparse matrices that enables global pivoting through parallel index modifications. Additionally, we demonstrate how pivoting can be introduced into register-focused GPU kernels, leading to a two-level, threshold a-posteriori pivoting scheme. Both concepts are validated on implementations of sparse $LDL^\top$ factorizations for GPUs.

Once we extend the block-sparse approach to other architectures, we risk maintaining divergent, device-specific code bases for batched kernels. Therefore, we present the source-to-source compiler borG. Based on a novel intermediate representation unifying two distinct parallel architectures programming models, borG compiles OpenCL code that uses a generalization of the warp register cache idiom, to AVX512-based CPUs, NVIDIA GPUs and NEC's SX-Aurora vector processor. The generated kernels may be specialized for a specific problem size, e.g., processing a batch of $m \times n$ matrices, and compare favorably to hand-coded kernels in the systems' native development stack.

Building on work so far, we extend the concept of block-sparse decomposition and generalize it into a meta-algorithm. Motivated by the rise of domain-specific, fixed function accelerators, we simplify the

device-side code for factorizations even further. The resulting concept, METAPACK, can be implemented not just on more traditional compute accelerators, but also on “non-Neumann” types of hardware. The latter includes systolic arrays and FPGAs. Relying only on host-side pipelining and batched kernel submission, we lay out a blueprint of a versatile factorization generator. As a full-stack system, METAPACK will allow rapid creation of customized, parameterized, sparse factorization code across many systems that support batched or pipelined parallelism. Several experiments confirm the validity of the concept and encourage a full-fledged implementation.

For the sake of simplicity, METAPACK regularizes sparse matrices by subdivision into a regular grid. In order to reduce the memory waste and exploit compute resources more efficiently, accelerators would need native support for work items of different sizes. By expanding on concepts from borG, we propose such support. We combine an incremental change of current accelerators’ architectures with a novel compiler pass to simplify such irregular scheduling on the hardware side. A prototypical software simulation shows this architectures’ potential to simplify support for irregular compute loads in the future.

In summary, our work can help to improve and simplify the handling of sparse matrix factorizations and their efficiency on acclerators. Through a block-centric decomposition of the factorization process and a simplification of the primitive numerical kernels, we enable the use of novel and future accelerator architectures for the sparse problems of computational science.

Zusammenfassung

In dieser Arbeit untersuchen wir neue Ansätze zur Minderung der unerwünschten Effekte bedingt durch irregulär strukturierte Berechnungen bei der Faktorisierung von dünnbesetzten Matrizen. Dazu zerlegen wir die Abfolge der Berechnungen in primitive Funktionen, die sich mittels unseres Compilers borG auf mehrere Arten von Rechenbeschleunigern übersetzen lassen. Sowohl bei der Wettervorhersage, als auch bei Klimamodellen, personalisierter Medizin, genetischer Analyse oder etwa beim autonomen Fahren werden heutzutage große Mengen an Daten verarbeitet, insbesondere durch hochauflösende Simulation oder Algorithmen des maschinellen Lernens. Aufgrund des weiterhin zunehmenden Umfanges dieser Probleme gerät die aktuelle Generation der Hardware sowohl hinsichtlich der Rechenpower als auch der Speicherkapazität an ihre Grenzen. Daher ist es entscheidend, dünnbesetzte Strukturen der Probleme auszunutzen. Diese Strukturen, die sich im Falle eines Graphen etwa dadurch auszeichnen, dass viele Knoten nicht direkt verbunden sind, führen oft zu dünnbesetzten lineare Gleichungssystemen, deren Lösung sehr rechenintensiv ist.

Gleichzeitig findet ein Prinzipienwandel in der Hardwareentwicklung im Bereich des Hochleistungsrechnens statt: Um den quadratisch zunehmenden Energiebedarf von höheren Taktfrequenzen zu umgehen, setzen Entwickler vermehrt auf Parallelisierung und Spezialisierung. So sind die heute meistverwendeten Rechenbeschleuniger für massive-parallele Berechnungen auf dichten und regulären Datenstrukturen ausgelegt.

Die effiziente Lösung dünnbesetzer Problemstrukturen auf solchen Beschleunigerarchitekturen stellt eine Herausforderung dar, der nur systemische Lösungen gerecht werden können. In dieser Arbeit untersuchen wir daher Strategien und Systeme, die sich mit einem wichtigen Fall dünnbesetzter Problemstrukturen beschäftigen: die der dünnbesetzten linearen Gleichungssysteme. Diese weist in zweierlei Hinsicht irreguläre Strukturen auf: Einmal führt die irreguläre Verteilung der Einträge in der Matrix zu Schwierigkeiten bei der gleichmäßigen Verteilung der Berechnungen auf die Hardware. Zusätzlich benötigen numerisch anspruchsvolle Matrizen eine Pivotisierung, die zu irregulärem Kontrollfluss im Programm führt. Dadurch ergibt sich unerwünschter Synchronisationsaufwand und es wird ein aufwändiger Neuaufbau vorhandener Datenstrukturen notwendig.

Daher schlagen wir zwei Bausteine vor, die die adversen Effekte dieser beiden Eigenschaften minimieren sollen. Zunächst generalisieren wir die Faktorisierung auf die Ebene von dichten Blöcken in der dünnbesetzten Matrix, was die Abbildung der notwendigen Berechnungen auf *batched* Funktionsaufrufe, die einem Template folgen, ermöglicht. Diese Funktionen sind relativ einfach und können mit wenig Aufwand an eine Beschleunigerarchitektur angepasst werden. Weiterhin beschreiben wir eine Datenstruktur, die globale Pivotisierung auf Block-Matrizen ausschließlich durch Index-Manipulationen ermöglicht. Als weitere Möglichkeit zur Pivotisierung zeigen wir registerfokussierte Implementierungen von linearer Algebra auf Grafikkarten auf. Die Kombination beider Arten der Pivotisierung ergibt schließlich eine a-posteriori-Pivotingstrategie, welche wir im Rahmen einer $LDL^\top$-Faktorisierung auf Grafikkarten validieren.

Wenn wir diese Methoden ebenfalls auf anderen Beschleunigerarchitekturen verwenden wollen, kann

es zu divergenten, architekturspezifischen Quellcodesammlungen kommen. Als Lösung prasentieren wir hier den Quellcode-zu-Quellcode Compiler borG. Dieser basiert auf einer neuartigen abstrakten Darstellung zweier verschiedener Konzepte des parallen Rechnens und übersetzt registerfokussierten Quellcode für Prozessoren mit AVX512, NVIDIA Grafikkarten und NECs SX-Aurora Vektorprozessor. Die generierten Programme können auf eine gewissen Problemgröße, etwa für $m \times n$ Matrizen, spezialisiert sein und sind von der Laufzeit her vergleichbar mit Programmen, die händisch in der jeweiligen nativen Entwicklerumgebung erstellt wurden.

Darauf aufbauend, erweitern wir das Konzept der Faktorisierung von Blockmatrizen hin zu einem Meta-Algorithmus der Faktorisierung. Inspiriert durch das Aufkommen von spezialisierten Hardwarebeschleunigern, vereinfachen wir den beschleunigerseitigen Code noch weiter. Diese Ansätze ergeben das Konzept "METAPACK", welches nicht nur für die verbreiteten Beschleunigerarchitekturen umgesetzt werden kann, sondern auch neue "non-Neumann" Typen von Architekturen unterstützt, darunter Systolische Arrays und FPGAs (rekonfigurierbare Hardware). Wir beschreiben die Blaupause einer Implementierung, die sich nur auf Pipeleine-Parallelisierung auf Host- und *batched*-Parallelisierung auf der Beschleunigerseite stützt. In seiner vollen Ausbaustufe wird METAPACK es Benutzern erlauben, schnell angepasste Pakete zur Faktorisierung dünnbesetzter Matrizen auf ihrer jeweiligen Beschleunigerhardware zu instanziieren. Mittels mehrerer Experimente belegen wir das enorme Potenizal dieses Konzeptes und leisten dadurch einer künftigen Umsetzung Vorschub.

Zur Reduktion der Komplexität verwendet METAPACK eine einheitliche Blockgröße zur Matrixzerlegeung. Um jedoch Speicherverwendung und Unterforderung der Berechnungseinheiten des Beschleunigers zu vermeiden, bräuchten ebenjene Beschleuniger native Unterstützung für *Batches* aus Problemen unterschiedlicher Größen. Basierend auf unseren Ideen zu borG, schlagen wir inkrementelle Änderungen an der Hardwarearchitektur vor und ergänzen Compiler um eine neue Phase, die zusammen die Verteilung dieser irregulären Probleme signifikant vereinfacht. Mittels einer Simulation in Software zeigen wir schlussendlich das Potenzial dieser Architektur für die Berechnungsprobleme der Zukunft auf.

Zusammenfassend lässt sich sagen, dass unsere Arbeit entscheidende Ansätze zur vereinfachten Entwicklung von effizienten Implementierungen von Lösern für dünnbesetzte Gleichungssyteme liefert. Durch unsere Blockmatrix-basierte Abstraktion des Faktorisierungsprozesses und der relativ simplen numerischen Funktionen ermöglichen wir die Nutzung aktueller und zukünftiger Beschleunigerarchitekturen für die Berechnungen der rechnergestützten Wissenschaft.

Acknowledgements

Research does not happen in a sealed box and neither does life. Without the support and influence of other people, much of this work would not have been possible.

First and foremost, I would like to thank my original supervisor Prof. Dr.-Ing. Michael Goesele for giving me a place to pursue my interests in his group and his continuing support and guidance even after he left TU Darmstadt. Not just by posing the right questions at the right time, but also his help with organizing and conducting research and communicating it through papers mattered. My gratitude goes to Prof. Dr. Marc E. Pfetsch for agreeing to jointly supervise me in my planned work in optimization and continuing to do so even while I ventured into the terrain of linear algebra instead. Marc, I am so sorry I got sidetracked and we never got to finish the optimization part. We will surely pick this up in the future! I thank my academic “stepfather” Prof. Dr. Kristian Kersting for academically adopting me into his AIML research group where I gained new perspectives into some of the research topics that define the field of computer science today. I also greatly appreciate Prof. Dr. Matthias Bollhöfer from TU Braunschweig for our immensely helpful meetings and e-mail conversations full of discussions about issues of computational linear algebra and, of course, for agreeing to review this thesis.

During the course of my PhD phase, I had the pleasure of being a part of three research groups: GCC and AIML at TU Darmstadt and now ISS at NLE. I had (and continue to have) a blast in all of them. I would especially like to thank my colleagues and friends from GCC where I spent my first years of research! Particularly, this involves the “dark side of GCC”: Sven Widmer, Dominik Wodniok, Nicolas Weber, Martin Heß and Stefan Guthe. The time in the “GPU + Rendering” office was both enlightening and fun, we even managed to throw a EGSR conference in the process. Of course, I am not forgetting the “light side”! Thanks, Michael Wächter, Samir Aroudj, Simon Fuhrmann, Fabian Langguth, Jens Ackermann, Xiang Chen and Mate Beljan for discussions while standing around Nele and always trying to balance the force in the group. This goes out to all our HiWis as well. A special award goes to Ursula Paeckel at GCC and Ira Tesar at AIML for helping all of us to navigate the jungle that TU Darmstadt’s administration.

This research has been supported by the ’Excellence Initiative’ of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.

I would like to thank Maxim Naumov, Alejandro Troccholi, Michael Garland and Jan Kautz from NVIDIA as well as Shoou-I Yu and Chenglei Wu from FRL Pittsburgh. It was a privilege working with all of you. Even in case we did not finish our joint projects - your lessons have helped my tremendously in the time after. Especially, it was Maxim who directed my interests to computational linear algebra; our joint work ended up as foundation for this thesis.

A special shoutout goes to my proofreaders: Joanna Mikolei, Nicolas Weber and Sven Widmer. Thanks for fighting your way through and your willingness to question my love for semicolons and double colons and nested scopes within a single sentence.

There’s always a life besides work (or so I have heard). This time is best spent in the company of friends. Florian and Katharina Herrmann and all the other “Flieger”, my former WG-co-inhabitants in Darmstadt

and all friends from Gelnhausen and Darmstadt - thanks for all the good times we had over the years. Friends like mine pick you up when your research disappointed you and make sure that you keep in mind that there are other things that matter in life as well. Since flying is also a big part of my life, all my fellow pilots and friends at EDFG should not go unmentioned. If I forgot to include you in any of these lists – this was a honest mistake, please feel included.

Last but not least, I cannot say how happy I am to have grown up in such a loving family. Thomas, Petra, Julian and Pauline ;) - thank you for your continuing love and support and the endless patience over all these years!

Contents

Frontmatter

Abstract II

Zusammenfassung IV

Acknowledgements VI

Main Content

1 Introduction 1

1.1 Issues of Sparse Processing on Accelerators . . . 3

1.1.1 Data and Control Flow Irregularities . . . 3

1.1.2 Hardware Support and Performance Portability . . . 4

1.1.3 Algorithm and Parameter Selection . . . 4

1.2 Problem statement . . . 5

1.3 Contributions . . . 6

1.4 Thesis outline . . . 7

2 Sparse Matrices In Computational Sciences 9

2.1 Differential Equations . . . 9

2.2 Constrained Optimization . . . 11

2.3 Deep Learning . . . 14

3 Sparse Numeric Computing 18

3.1 Data Formats . . . 18

3.2 Basic Operations . . . 19

3.2.1 SpMV . . . 19

3.2.2 SpTRSV . . . 20

3.3 Preprocessing . . . 21

3.3.1 Scaling . . . 22

3.3.2 Fill-reducing Ordering . . . 22

3.4 Sparse Linear Systems of Equations . . . 22

3.5 Iterative Methods . . . 23
3.5.1 Krylov Spaces . . . 23
3.5.2 Lanczos' Method . . . 24
3.5.3 MINRES . . . 24
3.5.4 Preconditioning . . . 26
3.6 Direct Method . . . 28
3.6.1 Dense Factorization . . . 29
3.6.2 Sparse Factorization . . . 30
3.6.3 Pivoting . . . 36

4 Heterogeneous Computing 39
4.1 Taxonomy and Setup . . . 39
4.2 SIMD Registers . . . 41
4.2.1 Architecture . . . 43
4.2.2 Developer Ecosystem . . . 45
4.2.3 Example . . . 46
4.3 GPUs . . . 46
4.3.1 Programming Model . . . 47
4.3.2 Architecture . . . 47
4.3.3 Developer Ecosystem . . . 49
4.3.4 Example . . . 49
4.4 Vector Processors . . . 49
4.4.1 Architecture . . . 51
4.4.2 Developer Ecosystem . . . 52
4.4.3 Example . . . 52
4.5 Irregularity Mitigation Strategies . . . 52
4.5.1 Multithreading . . . 53
4.5.2 Control Flow Divergence . . . 54
4.5.3 Data Irregularity . . . 54

5 Block-Sparse Indefinite Preconditioning on GPUs 56
5.1 Background and Related Work . . . 57
5.1.1 Dropping Strategies . . . 58
5.1.2 Parallel Approaches . . . 59
5.2 Blocking, Reorderings and Parallelism . . . 59

5.2.1 Blocking Strategies 59
5.2.2 Effectiveness and Parallelism 60
5.3 A Block-oriented $iLDL^T$ with Pivoting 60
5.3.1 Preprocessing and Setup 62
5.3.2 Factorization 62
5.3.3 Blocked Triangular Solves and SQMR 65
5.4 Evaluation 65
5.5 Conclusion 69

6 Supercharging the Block-Sparse Approach 70
6.1 Related Work 71
6.1.1 Fixed-point Methods 71
6.1.2 Context 71
6.2 Two-Level Pivoting 73
6.3 Backend 73
6.3.1 Full Pivoting in Registers 73
6.3.2 Evaluation 74
6.4 Frontend 77
6.4.1 Data Structure 78
6.4.2 Pivoting 78
6.5 A Modified Jacobi-$LDL^\top$ Algorithm 79
6.6 System Evaluation 80
6.7 Conclusion 84

7 Cross-Accelerator Programming for Batched Kernels 85
7.1 Background 87
7.1.1 The Warp Register Cache Idiom 87
7.2 PIRCH - A Common Virtual Accelerator Architecture 88
7.2.1 Compute and Execution Models 88
7.2.2 Register Format 89
7.2.3 InfReg-IR 90
7.3 borG 91
7.3.1 Frontend 94
7.3.2 SIMT Backend 97
7.3.3 SIMD Backends 98

7.3.4 LLVM backend 100
7.4 Evaluation 100
7.4.1 Comparison with Reference Kernels 102
7.4.2 Ablation Studies 105
7.5 Related Work 108
7.5.1 Architectures and Programming Models 108
7.5.2 DSLs and Code Generation 108
7.5.3 Vectorizing Compilers 109
7.5.4 Portability 110
7.6 Conclusion and Future Work 110

8 Customized Sparse Numerical Factorizations 112
8.1 Related Work 113
8.1.1 Implementation Techniques 114
8.1.2 Linear Algebra Program Synthesis 114
8.1.3 Sparse Matrix Factorizations Packages 115
8.2 A Factorization Meta-algorithm 116
8.2.1 Goal #1: Degrees of Freedom 117
8.2.2 Goal #2: Portability 117
8.2.3 The METAPACK Algorithm 117
8.3 System Architecture 119
8.3.1 Indexed BCSR and Pivoting 120
8.3.2 Fill-In Generation 122
8.3.3 Level Subset Scheduling 122
8.3.4 Interface to Accelerators and Scheduling 124
8.4 Inclusion of Domain-specific Accelerators 126
8.4.1 Pipelined Implementation 128
8.4.2 Systolic Implementations 130
8.5 Feasibility Studies and Discussion 131
8.6 Conclusion 135

9 Towards Architectural Support for Irregular Tasks 137
9.1 Related Work 138
9.2 Programming Model 138
9.3 Implementation 139

9.3.1 Front-End . . . 139
9.3.2 Back-End . . . 140
9.3.3 Integration into SX-Aurora . . . 141
9.4 Simulation Results . . . 142
9.5 Conclusion . . . 144

10 Conclusion 145
10.1 Summary . . . 145
10.2 Limitations . . . 146
10.2.1 No “one size fits all” . . . 147
10.2.2 Mitigation, not Resolution of Irregularity . . . 147
10.2.3 Cross-Architecture Efforts . . . 148
10.2.4 Consideration of Physical Constraints . . . 148
10.3 Lessons Learnt . . . 149
10.3.1 The Importance of Preprocessing . . . 149
10.3.2 Restricted Applicability of Fixed-Point Methods . . . 149
10.3.3 Parallel Complexities of Matrices . . . 149
10.3.4 Lack of Support for Irregular Workloads on Emerging Accelerators . . . 150
10.3.5 Top-Down vs. Bottom Up Systems Research . . . 150
10.3.6 Evaluating Research and Presenting Results . . . 151
10.3.7 Exchange of Ideas . . . 151
10.4 Open Problems and Remaining Challenges . . . 151
10.4.1 Linear Algebra . . . 151
10.4.2 Cross-Architecture Efforts . . . 152
10.4.3 Emerging and Future Hardware . . . 153

Appendix

Bibliography 156

(Co-)Authored Publications 174

Curriculum Vitae 175

Chapter 1

Introduction

Contents

1.1 Issues of Sparse Processing on Accelerators . . . 3
1.2 Problem statement . . . 5
1.3 Contributions . . . 6
1.4 Thesis outline . . . 7

The dead ~~speak~~ live[1]! Despite many predictions regarding the eventual demise of Moore's Law [Rotman 2020], it is still alive and doing well. That is, if you focus on its popular interpretation, predicting exponential growth in compute power. The most recent TOP500 [2020] list[2] roughly continues to exhibit the exponential peak performance growth of registered supercomputers which is expected (see Figure 1.1a). However, hidden behind that seemingly smooth growth lie major, disruptive changes in *how* those numbers come to be. First, Asanović et al. [2006] point out that most major performance increases in 2006 stem from a shift towards using multiple cores per die rather than continue to increase a single core's performance through, e.g., its clock frequency. Second, starting 2007, another major disruption made its mark on supercomputer architecture: the introduction of the General Purpose Graphics Processing Unit

[1] Warning: May contain a Star Wars reference.
[2] November 2020

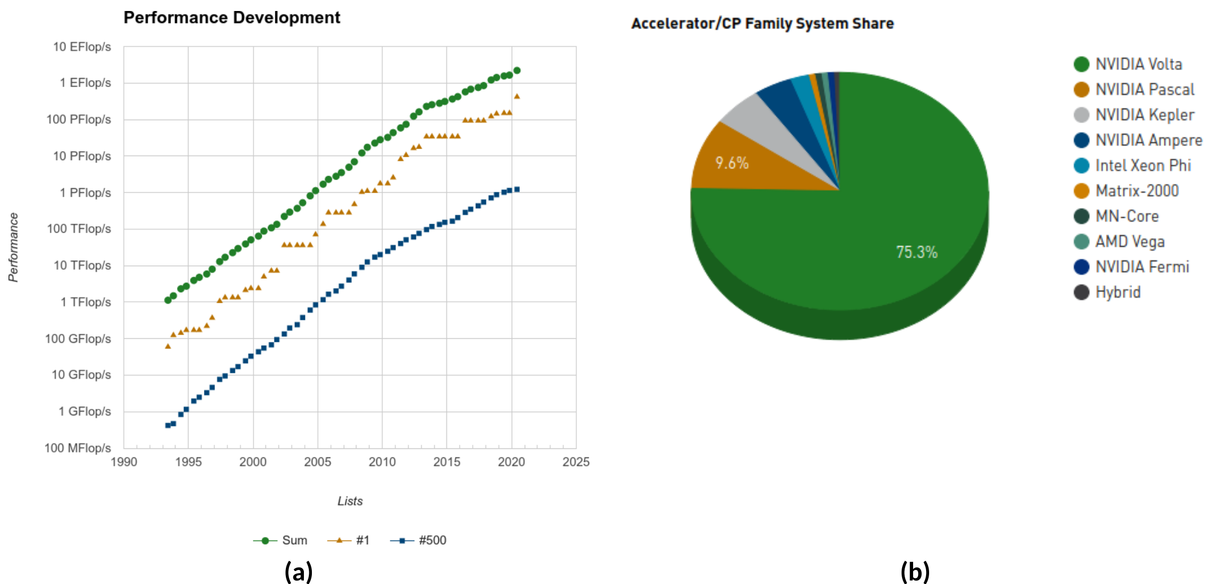


Figure 1.1: Statistics on the results reported in the biannual TOP 500. **(a)** shows the development of the sum, No.1 and No. 500 computer on the TOP 500 list. In all three, the trend continues to be exponential (mind the log-scale). About a third of the systems on the list are powered by compute accelerators whose performance share over various families is detailed in **(b)**. More than 95% of those systems rely on NVIDIA GPUs as accelerators. (Figures courtesy of the TOP500 [2020] statistics page).

(GPGPU) as a massively-parallel compute accelerator. Third, new and customized Central Processing Unit (CPU) architectures challenge the traditional x86 architecture, with a non-x86 based system being No. 1 on the list mentioned above. Today, many more distinct architectures for CPUs as well as accelerators enter the market at an increasing pace, leading Hennessy and Patterson [2019] to proclaim that "we are in [...] a golden age of computer architecture". So far, we have only considered accelerators as additional pieces of hardware in a High Performance Computing (HPC) setup. However, we see another trend that appears in waves: the tight integration of accelerators and CPUs on a single die. To some extent, the ideas of early vector computers were successfully integrated as Single Instruction Multiple Data (SIMD) registers into standard CPUs. Not long after the appearance of the Graphics Processing Unit (GPU), AMD integrated both GPU and CPU through a coherent cache as Accelerated Processing Unit (APU) on a single chip. The trend of integrating CPUs with a accelerator or even domain-specific hardware, continue. Recently, Apple followed by integrating an systolic array for machine learning inference on its ARM-based M1 processors.

As of November 2020, only roughly a third of the TOP500 compute performance is powered by accelerators (see Figure 1.1b), with NVIDIA dominating this segment. The requirements for HPC systems cause the statistics to be a bit skewed: When it comes to the top 10 systems, as of November 2020, 7 out of the 10 rely heavily on accelerators. Due to their cost, HPC setups often run for 5-10 years or longer, hence the lower list contains systems that were built before the widespread adoption of accelerators. Additionally, when highly specialized scientific software without lacks support for accelerators is used, HPC administrators might be hesitant to buy accelerators and rather invest the money into more CPU cores, attempting to avoid the effort of adapting or reimplementing software.

TOP500 ranks system according to their measured LINPACK [Dongarra et al. 1979] performance. LINPACK is a dense linear algebra-based benchmark that favors devices with massive parallelism and high Floating Point Operations (FLOPs) capabilities that process compute-bound workloads well. In practice, workloads and requirements on and for HPC systems are more diverse and have been increasingly so for some years now. With the rise of deep learning [LeCun et al. 2015], data science and artificial intelligence applications are catching up to more "traditional" science applications. On that front, the shift towards computational science and computational engineering [Wilson 1989] in many disciplines – most notably (virtual) engineering and biotech – require running ever-larger and more precise simulations. Prime examples are high-risk projects such as commercial airliners with billions of dollars in investments during their development. In such projects, not just the plane itself [Grihon et al. 2009], but also e.g., transportation routes [Lamiraux et al. 2005] are simulated and optimized.

Computational problems that arise in such scenarios or in Graph Neural Networks (GNN), a hot topic in the machine learning community, exhibit one trait that dramatically distinguishes it from traditional, LINPACK-like workloads: its underlying data structure is that of a sparse graph instead of a dense matrix.

In sparse structures, e.g., graphs or sparse matrices, only nonzero entries are stored explicitly, using an indexing structure to retrieve entries efficiently. Figure 1.2 shows a classic case for sparse matrices: discretizing of Partial Differential Equations (PDEs) leading up to their numerical integration. Only coefficients for the immediate neighborhood in the discretized grid are stored explicitly, leading to the five-point pattern in Figure 1.2b. In each step of the iterative PDE integration process, a system of linear equations using this matrix needs to be solved. In general, there are two classes of methods.

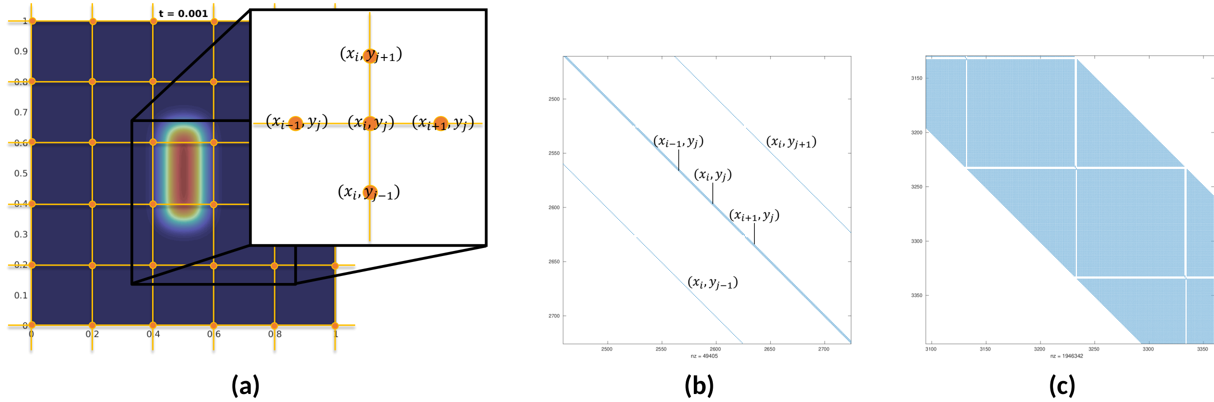


Figure 1.2: (a) Sparse matrices stem, among others, from discretizations of PDEs. With PDE's domain $\Omega = [0, 1]^2$ having been discretized into grid points (x_i, y_j), finite differences or elements-like methods approximate the PDE using local neighborhoods. **(b)** In this example, a grid with max. 4 neighbors per grid point leads to the well-known 5-point matrix. **(c)** When naïvely LU-factorized, the number of nonzero entries in these matrices can increase dramtically.

Iterative methods use Sparse Matrix-(Dense) Vector Multiplication (SpMV) as their main computational primitive. This class of methods performs altogether well on massively-parallel systems, but fails to solve numerically tough matrices. Direct methods, i.e., numerical matrix factorizations such as LU or Cholesky-Factorization, on the other hand, often suffer from serializing nonzero patterns and are sometimes hard to parallelize. Yet, through the means of pivoting, they often succeed in solving even numerically tough systems. However, the choice between direct and iterative methods is not a binary one: preconditioned iterative methods may use incomplete factorizations inside an iterative method to accelerate convergence, presenting a “middle ground” between both methods regarding their ability to be parallelized and their numerical capabilities.

Given a sparse matrix, selecting the solver with the best runtime and numerical performance is still an open issue. First works in this area are currently limited by the discrete selection of solvers and lack of large-scale experimental studies [Yeom et al. 2016].

1.1 Issues of Sparse Processing on Accelerators

All of the previously described developments that led us to this age of “accelerated computing” affect the design and architecture of HPC systems in deployment. This, in turn, affects the types of workloads that performs well on these systems. Some techniques for solving sparse linear systems of equations mismatch the current generation of accelerators that prefer regular workloads. Even if there are techniques that work well with the hardware at hand, we may lack a suitable implementation of that algorithm. In the following sections, we briefly outline expand on these problems as they pose the main motivation for the work presented in this thesis.

1.1.1 Data and Control Flow Irregularities

The preferred workloads for HPC systems with accelerators exhibit regular characteristics, both in the terms of of data and control flow. As in the LINPACK benchmark, achieving close-to-peak hardware performance requires data locality with simple access patterns, optimized and fixed loop tiling, “embarrassing” data-parallelism and branch-minimizing algorithms. The nature of sparse matrix algorithms does not

fulfill any of these requirements. In fact, sparse matrix factorizations suffer from data irregularity due to indirect memory addressing at runtime, control overhead, unbalanced or random sparsity patterns. SpMV, e.g., the central primitive of iterative methods, is directly impacted by such irregular nonzero patterns, i.e., patterns that have a high variance of nonzero entries per row or column. Splitting work along rows thus leads to varying resource requirements. Figure 1.3 shows some sparse matrices from numerical optimization, exhibiting the typically irregular and unstructured nonzero patterns.

In addition, factorizations can suffer from control irregularity. Small pivots that occur during factorization can lead to numerical instability, hence permutations are applied during the factorization to avoid numerical errors, also known as “pivoting”. Some matrices are numerically tough, meaning they require frequent pivoting operations. Besides the obvious control flow divergence that depends on runtime values, each such permutation changes the factors’ nonzero pattern. The resulting, expensive symbolic recomputation of the sparse factors’ nonzero layout leads to a global synchronization. Such computations are heavily sequential and thus unsuitable for accelerators. Furthermore, sparse workloads often lead to a irregular task dependency structure that may be expressed through, e.g., Directed Acyclic Graphs (DAGs) as opposed to regular, nested loops for dense computation.

1.1.2 Hardware Support and Performance Portability

Recently, the market has seen a stream of machine learning-focused, fixed-function accelerators [Reuther et al. 2020]. Additionally, Field Programmable Gate Arrays (FPGAs) are considered once again, as re-programmable, specialized compute accelerators. Instead of hand-written Verilog/VHDL codes, high-level synthesis [Lambert et al. 2018] and first configurable libraries [Matteis et al. 2020] appeal to a much wider base of HPC developers. All that adds to the more commonly used compute accelerators: SIMD extensions, GPUs and vector cards (see Chapter 4). With each architecture comes a set of capabilities, limitations and programming models and languages respective tools. Training developers for novel architectures can take months until they are ready to fully exploit the hardware.

While a large part of the complexities inherent in sparse factorizations can be executed to the host, good overall performance crucially depends on the availability of highly-optimized dense linear algebra kernels. As a result, responsibility is shifted onto vendors, who are required to hand-tune their set of Basic Linear Algebra Subprograms (BLAS) kernels[3] in a time-consuming and therefore expensive manner. Each BLAS implementation then becomes a highly complex and individual piece of software. As a consequence, most sparse packages only interface efficiently with a selected few BLAS implementations (“backends”).

Due the individuality of the BLAS kernels, it is close to impossible to infer any reasonable estimate of how performance varies between architectures. Ideally, we could decompose the factorization process into a set of simpler kernels that are straightforward to implement and perform well across devices. If we succeeded in achieving code- and performance portability across devices, the resulting package could contribute to reduced ramp-up times for software stacks across novel hardware architectures.

1.1.3 Algorithm and Parameter Selection

Considering the complexity of implementing well-engineered sparse matrix packages, it is no surprise that not all algorithms for sparse factorizations are available across all architectures. This holds even more for novel or “exotic” architectures. As stated above, sparse matrices exhibit a degree of numerical

[3] commonly: use *a* BLAS

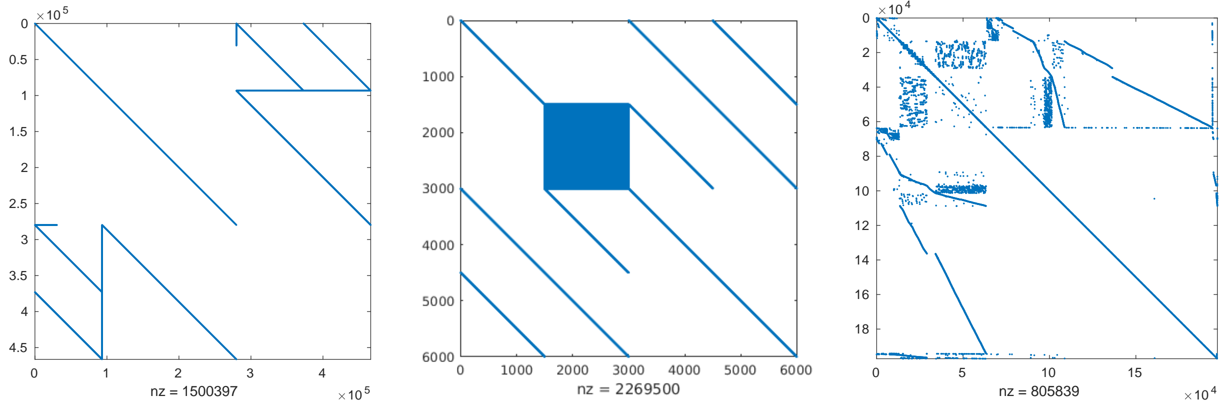


Figure 1.3: Nonzero patterns for matrices `boyd2`, `exdata1`, `power197k` (from **left** to **right**) in the SuiteSparse Sparse Matrix Collection [Davis and Hu 2011]. As clearly visible, the number and location of nonzero entries vary drastically, introducing data irregularity into the computation.

"difficulty", often related to their condition number. They require different algorithms to solve linear systems, ranging from simple iterative methods to full-blown numerical factorizations. In our experience, selecting the right algorithm and its parameters is the crucial decision when attempting to solve a sparse linear system. Selecting a concrete algorithm and tuning its parameters always results in a compromise between numerical accuracy and performance (i.e., time to solution). Most commercial and open source solvers offer a limited selection of algorithms or focus either on preconditioned or direct methods.

1.2 Problem statement

The goal of this thesis is to leverage the latest, regularity-focused architectures from accelerated computing for the solution of sparse linear systems, a potentially irregular workload. We assume a setup of a single machine with a multi-core CPU and an additional accelerator card. While we generally process matrices from problems in the computational sciences, we do not make any assumptions about their size or structure.

We aim at supporting arbitrary input matrices as problems and leveraging as many accelerator architectures as possible, adapting to the level of their support for irregular workloads. Our rather general setup mandates finding the right level of abstractions. Therefore, we examine every part of the process of solving a sparse system of equations. At each step, we attempt to find *configurable abstractions* that allow to minimize the potential performance gap compared to specialized approaches through parameter tuning only. Our focus lies on the development of abstractions for the irregular structure and control flow of sparse matrix factorizations, especially with respect to pivoting. Additionally, we consider features and programming models of various accelerator architectures to ensure performance portability. Ideally, the abstractions that we propose can be combined into a framework for sparse matrix factorizations that is easily adaptable to any given matrix and accelerator architecture by merely choosing some parameters.

In short, this thesis aims at finding abstractions for the solution process of sparse linear systems of equations that enable portability over problems as well as hardware architectures.

1.3 Contributions

In this thesis, we make several contributions addressing the issues motivated and outlined above. Some of our results have been previously published at international workshops, conferences or journals. Some of the following chapters contain these publications, in parts or as a whole, in verbatim. Where necessary, parts (e.g., background, related work) were moved into or referenced by the appropriate chapters of this thesis. Similarly, we adapted introductions and conclusions whenever we saw a potential benefit to the coherency of the overall thesis.

Our contributions are as follows:

- We first approach data irregularity by introducing the central device for sparse matrix factorizations in this thesis: the block-sparse matrix. A study considering the effects of popular matrix reorderings and fill-in levels on the parallelism inherent in the resulting structures leads to two heuristics for block selection. Furthermore, we develop a strategy for indefinite pivoting inside dense blocks, attempting to minimize the effects of control flow irregularity. These two ideas, implemented in an incomplete $LDL^\top$ factorization preconditioner for GPUs, result in a package that compares favorably in terms of numerical accuracy with mature, global-pivoting based software. In the end, we achieve significant improvements over their runtime.

  Chapter 5, [Thuerck et al. 2018]

  These techniques were developed and implemented by the author. The work started as an internship with NVIDIA Research and was supervised by Maxim Naumov, Michael Garland, and Michael Goesele. All three participated in discussions about the project and helped writing the paper.

- In order to solve even more numerically tough matrices, we improve upon the standard block-based CSR structure by adding an indexing layer, enabling cheap global pivoting operations for a 2-stage, threshold-delayed pivoting strategy that would otherwise lead to severe control flow irregularity. Within these dense blocks, we propose to use full pivoting kernels that following the warp register cache idiom. This results in speed-ups over shared memory versions of up to 3$\times$ for the LDU case. With these ingredients, we improve the classical Jacobi-method by the inclusion of the two proposed techniques, closing the gap between the Jacobi method and level set-based, incomplete factorizations.

  Chapter 6, [Thuerck 2019]

- Drawing on our experience from the last chapter, we identify the warp register cache idiom on GPUs as a candidate for performance portable, cross-architecture programs and generalize it to parameterized warp sizes. Using a novel Intermediate Representation (IR) unifying SIMD and Single Instruction Multiple Threads (SIMT) instructions, we propose the source-to-source compiler borG, which generates code for multiple accelerator architectures from a single OpenCL codebase. The generated kernels often exceed the performance of hand-implemented, high-level source code and compare favorably to vendor-tuned libraries.

  Chapter 7, [Thuerck et al. 2020]

Both the idea and implementation of borG and other mentioned ideas stem from the author's work. Nicolas Weber and Roberto Bifulco were involved into discussions about the work and helped with the evaluation and improving the corresponding journal paper.

- We generalize the block-sparse factorizations from Chapters 5 and 6 into a meta-algorithm and design a system that generalizes both incomplete and full sparse matrix factorizations with various parameters without device-specific code for accelerated computer systems using borG. The resulting concept, METAPACK, can even use fixed-function accelerators and reconfigurable hardware, showing that our concepts extend beyond the von-Neumann domain. Several experiments confirm the validity and potential of the concept.

  Chapter 8

- Lastly, we take a step back and take another look at the issue of irregularity on throughput-oriented architectures (e.g., GPUs) on the architectural level. Continuing from our idea of converging SIMT and SIMD representation in Chapter 7, we propose a method to execute SIMT-code on (wide-) SIMD hardware. To that end, we generalize CUDA's execution model and deal with the resulting data irregularity by parameterization through metadata registers. Control flow irregularity is mitigated by a novel method for exploiting register renaming for latency hiding. A simulation indicates that our approach leads to high utilization of the underlying hardware and dramatically simpler scheduling and implementation for developers.

  Chapter 9, [Thuerck 2020]

1.4 Thesis outline

Chapter 2: Sparse Matrices In Computational Sciences

This opening chapter serves as a motivation for our work. Through short deep dives into three topics from computational science and machine learning, we convey the importance of sparse matrix processing in these fields. For the computational sciences, we present two exemplary problems, demonstrating how their inherent sparsity leads to the formation of sparse matrices.

Chapter 3: Sparse Numeric Computing and Chapter 4: Heterogeneous Computing

These chapters outline the basics necessary for the understanding of this thesis. As fundamental topics covered in this thesis, sparse matrices and their decomposition and solution methods as well as HPC accelerator architectures are introduced in these chapters. In particular, we develop (preconditioned) iterative methods and the multifrontal algorithm for numerical factorizations. In Chapter 4, we give an overview over current-generation HPC accelerator architectures and their programming models.

Chapter 5: Block-Sparse Indefinite Preconditioning on GPUs

Chapter 5 introduces the first of two concrete instances of incomplete sparse $LDL^\top$ factorization algorithms for (massively parallel) GPUs, christened block-i$LDL^\top$. Herein, we focus on the inherent data irregularities in the application of this method to numerically tough matrices. We propose a block-sparse setup and use batched kernels where all working data is placed in the GPU's shared memory. This approach allows

partial pivoting to a certain extent and outperforms more traditional incomplete factorization packages.

Chapter 6: Supercharging the Block-Sparse Approach

The second method, discussed in Chapter 6, builds heavily on the the ideas presented in the context of block-iLDL$^\top$. In order to tackle the control flow irregularity from global pivoting, we design and implement a two-stage pivoting process. We introduce a dynamic data structure extending the block-CSR concept that supports parallel modification operations for fast permutations of block rows and columns. Improving on the shared memory-based kernels from above, we propose exploiting the idiom of the warp register cache and demonstrate an implementation of full dynamic pivoting despite its limits. The addition of pivoting elevates simpler parallel schemes (e.g., Jacobi method) to the same numerical performance as "classical" level set methods.

Chapter 7: Cross-Accelerator Programming for Batched Kernels

The pivoting Jacobi method in the previous chapter relied heavily on the availability of warp-register cache style GPU kernels. In order to port our sparse linear algebra packages to multiple accelerator architectures, we identify the warp register cache as an idiom that is ideally suited for performance portable, cross-device kernels. Abstracting away from the different types of architectures, we develop a novel IR that simultaneously represents SIMT and SIMD execution styles. We then discuss the design of the source-to-source compiler borG that automates the mapping process from OpenCL to the proposed IR. The resulting system automatically generates the desired kernels. As this system expands beyond linear algebra, we evaluate borG on a set of kernels not only from linear algebra, but also machine learning and numerical optimization.

Chapter 8: Customized Sparse Numerical Factorizations

Starting from the two concrete instances of block-based factorizations appearing in in Chapters 5 and 6, we develop a meta-algorithm for such factorizations. We decompose sparse matrix factorizations into a frontend and backend: the fronted relies on the data structure presented in Chapter 6 while kernels for the backend can be generated by, e.g., an extended version of borG. With this the system allows to create fully customized sparse factorizations. The pipelined nature of the resulting workloads are suitable even for simple, fixed-function accelerators. We present ideas for kernels on systolic arrays and propose an FPGA backend for borG. Several experiments with preprocessing strategies and our experiences with block-iLDL$^\top$ confirm the feasibility of the concept.

Chapter 9: Towards Architectural Support for Irregular Tasks

In Chapter 9, we present an approach that considers data and control irregularity both on the architectural level as well as the software stack and programming model. Our proposal repurposes some hardware units on vector processors, resulting in a hybrid processor that efficiently executes SIMT and SIMD code with irregularities. A model-based simulation of a SpMV kernel results in first promising results, indicating that our concept can dramatically simplify the implementation of irregular kernels by shifting complexity from the application down to the architectural level.

Chapter 10: Conclusion

Wrapping up this thesis, Chapter 10 summarizes our contributions and recaps the big picture of our work, discussing the merits of the proposed approaches. Moreover, we recap the lessons learnt over the course of our work. Finally, we close by briefly discussing open questions and possible future research.

Chapter 2
Sparse Matrices In Computational Sciences

Contents

2.1 Differential Equations 9
2.2 Constrained Optimization 11
2.3 Deep Learning 14

Sparse data structures are defined by the absence of most of the possible connections or entries. For a matrix, this means, e.g., a relevant part of its entries are zeros and it is beneficial to exclusively store the nonzero entries with their coordinates. In general, sparse matrices and data structures make sense whenever the potential memory savings outweigh disadvantages from indexing. Since the data formats and computational issues are discussed in Chapter 3, we briefly introduce two major areas of computational science where sparse matrices are standard: the numerical solution of partial differential equations and (non-)linear numerical optimization problems. Sparse matrices are not limited to computational science, though. With ever-increasing amounts of data, sparsity becomes more and more of a factor in data science as well. The third section of this chapter thus focuses on the data science and AI areas.

2.1 Differential Equations

PDEs model continuous phenomena in science and engineering, e.g., the airflow around an aircraft wing [Morton and Mayers 2005] or electromagnetic fields using Maxwell's equations [Strauss 2007]. For many types of PDEs, finding exact solutions can be tough or even impossible. Consequently, given their importance, numerical integration techniques have become a staple of engineering and computational sciences. In the following, we demonstrate how the numerical treatment of the simple heat dissipation PDE requires usage of a sparse matrix. For $\Omega = [0,1]^2$, we consider the PDE

$$\begin{aligned} \partial_t u(x,y,t) &= D\Delta_x u(x,y,t) \quad && (x,y) \in \Omega \setminus \partial\Omega \\ u(x,y,t) &= 0 && (x,y) \in \partial\Omega, \end{aligned} \tag{2.1}$$

describing a simple heat dissipation process in a homogeneous medium given some initial distribution $u(x,y,0), (x,y) \in \Omega$ and $D > 0$. The heat equation is, in this simple form, famously well-posed and may be treated with simple finite difference schemes, e.g., Back in Time, Central in Space (BCTS) [Strikwerda 2004]. First, we discretize the time steps into intervals $t_i = i\triangle t$ and the 2D plane Ω into the spatial regular grid $(x_i, y_j) = (i\triangle s, j\triangle s)$ with $i,j = 1...N$ $(N = \lceil \frac{1}{\triangle t} \rceil)$. Second, we introduce the shorthand $u_{i,j}^l = u((x_i,y_j),t_l)$. Using the following finite differences to approximate u's partial derivatives:

$$\begin{aligned} \partial_t u_{i,j}^{l+1} &= \frac{u_{i,j}^{l+1} - u_{i,j}^{l}}{\triangle t} \\ \partial_x u_{i,j}^{l+1} &= \frac{u_{i,j+1}^{l+1} - 2u_{i,j}^{l+1} + u_{i,j-1}^{l+1}}{\triangle x^2} \end{aligned} \tag{2.2}$$

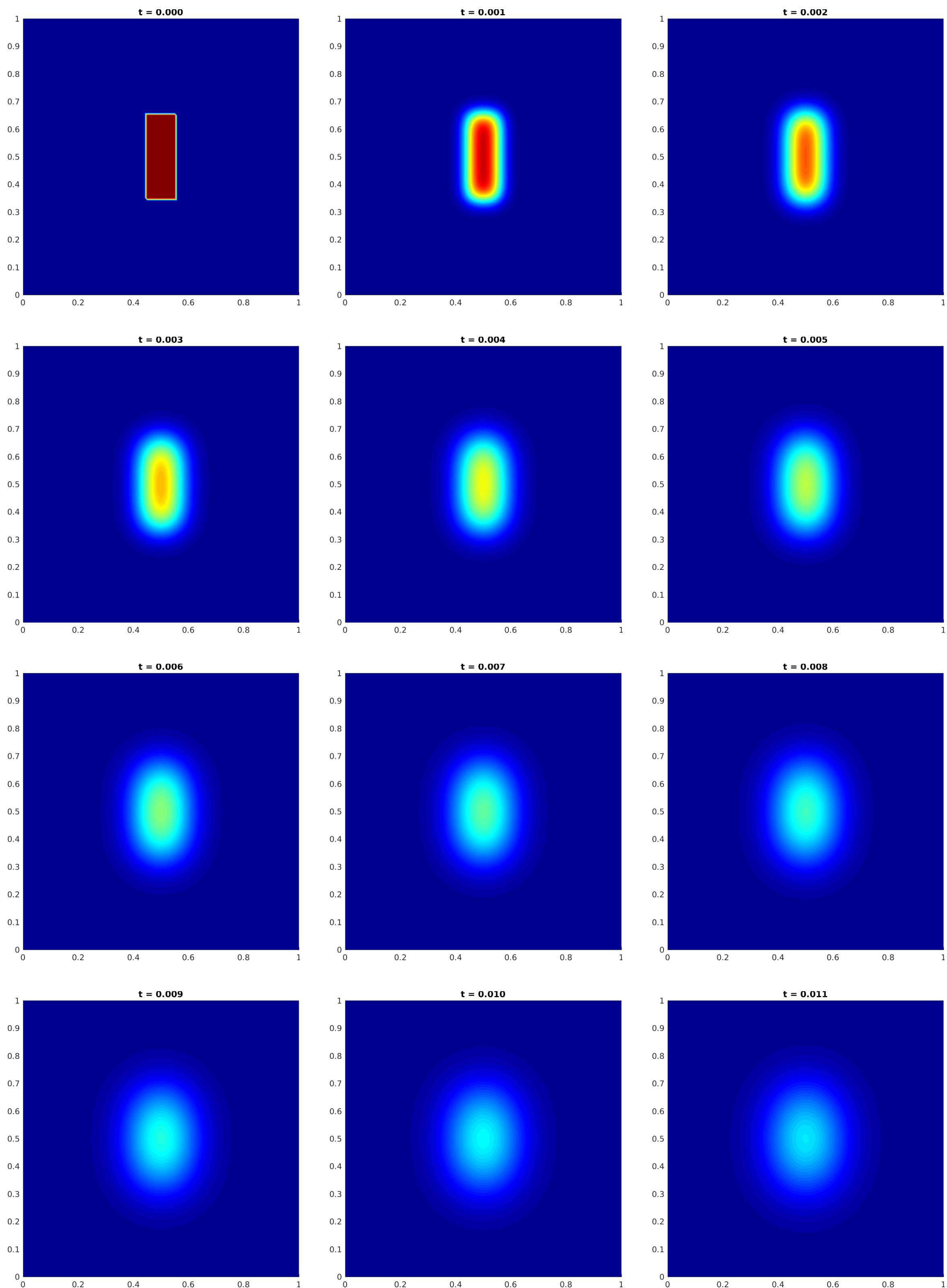


Figure 2.1: Visualization of an implicit finite difference scheme on the heat dissipation PDE $\partial_t u = \frac{1}{2}\Delta u$. The numerical integration process for each timestep leads to a sparse matrix with a five-point pattern, with temporal stepsize $\triangle t = 1\mathrm{e}{-3}$ and spatial stepsize $\triangle x = \triangle y = 1\mathrm{e}{-2}$.

and $\partial_y u_{i,j}^{l+1}$ defined likewise, we express the discretized equation as the following sparse linear system of equations:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & & & & \dots & & & & \\ 0 & \dots & -\frac{D}{\triangle s^2} & -\frac{D}{\triangle s^2} & (\frac{1}{\triangle t} + \frac{4D}{\triangle s^2}) & -\frac{D}{\triangle s^2} & -\frac{D}{\triangle s^2} & \dots & 0 \\ & & & & \dots & & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_{0,0}^{l+1} \\ \dots \\ u_{i-1,j}^{l+1} \\ u_{i,j-1}^{l+1} \\ u_{i,j}^{l+1} \\ u_{i,j+1}^{l+1} \\ u_{i+1,j}^{l+1} \\ \dots \\ u_{N,N}^{l+1} \end{pmatrix} = \begin{pmatrix} 0 \\ \dots \\ \frac{1}{\triangle t} u_{i,j}^{l} \\ \dots \\ 0 \end{pmatrix} \tag{2.3}$$

In order to solve the PDE, each iteration $t_l \longrightarrow t_{l+1}$ boils down to solving this system using sparse linear solvers. Figure 2.1 visualizes such a process for $t \in [0, 0.011]$ and $D = 0.77$. The sparse matrix' nonzero pattern is commonly known as five-point matrix (as in Figure 1.2b), referring to the 5 grid points involved for all interior nodes. Depending on the discretization and integration schemes involved, such matrices are often approximately diagonally dominant and exhibit a relatively regular distribution of nonzero entries per row. These properties, together with matrix sizes of over a million rows for large-scale Finite Element Method (FEM) problems, often mandates the use of an iterative method to solve the discretized linear systems (cf. Chapter 3 and [Koric et al. 2014]).

We would like to close this small example with a warning comment: finite difference type methods such as BCTS are all but state of the art. Rather, they serve as instructional examples [Strikwerda 2004; Moshfegh and Vouvakis 2017]. Still, they follow the same foundational ideas as the state-of-the-art finite element-type methods [Bathe 2006]. Both produce a sparse linear system reflecting, in some way, the discretizations' neighborhood structure. Hence, finite difference methods serve as a motivating good example.

2.2 Constrained Optimization

Optimization problems aim at selecting a solution out of a problems' solution set that is optimal in some sense, e.g., maximizes an objective function. Optimization problems can be classified into various classes according to the form of their objective function and their description of the set of solutions. One of the most basic class is that of Linear Programs (LPs), where program is an antiquated word for “problem”. An LP's description contains:

- a base set for the solution variables, e.g., $x \in \mathbb{X}^n$,
- a linear objective function $f(x) = c^\top x$, $c \in \mathbb{X}^n$ and
- s set of *constraints*, typically in the standard form $Ax = b, x \geq 0$ where $A \in \mathbb{X}^{m \times n}, b \in \mathbb{X}^m$

over some set $\mathbb{X}$. In the following, we assume $m, n > 0$ and $\mathbb{X} = \mathbb{Q}$ in order to exactly represent all solutions on machines with finite precision. The linear constraints $A_i x = b_i$ for $i \leq m$ each represent a halfspace in $\mathbb{Q}^n$; their intersection – if nonempty – implicitly constructs a polyhedron. In case $\mathbb{S} = \{x \in \mathbb{Q}^n \mid Ax = b, x \geq 0\}$ is bounded, we call it a polytope.

As proven by classic polyhedral theory (for the full details of the following brief tour, we refer the reader to Schrijver [1998]), each polyhedron can be expressed by a convex combination of its vertices V_i plus the conic combination of its extreme rays R_j, i.e.

$$\begin{aligned}\mathbb{S} &= \text{conv}\{V_1, V_2, \ldots\} + \text{cone}\{R_1, R_2, \ldots\} \\ &= \sum_i \lambda_i V_i + \sum_j \mu_j R_j\end{aligned} \tag{2.4}$$

where $\sum_i \lambda_i = 1, \lambda_i \geq 0$ and $\mu_i \geq 0$. In the interest of simplicity, we assume that $\mathbb{S}$ is a full-dimensional polytope for the following derivations. Then, for a standardized LP

$$\begin{aligned}\min \quad & c^\top x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0\end{aligned} \tag{2.5}$$

we can show that the optimal solution x^* is, in fact, a vertex of $\mathbb{S}$. We know that for the optimal solution (y^*, s^*) of the dual program

$$\begin{aligned}\max \quad & b^\top y \\ \text{s.t.} \quad & A^\top y + s = c \\ & s \geq 0\end{aligned} \tag{2.6}$$

it holds that $c^\top x^* = b^\top y^*$ (strong duality). Moreover, the so-called “complementary slackness” condition mandates that

$$x_i^* s_i^* = 0, i \leq n. \tag{2.7}$$

Together, all these constraints also form the Karush-Kuhn-Tucker (KKT) conditions for local optima (x^*, y^*, s^*) (see e.g., Nocedal and Wright [2006]):

$$\begin{aligned}&1.\ Ax^* = b && \text{(primal feasibility)} \\ &2.\ A^\top y + s = c && \text{(dual feasibility)} \\ &3.\ x_i^* s_i^* = 0, i \leq n && \text{(complementary slackness).}\end{aligned} \tag{2.8}$$

where $x, s \geq 0$. Since both constraints and objective functions are linear and therefore convex, all local optima are also global optima. When expressed as a mildly nonlinear (in fact, bilinear) system of equations on the nonnegative orthant, the KKT conditions become:

$$F(x, y, s) = \begin{bmatrix} Ax - b \\ A^\top y + s - c \\ XS\mathbb{1} \end{bmatrix} = 0 \tag{2.9}$$

where $X = \text{diag}(x)$ and $S = \text{diag}(s)$. F is differentiable, so we can apply Newton’s method for nonlinear

equations via a Taylor approximation (dropping the remainder term for notational convenience):

$$\begin{aligned} F(x+\triangle x, y+\triangle y, s+\triangle s) &= F(x,y,s) + JF(x,y,s)\begin{pmatrix}\triangle x\\ \triangle y\\ \triangle s\end{pmatrix} && = 0 \\ \Leftrightarrow \qquad \underbrace{\begin{pmatrix} A & 0 & 0\\ 0 & A^\top & I\\ S & 0 & X\end{pmatrix}}_{K_U}\begin{pmatrix}\triangle x\\ \triangle y\\ \triangle s\end{pmatrix} &= -F(x,y,z) && =: \begin{pmatrix}-r_b\\ -r_c\\ -r_{xs}\end{pmatrix} \end{aligned} \tag{2.10}$$

Taking a full step into the "Newton direction" $(\triangle x, \triangle y, \triangle z)$ may, however, violate the nonnegativity conditions $x, s \geq 0$. Hence, we use a line search to stay within the nonnegative orthant; this gives rise to the basic primal-dual Interior Point Method (IPM) in Algorithm 1. For LPs, this class of methods has been popularized by Karmarkar [1984], proving that LPs can be practically solved in polynomial time as opposed to the prior, more theoretical ellipsoid method [Khachiyan 1980].

Algorithm 1 Basic primal-dual interior point method applied to a linear program.

1. Start from a feasible point (x_0, y_0, s_0) and $k = 0$.
2. While $x_i s_i > \epsilon$ for any $i \leq n$:
   (a) Solve (2.10) for $(\triangle x_k)$.
   (b) Select $\alpha \in (0, 1)$ such that $(x_k + \alpha\triangle x_k, y_k + \alpha\triangle y_k, s_k + \alpha\triangle s_k) > 0$.
   (c) Set $(x_{k+1}, y_{k+1}, s_{k+1}) = (x_k, y_k, s_k) + (\alpha\triangle x_k, \alpha\triangle y_k, \alpha\triangle s_k)$
   (d) Set $k \leftarrow k + 1$

The matrix in Equation (2.10) is of size $(2m+n)\times(2m+n)$ and unsymmetric. Practical IPM implementations simplify this system by substituting $\triangle s = -X^{-1}r_{xs} - X^{-1}\triangle x$, leaving

$$\underbrace{\begin{pmatrix} -D & A^\top\\ A & 0\end{pmatrix}}_{:=K_0}\begin{pmatrix}\triangle x\\ \triangle y\end{pmatrix} = \begin{pmatrix}-r_c + X^{-1}r_{xs}\\ -r_b\end{pmatrix}. \tag{2.11}$$

with $D := X^{-1}S$. Further elimination of $\triangle y$ yields

$$\underbrace{AD^{-1}A^\top}_{:=K_+}\triangle y = -r_b + S^{-1}X(X^{-1}r_{xs} - r_c). \tag{2.12}$$

K_+ highlights one crucial feature of IPMs: in all iterations, $x_i s_i > 0$, i.e., the iterates never reach or cross the boundary. Otherwise, the computation of D fails, leading to the premature termination of the algorithm. Similarly, with $s_i \searrow 0$, D slowly becomes ill-conditioned. In the final few iterations of the IPM, when the iterates draw closer to the polyhedron's boundary, this causes severe numerical issues when solving for the Newton direction as some products $x_i s_i$ converge towards 0, others towards ∞ (see Figure 2.2). In order to minimize the effects, one strives to keep all iterates in the vicinity of the central path

$$x_i s_i = \tau \tag{2.13}$$

Matrix	Size	Symmetric?	Definite?	Condition	Nonzeros	Solvers
K_U	$(2n+m)^2$	✗	-	$\approx$ cond(A)	$2n$ + 2nnz(A)	LU, GMRES
K_0	$(m+n)^2$	✓	indefinite	$\approx$ cond(A)	n + 2nnz(A)	LDL$^\top$, SQMR
K_+	m^2	✓	positive	$\approx$ cond$(A)^2$	$\geq$ nnz(A)	Cholesky, PCG

Table 2.1: Important properties for all three classes of IPM Newton's matrices mentioned in the text: K_U, K_0, K_+. K_+'s number of nonzeros depends on A's structure: if A has dense columns, then K_+ will be a dense matrix.

for $\tau \to 0$. This changes the right hand side of the Newton system and forces all products $x_i s_i$ to converge at a similar rate.

There is an ample body of literature about solution techniques for both K_0 and K_+, but less concerning K_U. K_0 is commonly named Saddle-Point Matrix and K_+ its Schur complement. All three classes of matrices have different properties which we briefly summarize in Table 2.1. K_+ is used in most practical IPM implementations with special treatment for dense columns in A that would otherwise cause K_+ to be dense and careful handling of the squared condition number. Nevertheless, K_0 is frequently used by the Non-Linear Program (NLP) community (see e.g., Wächter and Biegler [2006]). We note that a similar derivation based on KKT conditions may also be applied to linearizations of NLPs, leading to algorithms which are similar to Algorithm 1 in terms of linear algebra. Furthermore, LPs often occur in the form of relaxations for (mixed) integer (non-) linear programs. In both the Integer Program (IP) and NLP cases, their solution is the major computational bottleneck; any progress here immediately transfers to those areas.

We close this section with a short detour concerning the Simplex algorithm. Other than IPMs, the Simplex uses the correspondence between vertices of a polyhedron and $n \times n$ submatrices of A. Roughly speaking, the Simplex algorithm progressively updates a set of n columns of A, the basis, and progressively improves the objective value of the corresponding vertex. Each iteration swaps one column in the basis with one outside of the basis. Thus, geometrically speaking, the Simplex moves on the boundary of the polytope, whereas IPMs traverse its interior.

2.3 Deep Learning

Around 2012, three things came together that led to an explosive development in the fields of Artificial Intelligence (AI) and, more specific, Machine Learning (ML):

1. The availability of huge amounts of accessible data, e.g., in the form of ImageNet [Deng et al. 2009]. Apart from being freely accessible, these datasets came with ground truth data for tasks such as object detection and a convenient suite for evaluation of classification systems, leading to the ImageNet challenge.
2. The widespread adoption of GPUs for throughput-oriented compute accelerators and their software stack, based on CUDA as high-level programming language.

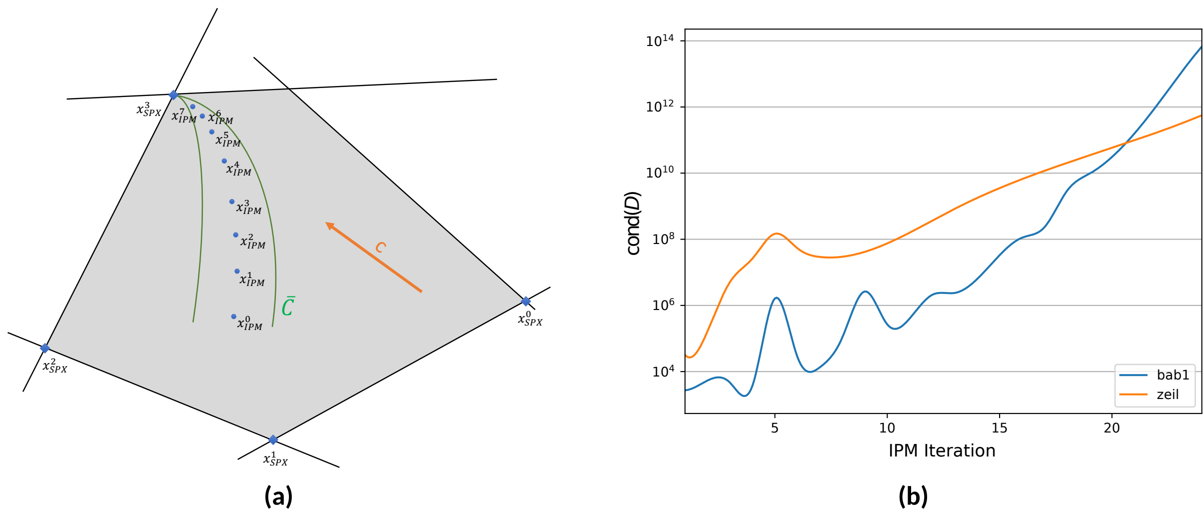


Figure 2.2: (a) IPMs (iterates x_{IPM}) vs. Simplex (iterates x_{SPX}) visualized - while IPM iterates try to stay near the central path ($\bar{C}$) in the Polyhedron's interior, the Simplex traverses the Polyhedron's boundary by following edges from vertex to vertex. **(b)** During an IPM's run, the condition number for the dominating matrix D (here: form $AD^{-1}A^{\top}$) increases exponentially due to some products $x_i s_i$ converging towards zero, others towards $+\infty$.

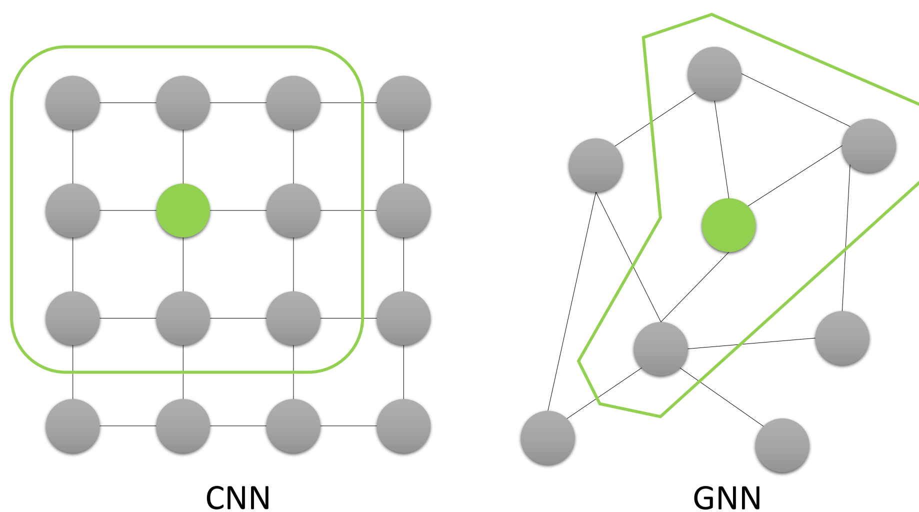


Figure 2.3: Pooling operations in dense, regular CNNs (convolutional neural networks) and sparse, irregular GNNs (graph neural networks). Values of all nodes in the green shapes are convolved with a *kernel* and written to the green neuron.

3. The resurgence of neural networks and and their training by backpropagation and stochastic optimization methods such as Adam [Kingma and Ba 2017].

After Krizhevsky et al. [2017]'s landmark ImageNet submission that exploited all of the above, the research community quickly converged towards comprehensive frameworks such as PyTorch [Paszke et al. 2019]. Vendor-driven, highly optimized libraries offered acceleration for the basic building blocks of the then-emerging convolutional neural networks, e.g, cuDNN Chetlur et al. [2014]. Based on this foundation, research in deep learning lead to significant progress in many fields, including natural language processing. Models such as GPT-3 [Brown et al. 2020] are now capable of writing semantically capable texts from just a few input notes.

Yet, GPT-3 itself is also a good candidate to underscore the current scalability problems of the field. With over 175 billion parameters and 45 TB of training data, it demands distributed training strategies in several datacenters, in sum performing over a PFLOP/s operations. GPT-3's inventors estimate that the compute costs on public clouds approach 12 Mio $. Training heavily utilized hardware accelerators designed to General Matrix(-Matrix) Multiplication (GEMM) operations for neural networks at TFLOP-throughput with orders of magnitude higher efficiency than GPUs [Dally et al. 2020].

Urged by these problems, researchers have begun investigating the inherent sparsity in DNNs. Han et al. [2015] noticed that after training, the majority of parameters in a deep neural network are close to zero or at least of comparatively small magnitude. This insight enabled a modified training process that allows to explicitly set some weights to zero, effectively removing them or incident neurons from the network. Compression rates of $10\times$ with an marginal accuracy loss of one percent point were achieved. Given that memory accesses are more energy-intensive than compute operations, these techniques enable significant cost reductions in practice.

We give an example for such a pruning procedure in Figure 2.4. In the (dense) input networks, each layer represented a GEMM call – a multiplication of two dense matrices, the second matrix containing the data of the current minibatch. After pruning, this operation changes into a Sparse Matrix-(Dense) Matrix Multiplication (SpMM). Furthermore, depending on the training procedure, the weight matrices' structure might change per epoch. Another example for sparsity in deep models are graph-neural networks (GNN) where the input data is a graph. GNNs extend the concept of convolution to irregular graphs. In Figure 2.3, the green frames mark the operands of a standard, regular 3-by-3 convolution and a graph convolution. Computationally, convolutions are mapped to either batched GEMMs or a SpMM call where the matrix has a regular pattern. For GNNs, the principal operation is Sparse Matrix-(Sparse) Matrix Multiplication (SpGEMM).

The resulting sparse linear algebra problems are different than in the case of PDE discretizations or constrained optimization: first, where the latter requires solving a linear system with a sparse matrix, training such networks only requires multiplication with sparse matrices. Second, as Gale et al. [2020] note, the sparse matrices occurring in deep networks are around 50% sparse where for scientific applications, more than 99% of the elements in a matrix are zero, as shown based on the SuiteSparse [Davis and Hu 2011] matrix database. For the remainder of this thesis, we focus on the scientific workloads introduced in the first two parts of this chapter.

For more details, we refer to the standard textbooks of Goodfellow et al. [2016] and LeCun et al. [2015].

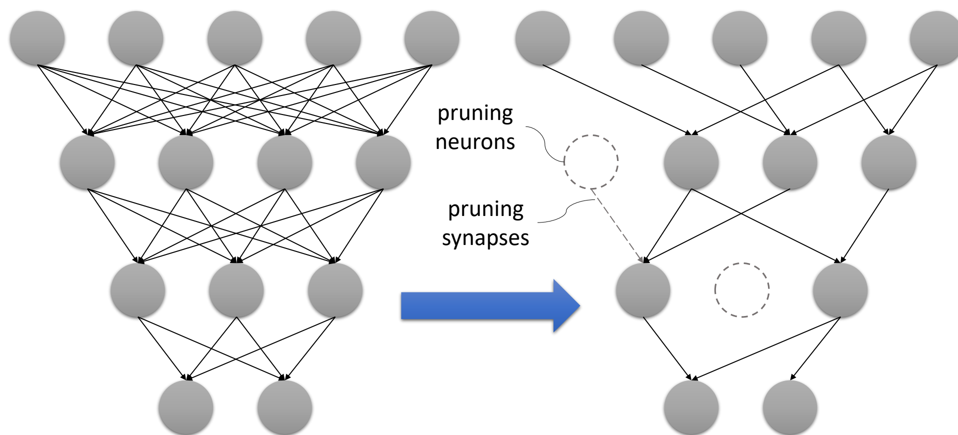


Figure 2.4: Pruning neural networks during or after training turns dense GEMV/GEMM operations into sparse SpMV/SpMM operations, reducing the number of parameters significantly at a moderate loss of accuracy.

Chapter 3
Sparse Numeric Computing

Contents

3.1 Data Formats . . . 18
3.2 Basic Operations . . . 19
3.3 Preprocessing . . . 21
3.4 Sparse Linear Systems of Equations . . . 22
3.5 Iterative Methods . . . 23
3.6 Direct Method . . . 28

This chapter introduces all relevant concepts for a thorough understanding of the subsequent chapters' contents. We formally introduce sparse matrices and relevant formats and primitives for their processing. Since the main focus of this thesis is on solving systems of linear equations we discuss both (iterative) preconditioned methods as well as direct methods. The latter includes the current state-of-the-art in the form of the multifrontal algorithm.

Note: Our treatment of these topics is far from complete. For more in-detail discussion, we refer the reader to standard textbooks which are cited whenever applicable.

3.1 Data Formats

In order to be able to store matrices for large optimization or PDE problems, we use *sparse* formats that mainly store the matrices' nonzero entries in some kind of index structure, generally requiring $O(\text{nnz})$ instead of $O(n^2)$ storage space, where nnz is the number of nonzero elements. However, any such storage format complicates the direct access to elements of the matrix, especially when it comes to structural modifications. In the following, we discuss the most prominent sparse matrix format and briefly outline

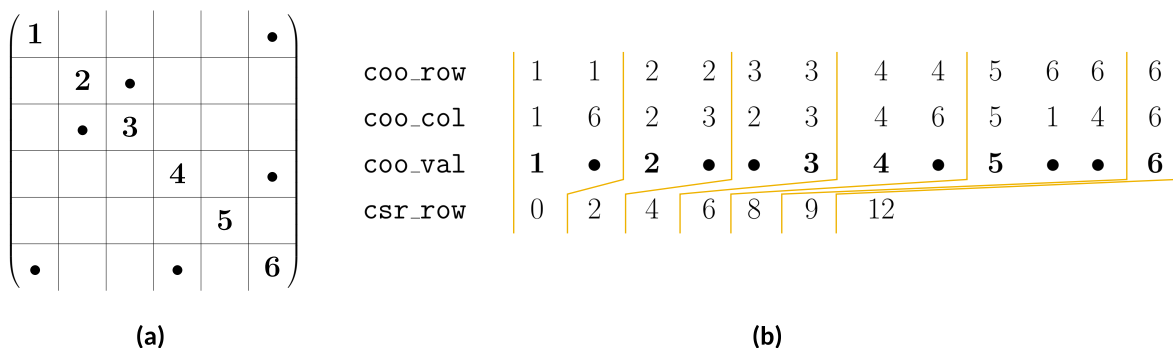


Figure 3.1: (a) Sparse matrix drawn following the scheme used in Davis [2006]. Each off-diagonal, nonzero entry is represented by a •. In this chapter, we always assume a zero-free diagonal and enumerate it for easier legibility. **(b)** The matrix on the left is stored as coordinate triplets `coo_row`, `coo_col` and `coo_val`. Alternatively, by compressing the `coo_row` array, one obtains the matrix' CSR representation.

basic computational kernels and preprocessing operations for solving linear systems.

By far the simplest sparse matrix format is the Coordinate List (COO). We enumerate all nonzero entries by (i, j, k) triplets where i denotes the row position, j the column position and k the numeric value. While the format imposes no order, it is common practice to order the triplets either by their row and column indices. This results in three arrays representing a sparse matrix: `coo_row`, `coo_col` and `coo_val`. As an example, we show the COO and Compressed Sparse Row (CSR) representation of Figure 3.1a in Figure 3.1b . Depending on the triplets' order, COO is the simplest format for modifications to the matrix. New triplets can be added to the end of the list or pre-existing triplets may be deleted – as long as we avoid introducing doublettes. For the latter, the use of a reduction step adding up all entries with the same indices, is quite common. In fact, iteratively reducing a set of unordered COO entries forms the basis of state-of-the-art SpGEMM on GPUs. Dalton et al. [2015] describe the method currently implemented in NVIDIA's cuSPARSE library [Nvidia 2014].

As Figure 3.1b confirms, COO contains redundant information. When entries are sorted by rows, the row index may appear multiple times. Hence, we compress this into the array `csr_row`: instead of listing the indices, we create an $m + 1$-sized array in which entry $\texttt{csr_row}[i]$ holds the offset into the COO arrays for row i's entries. Instead of (3nnz) values, the resulting format CSR requires $(2\text{nnz} + m + 1)$ entries. While the memory savings might seem negligible, CSR simplifies matrix traversals over rows, especially in parallel, by providing direct jump points. Instead of compressing the row indices, using the same method on the column indices results in a Compressed Sparse Column (CSC) matrix which is better suited for column-wise traversal.

We usually pick the sparse matrix format that is best suited for the most compute-intensive operations within an application. However, fast transformations between different formats become necessary when interfacing with e.g., external libraries. In addition to COO, CSR and CSC, there are numerous other specific and hybrid formats such as ELLPACK format (ELL), a densified CSR-like formats with a constant number of entries per row. We refer the reader to Anzt et al. [2020]'s presentation of different formats, which includes a study on format conversion costs. In numerical applications processing PDEs or optimization problems, we encounter mostly COO and CSR/CSC formats. For that reason, we restrict ourselves to the CSR format in the following sections.

3.2 Basic Operations

Other than "administrative" operations such as the creation of and conversion between sparse matrix formats, two operations are frequently used: SpMV and Sparse Triangular Solve (SpTRSV). In the following, we present the basic ideas for their efficient execution. We focus on the CSR format and refer the reader to Anzt et al. [2020] for the other formats.

3.2.1 SpMV

The SpMV operation is central to all iterative methods for linear equations (see Section 3.5). We present a simplified, sequential version of an SpMV-algorithm for a CSR matrix in Algorithm 2. The CSR format is well suited for this operation: using the offsets in `csr_row`, each row can be handled independently, giving rise to a simple parallelization: execute the `for`-loop in line 3 in parallel with each row being assigned to one processor. Each processor then only updates b at a unique location, avoiding costly atomic operations

Algorithm 2 Simple row-centric SpMV procedure for a CSR matrix $A \in \mathbb{R}^{m \times n}$ and a dense vector $x \in \mathbb{R}^n$.

```
1: procedure CSR_SpMV(A, x)
2:     b ← 0
3:     for i ∈ {0, . . . , n − 1} do
4:         for j ∈ {csr_row[i], . . . , csr_row[i + 1] − 1} do
5:             b[i] ← b[i] + coo_val[j] · x[coo_col[j]]
6:         end for
7:     end for
8:     return b
9: end procedure
```

Algorithm 3 Sequential SpTRSV procedure for a lower-triangular CSR matrix $L \in \mathbb{R}^{n \times n}$ and a dense vector $b \in \mathbb{R}^n$.

```
1: procedure CSR_SpTRSV(L, b)
2:     x ← b
3:     for i ∈ {0, . . . , n − 1} do
4:         x[i] ← x[i]/coo_val[csr_row[i + 1] − 1]
5:         for j ∈ {csr_row[i], . . . , csr_row[i + 1] − 2} do
6:             x[coo_col[j]] ← x[coo_col[j]] − coo_val[j] · x[i]
7:         end for
8:     end for
9:     return x
10: end procedure
```

or synchronization. In practical implementations, work is assigned to processors in the form of multiple rows at a time in order to avoid threading overheads.

For massively-parallel platforms, Algorithm 2 presents a problem. The presence of large variations of row length leads to significant thread divergence (for details, see Section 4.3). Operations on strongly irregular CSR matrices often present a challenge to those systems. Expensive preprocessing can help mitigating this data irregularity, but there is also active research into better workload distribution at runtime (see e.g., the MergePath approach by Dalton et al. [2015]). Due to its importance, SpMV is a frequent topic in active research works, especially since its performance is memory bound [Flegar and Anzt 2017; Merrill and Garland 2016; Liu et al. 2018; Bian et al. 2020].

3.2.2 SpTRSV

Solving a sparse system using the direct method involves two steps: factorizing the matrix into, e.g., a triangular factor L, then solving the right hand side with this factor and its transpose, requiring two SpTRSV operations. For CSR matrices, we give a brief, sequential SpTRSV pseudocode as Algorithm 3. Solving a lower triangular matrix in this way is called forward substitution, for upper triangular matrices we use the term backward substitution. In contrast to SpMV, rows are not independent, but may depend on the execution of a previous row. For SpTRSV, the parallelization has to consider the L's nonzero pattern; the most common approach divides the rows into sets of rows that may be executed in parallel [Anderson and Saad 1989; Naumov 2011], so-called “level sets”. Assuming L has a nonzero diagonal, a nonzero entry at (i, j) implies that row i (respective $x[i]$) has to be processed before row j (respective $x[j]$). Hence, L can also be interpreted as the adjacency matrix of a directed graph, containing all dependencies within the solution process. Running a breadth-first search then directly results in the desired level sets. Thus,

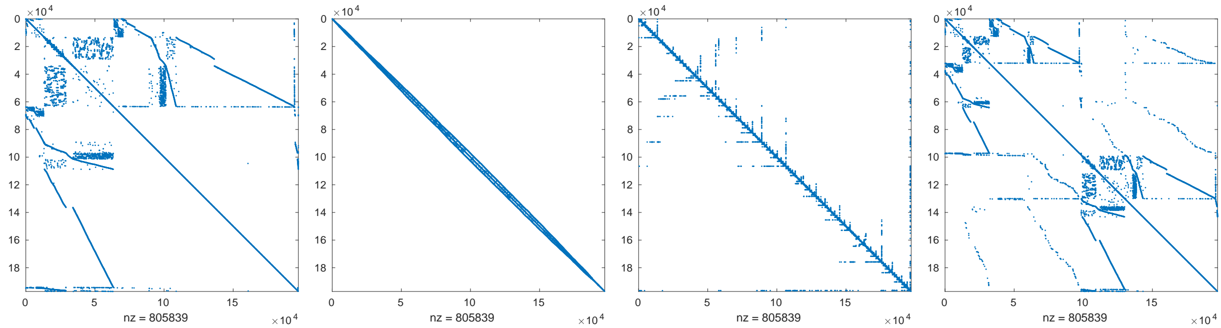


Figure 3.2: Reordering sparse matrices by permuting rows and columns can lead to significant runtime and memory savings in factorizations. Here, we applied the following techniques to matrix `power197k` (from **left** to **right**): None, RCM, AMD, Nested Dissection. Note that this matrix is well-suited for RCM and AMD, but nested dissection fails to retrieve significant block-structures.

SpTRSV implementations proceed in two phases. The analysis phase recovers the level set structure before we execute Algorithm 3, parallelized by level sets, in the numeric phase.

In case b is sparse, the analysis phase of SpTRSV becomes more complex. For that, Davis [2006] presents the theoretical foundation and a practical implementation. Through a row-wise traversal, a set $\mathcal{L}_i$ is determined for each row i such that

$$x_i = \frac{1}{L_{ii}}(b - \sum_{j \in \mathcal{R}_i} L_{ij} x_j). \tag{3.1}$$

With all sets $\mathcal{R}_\cdot$, we create a graph with vertices x_i and edges (x_j, x_i) if $j \in \mathcal{R}_i$. The resulting graph implies an ordering on the components of the solution vector, from which we ultimately derive the order of computation using a Breadth-First-Search (BFS). Each front of the BFS constitutes a level set, where Equation 3.1 can be applied to all its nodes simultaneously.

3.3 Preprocessing

In this thesis, we consider the problem of solving linear systems $Ax = b$ where A is large and sparse. Before we present the main classes of methods in the next sections, we briefly discuss preprocessing, i.e., making changes to (A, b) such that the solver in use can solve the system at hand either faster or more precise. State-of-the-art solvers such as PARDISO [Schenk et al. 2001] find permutation matrices P, Q and scaled permutation matrices S_r, S_c such that the preprocessed system

$$\begin{aligned} \underbrace{PS_rAS_cQ}_{:=B}\, y &= \underbrace{S_rPb}_{:=d}, \\ x &= QS_cy \end{aligned} \tag{3.2}$$

improves upon $Ax = b$ in two aspects. First, a factorization of B should result in less new nonzero entries introduced during the factorization, so-called fill-in, reducing the memory requirements significantly. If done wrong, the factorization could result in near-dense factors, requiring $O(n^2)$ storage space. Second, B's condition number should be smaller than A's or at least require less pivoting operations (again, see Section 3.6), reducing the solver's runtime. In standard implementations, preprocessing involves two

steps, each targeting one of the mentioned properties: fill-reducing ordering and scaling. Those steps are carried out independently. There are only few approaches combining the two (see e.g. Hogg et al. [2017]).

3.3.1 Scaling

In the first step, we find scaled permutation matrices S_r and S_c – or S_r only for symmetric matrices. Such matrices both permute and scale A at the same time, but can also be expressed as the product of a diagonal scaling matrix after a permutation matrix. State of the art, matching-based algorithms such as MC64 [Duff and Koster 2001] try to permute the matrix such that the largest entries are on the diagonal or the sub/super-diagonal. To that end, MC64 computes an unsymmetric matching between rows and columns on the graph induced by A as sparse adjacency matrix. As Hagemann and Schenk [2006] show, cycles in this matching can be broken up and formed into 1×1 and 2×2 blocks that are then permuted to the diagonal. In a second step, the resulting matrix is equilibrated, i.e., scaled in a way that the largest entry in each row and column has a magnitude of 1. For parallel systems, Hogg and Scott [2015] have evaluated replacing the Hungarian algorithm by a faster, approximate Auction algorithm.

3.3.2 Fill-reducing Ordering

In the second step, we find permutation matrices P and Q (where $Q = P^\top$ if A is symmetric) in order to reduce the number of fill-in elements during factorization. Since exactly minimizing fill-in is NP-hard [Yannakakis 1981], we employ one of the following three classes of heuristics:

- Reverse Cuthill-McKee (RCM) [Liu and Sherman 1976]: perform a BFS in order to move elements closer to the diagonal, reducing the bandwidth (maximum distance from the diagonal) of the matrix,
- Approximate Minimum Degree (AMD) [Amestoy et al. 2004]: perform a symbolic factorization on the matrix and optimize the permutation at each step in a greedy fashion or
- Nested dissection (e.g., by METIS [Karypis 1998]): recursively bipartition the matrix, minimizing the intersection between those partitions – this encourages good parallelizability of the ensuing factorization.

In Figure 3.2, we apply all three heuristics to the matrix `power197k` and plot the resulting sparsity patterns as a visual example. Each method leaves a characteristic type of patterns; which method works best depends on the factorization algorithm and input matrix.

3.4 Sparse Linear Systems of Equations

Solving both PDEs and optimization problems, as demonstrated in Chapter 2, requires the solution of large sparse linear systems of equations. We treat "solving sparse matrices" as synonym for "solving a sparse linear system of equations". The formal statement of this problems is as follows:

Problem 1 (Sparse Linear System of Equations). *Let $A \in \mathbb{R}^{n \times n}$ be a sparse matrix and $b \in \mathbb{R}^n$ a dense vector. Then, solving the system involves finding a vector $x \in \mathbb{R}^n$ such that $Ax = b$.*

The efficient treatment of such systems, using both iterative methods or the direct method (matrix factorization) on current, parallel hardware, is a central issue in HPC and computational science. The following sections introduce the basic algorithms for both methods with an explicit focus on the handling

of sparsity.

3.5 Iterative Methods

We classify the major approaches for solving linear systems into two groups: direct and iterative methods. The former is essentially a nickname for matrix factorization. We factor a matrix into a product of matrices that allows a solution of the system by forward or backward substitution. E.g., if A is positive definite, a Cholesky factorization leads to a (lower triangular) factor L such that $A = LL^\top$, with which we can solve the system with two SpTRSVs. In contrast to that, iterative methods start at a vector x_0 and iteratively improve upon that solution, minimizing $\|b - Ax_i\|$. One class of successful iterative methods are so-called Krylov subspace methods. In this section, we introduce their foundations and derive MINRES [Paige and Saunders 1975], a solver for indefinite matrices. For more details, we refer the reader to Saad [2003b]. For this section, we mainly use Saad's notation.

3.5.1 Krylov Spaces

Given a matrix $A \in \mathbb{R}^{m \times n}$ and vector $v \in \mathbb{R}^n$, the i-th Krylov space is defined as

$$\mathcal{K}_i(A, v) = \text{span}\{v, Av, \ldots, A^{i-1}v\}. \tag{3.3}$$

In the following, we use just $\mathcal{K}_i$ and imply using A, b from the linear system problem. Krylov spaces exhibit, among others, the following properties:

$$\begin{aligned} &\text{a) } \dim(\mathcal{K}_i) \leq \min\{m, n\} \leq n \\ &\text{b) } \mathcal{K}_i \subseteq \mathcal{K}_{i+1} \\ &\text{c) } \dim(\mathcal{K}_i) = \dim(\mathcal{K}_{i+1}) \Rightarrow \dim(\mathcal{K}_{i+j}) = i \text{ for all } j \geq 1 \end{aligned} \tag{3.4}$$

Assuming that A is a square $n \times n$ matrix and the degree of the minimum polynomial is n, then span$(\mathcal{K}_n) = \mathbb{R}^n$. Hence, there is an $i \leq n$ such that for a linear system $Ax = b$ it holds that $x \in \mathcal{K}_i$. In the worst case, $i = n$; however, Property (3.4) gives reasonable hope that we may find this solution in an "earlier" Krylov space. In fact, if we are willing to settle for an approximate solution, we may stop at a much earlier step. Each practical method based on this insight needs to deal with two challenges:

1. Find a basis for $\mathcal{K}_i$ that allows efficient projection into the Krylov space. The basis should be numerically stable considering that vectors $A^i b$ tend do be numerically linearly dependent for growing i's. It is just this fact that is exploited in the power iteration method for computing Eigenvectors to A. Ideally, systems with the basis matrix must be computationally efficient to solve. Most Krylov methods today are based on the Arnoldi [1951] or Lanczos [1950] methods that generate orthogonal bases for Krylov spaces.
2. With a basis for $\mathcal{K}_i$ at hand, we have to find a correction r_i in order to compute the next iterate $x_i = x_{i-1} + r_i$. State of the art methods follow one of two selection criteria: The Minimum Residual (MR) criterion $\min \|r_i\| = \|b - Ax_i\|$ or the Petrov-Galerkin approach $b - Ax_i \perp \mathcal{L}_i$.

Algorithm 4 Lanczos method for computing an orthogonal basis for the Krylov space $\mathcal{K}_m$.

```
1:  procedure Lanczos(A, b)
2:      v_0 ← 0, β_1 ← 0, v_1 ← b/||b||_2
3:      for j ∈ {1, ..., m} do
4:          w_j ← A v_j − β_j v_{j−1}
5:          α_j ← w_j^⊤ v_j
6:          w_j ← w_j − α_j v_j
7:          β_{j+1} ← ||w_j||_2, if β_{j+1} < ε then STOP
8:          v_{j+1} ← w_j / β_{j+1}
9:      end for
10: end procedure
```

3.5.2 Lanczos' Method

One method to obtain an orthogonal basis for $\mathcal{K}_i$ if A is symmetric is the Lanczos [1950] method given in Algorithm 4. Its execution results in a basis matrix $V_{m+1} = (v_0\ v_1 \ldots v_{m+1})$ such that

$$AV_m = V_{m+1}\overline{T}_m \tag{3.5}$$

where

$$\overline{T}_m = \left(\frac{T_m}{\beta_{m+1}e_m^\top}\right) = \left(\begin{array}{cccc} \alpha_0 & \beta_1 & & \\ \beta_1 & \alpha_1 & \beta_2 & \\ & \beta_2 & \alpha_2 & \ddots \\ & & \ddots & \ddots \\ \hline 0 & 0 & 0 & \beta_{m+1} \end{array}\right) \tag{3.6}$$

is a tridiagonal matrix (proof: by construction of the Lanczos method, also see Saad [2003b]). It remains to show that V_{m+1} is an orthogonal basis:

$$\begin{aligned} v_j^\top w_j &= v_j^\top (Av_j - \beta_j v_{j-1} - \alpha_j v_j) \\ &= v_j^\top (Av_j - \beta_j v_{j-1} - (v_j^\top (Av_j - \beta_j v_{j-1}))v_j) \\ &= v_j^\top A v_j - \beta_j v_j^\top v_{j-1} - v_j^\top A v_j (v_j^\top v_j) + \beta_j v_{j-1}(v_j^\top v_j) \\ &= v_j^\top A v_j - \beta_j v_j^\top v_{j-1} - v_j^\top A v_j + \beta_j v_{j-1} \\ &= 0 \end{aligned} \tag{3.7}$$

and thus $v_j^\top v_{j+1} = 0$. By the same argument and symmetry of the dot product, $v_j^\top v_{j-1} = 0$; by construction, $v_j^\top v_j = 1$. Plugging in w_{j-2} in the equation above, we get $v_j^\top w_{j-2} = 0$. Hence, the Lanczos method is correct to only orthogonalize explicitly against the last basis vector as all previous vectors are already orthogonal to v_j.

3.5.3 MINRES

MINRES [Paige and Saunders 1975] is a popular Krylov space method based on the Lanczos process for symmetric indefinite and positive definite matrices. The derivations for other methods are similar and some are expanded upon in Saad [2003b]. It employs a MR criterion when selecting the next iterate

$x_i = x_0 + V_i y$, starting with $x_0 = 0, r_0 = b/\|b\|_2$:

$$\begin{aligned} \min \|r_i\|_2 &= \|b - Ax_i\| \\ &= \|b - Ax_0 - AVy_i\|_2 \\ &= \|r_0 - AV_i y_i\|_2 \\ &= \left\|\beta_0 v_1 - V_{i+1}\overline{T}_m y_i\right\|_2 \\ &= \left\|V_{i+1}(\beta_0 e_1 - \overline{T}_m y_i)\right\|_2 \end{aligned} \tag{3.8}$$

Since V_{m+1} is orthogonal, we arrive at the following linear least squares problem:

$$\min || \begin{pmatrix} T_m \\ \hline \beta_{m+1} e_m^\top \end{pmatrix} - \begin{pmatrix} \beta_0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} ||_2 \tag{3.9}$$

$\overline{T}_m$ is tridiagonal, so we can solve this problem efficiently by Givens rotations [George and Heath 1980]. More specifically, in the first step, we eliminate β_1 in the following system:

$$\begin{pmatrix} \alpha_0 & \beta_1 & \\ \beta_1 & \alpha_1 & \ddots \\ & \ddots & \ddots \end{pmatrix} y = \begin{pmatrix} \beta_0 \\ 0 \\ \vdots \end{pmatrix} \tag{3.10}$$

by applying a Givens rotation with $\rho = \text{sign}(\alpha_0)\sqrt{\alpha_0^2 + \beta_1^2}, c = \alpha_0/\rho, s = \beta_1/\rho$:

$$\begin{pmatrix} \overbrace{c\alpha_0 + s\beta_1}^{\gamma_0} & \overbrace{c\beta_1 + s\alpha_1}^{\delta_1} & \\ 0 & \underbrace{-s\beta_1 + c\alpha_1}_{\gamma_1} & \ddots \\ & \ddots & \ddots \end{pmatrix} y = \begin{pmatrix} \overbrace{c\beta_0}^{\epsilon_0} \\ \underbrace{-s\beta_0}_{\epsilon_1} \\ \vdots \end{pmatrix}. \tag{3.11}$$

Repeatedly applying Givens transformations, we iteratively turn the system into upper bidiagonal form:

$$\begin{pmatrix} \gamma_0 & \delta_1 & & \\ & \gamma_1 & \delta_2 & \\ & & \ddots & \ddots \\ \hline & & & 0 \end{pmatrix} y = \begin{pmatrix} \epsilon_0 \\ \epsilon_1 \\ \vdots \\ \epsilon_{i+1} \end{pmatrix}. \tag{3.12}$$

where the last line is notably zero, leaving ϵ_{i+1} as the residual of our least squares problem. Finally, y is determined by backwards substitution.

MINRES integrates the presented treatment of the MR least squares problem into a Lanczos iteration.

After each Lanczos step (resulting in α_j and β_{j+1}), we compute a Givens rotation and update γ_i and δ_{i+1}, resulting in the current iteration's residual ϵ_{i+1}. Paige and Saunders [1975] additionally use an update scheme for solution vectors y_i. For the full MINRES pseudocode, we point readers to [Paige and Saunders 1975]. Notably, they show that MINRES

- monotonously reduces the residual $||b_i - Ax_i||$,
- reduces to the Conjugate Gradient [Hestenes 1956] for positive definite matrices and
- converges superlinearly in the worst case [Simoncini and Szyld 2013].

3.5.4 Preconditioning

Saad [2003b] shows that the error of the Conjugate Gradient (CG) algorithm, a Krylov method for symmetric positive definite matrices, is bounded by

$$||x^* - x_i||_A \leq 2 \left[\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right]^m ||x^* - x_0||_A \tag{3.13}$$

where x^* is the solution to $Ax = b$ and $\kappa = \lambda_{\max}/\lambda_{\min}$ is A's spectral condition number. For positive definite A, we have $\kappa \geq 1$ and

$$\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \approx \frac{(\sqrt{\kappa} + 1) - 2}{\sqrt{\kappa} + 1} = \frac{\sqrt{\kappa} + 1}{\sqrt{\kappa} + 1} - \frac{2}{\sqrt{\kappa} + 1} = 1 - \frac{2}{\sqrt{\kappa}} \text{ for } \kappa \gg 1 \tag{3.14}$$

Thus, the velocity of error reduction (respective convergence rate) largely depends on κ, i.e., it is fastest when $\lambda_{\max} \approx \lambda_{\min}$. Particularly those matrices that originate from constrained optimization regularly exhibit $\kappa \gg 10^3$, giving rise to the idea of preconditioning. Preconditioning means changing the linear equations in a way that reduces the condition number while not changing the system's solution vector x. For linear systems, that often means applying matrices M_1 and/or M_2 in one of the following three fashions:

- *Left* preconditioning: $(M_1 A)y = M_1 b$
- *Right* preconditioning: $(AM_2)y = b, x = M_2 y$
- *Split* preconditioning: $(M_1 AM_2)y = M_1 b, x = M_2 y$.

All three methods lead to applying the Krylov methods to other matrices, e.g., $M_1 AM_2$ with a smaller condition number. In general, we require that M_1, M_2 can be applied efficiently to a vector.

The ideal preconditioner is $M_2 = A^{-1}$ since $AM_2 = AA^{-1} = I$ and the iterative method would stop after one iteration with the exact solution. As this is not computationally feasible, given that Krylov methods are often applied to large sparse matrices, there are numerous preconditioner classes (see, e.g., Scott and Tůma [2016] for a survey on least squares problems). We briefly expand on 4 widely adopted classes:

Simple. This class contains preconditioners that are computationally simple to maintain. First and foremost, that includes the Jacobi-preconditioner which is just a fancy term for using A's diagonal as preconditioner. Jacobi can be helpful for near diagonally-dominant matrices. Its extension is the block Jacobi approach [Anzt et al. 2017b] that uses a block diagonal to capture most elements close to the diagonal. This approach works better if the resulting block diagonal dominates the remaining elements. The class also includes relaxation techniques that decompose the matrix into its diagonal and triangular

parts like Symmetric Successive Over-Relaxation (SSOR) [Axelsson 1972] and Gauss-Seidel.

Inner-Outer. The effectiveness of preconditioners relies on the ability to solve linear systems with it efficiently. Therefore, we usually restrict the nonzero pattern of preconditioner matrices to be products of diagonal, block-diagonal or triangular matrices. Simoncini and Szyld [2002] propose to use a different solver instead for the inner systems, i.e., the systems involving the preconditioner matrix. An inner iterative method slightly changes the actual linear system in each iteration, requiring special outer solvers. An example for the latter is Saad's FGMRES [Saad 1993]. Similar approaches have been applied to CG [Golub and Ye 1999; Simoncini and Szyld 2002]. This approach can deal with numerically difficult matrices, but may require high iteration numbers [Cui et al. 2019].

Incomplete factorizations. Other than full factorizations, e.g., a Cholesky factorization $LL^\top$ for positive semidefinite matrices A, incomplete factorizations only contain a subset of the nonzero elements. There are different methods to determine structured subsets: either we compute them a priori by, e.g., using level sets [Hysom and Pothen 2002] or we drop elements smaller than a threshold η in a thresholded incomplete factorization. We use IC(k) as shorthand for an incomplete Cholesky with level k and ICT(η) for the thresholded variant. As a rule of thumb, ICT often performs better than IC(k) due to its adaptivity to the matrix at hand, but often requires considerably more fill-in. Additionally, ICTs are hard to parallelize given their dynamic nonzero pattern. An exception here is the work by Anzt et al. [2019b] who investigate a parallel fixed-point algorithm that can work for some, often near-diagonal dominant, matrices. Otherwise, there are many more variants of incomplete factorizations, varying the pattern computation, pivoting scheme (if any) or structural setup (e.g., by blocking [Bollhöfer et al. 2019]). We discuss this further in Chapters 5 and 6. Ultimately, many of these approaches do not take the characteristics of the problem into account, but offer a black-box like interface. As Chow and Saad [1997] noted, successfully applying this class requires overcoming the black-box characteristic and finding a configuration of parameters that does take problem-specific information, e.g., the matrix' nonzero structure, into account. The combination of iterative methods with incomplete factorizations offers a continuous spectrum of algorithms between "pure" iterative methods and a full factorization of A.

Problem specific. Such preconditioners are tied to the concrete problem at hand, but can be immensely successful and efficient. One prime example here was developed by Schork and Gondzio [2020] who exploit the structure of Simplex-like bases for constructing preconditioning matrices for the later stages of an IPM. Compared to the application of inner-outer SSOR preconditioning to the same problems in Cui et al. [2019], this approach results in far shorter IPM runtime and higher stability in terms of the number of problems that can be solved accurately.

In practice, when applying a Krylov method to a linear system, researchers experiment with preconditioners of increasing complexity that are then tuned to the problem at hand. The success of a preconditioner depends on the cost of computing and repeatedly applying the preconditioner being smaller than the cost of the increased number of Krylov iterations.

We close this section with a small example for the effects of different preconditioners. In Figure 3.3, we used Preconditioned Conjugate Gradient (PCG) without and with 4 different preconditioners. While the simple Jacobi preconditioner does not improve upon the bare PCG much, a block Jacobi method with 16×16 blocks on the diagonal closes the gap to a zero-fill incomplete Cholesky method. All preconditioners so far fall wide behind a thresholded incomplete Cholesky with drop tolerance $\eta = 0.01$ – but that win

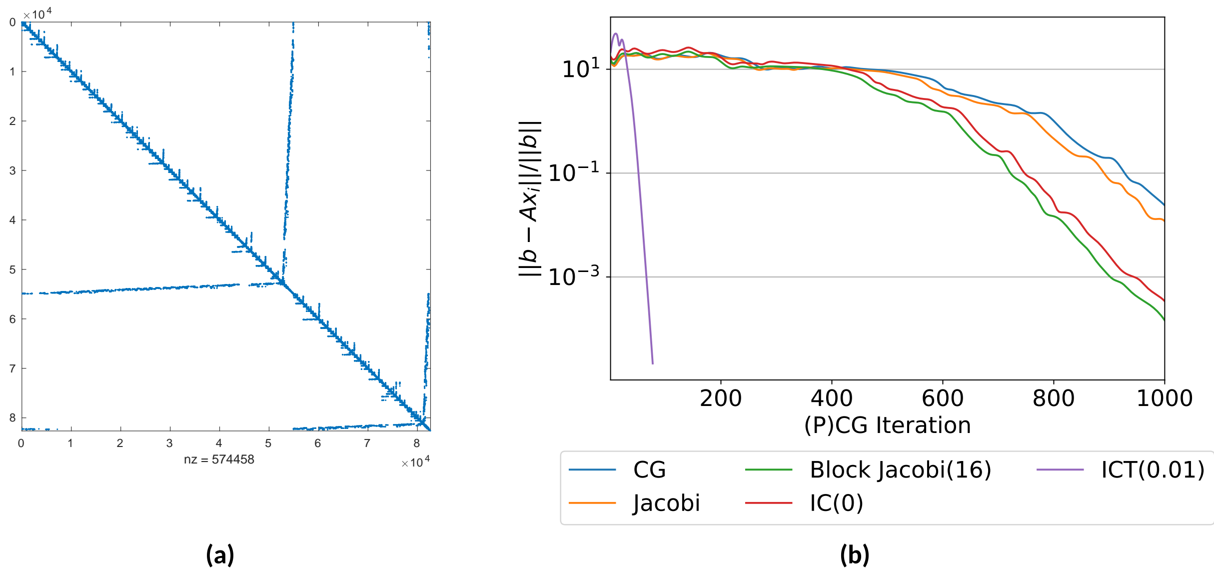


Figure 3.3: Solving the AMD-ordered matrix `thermal2` from Davis and Hu [2011] (**(a)**) with Conjugate Gradient and multiple preconditioners **(b)**: a simple Jacobi preconditioner, a block Jacobi preconditioner with 16×16 blocks, a zero-fill incomplete cholesky (IC0) and a thresholded incomplete Cholesky with drop tolerance $\eta = 0.01$ (the latter resulting in a fill-in factor of 2). The matrix' condition estimate is $4.96 \cdot 10^5$.

comes at a price. The incomplete Cholesky factor has about twice as many nonzero elements than A.

3.6 Direct Method

Rather than a class of methods, the direct method is a synonym of matrix factorization. For linear systems of equations, we distinguish between the following kinds of factorization of a real matrix $A \in \mathbb{R}^{m \times n}$:

- A QR-factorization $A = QR$ for rectangular A's. Q is an $m \times m$ orthogonal matrix whereas R is an $m \times n$ matrix $R = \left(\overline{R}^\top \mid 0^\top\right)^\top$ and $\overline{R}$ is an $n \times n$-upper triangular matrix. Q is a product of orthogonal transformations, either Givens transformations or Householder transformations [Householder 1958]. Solving with A then requires the application of the transpose of all transformations that make up Q. Since Q can become dense, we often apply orthogonal transformations directly to $Ax = b$, rather than explicitly storing Q.
- The LU-factorization $A = LU$ decomposes A into a lower triangular factor L and an upper triangular factor U and one of both ends up with an all-1 diagonal. LU is applied to nonsymmetric, square matrices.
- The Cholesky factorization $A = LL^\top$ returns a lower triangular factor L with a positive diagonal for Symmetric Positive Definite (SPD) matrices A. Apart from numerical errors, SPD matrices do not require pivoting. If the factorization “breaks down”, i.e., encounters a near-zero diagonal, A is numerically positive *semi*definite.
- The LDL$^\top$ or modified respective *indefinite* Cholesky factorization (or Bunch-Parlett factorization [Bunch et al. 1976; Bunch and Parlett 1971]) decomposes symmetric indefinite matrices A into a triangular factor L with ones on its diagonal and a (block-diagonal) matrix D with 1×1 or 2×2 blocks. The latter are created when a zero on the diagonal is encountered throughout the factorization

and helps stabilizing the factors.

We call the matrices whose product make up A "factors". The LU factorization may also be applied to symmetric matrices at the cost of saving both triangular factors. In this section, we always assume that A has full rank, i.e. rank$(A) = \min\{m, n\}$. We discuss the full setup of a sparse Cholesky factorization from a user's and programmer's point of view by deriving it from a dense factorization – for theorems and proofs, we refer interested readers to Davis [2006].

3.6.1 Dense Factorization

We start by presenting a dense Cholesky factorization. Both LDL$^\top$ and LU factorizations can be derived in the same way. There are multiple formulations of the Cholesky algorithm that ultimately all lead to the same result (up to numerical errors), but differ in the part of the (factorized) matrix that they access at each step. In the following, we derive the up-looking and the right-looking algorithms. If the Cholesky factorization was executed in "upper" mode where $A = U^\top U$ and U is upper triangular, those would become down- and left-looking. We start with the following 2×2 block matrices where lower case letters represent scalars and vectors, respectively, and the upper case letters represent square matrices:

$$\begin{pmatrix} A_{11} & a_{21}^\top \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ l_{21} & l_{22} \end{pmatrix} \begin{pmatrix} L_{11} & l_{21}^\top \\ & l_{22}^\top \end{pmatrix} \tag{3.15}$$

Comparing the left and right-hand side in this equation leads to the following set of equations that we then solve for l_{21}, l_{22} and L_{11}:

$$\begin{aligned} &\text{a) } A_{11} = L_{11}L_{11}^\top \\ &\text{b) } a_{21} = l_{21}L_{11}^\top \Leftrightarrow L_{11}l_{21}^\top = a_{21}^\top \\ &\text{c) } a_{22} = l_{21}l_{21}^\top + l_{22}l_{22}^\top \Leftrightarrow l_{22} = \sqrt{a_{22} - l_{21}l_{21}^\top} \end{aligned} \tag{3.16}$$

A head recursion on the Equation (3.16) gives rise to the up-looking algorithm that proceeds in row-wise order. Its main computational primitive is the Triangular Solve (TRSV) in b), see also Figure 3.4a. The name stems from the fact that each step of the factorization requires repeated access to the part of L upwards of the current row. The element on the diagonal in the current row, here a_{22}, is commonly referred to as the pivot.

Alternatively, we can start from a block matrix where the pivot is in the first row:

$$\begin{pmatrix} a_{11} & a_{21}^\top \\ a_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} l_{11} & \\ l_{21} & L_{22} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21}^\top \\ & L_{22}^\top \end{pmatrix} \tag{3.17}$$

Similar to the course of action above, we form the following equations:

$$\begin{aligned} &\text{a) } a_{11} = l_{11}l_{11} \Leftrightarrow l_{11} = \sqrt{a_{11}} \\ &\text{b) } a_{21} = l_{21}l_{11} \Leftrightarrow l_{21} = a_{21}/l_{11} \\ &\text{c) } A_{22} = l_{21}l_{21}^\top + L_{22}L_{22}^\top \Leftrightarrow L_{22}L_{22}^\top = A_{22} - l_{21}l_{21}^\top \end{aligned} \tag{3.18}$$

Finally, a tail recursion leads to the right-looking algorithm, whose main computational primitive is the so-called "Schur downdate" from Equation (3.18c). This formulation requires access to the lower, i.e., the remaining part of the matrix and proceeds by column. We visualize this in Figure 3.4b. Additionally, Algorithm 5 contains pseudocode for a basic right-looking Cholesky factorization on a dense matrix.

BLAS/LAPACK. Dense factorization kernels, among others, are crucial computations on today's supercomputers and form the basis of many algorithms. In particular, it is heavily used in the LINPACK [Dongarra et al. 2003] benchmark used in the TOP500 lists. Factorization kernels, with a suitable matrix blocking for effective use of all available caches, are prototypical for compute-bound codes. These codes' performance is limited by a system's FLOPS. Hence, they are commonly used to measure a systems' peak FLOPS capability. Normally, vendors hand-tune a set of these linear algebra procedures, so-called "kernels", for a given architecture. In order to be able to make code that uses these kernels portable, the community has agreed on a set of kernels with a standardized interface: the Basic Linear Algebra Subprograms (BLAS), see Lawson et al. [1979] and Dongarra et al. [1988]). There are 3 levels of BLAS kernels:

1. vector-vector interactions (e.g., `AXPY`: $y \longleftarrow \alpha x + y$),
2. matrix-vector interactions and (e.g., `GEMV`: $y \longleftarrow \alpha Ax + \beta y$)
3. matrix-matrix operations (e.g., `GEMM`: $C \longleftarrow \alpha A * B + \beta C$).

The BLAS, however, contains only the most basic linear algebra kernels. Kernels for linear systems, Eigenvalue problems, linear least squares and singular values decomposition – all based on BLAS operations – are packed in the Linear Algebra PACKage (LAPACK, see Demmel [1991] and Anderson et al. [1999]). Most state-of-the-art numeric software packages are based on the BLAS and LAPACK standards. In order to reach the possible performance, users may require to link in their vendors' implementations. Beyond dense linear algebra, these kernels are also of utmost importance for sparse matrix factorizations, as we will discuss towards the end of this chapter. Modern sparse matrix packages use the multifrontal algorithm to pack parts of a sparse matrix into dense blocks and apply these libraries, leading to effective cache use and orders of magnitude of possible speedups. For a closer look at dense linear algebra, especially its error analysis, we refer the reader to Van Loan and Golub [1983].

3.6.2 Sparse Factorization

Taking the step from a dense Cholesky to its sparse counterpart seems easy at first. In Algorithm 5, we would just skip any products in row 13 where one of the factors is zero. Unfortunately, that change is far from enough and introduces some disadvantages that renders the algorithm unusable for sparse matrices. First, its complexity would still be $\mathcal{O}(n^3)$ and we would traverse a lot of zeros. Second, its random access into the matrix requires a dense memory format and thus exhibits a memory complexity of $\mathcal{O}(n^2)$ – for a $100{,}000 \times 100{,}000$ matrix, this already means approximately 74.5 GB in double precision format (64 bit per entry)! Third, even the use of a sparse data structure for L requires frequent modifications owing to the fact that row 13 can create nonzero elements at places in L where there was a zero element in A. We call these elements "fill-in" (short: fill) elements. Thus, the nonzero pattern of L is the union of the nonzero pattern of the lower triangular part of A and all its fill-in.

This is, in fact, how an efficient sparse factorization works: In a "symbolic analysis" phase, we first determine the nonzero structure of L and then execute Algorithm 5 on the resulting nonzero pattern in the subsequent "numerical factorization" phase. We fulfill the requirement of both fast row- and column

Algorithm 5 Basic right-looking Cholesky factorization for a dense matrix A, returning the lower triangular factor L. This only serves as an instructional example; for instance, it requires access to the full matrix even though it just returns a triangular part.

1: **procedure** rchol(A)
2: $L \leftarrow A$
3: **for** $j = 0, \ldots, n-1$ **do**
4: **if** $L(j,j) < \epsilon$ **then**
5: **STOP** with "A is positive semidefinite"
6: **end if**
7: $L(j,j) \leftarrow \sqrt{L(j,j)}$
8: **for** $i = j+1, \ldots, n-1$ **do**
9: $L(i,j) \leftarrow L(i,j)/L(j,j)$
10: **end for**
11: **for** $j' = j+1, \ldots, n-1$ **do**
12: **for** $i' = j+1, \ldots, n-1$ **do**
13: $L(i',j') \leftarrow L(i',j') - L(i',j)L(j,j')$
14: **end for**
15: **end for**
16: **end for**
17: **return** tril(L)
18: **end procedure**

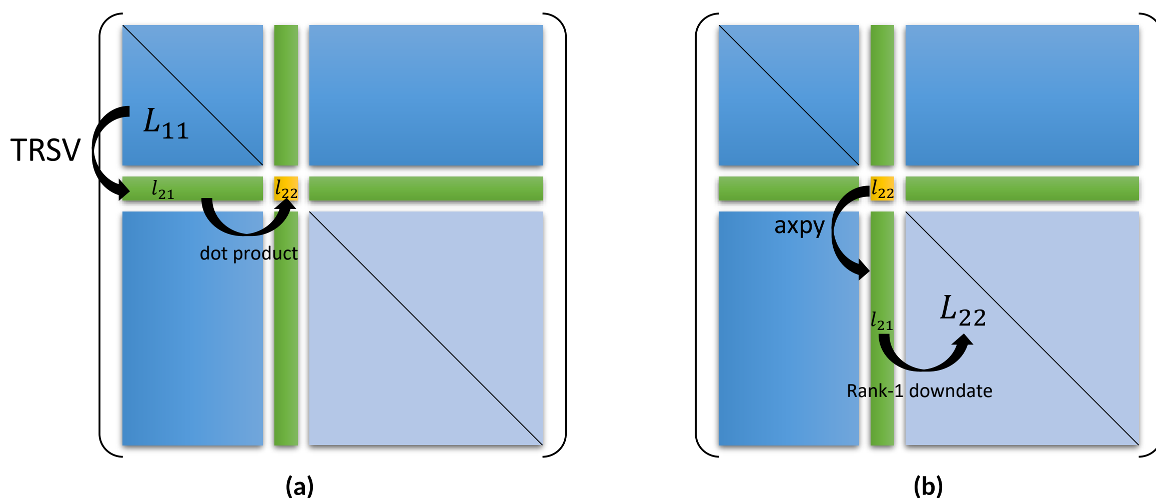


Figure 3.4: The principal operations of **(a)** up-looking and **(b)** right-looking Cholesky factorization. While the up-looking factorization performs triangular solves of the current row with the partial factor, the right-looking factorization downdates the remainder of the matrix by the product $l_{21}l_{21}^{\top}$ in each step. Note that both formulations are equivalent in exact arithmetic.

access by co-constructing both a CSR and CSC index layer on top of the `csr_values` array; this is feasible since the factor's nonzero layout is immutable during execution. For now, we will refer to the nonzero pattern of a matrix A using its calligraphic letter $\mathcal{A}$; moreover, $\mathcal{A}_i$ denotes A's nonzero pattern, limited to row i. If $a_{ij} \neq 0$, then $(i, j) \in \mathcal{A}$. All patterns are structural, meaning they do not take into account possible numerical cancellations during the factorization.

Symbolic Analysis. Each symmetric matrix A can be interpreted as an adjacency matrix, describing an undirected graph with n nodes. In the following part, we will refer to this graph as $\mathcal{G}(A)$ and assume that $\mathcal{G}(A)$ is connected, i.e., for each $i, j < n$ with $i \neq j$, there is a path from node i to node j. We use the shorthand $i \rightsquigarrow j$ for the latter case. In order to compute the nonzero pattern $\mathcal{L}$ of L, we first define a specific subset of $\mathcal{E} \subseteq \mathcal{G}_f(A)$, where $\mathcal{G}_f(A)$ is the Graph for matrix A including all fill-in entries ocurring in a factorization:

Definition 1. *The elimination tree $\mathcal{E} \subseteq \mathcal{G}_f(A)$ is a directed tree, i.e., for each node $i = 0, \ldots, n-1$, there is only up to one node $j = 0, \ldots, n-1$ with $i \neq j$ such that $(i, j) \in \mathcal{E}$, with the following properties:*

(1) $(i, j) \in \mathcal{E} \Rightarrow i < j$

(2) $(i, j) \in \mathcal{A} \Rightarrow i \rightsquigarrow j$ *in* $\mathcal{E}$

Based on the iterating property (2) in the definition above, Liu [1986a] presents an algorithm for the efficient computation. We include a simplified version as Algorithm 6. Essentially, starting from a forest of n nodes, this algorithm proceeds row-wise through $\mathcal{A}$. If a nonzero element A_{ji} is encountered, we proceed to j's highest ancestor (array `ancestor`) in the current forest and add an edge (`ancestor`, i) if `ancestor` $< i$. After the last row, the initial forest has been molded into a single tree.

The question now is: Given the elimination tree $\mathcal{E}$, how do we compute the pattern $\mathcal{L}$? The answer is based on the following theorems of Parter [1961] and Schreiber [1982]:

Theorem 1. *If $(i, j) \in \mathcal{A}$, then $(i, j) \in \mathcal{L}$.*

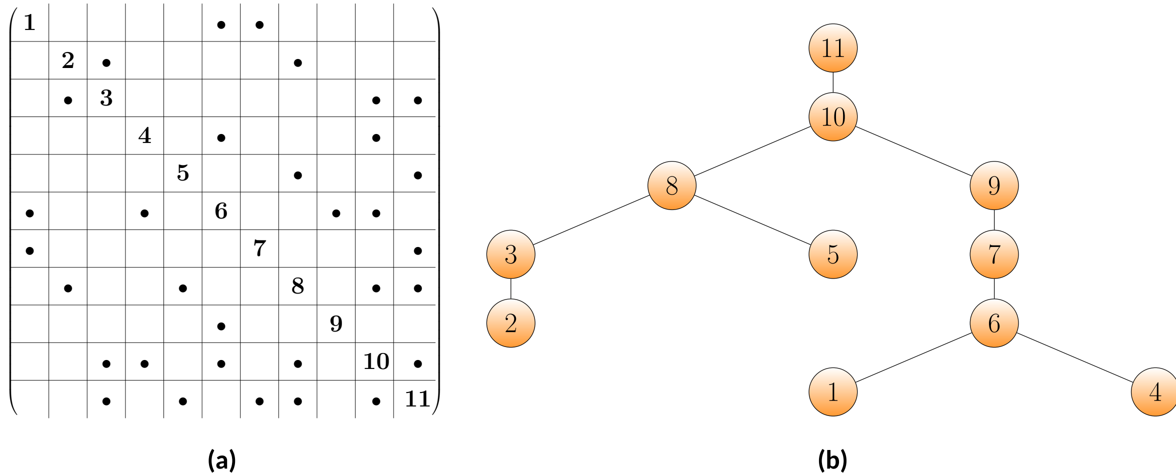
Theorem 2. *For $i < j < k$ and if $(j, i) \in \mathcal{L}$ and $(k, i) \in \mathcal{L}$, then $(k, j) \in \mathcal{L}$.*

For starters, Theorem 1 states that $\mathcal{L} \supseteq \mathcal{A}$. The second theorem leads to an interesting connection to the elimination tree. If we have $(j, i) \in \mathcal{A}$ and $(k, i) \in \mathcal{A}$, then we also have $(j, i) \in \mathcal{L}$ and $(k, i) \in \mathcal{L}$ which implies $(k, j) \in \mathcal{L}$. Similarly, due to $(j, i) \in \mathcal{A}$ and $(k, i) \in \mathcal{A}$, there are paths $i \rightsquigarrow j$ and $i \rightsquigarrow k$ in $\mathcal{E}$. Since $\mathcal{E}$ is a tree, $i < j < k$ implies that the path $i \rightsquigarrow j \rightsquigarrow k$ must be part of the tree. This leads us to the basic principle for the computation of $\mathcal{L}$:

Theorem 3. *If $(j, i) \in \mathcal{A}$, then all nodes $i, k_1, k_2, \ldots, j$ on the path $i \rightsquigarrow j$ in $\mathcal{E}$ are in $\mathcal{L}_i$.*

Davis [2006] proves this theorem, which gives rise to Algorithm 7 that computes $\mathcal{L}$. Please note that this formulation of the algorithm is not meant for implementation. Any efficient implementation requires explicit caching of the ancestor relationships within the elimination tree. If implemented on top of a state of the art disjointed set data structure, the algorithm has a worst case complexity of $\mathcal{O}(|\mathcal{A}| \log n)$.

We visualize a running example matrix in Figure 3.5a and its corresponding elimination tree as Figure 3.5b. Moreover, the last matrix in Figure 3.6 shows all of its fill-in elements. As discussed in Section 3.5, incomplete factorizations use only a subset of all fill-in elements. Threshold-based methods usually skip



(a) (b)

Figure 3.5: (a) Running example for this section. This matrix is symmetric and has a nonzero diagonal. **(b)** Therefore, positions for fill-in during its symmetric factorization can be computed using the *Elimination tree*. The matrix' nonzero pattern is taken from Davis [2006], while all figures were created by the author.

Algorithm 6 Computing the elimination tree $\mathcal{E}$ for the Cholesky factorization of a sparse matrix A (see Figure 3.5a). Arrays `csr_row`, `csr_col` should describe the lower triangular part of A. Since $\mathcal{E}$ is a tree, we describe its edges by the *parent* array `p` such that $\texttt{p}(i) = j \land i \neq j \Leftrightarrow (i, j) \in \mathcal{E}$.

```
 1: procedure etree(csr_row, csr_col)
 2:     p(i) ⟵ i for i = 0, . . . , n − 1
 3:     for i = 1, . . . , n − 1 do
 4:         for j = csr_row(i), . . . , csr_row(i + 1) − 1 do
 5:             ancester ⟵ csr_col(j)
 6:             while p(ancestor) ≠ ancestor do
 7:                 ancestor ⟵ p(ancestor)
 8:             end while
 9:             if ancestor < i then
10:                 p(ancestor) ⟵ i
11:             end if
12:         end for
13:     end for
14: end procedure
```

the symbolic analysis. As their pattern is subjected to frequent changes throughout the numerical phase, they make heavy use of dynamic data structures. Level-based incomplete factorizations use a structured subset of $\mathcal{L}$ and thus execute a symbolic analysis with a different algorithm, first presented by Hysom and Pothen [2002]. The 2nd and 3rd matrices in Figure 3.6 give an example, showing fill-in for levels 1 and 2. Note that in most cases, there is a level $l < n$ from which on the full set of fill-in entries has been included. The concept of the elimination tree can further be generalized. Instead of representing the structure of scalar entries in the matrix, we can build a tree over a matrix that has been divided into dense blocks [Gilbert and Schreiber 1992]. In this case, every node in the tree represents such a block, minimizing the overhead of the symbolic analysis at the cost of more explicit zeros in the final factor.

Numerical Phase. After we set up the nonzero structure $\mathcal{L}$, we then start the actual numerical factorization. All fill-in elements are initially set to zero, all other elements to their respective value in A, after

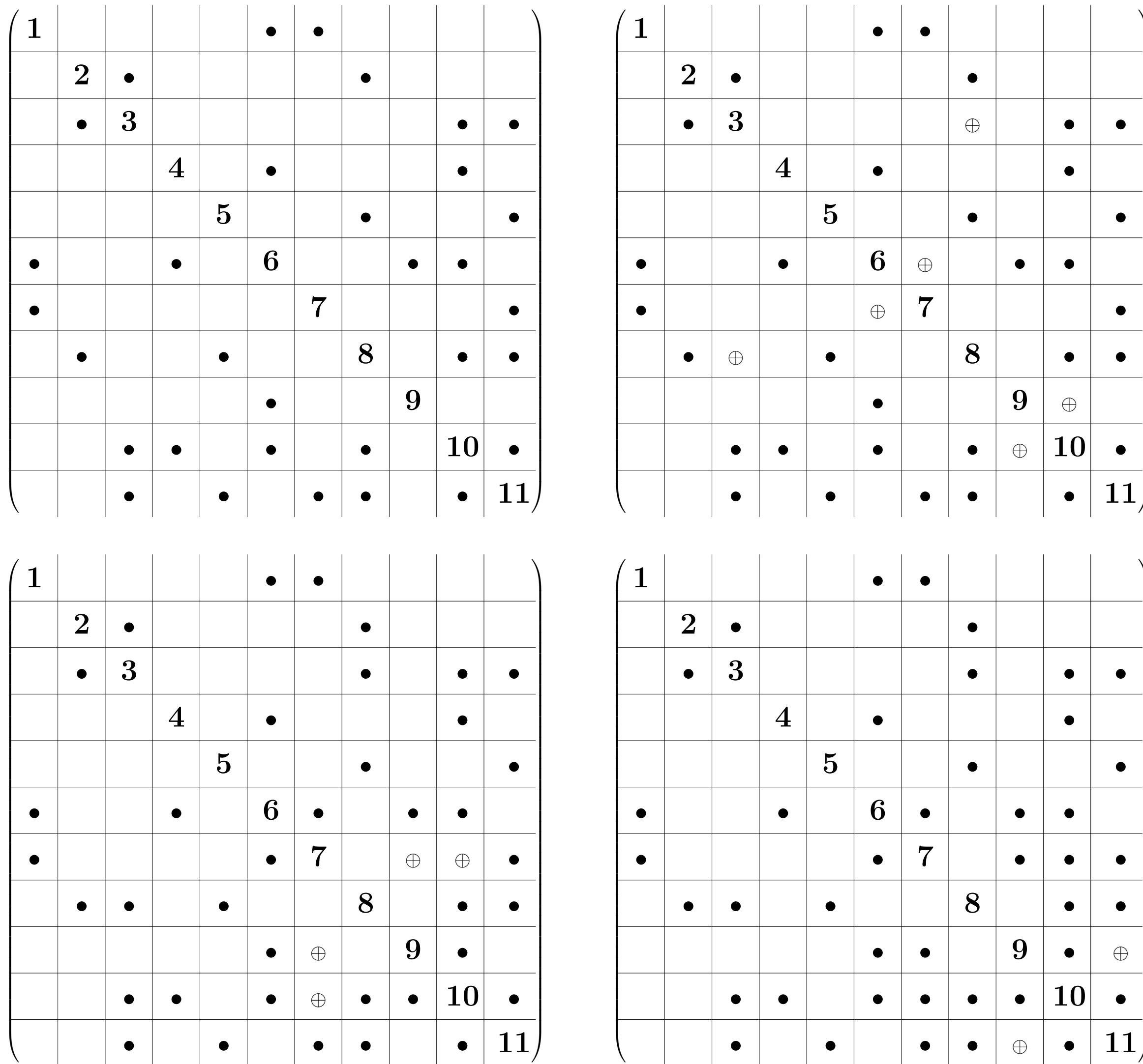


Figure 3.6: Visualizing fill-in levels for the matrix in Figure 3.5a. From **left** to **right** and **top** to **bottom**: No fill-in (input matrix), one level of fill, two levels of fill and full fill-in, corresponding to a complete matrix factorization. New fill-in is always marked with a ⊕ relative to the previous matrix. Running a factorization on an intermediate level of fill leads to incomplete factors that may be used as preconditioners in an iterative method.

which we compute both a CSR and CSC overlay over $\mathcal{L}$. This helps us to efficiently execute Algorithm 5 on $\mathcal{L}$: to execute lines 8 - 10 for column j, we traverse the CSC for column j, scaling each element. Next, we perform the rank-1 downdate in lines 11- 15. We traverse column j by CSC, but then for each traversed element l_{ij}, we traverse row i in the CSR overlay and execute line 13 on all elements $l_{ik}, k > j$. In order to access l_{kj}, it is convenient to store column j temporarily in a dense array.

Alternatively, we can reorganize the execution order: Instead of pushing the downdates caused by column j, we can store them and postpone them to later. In this case, when processing column j, we first pull in all rank-1 updates from previous columns, which each can be computed in the same fashion as outlined above. This order of execution is a specialization of the right-looking factorization, commonly called "left-right looking"; it is further possible to derive it by a similar approach as in Equation 3.18 using a 3×3 block matrix.

Algorithm 7 Computing the full nonzero pattern of L with all fill-in given an elimination tree as well as A's lower triangle in CSR form. Each set $\mathcal{L}_i$ then contains the nonzero pattern of row i.

```
 1: procedure etree2pattern(csr_row, csr_col, p)
 2:     𝓛_i ← i for i = 0, ..., n − 1
 3:     for i = 0, ..., n − 1 do
 4:         for j = csr_row(i), ..., csr_row(i + 1) − 1 do
 5:             𝓛_i ← 𝓛_i ∪ {csr_col(j)}
 6:         end for
 7:     end for
 8:     for i = 0, ..., n − 1 do
 9:         for j ∈ 𝓛_i do
10:             ancestor ← j
11:             while p(ancestor) ≠ ancestor do
12:                 ancestor ← p(ancestor)
13:             end while
14:             𝓛_i ← 𝓛_i ∪ {ancestor}
15:         end for
16:     end for
17: end procedure
```

Parallelization. As sparse factorizations are a workhorse of computational science, researchers started early on to investigate parallelization opportunities, especially to exploit more and more parallel hardware provided by supercomputers. While we proceed column by column from $j = 0$ to $j = n - 1$ in the sequential factorization, the elimination tree already contains all data dependencies required for the parallel factorization. If node j' is an ancestor of node j, then j has to be processed before j'. Thus, the first opportunity for parallelization is the bottom-up traversal of the elimination tree, where independent branches are handled either on different cores, threads or machines; see Gilbert and Schreiber [1992]. Processing a node in the elimination tree requires a subset of all nodes in its subtree, which may also be interpreted as looking at submatrices of L that increase in size while going up the tree. Whenever we encounter a joint parent, these matrices are concatenated. We illustrate this in Figure 3.7b (left). There, the concatenation of previous' nodes submatrices are shaded in gray. Such matrices are commonly called "frontal matrices".

If the nonzero patterns of columns in these frontal matrices are similar enough, they form near-dense matrices. As we discussed above, dense factorizations are often included in vendors' LAPACK implementations, giving rise to the idea of explicitly representing frontal matrices and having them processed by a LAPACK kernel. If the frontal matrix fits into a devices' cache, this idea can lead to runtime improvements of up to an order of magnitude. Moreover, this easily extends to accelerator hardware (see Chapter 4). Individual frontal matrices are transferred asynchronously to the device and processed there. We note that such an "out-of-core" factorization requires a cost function that takes the memory transfer costs into account when determining where to process a frontal matrix [Chen et al. 2008; Haidar et al. 2017; Hogg et al. 2016].

A third opportunity for parallelization lies in the simplification of the elimination tree: If successive columns within the tree have similar nonzero patterns, their nodes can be amalgamated, leading to frontal matrices with a higher proportion of unprocessed column and less small memory transfers respective less overhead for packing and unpacking these matrices. Such "supernodal" approaches

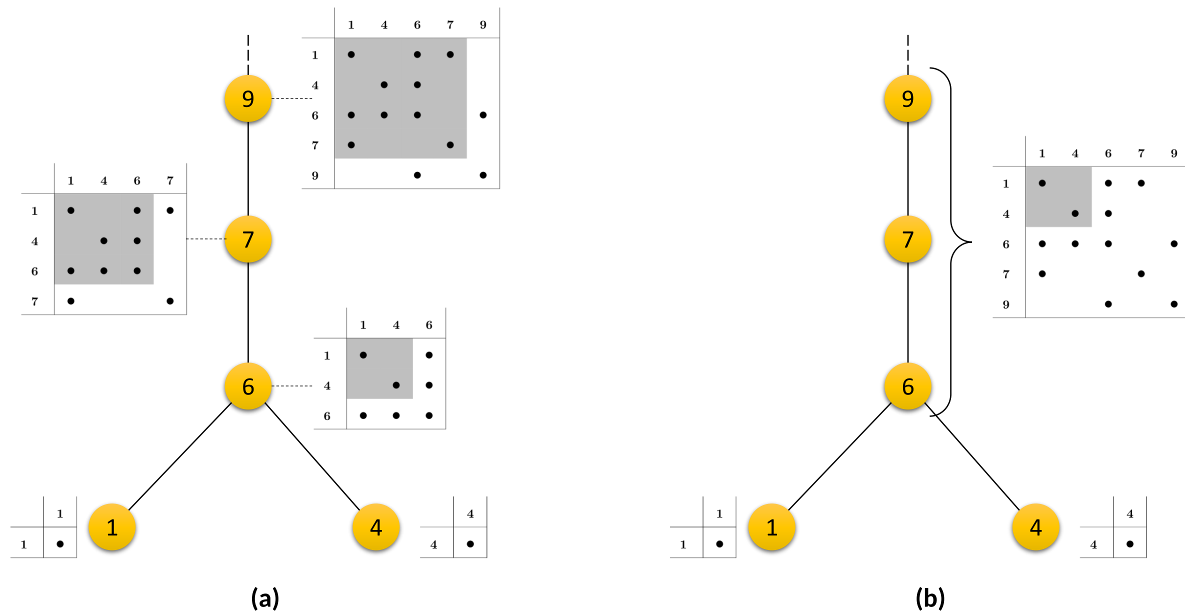


Figure 3.7: When considering only nodes of the elimination tree in the currently processed node's subtree, the induced submatrices are relatively dense. **(a)** Each matrix includes concatenates the children's matrices; these submatrices are called "frontal" matrices. **(b)** In order to efficiently use dense BLAS kernels, we amalgamate multiple nodes of the elimination tree into "supernodes", leading to the "multifrontal" method.

have been first proposed by Demmel et al. [1999b]. The integration of supernodes and frontal matrix processing leads to the "multifrontal" method [Demmel et al. 1999b], the state-of-the-art method for modern solver packages.

As an alternative to the conventional, elimination tree-based scheduling, Hogg et al. [2010] refine the dependency tracking by using a DAG, which can enable a higher degree of parallelism if the matrix' nonzero pattern allows it.

3.6.3 Pivoting

Factorizations based on Algorithm 5 do not consider numerical issues. Since the precision of computers is finite, numerical errors are bound to pop up during factorization and increase in frequency and severity with the matrix' condition number. Line 9 in Algorithm 5 is particularly susceptible: here, we divide by an element on the diagonal. If that element happens to be orders of magnitude smaller than the elements under the diagonal, large numerical errors start to appear. If this happens early in the factorization, they will propagate and lead to a large reconstruction error ($\|A - LL^\top\|_\infty$). Thus, an accurate factorization must avoid this case at all cost.

The first treatments for this issue stem from Gauss-Jordan elimination [Peters and Wilkinson 1975], which is essentially equivalent to LU factorization. Peters and Wilkinson [1975] perform so-called "partial pivoting", i.e., in each step of the factorization, they permute rows such that the subdiagonal element with the largest magnitude in column j is permuted onto row j. This, however, is not guaranteed to bound the growth rate of the factor's entries. In order to guarantee the latter, we extend the search space for large entries to the full remainder of the matrix, so-called "full pivoting". In turn, this increases the complexity of dense LU factorization from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^4)$. As a sort-of in-between, Poole and Neal

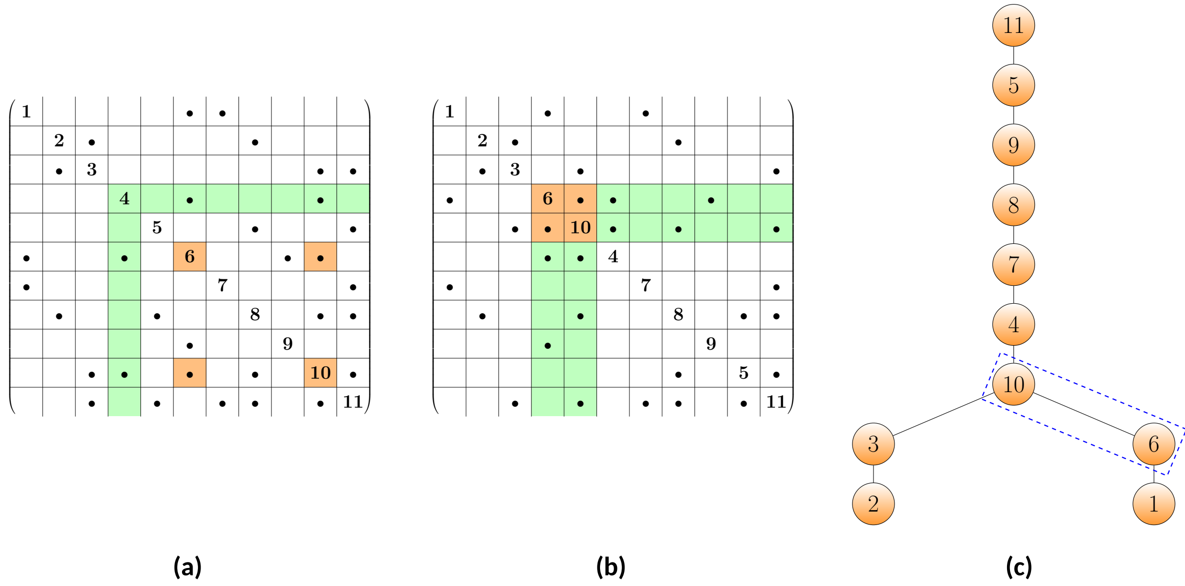


Figure 3.8: Full 2×2 pivoting in the 4th factorization step for the matrix in Figure 3.5. For full pivoting, we query the whole remainder matrix after applying all rank-1 downdates. **(a)** In this example, a 2×2 pivot is formed from frames 6 and 10. Before continuing with the factorization process, we permute both frames to the currently processed column and its successor, resulting in the matrix **(b)**. This pivoting operation did not consider the sparseness of the resulting matrix, leading to an unfavorable updated **(c)** elimination tree and thus potentially many more nonzeros. Note that some implementations fuse the 2×2 pivot into one supernode, illustrated here by the blue box.

Algorithm 8 Rook pivoting, a pivot selection strategy located between partial and full pivoting, in a version for symmetric indefinite matrices, as given by Greif et al. [2017]. This algorithm is guaranteed to avoid exponential element growth [Foster 1997]. Our notation assumes that the factorization is currently processing column j.

1: $\alpha \leftarrow (1 + \sqrt{17})/8$
2: $\omega_1 \leftarrow$ arg max$_{i \geq j}\{|l_{ij}|\}$
3: **if** $|l_{jj}| \geq \alpha\omega_1$ **then**
4:     Use l_{jj} as 1×1 pivot, **return**
5: **else**
6:     $k \leftarrow j$
7:     **while** `true` **do**
8:         $r \leftarrow$ arg max$_{i > k}\{|l_{ik}|\}$
9:         $\omega_r \leftarrow \max_{i \neq r}\{|l_{ir}|\}$
10:         **if** $|l_{rr}| \geq \alpha\omega_r$ **then**
11:             Use l_{rr} as 1×1 pivot, **return**
12:         **else if** $\omega_k = \omega_r$ **then**
13:             Use $\begin{pmatrix} l_{kk} & l_{kr} \\ l_{rk} & l_{rr} \end{pmatrix}$ as 2×2 pivot, **return**
14:         **else**
15:             $k \leftarrow r$
16:             $\omega_i \leftarrow \omega_r$
17:         **end if**
18:     **end while**
19: **end if**

[Poole and Neal 1991; Neal and Poole 1992] introduced “Rook pivoting”, which also scans all remaining columns in the worst case. In practice, it often terminates after only scanning few columns.

While pivoting is theoretically not necessary for Cholesky factorizations, we might still employ it to improve accuracy. In order to maintain symmetry, pivot candidates can only lie on the diagonal. Symmetric indefinite matrices pose another problem: during factorization, we may encounter numerical zeros on the diagonal, leading to a breakdown of the factorization. In order to overcome this, we allow the use of 2×2-pivots, which are then reflected in D of $LDL^\top$. We show an example in Figure 3.8. In Figure 3.8a, we visualize the 4th step of a $LDL^\top$ factorization where we happen to encounter a small pivot below the threshold. Therefore, we use a Rook-like pivot search procedure that locates a 2×2 pivot candidate. Those two rows and columns are then permuted onto rows and columns $\{j, j+1\}$ in Figure 3.8b and treated as a block inside the rest of the factorization.

Permuting rows and columns of a sparse matrix changes its nonzero pattern and thus also the pattern of the generated factor. Consequently, each pivot operation requires then a partial re-execution of the symbolic analysis phase on the remaining, unprocessed columns. The elimination tree in Figure 3.8c is, as a result of these permutations, almost degenerated to a chain. This underscores how important it is to consider both structural and numerical aspects within pivot selection. Symbolic analysis mandates a global synchronization and possibly, expensive memory (re-)allocation. Moreover, other threads working in different branches of the elimination tree must be stopped as well. Therefore, researchers have investigated limitations to the pivot search, e.g., onto the current frontal matrix [Becker et al. 2012; Schenk and Gärtner 2006; Schenk et al. 2001]. While this avoids repeated symbolic analysis phases, it reduces the factors’ accuracy, leading Schenk et al. [2001] to embed their method into an iterative refiner to postprocess the results of their triangular solves. For matrices from optimization, Zhang [1998] proposes to replace small pivots by a very large number. So far, this strategy has not been expanded to general matrices. Lastly, we repeat our statement from above: Good preprocessing can avoid many pivoting operations, by, e.g., initially moving all large elements closer to the diagonal, possibly embedded within a fill-reducing approach [Hogg et al. 2017].

Chapter 4
Heterogeneous Computing

Contents

4.1 Taxonomy and Setup 39
4.2 SIMD Registers 41
4.3 GPUs 46
4.4 Vector Processors 49
4.5 Irregularity Mitigation Strategies 52

Since the widespread adoption of CUDA on the GPU around 2008, heterogeneous computing has become a staple of HPC. As we discussed in Chapter 1, heterogeneous systems are now the main drivers of recent performance increases in the TOP500 list. In this context, *heterogeneous* computing refers to the combination of multiple different architectures in one computing system. In the main part of this thesis, we narrow our scope to the combination of a standard x86 CPU and a massively-parallel compute accelerator card. There is a plethora of accelerator architectures in use in HPC systems everywhere. In this chapter, we focus on three widely adopted architectures and discuss them with their software stacks and developer interfaces. All three classes can run general-purpose programs, unlike recent fixed-function hardware such as Google's Tensor Processing Units (TPUs).

4.1 Taxonomy and Setup

Various parallel architectures, both for CPUs and accelerators, have been proposed in the past. Flynn [1972] introduced a coarse taxonomy that classifies parallel systems by the multiplicity of both the

Class	**GPU / SIMT**	**SIMD (short)**	**Vector / long SIMD**
Representative	NVIDIA Titan V	Intel Xeon 6126 Gold	NEC SX-Aurora 10B
Device RAM [GB]	12	≤ 768	48
Bandwidth [TB/s]	0.653	0.119	1.2
LLC [MB]	4.5	19.25	16
Tensor Ops [TOPS]	110	n.a.	n.a.
FP16 [TOPS]	27.6	n.a.	n.a.
FP32 [TOPS]	13.8	2.6	4
FP64 [TOPS]	6.9	1.8	2.15
Cores	80 (SM)	12	8
TDP [W]	250	125	300
Release	2017	2017	2017
Price at release [$]	3,000	2,100	3,000

Table 4.1: Comparing technical specifications of one representative product per class of accelerator of those presented in this chapter. We note that the definition of "core" varies between the architectures and will be specified in the corresponding sections. TOPS stands for a trillion operations (10^{12}) per seconds in the respective data types.

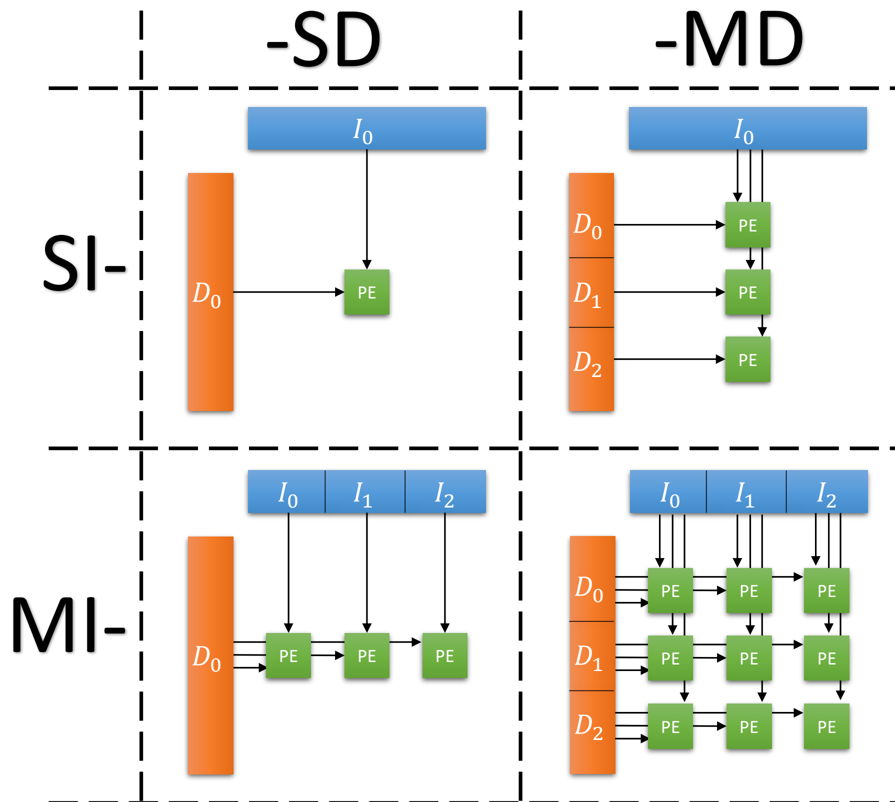


Figure 4.1: Flynn's Taxonomy: We classify systems by the multiplicity of both instructions and data consumed by its processing elements (PEs) in each cycle. In order to get the name for one of the schematic systems, select a row first (instructions; **SI**: single instruction, **MI**:multiple instruction) and then append a column (data; **SD**: single data, **MD**: multiple data). This pattern leads to the well-known classes SISD, SIMD, MISD, MIMD – where MISD is, as far as we know, not realized yet.

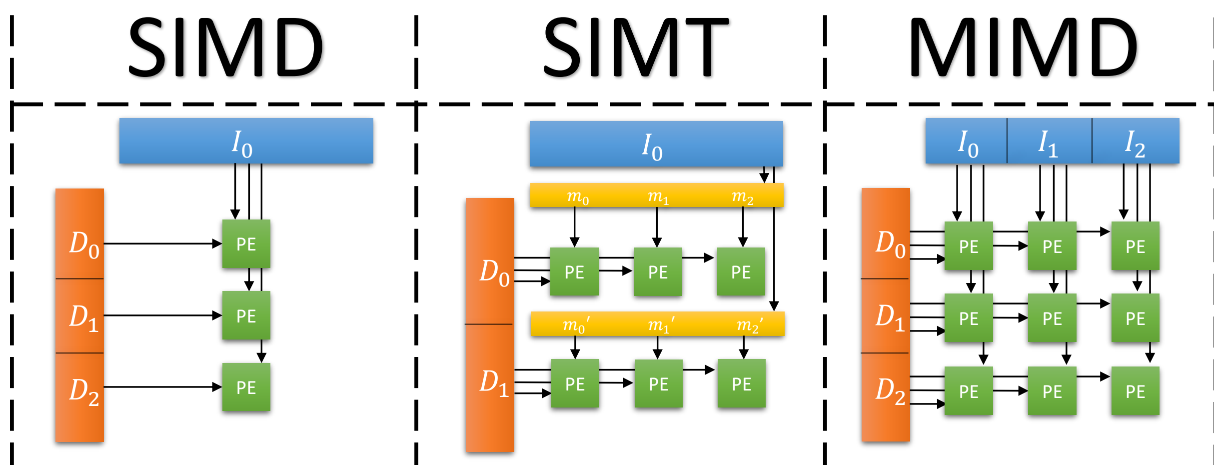


Figure 4.2: SIMT (single instruction, multiple threads) is a term that recently rose to popularity due to its use in conjunction with GPUs. It represents an extension to SIMD. Essentially, groups of PEs (columns) work on multiple data points each, reading the same instruction – but some PEs may be skipping the instruction by masking them out using the (yellow) mask registers. This process is commonly called (SIMT) "predication".

instructions and the data they consume. Flynn's model focuses on Processing Elements (PEs) as the smallest independent units of compute. Each architecture consists of one or more PEs and each PE receives both a stream of instructions (e.g., a stream of assembly code) and data. The resulting "Flynn's taxonomy" proposes organizing architectures in a matrix, where

- the system's instruction multiplicity (single instruction (**SI**) and multiple instructions (**MI**)) are put into the matrix' rows and
- the system's data multiplicity (single data (**SD**) and multiple data (**MD**)) are the matrix' columns.

We visualize that scheme in Figure 4.1. In this context, multiple refers to the number of streams, e.g., a SIMD system (single instruction, multiple data) executes the same instruction on multiple data points. Hence, such a system uses a single, shared instruction stream and different data streams.

With the exception of the Multiple Instructions Single Data (MISD) class, we see practical implementations for all three other classes in the taxonomy: scalar, single-core CPUs implement SISD and multi-core CPUs implement Multiple Instructions Multiple Data (MIMD). When taking vector registers (e.g., Intel's SSE) into account, each CPU also doubles as a SIMD unit, where each lane of the SIMD register processes one data stream.

With CUDA, NVIDIA's marketing department has heavily touted its GPUs as implementations of a SIMT-paradigm in order to distinguish them from multi-core CPUs. While not part of Flynn's original taxonomy, SIMT can be seen as a "in-between" between SIMD and MIMD architectures. As can be seen in Figure 4.2, SIMT extends the traditional SIMD by the introduction of masks (the yellow bars). Using these masks, some of the PEs can skip the current instruction. Thus, a MIMD model may be emulated by concatenating all MIMD instruction streams into a single stream and masking out all but one PE per instruction stream. We say that PEs are predicated by masks, i.e., the execution of an instruction depends on a boolean value (mask).

The combination of multiple architectures, both CPUs and accelerators, promises efficiency by adapting the hardware setup to the tasks at hand. This, however, requires more fine-grained and complicated scheduling, resource allocation and communication between all components of the system, which is a field of research of its own right. In this thesis, we consider only the simplest case: systems that have one (multi-core) CPU as well as (up to) one accelerator card connected via PCIe. In the following three sections, we discuss the architectures and software stacks for three classes of accelerators: short-SIMD registers, GPUs and long-vector cards. An example implementation for each category and their specifications is listed in Table 4.1.

Running example. In order to briefly demonstrate the most popular programming models from a developer's perspective for accelerators, we present an implementation of a basic kernel: SAXPY. SAXPY updates a vector $y \in \mathbb{R}^n$ by $y \leftarrow \alpha x + y$ given a scalar α and another vector $x \in \mathbb{R}^n$. The kernel trivially parallelizes over vector dimensions which results in straightforward implementations in CUDA, AVX512 intrinsics as well as auto-vectorizable code for SX-Aurora (see Listing 1 top to bottom).

4.2 SIMD Registers

We start our presentation by slightly bending the definition of "accelerator". SIMD registers – an implementation of Flynn's SIMD concept – are on-chip extensions to CPUs. At their core, SIMD registers are CPU

Listing 1 SAXPY $y \leftarrow \alpha * x + y$ for $x, y \in \mathbb{R}^n$ and scalar α code example for – **top** to **bottom** – GPU (CUDA), AVX512 (Intrinsics) and Aurora (auto-vectorization via NCC).

```
__global__
void cuda_saxpy(const float alpha, const float * x, float * y, const int n)
{
    const int ix = blockIdx.x * blockDim.x + threadIdx.x;

    if(ix < n)
    y[ix] += alpha * x[ix];
}

void avx512_saxpy(const float alpha, const float * x, float * y, const int n)
{
    const int num_chunks = n / 16 + (n % 16 == 0 ? 0 : 1);
    __m512i ix = _mm512_set_epi32(15, 14, 13, 12, 11, 10, 9, 8, 7, 6,
    5, 4, 3, 2, 1, 0);

    __m512 vx, vy;
    __m512i ix;
    __mmask16 mask;
    for(int i = 0; i < num_chunks; ++i)
    {
    mask = _mm512_cmplt_epi32_mask(ix, _mm512_set1_epi32(num_chunks));

    vx = _mm512_load_ps(x + 16 * i);
    vy = _mm512_load_ps(y + 16 * i);

    vy = _mm512_fmadd_ps(_mm512_set1_ps(alpha), vx, vy);
    _mm512_store_ps(y + 16 * i, mask, vy);

    ix = _mm512_add_epi32(ix, _mm512_set1_epi32(16));
    }
}

void ve_saxpy(const float alpha, const float * x, float * y, const int n)
{
    for(int i = 0; i < n; ++i)
    y[i] += alpha * x[i];
}
```

registers that are wider than 32/64 bit, allowing them to hold multiple 32/64 bit operands and execute the instructions on all of these "lanes". As part of the register file in modern CPUs, they offer a latency of one cycle and have dedicated Arithmetical-Logical Units (ALUs) matching their width. As the first SIMD instruction set for x86 CPUs, MMX (trademark acronym with no official expansion) was introduced in 1997 by Intel. MMX introduced registers `MM0` through `MM7` with a capacity of 64 bit, compared to 32 bit for conventional registers, capable of processing two 32-bit operands at once. Originally, all registers were aliased to the x87 Floating Point Units (FPUs) registers for backwards compatibility. MMX's successor instruction set, Streaming SIMD Extensions (SSE) increased the SIMD width to 128 bit, with the lower 64 bit aliased to the MMX registers. Subsequently, Intel kept expanding its SIMD widths and capabilities of the instruction set through multiple versions of SSE, then Advanced Vector Extensions (AVX) with 256 bit and, most recently, AVX512 with 512 bit. The lower half of each generation's registers are aliased to the previous generation's registers for backwards compatibility as visualized in Schematic 4.4b.

AVX512 was originally introduced as an integral feature of Intel's Xeon Phi line of HPC accelerators. After Phi's discontinuation, they have been transplanted to newer Xeon processors. As long as the AVX512 part of the chip is accessed explicitly in application code, we consider the AVX512 register set to be an accelerator in the sense of this chapter. Hence, all discussion of AVX512-enabled hardware in the remainder of this thesis will only consider AVX512-enabled CPUs. Through licensing deals, modern AMD processors support all SSE/AVX instruction sets through to AVX2. At the time of writing this thesis, AVX512 is still an Intel exclusive. In the area of non-x86 processors, ARM offers its licensees to integrate its own SIMD extension (NEON) into their chips – unfortunately, NEON is not SSE-compatible and requires separate implementations. Recently, ARM introduced its IP extension SVE, with which customers can synthesize arbitrary SIMD widths up to 2048 bit.

4.2.1 Architecture

CPUs have traditionally been optimized for latency-oriented computing, i.e., finishing sequential tasks as early as possible. This paradigm is opposed to throughput-oriented architectures such as GPUs (cf. below) which are optimized to process many independent work items in parallel, maximizing throughput. With that goal in mind, CPU designers have added scores of complex features such as superscalar pipelines, speculative and out-of-order execution. Since it is our intention to compare the "accelerator" parts of different architectures, we only touch those features very briefly and refer the interested reader to Shen and Lipasti [2013].

In the following, we use Intel's 2011 Sandy Bridge microarchitecture for our discussion, since it is well-documented and its design still forms the core of more modern Intel designs. A schematic for such a CPU chip is given in Figure 4.3. Each CPU core offers a dedicated register file, L1 and L2 cache and can scale its clock frequency to stay in the TDP (thermal design power) envelope. All CPU cores (for Sandy Bridge Core i3/5/7, that is up to 8 cores) are connected by a ring bus and share a L3 Last-Level Cache (LLC). In addition to the latter, the ring bus allows simple inclusion of additional hardware such as an integrated GPU. At the same time, it connects cores to the system controller that interfaces with the system's Random Access/Main Memory (RAM) and PCI Express (PCIe) bus. Modern CPU cores support Hyper-Threading (HT), the ability to execute context switches between two threads in a short time. In hardware, HT is realized by using two separate register files. Through HT, pipeline stalls after memory accesses can be hidden by issueing executions from another thread after a context switch. Without HT

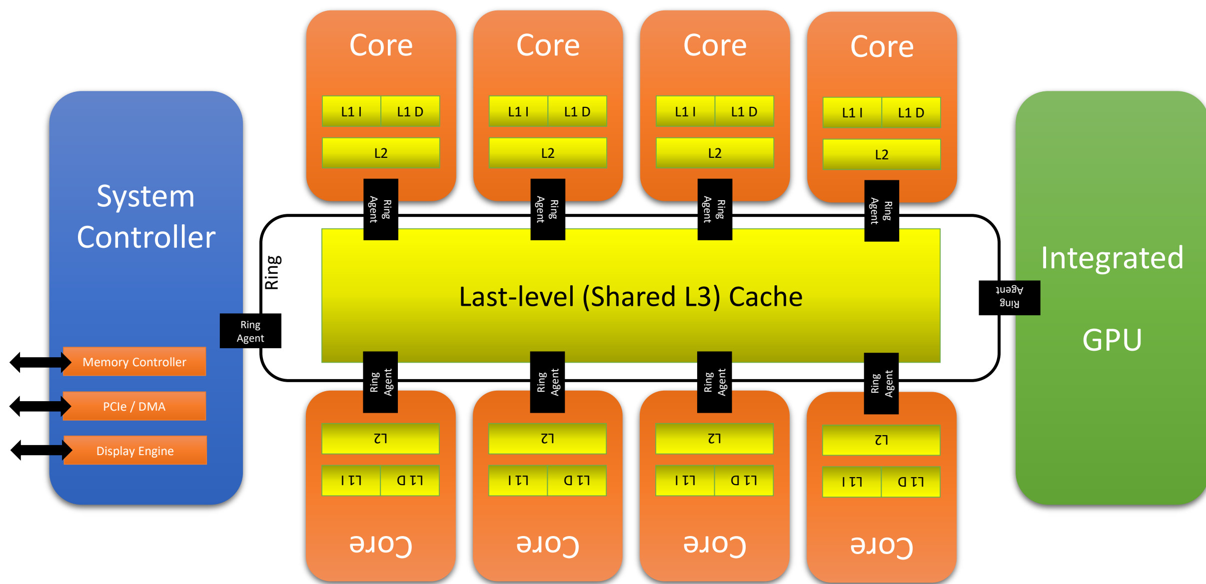


Figure 4.3: Schematic for a modern Intel multi-core CPU (octacore) with Sandy Bridge microarchitecture (introduced in 2011). While each core has its own Level 1 and Level 2 caches, the last-level cache is shared between all cores. The ring bus between cores simplifies scaling the architecture to both desktop and server CPUs.

context switches require a backup of the core state (i.e., register file and status registers) to the cache, which can be prohibitively expensive. Through the combination of HT, the ring bus and frequency scaling, the Sandy Bridge microarchitecture can scale up from mobile processors (15W Thermal Design Power (TDP)) to Xeon-line server processors for high performance (150W TDP).

We continue with a brief analysis of the architecture of a single CPU core as in Figure 4.4a. In the past, there has been a fierce debate about the benefits of RISC and CISC (reduced vs. complex instruction set computing) style Instruction Sets (ISAs) [Bhandarkar and Clark 1991]. Hennessy and Patterson [2019] witnesses that the industry has mostly settled for RISC. In order to maintain support for the original x86 architecture, Intel has implemented a hybrid CISC-RISC ISA. While the ISA follows the CISC paradigm on the outside, in the first pipeline stage, CISC-flavored operations are decomposed into μOps that resemble RISC instructions. Those μOps are then executed out-of-order from the instruction queue by reordering and register renaming to avoid hazards and extract Instruction-Level Parallelism (ILP) by detecting independent instructions. Since instructions are executed in different units with different latencies, instruction buffers help avoid pipeline stalls. Additionally, Intel's core design uses a branch predictor to speculatively execute later parts of the program when there are idle execution units to minimize latency. While very effective, this method came under scrutiny as the cause of the Spectre [Kocher et al. 2019] and Meltdown [Lipp et al. 2018] flaws that allow undetected leaking of information. After reordering and register renaming, the resulting μOps are scheduled onto different ports which feature execution units such as ALUs or Load/Store controllers; given enough register space, these ports can be used in parallel. There are ports for both scalar and vector operands which are handled simultaneously in the μOp pipeline. We note that each core uses a single L2 cache for data and instructions, but splits them successively into two separate L1 caches that amount to few kilobytes, but come close in latency to registers. By balancing the mentioned features that are geared towards minimizing latency with enough SIMD additions, CPU cores can be adapted to handle throughput-oriented computing tasks gracefully, as long as the combined

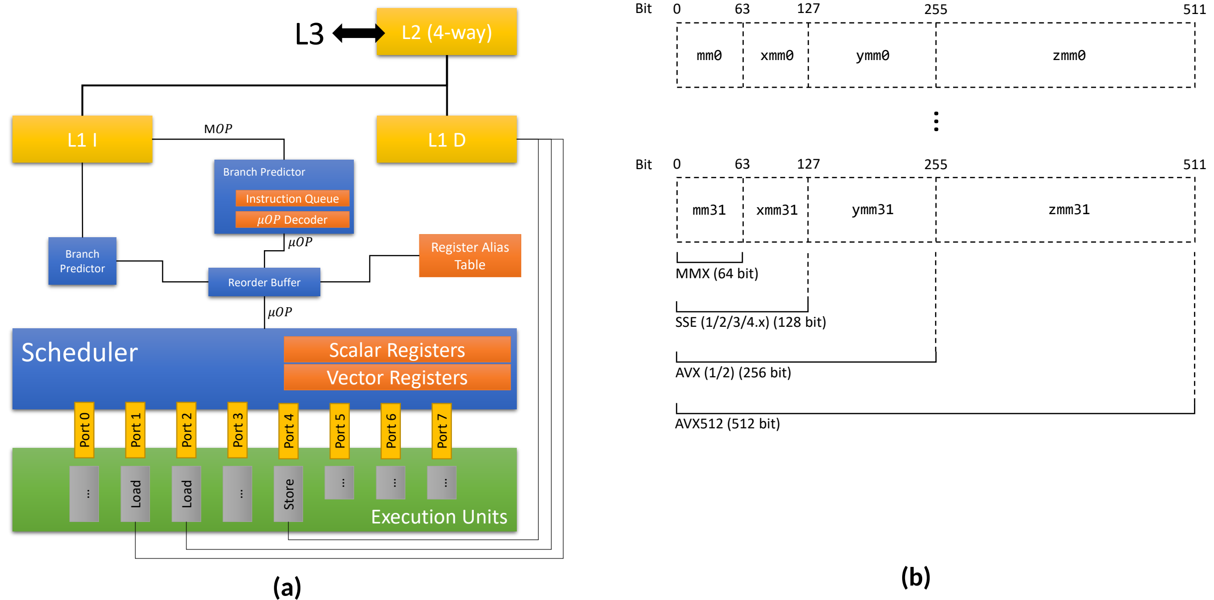


Figure 4.4: (a) Schematic of a core of Figure 4.3. Notably, instructions from the assembly are decomposed into smaller μOPs and re-merged on demand in order to more efficiently use all execution units via their respective ports. Each port can only execute a subset of operations. **(b)** Intel has successively extended the width of their vector registers. Compatibility between the instruction sets is maintained by aliasing the previous generations' registers to the lower halves of current-gen vector registers.

design adheres to the financial and thermal budget.

4.2.2 Developer Ecosystem

CPUs have a long tradition, so there are various programming models, compilers, libraries and software stacks. Due to our focus on the SIMD extension of CPUs, we restrict ourselves to models and tools that are tailored for SIMD programming. In general, there are three ways of accessing SIMD registers:

1. explicitly by (inline) assembly or intrinsics,
2. implicitly by either auto-vectorization or compiler hints (e.g. `pragma simd`) or
3. Single Program Multiple Data (SPMD)-on-SIMT.

Intrinsic instructions mirror assembly mnemonics almost 1:1 into the high-level language and are therefore direct, but rather complicated to use. Auto-vectorization respective compiler hints are simple to use and require almost no additional development time, but may be brittle. As stated for the NEC auto-vectorizer [NEC Corporation 2020], compilers try to find patterns in the code or analyze loop indices [Moll et al. 2019]. Depending on the code normalization before the analysis, some opportunities for vectorization may be missed. Pragmas that can help the compiler with that have recently been added to the OpenMP standard. However, those are pragmas mostly limited to chunking and vectorizing for-loops. The last category, "SPMD-on-SIMD", was pioneered by ISPC [Pharr and Mark 2012] and later consolidated in the OpenCL standard [Stone et al. 2010]: similar to CUDA (cf. below), we implement scalar, parameterized programs that are embedded in a virtual compute grid and the compiler maps each program instance to a SIMD lane, generating intrinsic code in the process. Without further hardware support for branching, this approach only shines for regular, non-diverging code – but in the best case, it can reach speed-ups in the order of the number of SIMD lanes.

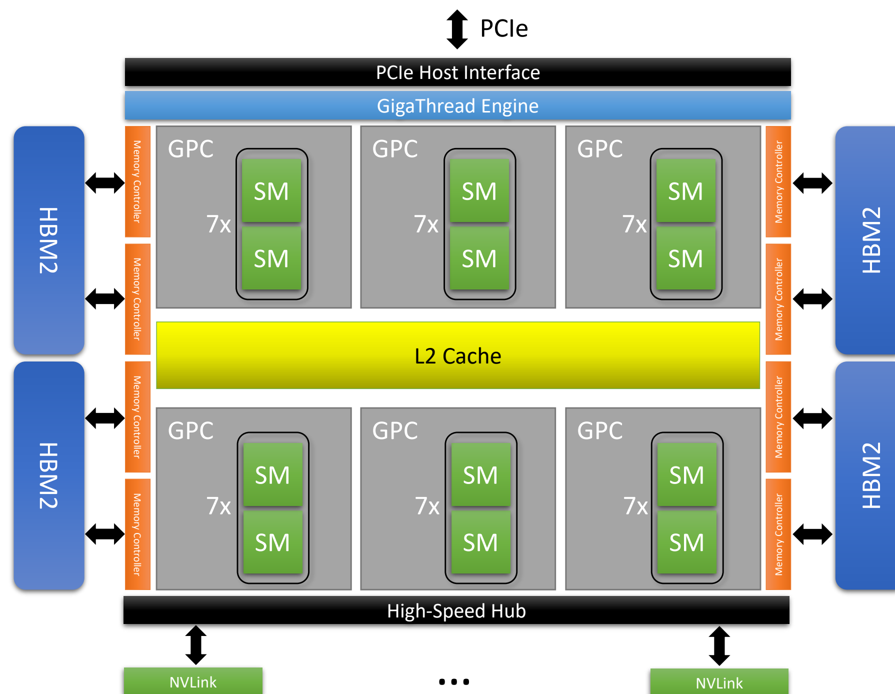


Figure 4.5: Schematic of the GV100 GPU chip (Volta generation). The 24GB of HBM memory are split into stacks connected to two 512 bit memory controllers each. The computational cores are split into 6 global processing clusters (GPC) with 7 texture processing clusters (TPCs) of two streaming multiprocessors (SMs) each, all sharing 6144 KB of L2 Cache. The SMs are the central computational primitives where CUDA blocks are scheduled to. Schematic similar to [NVIDIA Corporation 2017].

So far, we have mentioned programming models that only target SIMD execution. In addition to that, we can also use processes and threads as another layer of parallelism on CPUs. Frameworks like Intel's TBB can simplify the combination of threading and SIMD programming.

4.2.3 Example

Since intrinsics best illustrate the direct use of SIMD units, we use Intel's AVX512 intrinsics for SAXPY in our running example (Listing 1, second function). Here, we take the approach of chunking the loop over vector dimensions and processing each chunk of 16 floats inside a SIMD register (AVX512). Therefore, we load chunks of x, y in lines 14 and 15, use a fused multiply-add to compute $y + \alpha x$ in line 17 and store the result to y in line 18. If n is not a multiple of 16, the load and store operations are at risk of accessing invalid memory areas. Thus, we maintain the current vector indices in variable `ix` and use it to generate a mask that removes all lanes with `ix ≥ n` from consideration (line 12). AVX512 introduces the use of such masks for many of its instructions. In a performance-oriented implementation, one would factor out the last chunk of the loop and avoid the overhead of masking in all other chunks. Other SIMD instruction sets (ARM NEON, NEC Aurora) resemble the AVX512 instruction set, leading to similar code.

4.3 GPUs

While GPUs have been around since the 1970s as coprocessors for 2D framebuffer rendering, the modern GPU was not popularized until the early 1990s. Back then, the PowerVR and 3dfx Voodoo products became affordable for a larger audience. In each generation, chip designers moved towards more and more freely

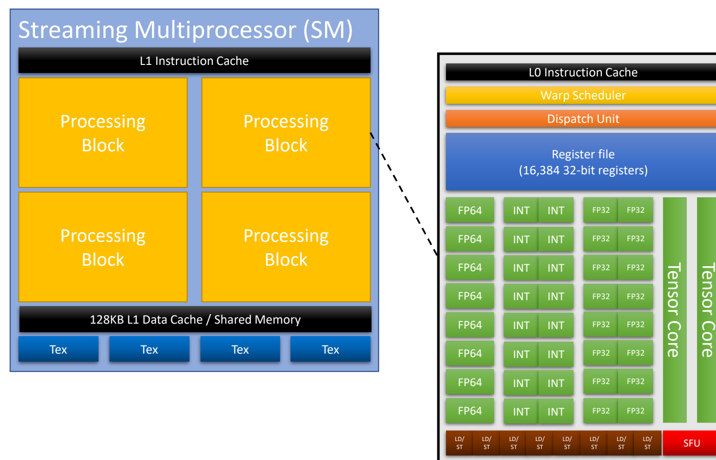


Figure 4.6: Schematic of one GV100 SM, the execution units in Figure 4.5. Apart from the L1 instruction cache, each SM contains up to 96KB of shared memory that is shared between all threads in a CUDA block. The L1 data cache and shared memory share the same 128KB storage and users are able to select different splits between the two. Each processing block handles a subset of a blocks' warps, including a register file and actual compute units for 32 and 64 bit operands. Schematic similar to [NVIDIA Corporation 2017].

programmable compute units (called "shaders"). As early as 1999, researchers figured out how to use the throughput-oriented, fixed-function pipelines of graphics chips for other tasks such as convolution [Hopf and Ertl 1999] or matrix multiplication [Larsen and McAllister 2001]. First programming systems such as Brook [Buck et al. 2004] made this approach more accessible, but it was not until the release of NVIDIA's CUDA 1.0 (Compute Unified Device Architecture) [Garland et al. 2008] in 2008 that opened up its Tesla architecture[1] to general-purpose computing, coining the term GPGPU (general-purpose GPU). CUDA's programming model was later generalized to non-NVIDIA GPUs and other hardware through standardization in Khronos' OpenCL standard [Stone et al. 2010]. However, due to the popularity of NVIDIA's products and its dominating position in the datacenter GPU market, CUDA has prevailed as the quasi-standard for programming GPUs.

4.3.1 Programming Model

CUDA's programming model mirrors the throughput-oriented nature of GPU hardware. GPU programs, so-called "kernels", are implemented as C++ code describing the computation a single thread is doing. At runtime, thousands of threads are launched. These threads are organized into a 3D grid that contains 3D "blocks" of up to 1024 threads. All threads inside a block can synchronize and exchange data in a shared memory area, but there are no effective means to communicate between blocks. Other than a CPU thread, a CUDA thread resembles more a SIMD lane with its own program counter (PC) and hardware mechanisms for masking and lanes. In our running SAXPY example, one thread would e.g., compute one element of the vector y, meaning we launch n threads at runtime. Then, the size of the CUDA blocks is a parameter up for tuning.

4.3.2 Architecture

Compared to CPUs, GPUs sacrifice control logic (e.g., Simultaneous Multi-Threading (SMT) and Out-of-Order Execution (OoO)) in favor of simple, but orders of magnitude more ALUs. For that reason, GPUs are

[1]Not to be confused with the Tesla GPU brand!

also characterized as “massively-parallel” processors. As for CPUs, we sketch the chip architecture of the NVIDIA Volta generation’s GV100 in Figure 4.5. There, the rough analogue to a CPU core is the Streaming Multiprocessor (SM). 14 of such SMs are grouped each into are Global Processing Cluster (GPC) of which there a 6 on GV100. All GPCs share a L2 cache and 8 memory controllers that interface with up to 24GB of HBM2 memory. For multi-GPU systems, NVIDIA offers its own interconnect, NVLink, which is accessed by the SMs through a shared high-speed hub.

Within warps, threads operate in lockstep and inactive threads (e.g., due to branching or conditionals) are masked out (“predicated execution”). In GPUs, instruction can be predicated by a boolean predicate, akin to mask registers in SIMD processors. In case of diverging threads, i.e., threads inside a warp that take different paths in the code, GPUs use hardware logic to trace the partitions’ execution paths and reconcile different partitions during execution. At runtime, multiple blocks are scheduled to an SM (see schematic in Figure 4.6). Other than threads on CPUs, these blocks partition the resources on an SM. Warps from all blocks are partitioned onto the SM’s 4 processing blocks. Through that, issuing instructions from different warps respective blocks inside the same processing block does not require any context change. Essentially, each warp scheduler collects the next instruction from all warps that are scheduled to its processing block and can submit any of those at any time – we say that “multiple warps are kept in-flight”.

This hardware multithreading as well as the reliance on ILP within the same warp are two key features that GPUs rely on for latency hiding. Due to the lack of OoO, any memory access that produces a cache miss stalls the respective warp for 100s of cycles. During that stall, hardware multithreading kicks in and continues to execute instructions from other, in-flight warps. GPUs are most effective when their task has a high ratio of arithmetic operations over bytes transferred from memory. In order to minimize the memory transfers as much as possible, the architecture offers a multi-level memory hierarchy apart from the caches. On top of the HBM2 device memory, each SM has 128 KB local memory that is split between L1 cache and shared memory[2]. Through the latter, threads from different warps in the same block can exchange information about 10 times faster than through the device memory (“global memory”). On top of that, each processing block offers a register file of 16,384 registers with 32 bit each. Through that, threads within the same warp can exchange data in 1 cycle through “shuffle” instructions. Thus, exploiting both shared memory and shuffle operations can lead to large speedups. For more details, we refer to Weber and Goesele [2017]. When accesses to global memory are necessary, all accesses from all (active) threads in a warp are collected and collated into one access per 128B cache line. Whenever consecutive threads access consecutive memory addresses with a stride of 4B, we talk of “coalesced” memory accesses. Such accesses use the cache to the optimum advantage and achieve the highest throughput rates on a GPU.

In recent GPU architectures (Pascal and subsequent), NVIDIA has introduced major new hardware features. “Tensor cores”, two in each processing block, speed up reduced-precision GEMM-like operations for deep learning. Additionally, separate integer and floating point execution units allow dual issue of instructions. Lastly, with “independent thread execution”, each CUDA thread maintains its own Program Counter (PC) which allows ILP between multiple subgroups of threads that occur, e.g., through conditionals (cf. below). Additionally, unified memory allows to create memory buffers that are transparently transferred between

[2] On some GPUS, users can customize this split.

host and device and between multiple devices using a unified address space and address translation in hardware.

4.3.3 Developer Ecosystem

The widespread adoption of GPUs, especially with the deep learning boom, has lead to a rich ecosystem around GPUs. The central interface, though, is still the "native": CUDA. NVIDIA continuously updates CUDA to reflect the latest modifications in the hardware and documents it in a programming guide [NVIDIA Corporation 2020a]. Using CUDA, both GPU and CPU code can be written into the same file.

The CUDA build pipeline then processes this file (usual extension: `.cu`) as follows: first, the compiler driver `nvcc` splits host- and device code. The host code is then pipelined into the host compiler (e.g., `gcc`). The device code is first compiled into PTX code [NVIDIA Corporation 2020c], an assembly-like language for a virtual GPU architecture. PTX uses an infinite number of registers and follows – up to branching – the SSA (single static assignment) format. Only in the last compilation step, when the exact model of the GPU with its characteristics and number of registers is known, the hardware registers are assigned and the PTX code is compiled to the final machine code. Machine codes for multiple GPUs can be packed into a "fat binary" in order to avoid runtime recompilation. At launch time, kernel calls are configured by their compute grid dimensions and the amount of shared memory per block.

By continuous investment, NVIDIA has build a whole ecosystem around GPUs with all sorts of libraries and integrations into frameworks such as PyTorch and TensorFlow. As an alternative to CUDA, the open standard OpenCL by Khronos extends the SIMT programming model to devices of multiple vendors. Unfortunately, NVIDIA has been slow to improve its support for the standard so that OpenCL has failed to gain the community's support and is largely irrelevant now. Apart from OpenCL, other projects have tried to create open alternatives to the CUDA framework. gpucc [Wu et al. 2016] has recently been integrated into the LLVM project as an alternative device-side compiler. In order to simplify GPU programming for non-experts, unified programming models for host and device like PacXXV2+ [Haidl et al. 2017] or support for offloading computations by compiler directives in the form of OpenACC [Wienke et al. 2012] have been proposed. As the "spiritual successor" of OpenCL, SYCL [Subgroup 2015] tries to re-capture the HPC audience by offering an open standard for unified programming over both host and device with an integrated memory management.

4.3.4 Example

As before, we implement SAXPY, using CUDA in Listing 1 (first function). As CUDA's programming model is thread-centric, the code itself is written in a scalar manner. In SAXPY, CUDA blocks do not hold a special meaning as there is no cooperation between threads. Hence, all threads are equal and compute their vector index in line 4. Line 7 then contains the SAXPY operation, potentially being masked out through row 6. Since `y` is kept and updated directly in global memory, we do not need an explicit store operation. We note that line 6 leads to the creation of a predicate for line 7 and that line 7 is converted into a hardware-supported Fused Multiply-Add (FMA) operation by the compiler.

4.4 Vector Processors

Vector computers (also: array processors) have been prominent in the early years of HPC, with notable instances being the Cray-1 in 1975 or NEC's SX-line after 1983. In fact, systems of that line held the first

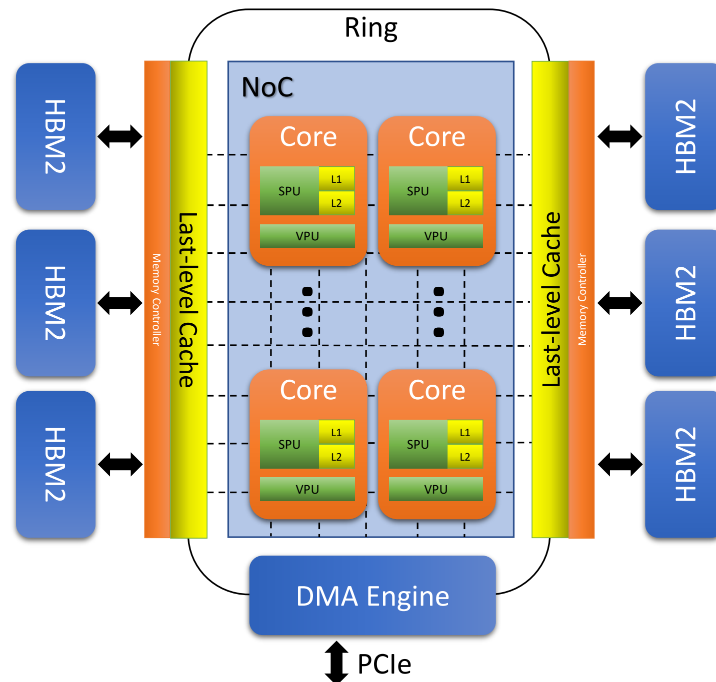


Figure 4.7: Schematic of NEC's SX-Aurora architecture. Up to 48GB of HBM2 memory are accessed through memory controllers and cached in a 16MB last level cache (L3) shared by all cores. Pieces of the LLC are connected through a ring and also to a network on chip (NoC). Each core is placed on the NoC as well and acts similar to a CPU core, integrating a scalar unit (SPU), vector unit (VPU) and instruction/data caches. Schematic from [Yamada and Momose 2018].

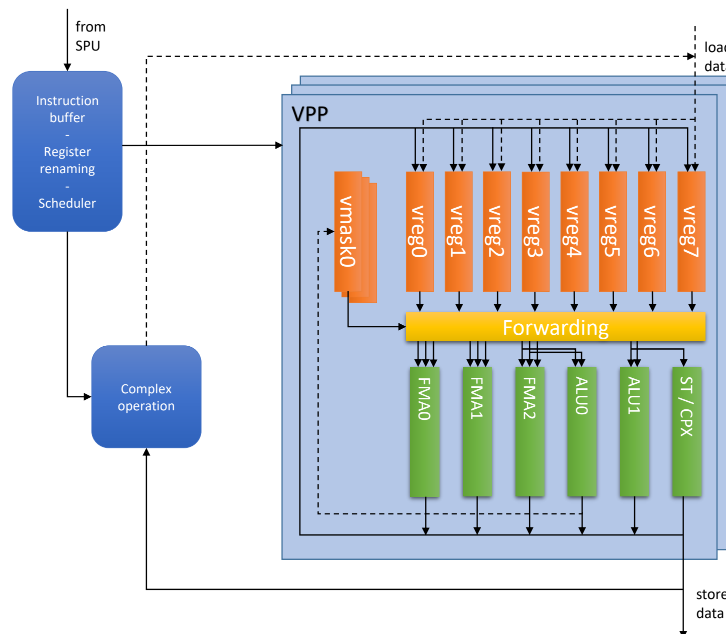


Figure 4.8: Schematic of the VPU inside a SX-Aurora core as in figure 4.7. In each clock cycle, the SPU can submit one vector instruction to the VPU where it is renamed and put into an instruction buffer. From there, it is scheduled to one of eight vector processing pipelines (VPP), each with a set of 8 vector registers and 2 vector mask registers as well as ALU/FMA and complex functional units. The latter processes instructions sequentially over the vector registers' lanes. Schematic from [Yamada and Momose 2018].

place in the TOP 500 list from 1992 to 1993 (SX-3/44) and again from 2002 to 2004 (SX-6 / Earth simulator). From a modern perspective, vectors are just wide SIMD registers with, e.g., 16,384 bit compared to the 512 bit of modern Intel SIMD registers (short SIMD). However, vector computers come from a different place, historically. Their processors were always designed around vector registers while CPUs were original scalar machines and later added SIMD registers. Today, the architecture of vector processors does feature elements from both, though with lane masking and runtime-configurable vector lengths, some features from early vector machines have prevailed. In that sense, modern vector processors represent an interesting compromise between the latency-oriented approach of CPUs and the throughput-oriented approach of GPUs by combining some aspects of both.

Today, the SX-Aurora "TSUBASA" from NEC is the latest implementation of a vector computer, but this time on a PCIe card instead of a physical, separate machine. While it was originally designed to execute programs autonomously from the host machine, recent additions allow to use it as a accelerator in offload mode, similar to GPUs. In this thesis, we use SX-Aurora only in this mode.

4.4.1 Architecture

We present the usual schematic for the SX-Aurora "TSUBASA" (in the following: Aurora) architecture in Figure 4.7. As we have teased, its design contains elements of both CPUs as well as GPUs. Up to 48 GB of HBM2 memory are accessed via an 16 MB L3 (LLC) cache whose chunks are connected through a ringbus with a DMA engine that interfaces with PCIe. Other than Sandy Bridge's ringbus, Aurora's ringbus does not directly connect to its cores, but rather to a mesh-shaped Network-on-Chip (NoC). This NoC allows for lower-latency point-to-point communications between cores compared to a ringbus. While the first generation of Aurora (10x) has up to 8 cores, the second generation (20x, released in 2020) expands that limit to 10 cores and increases the bandwidth to 1.5 TB/s while the principal architecture and ISA is unchanged. Each core integrates both a Scalar Processing Unit (SPU) and a Vector Processing Unit (VPU) as well as dedicated L1 and L2 caches.

A SPU is an out-of-order scalar core with 4-fold instruction fetch per cycle and 1.6 GHz clock frequency with a custom ISA [NEC Corporation 2019]. Each SPU has its own L1 data (operand) and instruction cache as well as an L2 operand cache. When using Aurora in the host mode, the SPU mainly executes all scalar code. In the offload mode, it is merely degraded to submitting one vector instruction per cycle to the VPU. The VPU (schematic in Figure 4.8) operates purely on instructions passed from the SPU. The VPU, much like the SPU, also executes the passed instruction out-of-order. Therefore, each VPU has its own (vector) instruction buffer, a dedicated register renaming unit and 256 vector registers of 16,384 bit each plus 16 vector mask registers. The registers are distributed over 8 VPPs (vector processing pipelines) which each fill 2 FMA units, 1 combined FMA/ALU unit and one combined ALU/ST/CPX unit. The latter executes so-called "complex" instructions such as `VMV` that require communication between lanes of the same register. Complex instructions are executed iteratively, one lane per cycle, and controlled by a dedicated scheduler that (re-)fills the vector instruction buffer detached from the SPU. Thus, such complex operations have a long latency of up to 256 cycles (for 256 double precision operands in a vector). Examples here are reductions inside a register or the lane-shift `VMV`. When not processing complex instructions, the CPX unit also takes care of memory store operations; when scatter instead of a contiguous store is used, the complex unit again processes this lane by lane.

Aurora cores, similar to CPU cores, also support SMT via frameworks like POSIX or OpenMP. However,

each context change must store both scalar and vector registers on the stack and thus takes long. Measurements with OpenMP have shown that, in the worst case, a context switch can take up to a second. However, since both VPU and SPU have their own instruction buffer, latencies can be hidden without context changes – if there is enough ILP in the code. Still, due to its relatively complex layout with different execution units and the long registers, Aurora's hardware is notoriously hard to exploit in HPC codes. However, if a suitable code permits it, only about 2 flops per byte are required for efficient usage, compared to 8 flops per byte in earlier GPU architectures. In Chapter 9, we use this architecture as a basis for an efficient accelerator tailored towards irregular, throughput-oriented codes.

4.4.2 Developer Ecosystem

Due to the mentioned difficulty in developing efficient vector code and the rather limited market penetration of Aurora, NEC has largely relied on its compiler's auto-vectorization functionality (NCC) as a primary means of development. However, auto-vectorization can be brittle since it relies on the recognition of certain patterns that can be converted into vector instructions in the code. NCC offers a great deal of diagnostic output, but its inner workings are opaque to the user. As an attempt to open up the system, NEC supports the LLVM project by contribution the support for both Aurora intrinsics (LLVM-VE)[3] as well as an open, principled auto-vectorizer [4] [Moll et al. 2019] to the LLVM project. Futhermore, open source projects AVEO[5] and VEDA[6] simplify the use of Aurora in offload-mode by mirroring the respective CUDA API. Still, compared to Intel and NVIDIA, the Aurora software stack is not as mature or integrated to the point where it is suitable for widespread use.

4.4.3 Example

As auto-vectorization is the recommended programming model for SX-Aurora, our SAXPY example (Figure 1, bottom) contains just a sequential loop. NCC supports multiple patterns that can be vectorized; our code matches one of these patterns as witnessed by NCC's vectorization report:

```
ncc: vec( 101): test.c, line 3: Vectorized loop.
ncc: vec( 128): test.c, line 4: Fused multiply-add operation applied.
```

The vectorized code (in assembly) largely resembles the manually vectorized code of the AVX512 example.

4.5 Irregularity Mitigation Strategies

So far, we have discussed three different (accelerator) architectures in this chapter. Regarding their design goals, some architectures are in stark contrast. While CPUs with SIMD registers are designed to minimize latency of sequential computations, GPUs originate in 3D rendering and thus are constructed with throughput maximization in mind. The vector processor Aurora is closer to a CPU + SIMD design, but requires extra consideration due to practical issues (e.g., time needed for context switches).

In our case, the term "irregularity" refers to the situation when data accesses or control flow depends on runtime values and can vary dramatically between multiple instances or runs of the same program. Consequently, we distinguish between *data* and *control flow* irregularity. Data irregularity results in

[3]https://github.com/sx-aurora-dev/llvm
[4]https://github.com/sx-aurora-dev/llvm-project/releases/tag/llvm-ve-rv-v1.7.0
[5]https://github.com/SX-Aurora/aveo
[6]https://github.com/SX-Aurora/veda

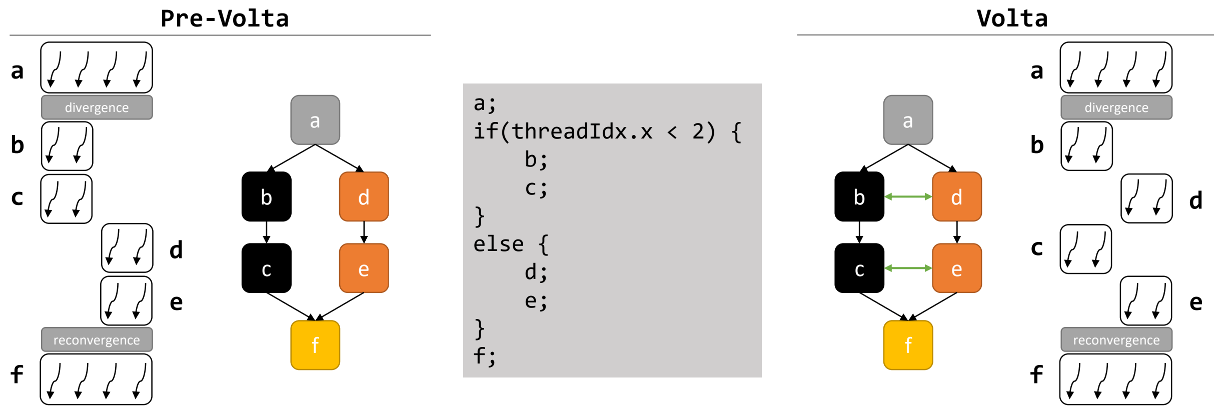


Figure 4.9: Thread scheduling for NVIDIA GPUs before the Volta generation (**left**) and from Volta on (**right**). From inside to outside: Simplified code example, control flow diagram and resulting execution flow for an imaginary thread of size 4. Before Volta, the whole warp was only following one PC, having to execute full branches in order. Since Volta, each thread has its own PC, so multiple branches may be interleaved.

unpredictable and unstructured memory accesses and variable data length or data imbalances between instances of the same program. Control flow irregularity refers to divergent control structures (e.g., conditions and loops) or fine-grained synchronization. Here, the semantic meaning of "program instances" depends on the hardware architecture: on GPUs, we consider different threads or warps whereas in the SIMD world, we focus on different lanes. Irregularity on a higher level, i.e., SMT threads or SMs on GPUs are an issue of their own in the presence of lock-like data structures. In this thesis, we try to avoid these issues by using a batch-like job model: jobs are submitted in batches of independent instances.

Our main focus in this thesis is the avoidance or, at least, mitigation of data and control flow irregularities inherent in sparse numerical factorization on accelerators. Each architecture already offers some features to support us in that goal. In this closing section of the background chapters, we highlight those (some having been briefly mentioned in the previous sections) and compare them.

4.5.1 Multithreading

On most hardware designs, instructions can have varying latencies, especially when they involve access to memory. In an in-order processor, hitting a high-latency instruction blocks processing all further instructions, leading to underutilization of the functional units. Hence, avoiding or hiding that latency is a key minimize latency or maximize instruction throughput. A common technique is multithreading where the processor swaps between multiple threads (or contexts) and issues instructions from one context while another one is stalling. All three classes of accelerators support multithreading, although in different implementations.

Both SIMD-augmented CPUs and Aurora, use SMT, where each thread accesses the complete register set and full resources (functional units, caches) of the core. At each point in time, only one thread is run on the core and switching to a different thread requires a so-called "context switch". Such a context switch writes out the state, register and stack contents, of the current thread to memory and loads the state of another thread before continuing execution. With larger register files and SIMD registers, switches may take a considerable amount of cycles. Processors that implement Hyper-Threading [Shar and Davidson 1974] duplicate the registers that contain the architectural state – thus, two threads can interleave their

instructions without requiring context switches.

In throughput-oriented systems such as GPUs, SMT is not an option, since maximizing instruction throughput means keeping execution or functional units busy. NVIDIA employs an alternative form of multithreading dubbed “latency hiding”. Since the number of registers required per thread are known at compile time and the block size is constant for each kernel call, a GPU can schedule multiple blocks onto each SM and divide the register file between them. As a result, multiple blocks with their warps are kept “in-flight”. Each warp has its full state stored in a static piece inside the register file and issues its instructions in order. If any thread within the warp stalls, an SMs warp scheduler can just issue an instruction from another in-flight warp to keep the functional issues busy. No context switch is necessary; in comparison to SMT, this form of multithreading rather resembles a multiplexing of instruction streams. Latency hiding also comes with its own disadvantages: the resource requirements (registers, shared memory, number of warps per block) must be constant per kernel call, limiting its use to regular execution models.

4.5.2 Control Flow Divergence

Divergent control flow is usually the consequence of conditionals, loops or branches respective jumps in the program code. While in SIMT programming models, each thread may branch and jump on its own, the SIMD model has only one global context and has to model divergent behavior between lanes using masks. The SPMD-on-SIMD compiler ISPC [Pharr and Mark 2012] supports divergent control flow between SIMD lanes by automatically inserting the appropriate masks.

Originally, warp divergence was handled similarly in GPUs. Each warp had one program counter (PC), so any divergent path through the program was serialized and stored on a stack, each with the PC and mask of active threads for the prospective branch (“subwarps”) [Fung et al. 2009a]. From a static analysis of the PTX code, reconvergence points were derived from the control flow graph to enable merging of convergent control paths. Each individual control path, however, was executed continuously until the next branch occurred. Hence, a memory access in the active path would stall the whole warp’s execution (see Figure 4.9 (left)). With the Volta-generation of their GPUs, NVIDIA introduced the “Independent Thread Scheduling” feature. Each thread now has its own PC, but only one PC per cycle may be picked up by the warp scheduler and executed with an implicit mask, activating all threads that share the same PC. This effectively allows interleaving of different code paths, allowing latency hiding between subwarps (see Figure 4.9 (right)). As a consequence, developers must consider this deviation from the originally bulk-synchronous execution model of a warp and add explicit synchronization points with `__syncwarp()`.

4.5.3 Data Irregularity

In our scenario, data irregularity occurs when different instances have varying requirements concerning their data structures, e.g., size or formats. One example is the “vector length”: this is taken literally for SIMD systems, where it describes the length of a vector stored in one SIMD register. While vector systems have always been able set a vector length at runtime (less or equal to the hardware SIMD length), AVX512 has only recently implemented this with the extension AVX512VL. While the latter is only a matter of convenience as we can have the same by creating mask vectors, it can make a striking difference for Aurora. In case of complex instructions that are pipelined, the vector length determines the instructions’ latency. On GPUs, the closest analogue is the block size, which may also be set at runtime – although only globally, for all blocks equally. The block size is configurable up to 1024 threads, but execution is internally

chunked into warps of 32 threads. Smaller block sizes can be emulated by masking, but all blocks still require the same (upper limit) of resources, without consideration of their actual requirements. We outline a possible solution to this in Chapter 9.

Memory access patterns can be irregular as well. Here, irregularity refers to the deviation from simple, contiguous and coalesced access patterns, i.e., gather and scatter operations. Caches work best for contiguous access, a high amount of scatters or gathers can lead to repeated cache trashing and latencies through cache misses. GPUs always organize all concurrent accesses from a warp into 128 byte cache lines and then load only those cache lines, not the individual scalar loads. This benefits memory access patterns that are random, but spatially close. Access to shared memory is handled via 32 banks, where accesses are only serialized when they fall into the same bank. On Aurora, all scatters and gathers are serialized to scalar load/stores, but pipelined, through the complex instruction scheduler that bypasses the SPU's instruction fetch unit. Intel originally implemented their scatter and gather operations by breaking them up in scalar operations, but have improved upon that in their recent microarchitectures (since Haswell). Unfortunately, details are not publicly known.

Chapter 5

Block-Sparse Indefinite Preconditioning on GPUs

Contents

5.1 Background and Related Work 57

5.2 Blocking, Reorderings and Parallelism 59

5.3 A Block-oriented $iLDL^T$ with Pivoting 60

5.4 Evaluation 65

5.5 Conclusion 69

We consider the solution of linear systems $A\mathbf{x} = \mathbf{b}$ where the coefficient matrix $A \in \mathbb{R}^{n \times n}$ is sparse and symmetric indefinite with a solution vector $\mathbf{x} \in \mathbb{R}^n$ and right-hand-side vector $\mathbf{b} \in \mathbb{R}^n$. These linear systems appear frequently in scientific computing, for instance in the three areas mentioned in Chapter 2. Direct methods for such systems rely on the $LDL^\top$ factorization that often suffers from heavy pivoting. On parallel platforms, pivoting leads to irregular control flow and synchronization points.

As we discussed in Section 3.5, iterative methods can be an alternative to the direct approach. Iterative methods feature sparse matrix-vector multiplications as their major computational primitive, attractive for SIMD-style execution, but suffering from data irregularity. An additional drawback of these methods is the need of finding a good preconditioner, a matrix M such that $M^{-1} \approx A^{-1}$ and linear systems with M being relatively easy to solve. Incomplete factorizations are common "black-box" solutions [Chow and Saad 1997]. They combine the advantages (and disadvantages) of direct and iterative methods. While they benefit from techniques used by direct solvers, they need to minimize computation in order to leave enough time for multiple solves during steps of the iterative method.

In order to be efficient, modern incomplete factorization algorithms must make good use of available hardware with its complex memory hierarchy. By tuning several knobs (preprocessing mechanisms, reordering schemes as in Figure 5.1, assembly of dense operations and the amount of allowed fill-in),

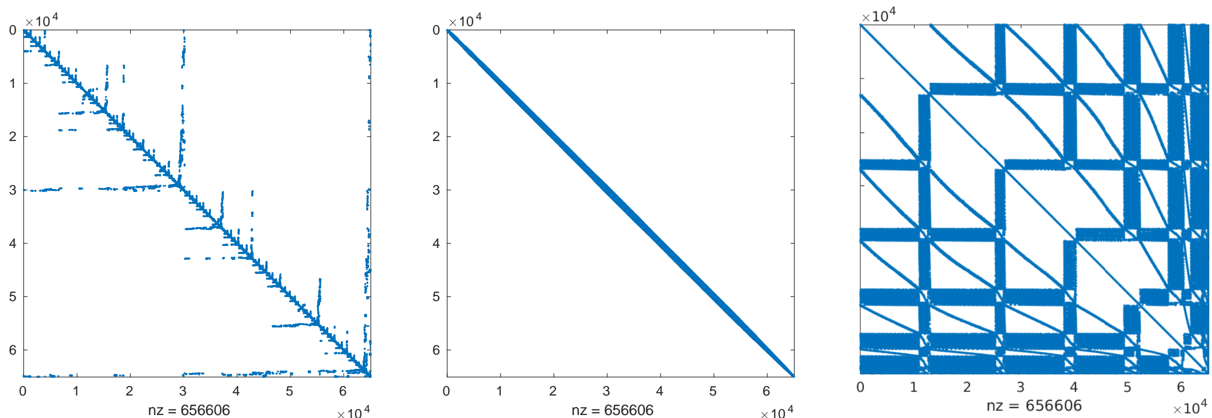


Figure 5.1: Block-reordered sparse symmetric matrix (from **left** to **right**: AMD, RCM, GMC). Once again, we notice how drastically different layouts can be generated through permutations.

the user can balance between parallelism in the construction and solve phases as well as quality of the preconditioner to minimize the overall time-to-solution.

We propose a compromise that combines techniques from full-blown $LDL^\top$ factorizations with dropping techniques from incomplete factorizations. We show that this mix is well-suited for parallel computing platforms through its performance on GPUs. Additionally, we investigate the effect of widely used preprocessing methods, cornerstones of modern indefinite solvers, with respect to their effect on parallelism in the factorization. The techniques presented in this chapter especially aim at reducing the effects of the data irregularity in sparse matrix accesses and control flow irregularity due to necessary pivoting.

Our contributions in this chapter are:

- We propose two blocking and reordering schemes which result in a tile-based matrix structure, that is ideally suited to exploit local operations on the GPU cores avoiding data irregularity.
- We propose a protocol for handling the case of limited fill-in through memory re-distribution between blocks.
- We develop block-iLDL$^\top$, the first tile-based incomplete factorization that runs completely on the GPU and offers supernodal partial pivoting, fill-in and a dual threshold dropping strategy. Our method solves more systems than a state-of-the-art CPU-based preconditioner with full pivoting and achieves speed-ups of up to $\sim 22\times$ with an average speedup of $\sim 6\times$ on a benchmark set compromising 17 numerically challenging matrices.

5.1 Background and Related Work

As part of the background chapters, Sections 3.5 and 3.6 introduced iterative and direct methods for sparse linear systems. Furthermore, we explicitly named incomplete factorizations as a class of preconditioners. This chapter focuses particularly on symmetric indefinite systems in scientific applications. Indefiniteness, i.e., the presence of both negative and positive Eigenvalues, can cause numerical zeros to appear on the diagonal of the matrix during the factorization process. In order to avoid division by zero, we use a $LDL^\top$-type factorization. Other than in the Cholesky ($LL^\top$) variant, its D is block-diagonal with 1×1 or 2×2 blocks.

Since these 2×2 pivots usually result in more fill-in, we try and avoid them through a standard preprocessing pipeline. A matching-based permutation first brings the largest entries close to the diagonal; a symmetrization of the optimal matching from MC64 (see Duff and Koster [2001] and Hagemann and Schenk [2006]) also delivers a scaling matrix as well as a static pivot order with 1×1 and 2×2 pivots. The largest elements in the resulting matrix have an absolute value of 1.0, with the largest elements mostly near the diagonal.

Therefore, this pipeline leading to Equation (3.2) serves three purposes: Find an initial pivoting order, reduce the condition number of the matrix, and reduce the amount of fill-in, thereby decreasing time and memory consumption. Of the different formulations for $LDL^\top$ factorization, we use a left-right-looking, “supernodal” variant derived from unrolling the following 3×3 block factorization from the top

$$\begin{pmatrix} A_{11} & A_{21}^\top & A_{31}^\top \\ A_{21} & A_{22} & A_{32}^\top \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} D_{11} & & \\ & D_{22} & \\ & & D_{33} \end{pmatrix} \begin{pmatrix} L_{11}^\top & L_{21}^\top & L_{31}^\top \\ & L_{22}^\top & L_{32}^\top \\ & & L_{33}^\top \end{pmatrix} \tag{5.1}$$

leading to the following steps, where $(A_{11}^\top\ A_{21}^\top\ A_{31}^\top)^\top$ denotes the previously processed columns and $(A_{21}\ A_{22}^\top\ A_{23}^\top)^\top$ the current (block) column, where A_{22} is a $k \times k, k > 0$ block:

$$L_{22}D_{22}L_{22}^\top = A_{22} - L_{21}D_{11}L_{21}^\top \tag{5.2}$$

$$L_{32} = (A_{32} - L_{31}D_{11}L_{21}^\top)L_{22}^{-\top}D_{22}^{-1} \tag{5.3}$$

$$L_{33}D_{33}L_{33}^\top = A_{33} - L_{31}D_{11}L_{31}^\top - L_{32}D_{22}L_{32}^\top \tag{5.4}$$

Each step is defined on the Schur complement, i.e., the result of previous rank-k updates (5.4). In a scalar $LDL^\top$ factorization with $k \leq 2$, this yields $L_{22} = I$ and thus 1×1 and 2×2 blocks on D's block-diagonal. For larger blocks, (5.2) requires a separate, often dense factorization of the updated A_{22}. The order of the updates in Equations (5.2) and (5.4) lead to “fan-out” and “fan-in” variants of the factorization. We use the latter, where updates to a block are all pulled in and applied at once before further operations on that block.

Apart from computing the D_{11}-weighted outer product of L_{21} in (5.2) and simple (block-)diagonal scaling in (5.3), updating the A_{33} block in (5.4) is the most time-consuming computational primitive. For sparse matrices, this yields many indexing operations and irregular memory accesses.

For a reliable $LDL^\top$ factorization, partial pivoting is often mandated. In general, pivoting operations change the elimination tree, thus triggering an additional round of symbolic analysis. Highly optimized packages thus use methods such as deferred pivoting [Hogg et al. 2016] or restrict pivots to supernodes [Schenk and Gärtner 2006].

5.1.1 Dropping Strategies

An incomplete version of the $LDL^\top$ factorization results in matrices $\tilde{L}, \tilde{D}$ such that $(\tilde{L}\tilde{D}\tilde{L}^\top)^{-1} \sim A^{-1}$. Incomplete factorizations drop elements according to a set of rules. Most implementations adhere to one or a combination of the following principles:

- Level-of-fill: Let the level of all nonzero elements of A be defined as 0. Then, other element's level is recursively set to $\text{lvl}(l_{ij}) = \max_{k<\min(i,j)}\{\text{lvl}(l_{ik}), \text{lvl}(l_{kj})\} + 1$. With the assumption that the magnitude of elements shrinks with higher levels, which holds for specific matrices (e.g., five-point matrices [Saad 2003b]) the permitted level-of-fill is bounded by the user [Naumov 2011; Chow and Saad 1997]. Such factorizations are often denoted by $LDL^\top(k)$ for level k.
- Threshold dropping: During factorization, all elements with magnitude smaller than a fixed threshold τ times the row/column 2-norm are dropped. Additionally, a specific capacity threshold c_i per i-th row/column can be set prior to starting the factorization so that only the largest c_i elements are kept per row [Saad 2003b; Greif et al. 2017; Bollhöfer and Saad 2006; Bollhöfer et al. 2019]. These threshold-based methods have been found to work better on general matrices, where the assumption regarding magnitude decay over levels of fill does not apply.

Due to dropping, near-zero pivots occur and can be dealt with by adding a perturbation onto the input matrix or postponing that pivot to a later step [Bollhöfer and Saad 2006].

5.1.2 Parallel Approaches

There are mainly three sources of parallelism to be explored when solving linear systems: and the preprocessing steps employed in the solution. First, replacing dense linear algebra calls by parallel alternatives [Bollhöfer et al. 2019; Nvidia 2008; Schenk and Gärtner 2006; Li and Shao 2011] to offset the memory transfer cost. Second, analyzing the dependecies in the nonzero sparsity pattern of the matrix. Then, the resulting dependency graph is either executed directly as in DAG-based methods [Hogg et al. 2010] or level sets of rows/cols are discovered and explored [Naumov 2011; Aliaga et al. 2011]. In particular, Greedy Multicolor (GMC) reorderings have been shown to shrink the number of level sets [Saad and Zhang 1999; Lukarski et al. 2014; Naumov et al. 2015]. Third, a number of authors have proposed methods that compute incomplete factorizations for near diagonal-dominant matrices by iteratively solving a system of nonlinear equations by a fixed-point, asynchronous Jacobi method. This approach is applicable to both solution of triangular systems [Anzt et al. 2015] and ILU(k)/IC(k) factorizations [Chow and Patel 2015]. A recent extension enables a form of threshold dropping [Anzt et al. 2018].

5.2 Blocking, Reorderings and Parallelism

The effectiveness of a block-based factorization crucially depends on the partition of a sparse matrix into blocks. Finding and exploiting block-structures has been investigated by many authors (see, e.g., [Bollhöfer et al. 2019; Polok et al. 2013; Saad 2003a; Saad and Zhang 1999]). In general, the objective of "blocking" the matrix A, i.e., permuting and determining the block starts, is to minimize the number of filled blocks while maximizing the number of nonzero entries in each block. Ideally, not just A's entries, but also future fill-ins during the factorization are considered. Since we use GPUs and heavily rely on warp-centric operations, most block-based operations take roughly the same amount of time as a 32×32 block. Consequently, we need to find much larger blocks than in previous work. If these blocks happen to be sparse, our hybrid block format still avoids wasted storage. As outlined in Section 3.3, several reorderings have been used for varying purposes, e.g., reduce fill-in or the number of level sets. Since both reordering and blocking affect the number of level sets as well as the quality of the resulting preconditioner, we outline two strategies for computing blockings. The two strategies are then compared considering the level-of-fill in a factorization and their effect on parallelism.

5.2.1 Blocking Strategies

We distinguish two different blocking strategies: blocking before reordering ("BR") and reordering before blocking ("RB").

Strategy BR. In order to find blocks, we first group rows of the matrix with similar patterns and permute them next to each other. Applying that permutation to both sides of the matrix immediately yields a blocking structure. To detect rows with similar layouts, we use cosine-based blocking [Saad 2003a].

As Bollhöfer et al. [2019] note, this often leads to small blocks far from 32×32. We thus adapt a multilevel strategy: since our hybrid block format can efficiently handle larger, sparse blocks to a certain degree, we apply the same cosine-based dropping to the coarse matrix, built from the blocks found in the earlier step. This method can be repeated recursively until the coarse matrix is sufficiently small. To facilitate merging

on coarser levels we decrease the similarity threshold over the hierarchy; we start with a threshold of 0.8 and reduce it by a factor of 0.9 per iteration. Finally, on the last level, we apply the selected reordering (AMD, RCM or GMC).

Strategy RB. We first apply a reordering to the coefficient matrix and then perform blocking. Again, we use cosine-based blocking; different than before, the blocking must preserve the ordering of rows and columns - thus, only neighboring rows can be clustered into the same block.

In the former strategy we are applying the reordering on a smaller coarse matrix C, while cosine-based blocking requires, in the worst case, to compute $C_k C_k^\top$ at each level k of the hierarchy. In the latter strategy we only need to compute a superdiagonal of $AA^\top$ once, but the effectivenes of the blocking depends highly on the reordering success in clustering rows with similar sparsity pattern.

5.2.2 Effectiveness and Parallelism

We applied both strategies to a test set of 17 sparse matrices. Fig 5.2 shows results for three matrices representing the trends found in the whole dataset. The matrices in the test set can effectively be separated into "deep" and "shallow" matrices with regard to their number of level sets. We omitted the option to not reorder the matrix as these initial layouts often led to a massive fill-in, exceeding the host's memory capacity.

The first two rows in Figure 5.2 show deep matrices. Here, fill-in reducing orderings succeed in keeping the number of fill-in blocks low over the levels of fill; GMC reordering, on the other hand, quickly adds a high amount of fill-in and looses its level-set reducing effect. While RCM sometimes creates less fill-in than AMD, especially for a small bandwidth matrix, packing elements close to the diagonal increases the number of level sets fast. Interestingly, in G3_circuit with RB ordering, the level sets created from RCM saturate fast; a sign that few levels of fill-in get close to the full factorization. This matrix is also one of the few examples where the RB strategy dominates BR; the initial layout of the matrix already exhibits similar row patterns close to the diagonal. In both shallow matrices, the GMC reordering distributes relatively dense submatrices over the independent sets, leading to fill-in later. nlpkkt120 is an example for a "shallow" matrix. Even though GMC quickly adds more fill-in over the levels, it keeps the number of level sets small in both strategies. Shallow matrices often benefit from parallel processing for both incomplete and full factorizations.

These plots highlight two points: First, the effect of reorderings when adding fill-in is highly matrix-dependent, including the choice of strategy. For the majority of the matrices, BR was the better choice as the early row amalgamation changes rows' patterns, making them more similar. Contrary to zero-fill preconditioners, the GMC reordering delivers poor results on deep matrices. AMD and RCM, as fill-in reducing permutations, consider future fill-in when permuting the matrix, GMC does not. Second, the classical choice for full factorizations, AMD, offers a decent compromise. We did not find any example where AMD delivered the worst results of these three reorderings.

5.3 A Block-oriented $iLDL^T$ with Pivoting

We now present a block-oriented $iLDL^\top$ preconditioner that benefits from the recent trends in GPU architecture briefly mentioned in Section 4.3. We assume that the input matrix A has been appropriately permuted and scaled in the preprocessing phase. Additionally, we assume a partition of rows (and,

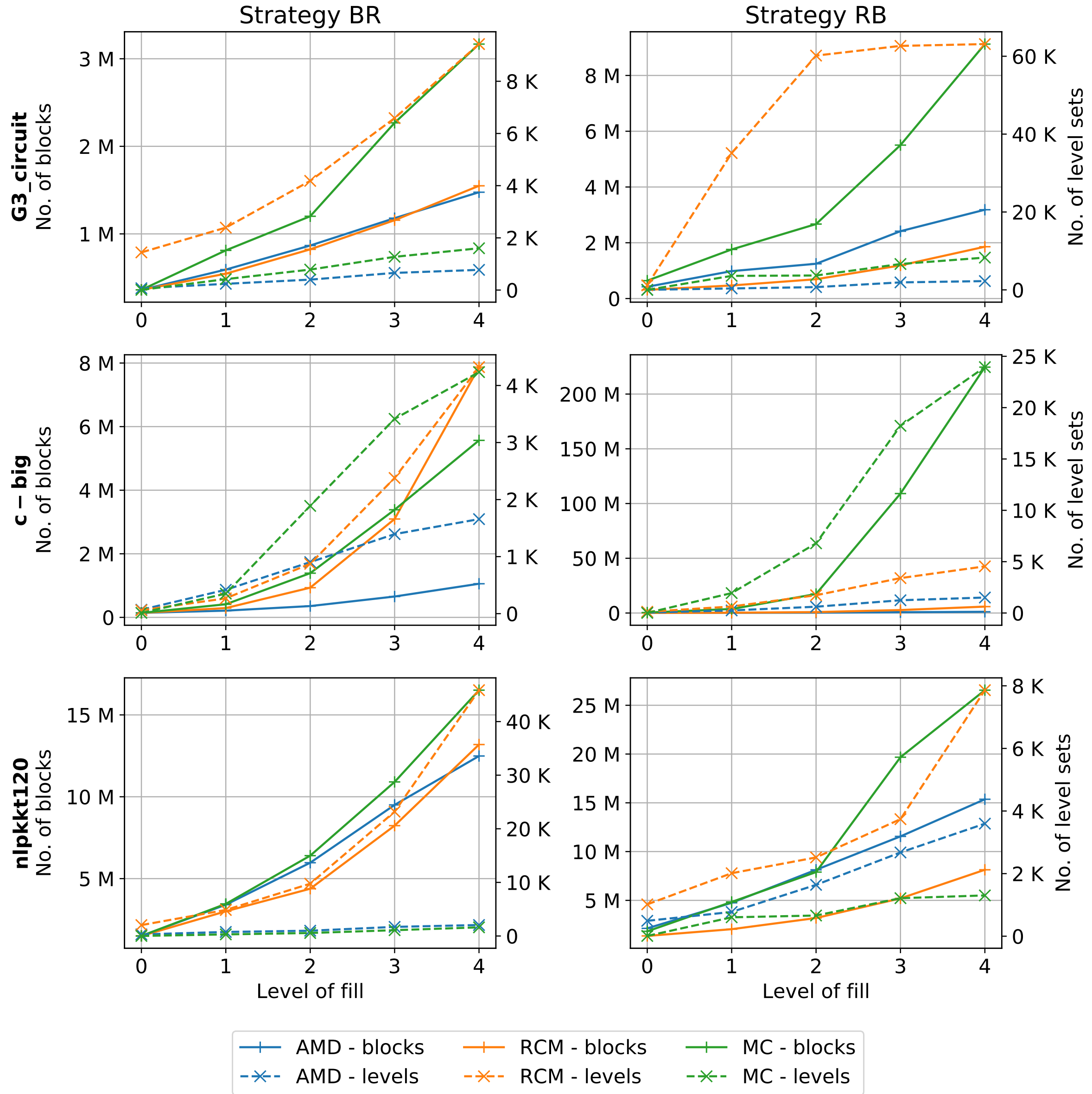


Figure 5.2: Statistics and patterns for BR and RB strategies, including the number of nonzero blocks (solid line) and level sets (dashed line) for different levels of fill with AMD (red), RCM (green) and GMC (blue) reorderings.

due to symmetry, columns) has been computed, as shown in the previous section. We use the blocks resulting from the partition in two ways: first, they fit into an 32×32-sized tile of shared memory for on-chip processing; second, we allow arbitrary fill-in inside of all allocated blocks but none outside. The resulting preconditioner is thus a hybrid of a level-based $iLDL^\top(p)$ and a pure, thresholded $iLDL^\top(\tau)$ factorization. It is characterized by three parameters: The level-of-fill l_f that defines which of the blocks in the partitioned matrix may contain entries, a threshold τ for dropping of elements and a fill ratio r_f that determines how many elements each block may contain. All set-up is executed on the CPU; after upload, the complete factorization is performed by the GPU.

Algorithm 9 Block-oriented $iU^\top DU$ factorization (with deferred updates for one level $\mathcal{L}_i$)

```
1:  for block row i ∈ L_l do
2:      [U_ii] ← apply_sym_updates(U_ii)
3:      [U_ii, D_ii, P_ii] ← factor_diagonal_block(U_ii)
4:      for block (i, j), j > i in block row i do
5:          [U_ij] ← apply_nonsym_updates(U_ij)
6:          [U_ij] = (P_ii U_ij) U_ii^-1 D_ii^-1
7:      end for
8:      [r] = rowwise 2-norms of [U_ii ... U_in]
9:      for block (i, j), j > i in block row i do
10:         [U_ij] ← = block_row_dualdrop(U_ij)
11:         compress_store(U_ij)
12:     end for
13: end for
```

5.3.1 Preprocessing and Setup

Initially, the sparse input matrix A in CSR format is expanded into COO-format and all elements are mapped to their respective blocks. We then sort the resulting coordinate pairs and, after reduction, produce a COO description of the blocked ("coarse") matrix. We then generate CSR and CSC representations for the coarse matrix. We work on a transpose of L and thus actually compute an $iU^\top DU$ factorization.

We add blocks for fill-in according to l_f. Hysom's algorithm [Hysom and Pothen 2002] allows computing such fill-in in parallel per row. With new blocks integrated into the coarse CSR representation, we then collect A's nonzero entries per block and determine the maximum number of elements per block as max_nnz$(U_{ij}) = r_f \cdot$ nnz(U_{ij}). For (empty) fill-in blocks, we base their allowed fill on the average number of nonzero elements in all occupied blocks in the same block row.

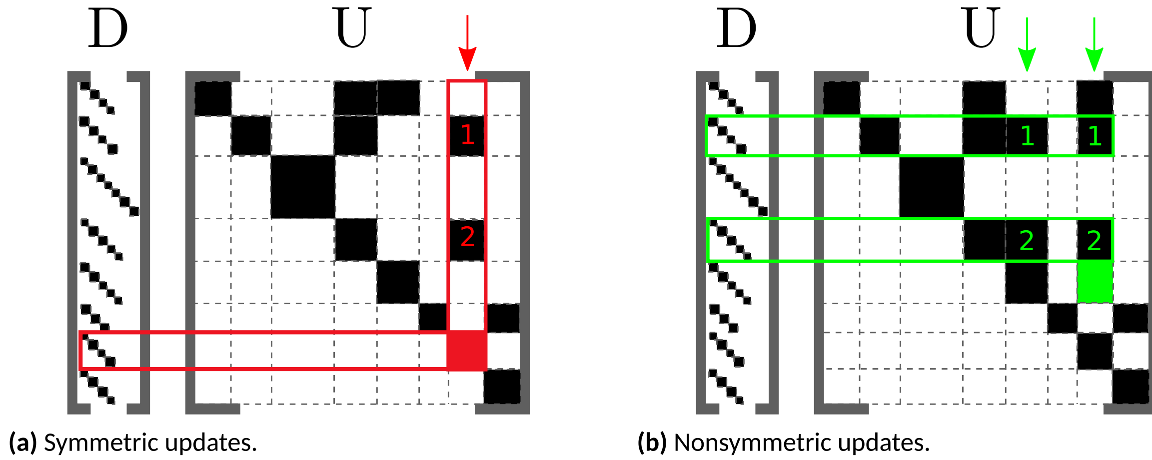
Our blocks are saved in a hybrid format: either sparse, with indices and values, or dense. We choose the format with smaller memory footprint given the maximum number of elements computed earlier. Blocks on the block-diagonal of U (dubbed "diagonal blocks") are, however, always set to dense - as well in the bottom 1% of blocks, where many updates occur. A compensation factor of 0.5 accounts for the additional indirect addressing when working with sparse blocks.

Finally, similar to Naumov et al. [2015], we compute level sets on the coarse graph which dictate the processing order for factorization and sort the blocks in memory after the number of their level set. We allocate an array of 32×32-sized, dense blocks to keep dense blocks processed in the curent level in memory ("in-flight" blocks).

5.3.2 Factorization

The high-level structure of our factorization algorithm is given in Algorithm 9. In order to avoid repeated load and store operations, some functions are merged into one kernel. This section describes the resulting kernels.

A notable change to the derivation in Equation (5.1) is the handling of updates: While Equation (5.4) amounts to *pushing* the results of the outer products $L_{31}D_{11}L_{31}^\top, L_{32}D_{22}L_{32}^\top$ to the Schur complement, we use a *pull* principle: A block pulls in, i.e., applies, all updates once it is in the current level. This strategy avoids multiple applications of the dropping rules.



(a) Symmetric updates.

(b) Nonsymmetric updates.

Figure 5.3: Block updates. (a) For diagonal block (i, i), all blocks (k, i) (column i marked with red arrow) for $k < i$ are required. Similarly, the dense factorization step modifies D's ith block. (b) For off-diagonal block (i, j), all blocks (k, i) and (k, j) (columns i, j marked with green arrows) for $k < i$ as well as the kth block of D are required.

Following the architectural changes in GPUs, we process blocks in shared memory whenever possible and use collective operations to avoid coarse-grained synchronization inside of blocks. To that end, we implemented most BLAS-like operations in a cooperative manner, where tasks are mapped onto groups that correspond to (parts of) a warp. Whenever a matrix product needs to be computed, we transpose one of the operands to avoid bank conflicts during the multiplication.

Diagonal blocks. Before factorizing a (dense) diagonal block U_{ii}, we pull in the updates $U_{ki}^{\top} D_{kk} U_{ki}$ for $k < i$, which are all found in the column i (see Figure 5.3a), using the coarse CSC representation on the device. Each diagonal block is processed by one CUDA block. All updates, are also expanded into shared memory. The outer product is computed row-wise, each row assigned to one warp.

All blocks have at most size 32×32, enabling us to perform the subsequent, dense $U^{\top} DU$ factorization step on the result of the updates using a single warp. The dense factorization follows the same steps outlined so far, except for a *push* strategy for updates. The application of small, independent and dense block-factorizations allows efficient supernodal pivoting; we implemented three pivoting strategies using 1×1 and 2×2 pivots:

1. None/static pivoting with 1×1 or 2×2 pivots from MC64
2. Bunch-Kaufmann (partial) pivoting (scans 2 cols / step)
3. Rook (full) pivoting (possibly scans all columns)

With one warp computing the factorization, all comparisons can be handled efficiently with warp shuffle operations.

The resulting triangular matrix with a pivoting permutation, is stored as dense, 32×32 block in the shared memory (as B_{ii}). This kernel also contains a step that sums up the square of entries per row in the block for later use during dropping.

Off-diagonal blocks. Similar to the diagonal blocks, off-diagonal blocks U_{ij}, for $j > i$ first pull in their updates. In order to do that, the block columns i and j have to be traversed. The intersection of the

Algorithm 10 Conservative protocol for distributing leftover memory to an oversubscribed in-flight block $B_{i,j}$

1: required = nnz(B_{ij}) − capacity(U_{ij})
2: wasfree = atomic_subtract(leftover, required)
3: granted = $\max(\min($required, wasfree$), 0)$
4: giveback = required − granted
5: **if** granted > 0 **then**
6: atomic_add(leftover, giveback) **return** granted ▷ extra units of memory for saving B_{ij} to block $U_{i,j}$
7: **end if**

columns' row indices yield the necessary updates (see Figure 5.3b). Per common row, the pair of update blocks is loaded into shared memory and the outer product is computed (again in one CUDA block, as above). Once all updates are applied, we first permute rows according to the block rows' permutation vector from pivoting. Next, each column of the block is solved with U_{ii}, D_{ii}. The dense triangular solve is performed with one warp per column. Finally, the block is saved to shared memory (as B_{ij}). Using atomics, finalized blocks contribute the row-sums of the squared entries for the following threshold dropping.

Dropping. If pivoting was used, an additional step becomes necessary before dropping: All superdiagonal blocks in the current block column need to be column-permuted according to the diagonal blocks' permutation vectors. Since permuting entries does not add additional elements, the operations may be done in-place in global memory (for sparse blocks, only the indices need to be changed; for dense blocks, warps reorder the columns).

We implement a dual pivoting strategy, following Saad [1994]: First, we drop all elements $u_{kl} \leq \tau ||u_{k\cdot}||_2$ by setting them to 0. The number of remaining nonzeros is then counted and, if it exceeds max_nnz(U_{ij}), only the elements with largest magnitude are kept. For the latter, every thread in the block stores 4 matrix elements. A collective block-wide radix-sort yields the elements in the order of their magnitude. The first max_nnz(U_{ij}) can then directly be stored in the block storage in the case of a sparse block. For dense blocks, naturally, no dropping is performed.

By setting a maximum number of entries to a fixed number per block, we enable each block in the current level to be processed separately. However, this restricts the layout of the preconditioner and could worsen its quality. As a relief, we propose a protocol that roughly emulates an area-based heuristic by Li and Shao [2011]. Memory that was reserved for blocks in previous levels but not used is redistributed. As a consequence, the fill-ratio r_f holds for the whole matrix, not for each block.

An outline of the protocol, applied to a single dense block B_{ij}, is given in Algorithm 10. During the whole execution, we keep an atomic counter (leftover) for the space left. In our protocol, each thread first determines how much additional memory is needed and subtracts that amount atomically from the leftover counter, returning the old value. If the latter is positive, we compute the amount of extra space provided. Since we subtracted the complete amount of memory requested, we then add the difference to the amount of memory granted back ("giveback") to the leftover counter, atomically. This protocol, however, is not exact, but conservative: If a block asked for more memory than was left over, the leftover counter becomes negative. Blocks that return their giveback after that increase the leftover counter,

but as long as it is negative, no further memory can be distributed to other blocks. Only when the first block returns its giveback, the leftover counter is positive. This is due to the separation of the two atomic updates to leftover; the protocol thus may underestimate the amount of memory to distribute, but it never overallocates it.

In the setup phase, we order the blocks in memory by levels. Using an additional atomic counter for the total memory used, we can save the extended blocks in-place.

5.3.3 Blocked Triangular Solves and SQMR

We solve (3.2) using iterative methods, with the block-oriented $iLDL^\top$ preconditioner. In each step of the iterative methods, solve operations with L, D and $L^\top$ are executed. Since D is block-diagonal with 1×1 and 2×2 blocks, the operation is trivially parallelizable. For L and $L^\top$, we can (as in the factorization) exploit the discovered level sets, see [Naumov 2011], or use fixed-point methods, see [Anzt et al. 2015; Chow and Scott 2016]. We reuse the level sets from the factorization – in the same order for L and inverted for $L^\top$. By replacing the scalars in a level-scheduled triangular solve algorithm, we arrive at a block-based algorithm. In our implementation, each CUDA block handles one matrix block; in the first kernel, each off-diagonal block is multiplied with the corresponding solution components by a single warp; lastly, a solve operation in a second kernel results in another part of the solution vector.

Since both A and $LDL^\top$ are indefinite, we use SQMR [Freund and Nachtigal 1995] as the method of choice. To allow for an indefinite preconditioner with split preconditioning, we use the D-symmetric version from [Freund and Jarre 1997].

5.4 Evaluation

We use a test set with 17 symmetric indefinite matrices also used in [Greif et al. 2017; Hogg et al. 2016]. All matrices are taken from [Davis and Hu 2011] and preprocessed using symmetrized MC64 [Hagemann and Schenk 2006]. Also, we added KKT matrices resulting from an interior point method for linear programs. We created the test set by downloading mps files from MIPLIB [Koch et al. 2011], geometrically scaling the LPs, standardizing the LP format and then adding a diagonal in the (1,1)-block of the KKT matrix with random positive elements from $(0, 1)$ that span a logarithmic range of 10^{-6}. All right-hand sides in the experiments were created by multiplying the input matrices with an all-1 vector. We computed the blockings and permutations for both strategies offline. For fairness, the preprocessing time for all evaluated packages was excluded from the comparison.

We compare our preconditioner with SYM-iLDL [Greif et al. 2017] (Github commit d4b862b), using the CLI “ldl-driver” and varying levels of fill between 4.0 and 8.0, following their paper. When handling difficult, indefinite linear systems, pivoting is often required for good accuracy despite its negative effect on parallelism (pivoting and parallelism are often two conflicting design goals). Thus, it is valid to compare our approach to this sequential package. Note that while the solve phase using SQMR could be parallelized, we often received large numerical errors with cuSPARSE for the resulting triangular factors. We also compare to a fixed-point method, ParILU [Chow and Patel 2015] with levels-of-fill between 1 and 3 using the authors’ implementation in their MAGMA library [Tomov et al. 2011]. All solvers use the same input permutation for each matrix.

All experiments are performed on a system with an Intel i7 3930K processor, 64 GB RAM and Ubuntu 16.04.

We benchmark on a NVIDIA K40 and a TITAN V GPU with CUDA 9.2 (driver 396.44). With the blocking strategy, reordering, level-of-fill, fill factor, dropping threshold, pivot strategy and precision (given in that order in Table 5.1) we handle a huge parameter space. Since the influence of these parameters can be highly nonlinear (see below), we manually explored the space and only present the best result in terms of runtime that we found. These results are thus upper bounds on the best possible runtime. For a fair comparison, we did the same with our competitors: SYM-iLDL's fill factor was varied between 4.0 and 8.0, ParILU's level-of-fill between 1 and 3. We split all reported times in factorization and solve phase (and, in our case, an additional setup for the triangular block-solve).

Table 5.1 shows overall runtime. The experiments with MAGMA confirm that indefinite matrices are notoriously hard to precondition: It fails to converge after 1000 iterations with its ILU in all cases but one; it is thus mostly excluded from Table 5.1. A second observation regarding convergence to the desired relative residual of 10^{-6} is shared between our approach and SYM-iLDL: a large fill factor of between 4.0 and 8.0 is necessary as well as a smaller dropping threshold than often used by default (10^{-4}). SYM-iLDL, being a scalar, sequential implementation, works best when the factors are relatively sparse (`Si10H16`) or our code suffers from tiny blocks (`boyd1`). The choice of blocking and permutation method is critical for our code's performance. Setup and overhead are comparable between the two implementations, as shown by the similar performance on smaller matrices where the overhead dominates factorization time.

Table 5.2 reports the block sizes and the amount of nonzero elements inside these blocks (dubbed "fill") for input and factorized matrices. Dense blocks are counted with fill = 1, which explains the differences of fill increase compared to the selected fill factor. These statistics correlate well with the matrices where we achieve the highest speed-ups compared to SYM-iLDL: most fill-in elements fall inside of our blocks, allowing effective pivoting and capturing the preconditioner better. For the large nlpkkt* matrices, this leads to an excellent preconditioner quality and runtime. For the matrix `bley_xl1`, no working preconditioner could be generated by any tested package. In such hard cases, a direct methods is the method of choice. Even in easier cases, double precision was necessary for a successful preconditioner generation and application. Notable exceptions are `co-100` and `nlpkkt80`, where single precision and a moderate fill and dropping threshold were sufficient for the desired accuracy.

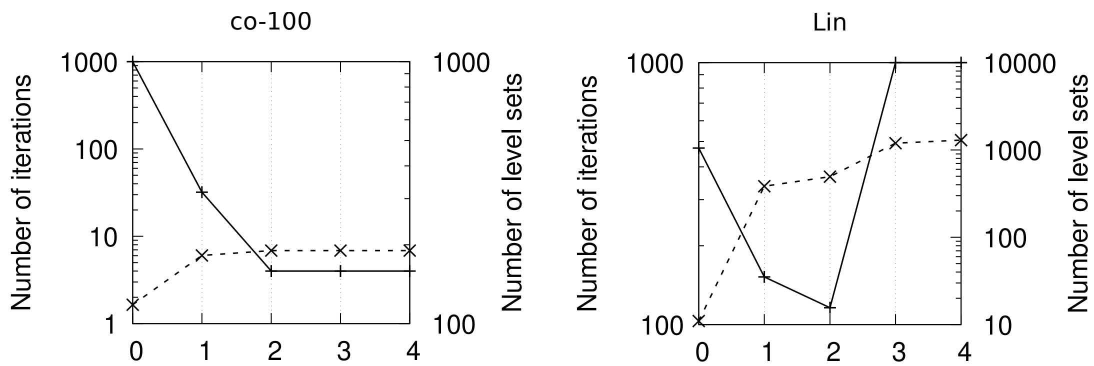
We also include results for both the K40 as well as the TITAN V. Our approach was specifically designed with the state of the art in GPU architecture as of 2018 in mind – and that is visible in the results. Besides an 2x improvement across-the-board for the factorization phase, the solve phase benefited more: Our implementation relies on double precision atomics, which are implemented in hardware on the TITAN V but are emulated through integer atomics on the K40.

The effects of parameters on our preconditioners are often nonlinear and counter-intuitive; the characteristics vary between matrices. We try to capture some of that by presenting two extremes per parameter in Figure 5.4. First, the level of fill (Figure 5.4a): some matrices, here `co-100`, behave as expected with better preconditioners for higher levels that reduce the number of iterations but increase the number of level sets. For other matrices, such as `Lin`, a higher number of levels after a threshold results in an unstable preconditioner. Here, more aggressive dropping might help. Alone, the dropping threshold (Figure 5.4b) often has small influence on the results, especially if the matrix has mostly dense blocks, where dropping is not applied. The fill factor often has a much larger effect (Figure 5.4c). Naturally, the average block fill increases in conjunction with that parameter – but the number of SQMR iterations does

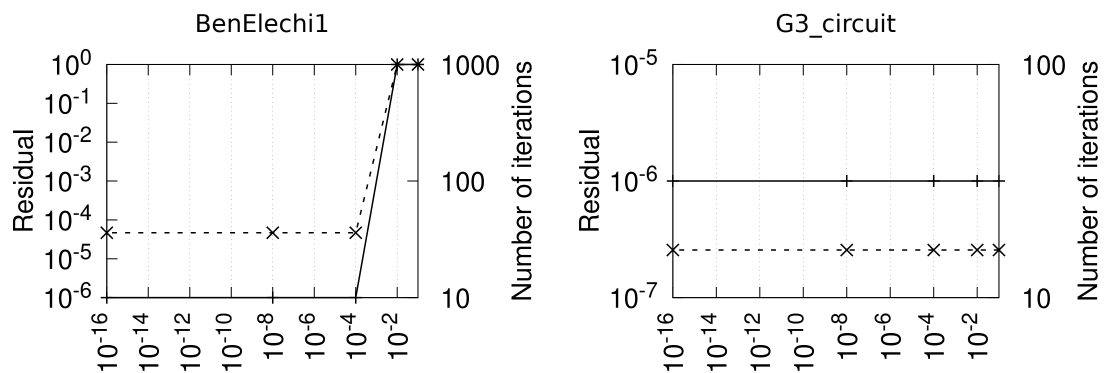
Matrix	Package	Runtime (s)
qpband [Greif et al. 2017]	SYM-ILDL (8.0)	0.01 + 0.04
$m = 20,000$	ours (BR, AMD, 0, 4.0, 10^{-4}, BK, D)	3.13 + 0.05 + 4.49, 1 it., K40
$nnz = 45,000$		**0.01 + 0.00 + 0.01**, 1 it., Titan V
mario001 [Greif et al. 2017]	SYM-ILDL (8.0)	**0.14 + 0.01**
$m = 38,434$	ours (BR, AMD, 3, 8.0, 10^{-8}, Rook, D)	0.26 + 0.00 + 0.65, 30 it., K40
$nnz = 204,912$		0.12 + 0.00 + 0.29, 30 it., Titan V
bab5 [Koch et al. 2011]	SYM-ILDL (8.0)	0.31 + 0.81
$m = 30,199$	ours (RB, AMD, 1, 8.0, 10^{-8}, Rook, D)	0.17 + 0.00 + 0.18, 8 it., K40
$nnz = 343,545$		**0.10 + 0.00 + 0.05**, 8 it., Titan V
c-72 [Greif et al. 2017]	SYM-ILDL (4.0)	0.96 + 2.58
$m = 84,064$	ours (BR, AMD, 1, 8.0, 10^{-8}, BK, D)	2.25 + 0.01 + 1.87, 11 it., K40
$nnz = 707,546$		**0.95 + 0.01 + 0.52**, 11 it., Titan V
Si10H16 [Hogg et al. 2016]	MAGMA PBiCGStab + ILU(3)	3.16 + 0.54
$m = 17,077$	SYM-ILDL (8.0)	**0.32 + 0.37**
$nnz = 875,923$	ours (RB, AMD, 3, 4.0, 10^{-4}, Rook, D)	2.62 + 0.01 + 0.39, 8 it., K40
		2.62 + 0.01 + 0.37, 8 it., Titan V
boyd1 [Greif et al. 2017]	SYM-ILDL (8.0)	**0.06 + 0.06**
$m = 93,279$	ours (RB, AMD, 0, 4.0, 10^{-4}, BK, D)	1.62 + 0.02 + 2.98, 1 it., K40
$nnz = 1,211,231$		0.98 + 0.01 + 0.1, 1 it., Titan V
Lin [Hogg et al. 2016]	SYM-ILDL (8.0)	1.7 + 8.47
$m = 256,000$	ours (RB, AMD, 1, 8.0, 10^{-8}, BK, D)	4.73 + 0.04 + 18.1, 168 it., K40
$nnz = 1,766,400$		**1.4 + 0.04 + 5.96**, 168 it., Titan V
map06 [Koch et al. 2011]	SYM-ILDL (4.0, 8.0)	†
$m = 703,690$	ours (BR, AMD, 0, 4.0, 10^{-4}, BK, D)	3.44 + 0.03 + 2.5, 16 it., K40
$nnz = 1,895,362$		**1.35 + 0.02 + 0.33**, 16 it., Titan V
bley_xl1 [Koch et al. 2011]	SYM-ILDL (4.0, 8.0)	†
$m = 354,783$	ours (all)	†
$nnz = 2,264,609$		†
c-big [Greif et al. 2017]	SYM-ILDL (8.0)	†
$m = 345,241$	ours (BR, AMD, 1, 8.0, 10^{-8}, BK, D)	8.34 + 0.02 + 17.6, 56 it. K40
$nnz = 2,340,859$		**3.53 + 0.02 + 3.38**, 56 it., Titan V
co-100 [Koch et al. 2011]	SYM-ILDL (8.0)	†
$m = 52,618$	ours (BR, AMD, 2, 4.0, 10^{-8}, Rook, S)	8.61 + 0.02 + 0.47, 4 it., K40
$nnz = 4,046,093$		**4.61 + 0.02 + 0.11**, 4 it., Titan V
bab3 [Koch et al. 2011]	SYM-ILDL (8.0)	†
$m = 435,381$	ours (RB, AMD, 1, 8, 10^{-8}, Rook, D)	3.55 + 0.01 + 4.70, 36 it., K40
$nnz = 7,053,012$		**1.74 + 0.01 + 1.26**, 36 it., Titan V
G3_circuit [Hogg et al. 2016]	SYM-ILDL (8.0)	4.35 + 11.93
$m = 1,585,478$	ours (RB, AMD, 1, 8.0, 10^{-8}, BK, D)	7.93 + 0.12 + 21.6, 72 it., K40
$nnz = 7,660,826$		**2.93 + 0.12 + 5.05**, 72 it., Titan V
BenElechi1 [Hogg et al. 2016]	SYM-ILDL (8.0)	8.09 + 27.3
$m = 245,874$	ours (BR, AMD, 4, 8.0, 10^{-8}, BK, D)	2.96 + 0.01 + 7.02, 88 it., K40
$nnz = 13,150,496$		**1.82 + 0.01 + 1.57**, 88 it., Titan V
af_shell7 [Hogg et al. 2016]	SYM-ILDL (4.0, 8.0)	†
$m = 504,855$	ours (BR, AMD, 4, 8.0, 10^{-8}, Rook, D)	5.34 + 0.02 + 18.2, 64 it., K40
$nnz = 17,579,155$		**3.15 + 0.02 + 10.1**, 64 it., Titan V
nlpkkt80 [Greif et al. 2017]	SYM-ILDL (4.0)	56.16 + 118.55
$m = 1,062,400$	ours (BR, AMD, 0, 8.0, 10^{-4}, BK, S)	1.87 + 0.04 + 7.58, 196 it., K40
$nnz = 28,192,672$		**2.65 + 0.04 + 3.86**, 196 it., Titan V
nlpkkt120 [Greif et al. 2017]	SYM-ILDL (4.0, 8.0)	†
$m = 3,542,400$	ours (BR, AMD, 0, 8.0, 10^{-8}, Rook, D)	23.89 + 0.21 + 67.4, 96 it., K40
$nnz = 95,117,792$		**11.04 + 0.19 + 19.4**, 96 it., Titan V

Table 5.1: The runtime of the software packages. It is reported as $x + [y] + z$, where x denotes preconditioner computation, y optional preconditioner analysis and z time taken by the iterative method. We also report number of iterations (it.) taken to convergence, while † denotes the lack of convergence.

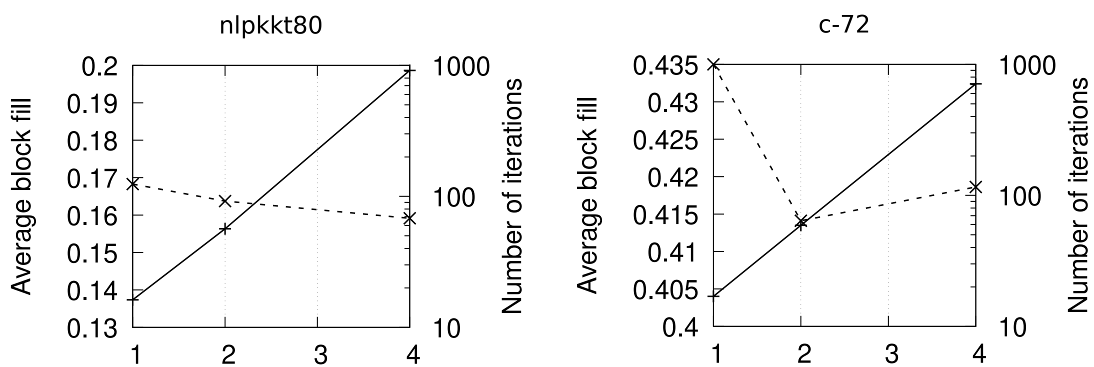
(a) Fill-in level



(b) Dropping threshold



(c) Fill factor



(d) Pivoting strategy

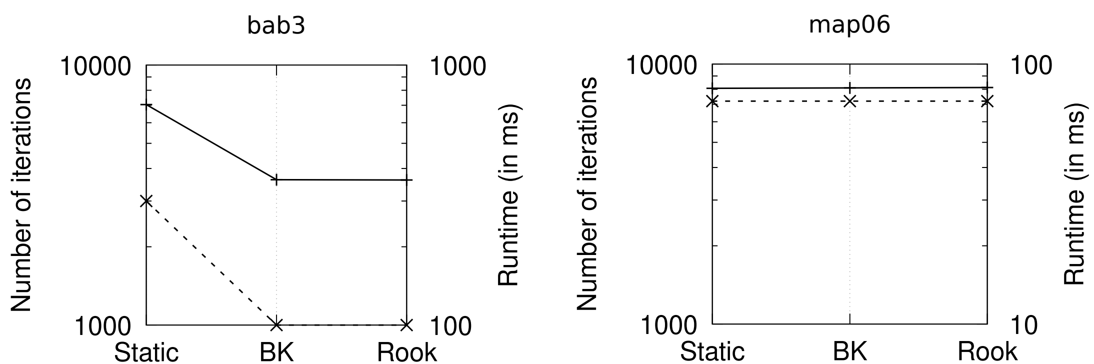


Figure 5.4: Influence of factorization parameters. The solid line corresponds to the left axis, the dashed line to the right axis.

Matrix	Mean block size	Mean fill Input	Mean fill Output
qpband	16	0.375	1
mario001	712.68	0.05	0.25
bab5	400.05	0.04	0.49
c-72	698.14	0.1	0.28
Si10H16	744.53	0.01	0.08
boyd1	15.02	0.58	0.99
Lin	708.24	0.01	0.19
map06	146.83	0.16	0.66
bley_xl1	136.87	0.12	0.42
c-big	787.99	0.07	0.25
co-100	299.99	0.09	0.47
bab3	649.77	0.04	0.41
G3_circuit	265.99	0.03	0.61
BenElechi1	590.9	0.08	0.99
af_shell7	467.17	0.1	0.98
nlpkkt80	989.39	0.03	0.31
nlpkkt120	1001.67	0.03	0.32

Table 5.2: Fill statistics for matrix blocks.

not necessarily decrease accordingly; often, tiny elements are kept in denser blocks that do not effect the solution much. Lastly, the effect of supernodal pivoting (Figure 5.4d) is split: For matrices such as `map06` where MC64 was able to determine good pivots and the off-block-diagonal elements are small, there is not much difference to static pivoting. For other matrices, see e.g. `bab3`, pivoting is necessary for a working preconditioner. We observed multiple cases where pivoting allowed a preconditioner computed in single-precision to be effective. Combining a single-precision, pivoted preconditioner with a double-precision solve and SQMR could be advantageous.

5.5 Conclusion

Symmetric indefinite systems can be hard to precondition. Our approach combines techniques from full and incomplete factorizations and leverages recent changes in GPU architecture, yielding short runtimes and effective preconditioners for large, sparse matrices. The approach, rare for GPU-based implementations, allows restricted fill-in and pivoting. Through the use of intra-block pivoting and other degrees of freedom, we were able to compete with a sequential preconditioner that relies on global pivoting.

These results show us that using the right combination of techniques with regular compute patterns (block matrix operations with pivoting, fill-in in dense blocks) can help avoid highly irregular techniques such as global pivoting and still perform well in comparison.

As a by-product, we also found that the fixed-point ("Jacobi") technique [Chow and Patel 2015], an ideal candidate for massively-parallel systems, is unable to solve these test matrices. The reasons for this deficiency remain a subject of further investigation.

Chapter 6

Supercharging the Block-Sparse Approach

Contents

6.1 Related Work 71
6.2 Two-Level Pivoting 73
6.3 Backend 73
6.4 Frontend 77
6.5 A Modified Jacobi-$LDL^\top$ Algorithm 79
6.6 System Evaluation 80
6.7 Conclusion 84

In Chapter 5, we presented two approaches combating irregularity in incomplete sparse matrix factorization: working on block-sparse matrices with local, collective kernels and augmenting a local pivoting and fill-in strategy inside these blocks. As a result, we were able to compete with a sequential, yet full pivoting-based preconditioner package. Our ideas were carefully implemented in order to suit the execution model of GPUs.

However, the dense kernels in question relied on shared memory-based block storage to provide the dynamic accesses required for local pivoting. Modern GPUs have large register files that not only exceed the capacity of shared memory but also provides a bandwidth advantage of up to $10\times$[1]. This comes at a price, though: data accesses and control flow must be static to be cached in registers by compilers. At first sight, the use of a register cache renders intra-block pivoting impossible.

If we want to extend the concepts from Chapter 5 to more precise preconditioners or even direct methods, local pivoting will not suffice. As Schenk et al. [2001] note, even the minor loss of accuracy through a pivoting procedure restricted only to supernodes requires fixing by an iterative correction. Hence, we suggest extending our block-based approach's pivoting to a *global* level. As we have frequently mentioned, global pivoting always implies control flow irregularity and an additional factor of data irregularity due to modifications in the underlying control structure. Each such operation poses a synchronization point, limiting the progress through novel hardware generations that offer more potential for fine-grained parallelism.

Lastly, the parallelization of our previous efforts was build upon a matrix' level sets. The number of level sets depend on multiple factors, including the reordering from a preprocessing pipeline. Furthermore, every pivoting operation requires a re-computation of the level sets in the Schur complement. In summary, it is worth considering alternatives to this scheduling strategy. While the iterative Jacobi method [Chow and Patel 2015] presents itself as a viable candidate, last Chapters's experiments clearly demonstrated its shortcomings.

[1]This number is taken from NVIDIA's 2012 Fermi generation of GPUs. No bandwidth numbers for register files have been reported for subsequent architectures

We continue our work on all three mentioned fronts in this chapter: register-cache based pivoting kernels, a data structure for efficient pivoting and the adoption of the Jacobi method for tough linear systems. Our contributions in this chapter are as follows:

- We present a blocking-based data structure that allows parallel matrix permutations on the block level, enabling pivoting over the whole matrix.
- In order to maintain accuracy inside these blocks, we develop CUDA kernels that, for the first time, allow irregular *full* and *symmetric* pivoting with regular, static memory accesses.
- We modify the Jacobi factorization scheme, enabling convergence even for the tough cases: indefinite, non-diagonal dominant matrices.

6.1 Related Work

The fundamentals of iterative methods, numerical factorizations and preconditioners have been laid out in Chapter 3. In the following, we complement its contents by introducing the Jacobi iteration and listing notable related works.

6.1.1 Fixed-point Methods

Departing from 'classic' methods that process the matrix along its nonzero structure, Chow and Patel [2015] proposed to use a fixed-point iteration scheme to approximate an Incomplete LU (ILU) factorization. Their main insight is that for an incomplete preconditioner $A \approx LU$ and a predefined sparsity pattern S, it holds that

$$A_{ij} = (LU)_{ij}, (i, j) \in S \tag{6.1}$$

which leads to the following equations:

$$L_{ij} = (A_{ij} - \sum_{k=1}^{j-1} L_{ik}U_{kj})U_{jj}^{-1}, U_{ij} = L_{ii}^{-1}(A_{ij} - \sum_{k=1}^{i-1} L_{ik}U_{kj}). \tag{6.2}$$

These Equations can then be solved in a fixed-point manner, always taking the initial matrix A on the right hand side as input and consecutively updating L and U - each iteration is called a *sweep*. Following up on that work, Anzt et al. [2019b] used these so-called *Jacobi sweeps* to construct a highly-parallel version of Thresholded Incomplete LU (ILUT) algorithm. We note that the term "Jacobi" is a little overloaded since it is used for both the fixed-point iteration scheme as well as diagonal preconditioning. In general, which meaning is used can be inferred from context.

6.1.2 Context

As discussed, allowing pivoting causes sequential symbolic operations and irregular memory accesses. Reconciliating pivoting and parallel computations has often been discussed in the literature concerning direct methods (i.e., full factorizations), but only rarely in conjunction with incomplete factorizations.

Pivoting. We find that the scope and amount of pivoting allowed in a package can be represented on a quasi-continuous scale: The simplest approaches process the factorization in the order as supplied by the input matrix [Bollhöfer et al. 2019]; whenever a small pivot is encountered, they usually add a small numerical shift to it and rely on iterative refinement in the Krylov solver later on. For these cases, the mentioned preprocessing and scaling of the input matrix is crucial. A step more towards global pivoting is

given in the previous Chapter: When the matrix is cut into dense blocks a priori, then pivoting operations inside these blocks come without the burden of requiring a data structure modification. Increasing the size of these blocks more and more approximates full pivoting; frontal matrices as in PARDISO [Schenk et al. 2001; Schenk and Gärtner 2006] can reach sizes up to 1024×1024. Lastly, the extreme case allows unlimited, arbitrary pivoting as in Greif et al. [Greif et al. 2017]. With this work, we add another option that is located somewhere around the center of this axis: pivoting inside modestly-sized blocks, but we also permit the permutation of the blocks themselves.

Blocking. Any successful operation on such blocks requires heuristics to initially find dense blocks in the sparse matrix without wasting too much storage saving numerical zeros. Mostly, we distinguish between two methods: a priori [Saad 2003a; Thuerck et al. 2018] and dictated by amalgamating nodes of the elimination tree [Saad and Zhang 1999]. While left-looking blocked approaches permit to initially set up these block structures, multifrontal packages with a right-looking computation order often pack/unpack scalar rows and columns of the matrix into so-called *frontal* matrices, which are, by design, almost fully dense. Apart from minimizing the symbolic computations, using blocks also permits us to exploit dense BLAS and LAPACK kernels on the blocks. On the CPU, such cache-friendly kernels even offset the additional work of increasing the size of blocks progressively during the computation as done in Bollhöfer et al. [2019]. Other than these traditional methods, Götz and Anzt [2018] have presented the first deep learning based approach to detect dense blocks near a matrix' diagonal.

Parallelization. Sparse linear algebra, due to its content of symbolic operations and irregular memory accesses, presents a challenge for the effective use of SIMD-based, throughput-oriented accelerator cards. In the case of factorizations, parallel performance rests on two pillars: (1) using dense, parallel kernels on blocks and (2) detecting independent operations due to the dill structure of the matrix. The latter leads to multilevel-types of factorizations [Naumov 2011] that process rows respective columns of matrix that are independent, at the same time. Those sets are discovered by a breadth-first search on the matrix' adjacency graph. Since the latter changes when permuting the matrix during preprocessing, Naumov investigated [Naumov et al. 2015] graph coloring as a means to expose more parallelism; Chapter 5 studied the effects of such a reordering. In case of direct factorizations or significant expected fill-in, the computation is guided by the matrix' elimination tree [Liu 1986a; Davis 2006]. Recently, Hogg et al. [2010] has also explored the use of DAGs for scheduling. Lastly, we note the efforts made to include more and more special abilities and hardware of GPUs, such as NVIDIA's Tensor Cores for fast processing of half precision; mixed-precision approaches [Anzt et al. 2019a; Grützmacher et al. 2020] could help to save computation where lower precision is sufficient.

Implementing the fixed point equations 6.2 from above leads to an embarassingly parallel method that is ideally suited for the GPU. This method, often referred as 'Jacobi style' (not to be confused with diagonal preconditioning!) is used in subsequent works as a building block for e.g. triangular solves. ParILUT, an iterative threshold-dropping based Jacobi method [Anzt et al. 2019b], enables the generation of fill-in by alternating between a step of matrix-matrix multiplication to determine new candidate locations for explicit zero elements and Jacobi iterations to determine their value. All these methods' drawback is their applicability, a high degree of diagonal dominance seems to be a condition for convergence. The methods we present in this chapter are an attempt at creating a remedy to this problem: (a) Using blocks naturally opens up many different locations for fill-in and (b) full pivoting improves the dominance of diagonal entries during computation.

6.2 Two-Level Pivoting

Our factorization code works as follows: First, we overlay the input matrix A with a regular grid of a user-defined size and extract all nonzero elements which are then combined in dense blocks. We then set up data structures for organizing these blocks (see Section 6.4). This block structure is fixed for the remainder of the factorization, i.e., blocks are neither created nor deleted; however, larger blocks are notably better at capturing possible fill-in on the sparse level. Thus, the ordering of the matrix together with the size of the regular grid leads to a trade-off between more flexible fill-in and inefficiencies caused by empty blocks.

While factorizing the matrix, we perform all operations on these blocks, including dense factorization of blocks on the block-diagonal. These batched factorizations use special kernels (see Section 6.3). Furthermore, we also permit to postpone whole blocks to the end of the matrix and factor them later. In sum, we offer two levels of pivoting: *inner* pivoting inside of dense blocks and *outer* pivoting.

Our code splits the calling-the-actual-kernels from the analysis and planning phase. We refer to the symbolic part as the frontend on the host while the set of kernels makes up the backend on the device.

6.3 Backend

We start by discussing inner pivoting in the backend. This mainly covers the kernels performing a dense numerical factorization of a block. In the previous chapter, we have presented GPU kernels for $LDL^\top$ factorizations, which similarly process one block per warp and heavily make use of collective operations. Our previous code, however, stores the block in shared memory during factorization, which enables dynamic accesses and therefore allows straightforward pivoting. Recent GPU generations, especially of the Volta and Turing generation, offer an register file of 256 kB per streaming multiprocessor; shared memory is limited to 96 kB for the same hardware unit. Registers greatly exceed shared memory in bandwidth; the downside, however, is that dynamic accesses to registers get translated to local memory accesses – which are just cached, slow global memory accesses. A task like pivoting with its irregular (i.e., dynamic at runtime) access pattern is not a natural fit to register-based kernels.

6.3.1 Full Pivoting in Registers

Previously, Anzt et al. [2017b] reported a way to support partial pivoting in register-based kernels: they assign one row each to a thread in the warp; with shuffle-instructions, these threads then swap values with destinations determined at runtime. In the LDU-case, that constitutes partial (row-) pivoting.

Limiting a kernel to either row interchanges or column interchanges is straightforward. The situation, however, is more complicated when handling both. A possible alternative traverses the whole matrix after each factorization step and applies masks during the rank-1 downdate. While this eliminates all dynamic accesses, the size of the generated code for $k \geq 16$ negatively affects the instruction cache and lets performance drop below a baseline shared memory version. As a remedy, we propose to allow one dynamic access via binary search per factorization step and amortize this through two measures: (1) explicit permutation to the current column avoids repeated iteration over all columns and (2) usage of masks to fuse rank-1 update, row scaling and pivot update into one step. Essentially, we fuse the last two steps of a classical right-looking factorization into one fused Schur-complement. The resulting code is presented in Listing 2. In the interest of notational brevity, we use 'telling' pseudo-function names, e.g.,

`swap_with_ix`, for collective primitives that are straightforward to implement with shuffle instructions. Similar to a CUDA kernel, the code snippet is written for one thread with id `t_ix` in its local (sub-)warp. It accesses the `j`-th element of its row by `reg[j]` – iff the index `j` is static. Otherwise, we mark it as a costly dynamic access – requiring a binary search on a runtime index to access the register. An important part of Anzt' kernel is handling the row interchanges implicitly - rows are never actually interchanged, each thread only stores the position of its row in the matrix. The actual permutation is postponed to when the factorized block is written back to global memory, where dynamic accesses are cheaper.

For full LDU pivoting, we always select the entry of maximum magnitude in the Schur complement. Initially, each thread iterates over its row to determine the maximum (line 2); a warp reduction then determines the first pivot element – (piv_ix_r, piv_ix_c) (line 6). During the whole factorization, thread i keeps a record of which of the original rows respective columns are permuted to position i in `(t_pivot_r, t_pivot_c)`; these values are exchanged accordingly (lines 9, 10). Since we handle column permutations explicitly, the only necessary dynamic access occurs in line 14: each thread copies the value from the column selected as the next pivot column. These values are scaled and written into the current column at step `s` (line 20). Doing the same with the pivot column would warrant another dynamic access; instead, we defer this part of the permutation to line 34 in the rank-1 downdate: This expression combines copying entries from column `s` to the pivot column and executing the rank-1 downdate for other columns into one expression via operand semantics. We replace all conditions by explicit multiplication with masks to avoid warp divergence. While traversing the columns in the downdate, we also select the maximum magnitude entry per row; by putting all these steps in one loop, we incentivize the compiler to make use of ILP inside the loop. Since registers cannot be manually addressed in CUDA code, we use Cog [Batchelder 2004] as code generator to explicitly unroll all loops and avoid any dynamic arrays accesses.

The symmetric ($LL^\top$) case generally mirrors the method in Listing 2; row and column permutation is then the same. The $LDL^\top$ case with its 2×2-pivots is more complex and generally requires 2 dynamic accesses, but otherwise follows the same ideas.

As an additional measure, we apply pivot pertubations with a threshold ϵ. During factorization of a block B of the sparse matrix A, whenever the largest pivot has a norm smaller than $\epsilon||A||_F$, we replace it by $\epsilon||A||_F$. This helps avoiding small pivots than can occur in incomplete factorizations due to the lack of fill-in. Its implementation in our kernels comes almost for free. The concept is considered a necessity for indefinite matrices [Hagemann and Schenk 2006]. Usual values of ϵ are around 10^{-6}.

6.3.2 Evaluation

We evaluate the resulting kernels in Figures 6.1 (single precision) and 6.2 (double precision) on a NVIDIA Titan RTX with 24 GB memory and CUDA 10.1 on Ubuntu 18.04 with driver 418.39. We compare the register-based ("GPU-Reg") with the shared memory version from Thuerck et al. [2018] ("GPU-SMem") as baseline. Both cuBLAS [2] and MAGMA [3] [Tomov et al. 2011] also offer a batched LU-kernel with partial pivoting; the latter being the implementation of Abdelfattah et al. [2017]. While our code uses CUDA's unified memory interface, we manually prefetched all data and used the profiler to make sure no page faults affect the benchmark results. Our backends offer static and full pivoting for all three factorizations as well as a partial LDU-kernel for the sake of comparison. We execute all kernels on random block

[2] `cublasSgetrfBatched`
[3] `magma_sgetrf_batched_smallsq_shfl`

Listing 2 Full pivoting kernel for an $n \times n$ - sized LDU factorization (see 6.3 for definitions of variables and symbols used). This code snippet assumes that `t_ix` is the current thread's index into the (sub-) warp. Here, we use Python syntax for improved readability and to reduce the level of detail.

```
1  processed_r = False
2  [t_piv_val, t_piv_ix_c] = max(abs(reg[0:(n - 1)]))
3  [t_pivot_r, t_pivot_c, t_piv_ix_r] = [t_ix, t_ix, t_ix]
4
5  for s in range(0, n):
6
7    # find maximum value in Schur complement as next pivot
8    [piv_val, piv_ix_r, piv_ix_c] = reduce_with_ix(t_piv_val, t_piv_ix_r, t_piv_ix_c)
9
10   # record row and column permutation
11   swap_threads(t_pivot_r, s, piv_ix_r)
12   swap_threads(t_pivot_c, s, piv_ix_c)
13
14   # fetch values in columns s and the pivot
15   cur_col = reg[s]
16   piv_col = dynamic(reg, s)
17
18   if(t_ix == piv_ix__r):
19     processed_r = True
20
21   # explicitly permute pivot column to position s
22   reg[s] = piv_col / (1.0 if processed_r else piv_val)
23
24   # fused rank-1 downdate: includes row scale, column swap and pivot search
25   t_piv_val = 0.0
26   t_piv_ix = s
27
28   # mult0: scales the pivot row
29   mult0 = 1.0 / (piv_val if (t_ix == piv_ix_r) else 1.0)
30
31   for j in range(s + 1, n):
32     src = cur_col if (j == piv_ix_c) else reg[j]
33     o_val = shuffle(src, piv_ix_r) / piv_val
34     update = o_val * piv_val * reg[s]
35
36     # switch between update and relacement
37     reg[j] = mult0 * src - (update if not processed_r else 0.0)
38
39     # prepare next pivot selection
40     if(abs(reg[s]) > abs(t_piv_val)):
41       t_piv_val = reg[s]
42       t_piv_ix = j
```

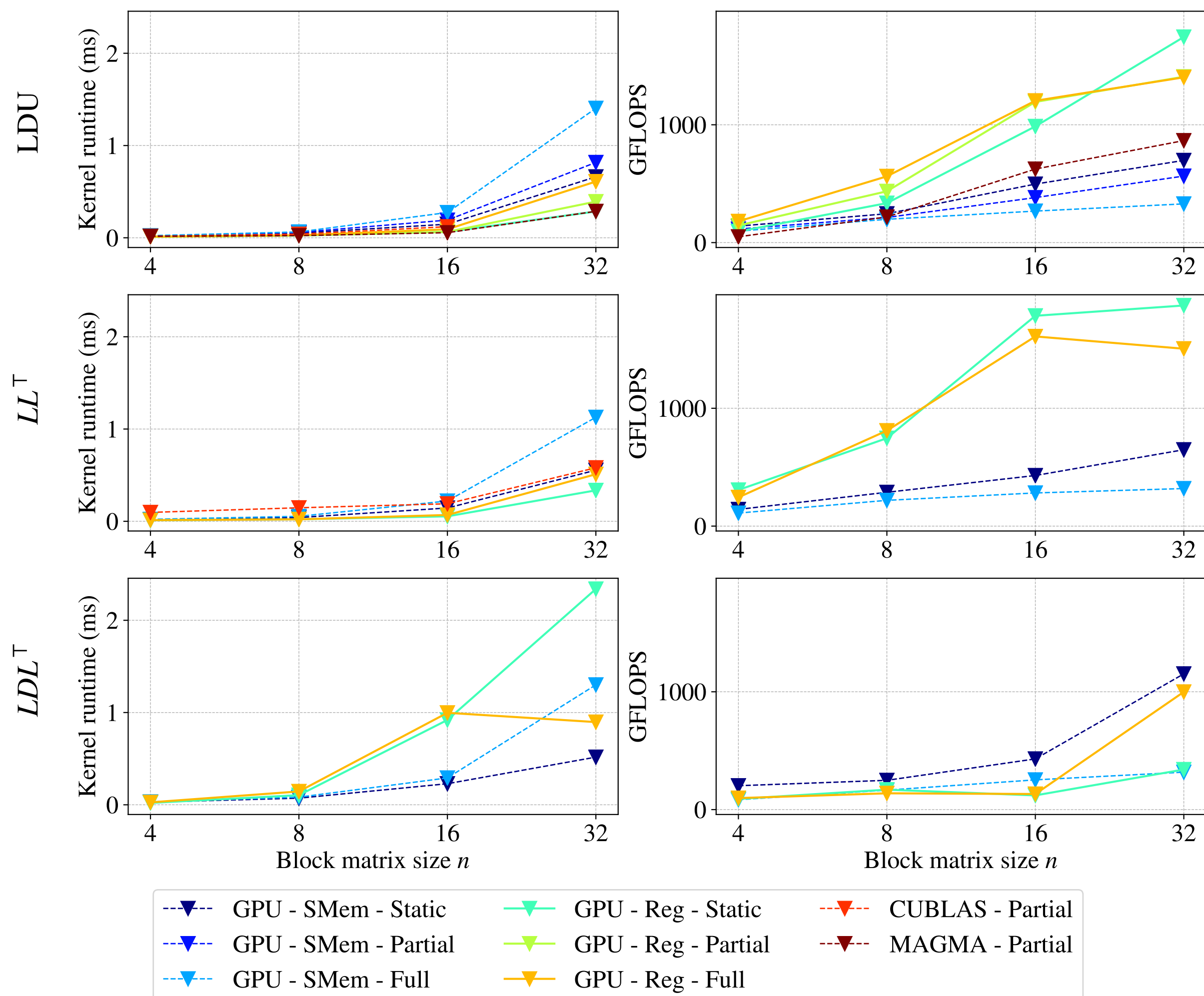


Figure 6.1: Runtimes and GFLOPs over block size (x-axis) for pivoting kernels of different factorization methods. All runs use a batch size of $10,000$ matrices that are either unsymmetric (LU), symmetric positive definite ($LL^\top$) or symmetric indefinite ($LDL^\top$).

matrices of sizes 4×4, 8×8, 16×16 and 32×32 with a batch size of $10,000$. Additionally, we use nvprof to get the FLOP-count of each kernel. While the results for our kernels and MAGMA are on par, we exclude cuBLAS from the GFLOPS plot as the profiler returned results that were beyond the theoretical maximum of the TITAN RTX (16.31 TFLOPS FP32, 0.509 TFLOPS FP64). All timings include load and store operations from and to global memory as well as fetching the frontends' job descriptions. For 32×32 matrices, our LDU-kernels achieve $1,745/1,407/1,402$ GFLOPS for static/partial/full pivoting and single precision using the register-based method from above. MAGMA's partial LDU kernel achieves 898 GFLOPS. Runtimes for the batch were $0.28/0.39/0.60$ ms for ours and 0.28 ms for MAGMA. The full pivoting kernel, despite the much higher FLOP count and symbolic intensity, is only 50% slower than the partial pivoting case; this confirms that Listing 2 allows the GPU to leverage mainly ILP to compensate for many of the additional FLOPS. In general, the register-based backend beats the shared memory-based kernels; full pivoting in registers is often as fast as static pivoting in shared memory.

A notable exception is the case of $LDL^\top$: To maintain numerical accuracy, static pivoting selects 2×2 pivots in roughly 40% of steps; since we use Givens transformations to solve the resulting 2×2 systems, this especially affects larger block sizes. Hence, different block sizes, operating on different random

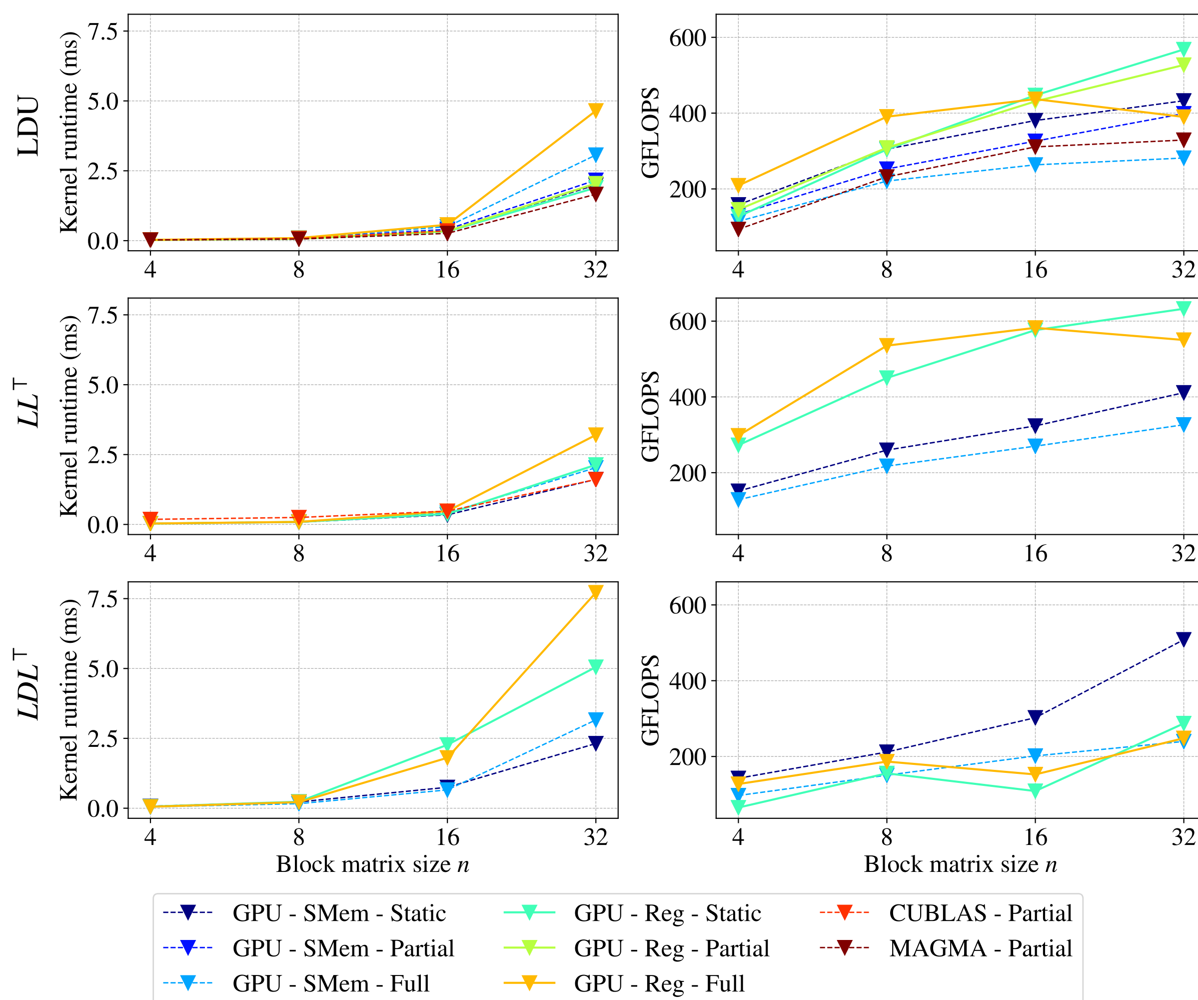


Figure 6.2: Experiments from Figure 6.1, repeated with double precision. Since the double-precision kernels also contain single-precision operations, we count 1 DP-Flop as 2 SP-Flops.

matrices, cannot be compared - their ratio of 2×2 pivoting is just too different. The full pivoting variant only requires 2×2 pivots 10% of the time on average; thus, it is faster than the static case. here, shared memory variants are generally faster since the high register pressure in the register-based variants cause many accesses to go to local memory.

For double precision, the register-based kernels suffer from high register pressure as well and are also restricted by the FP64-hardware on the TITAN RTX. As a consequence, shared memory kernels are preferable in this situation. Note that, since the double precision kernels also include single precision operations, we regard one FP64 operation as two FP32 operations in Figure 6.2. Notably, despite the slower runtime, the register-based kernels still manage to come close to approx. 60% of the theoretical FP64 performance due to the additional operations resulting from pivoting.

6.4 Frontend

For very tough matrices where inner pivoting still leads to small pivots, we propose to allow outer pivoting, i.e., change the structure of the matrix by pushing blocks from the block diagonal and their respective rows and columns to the end of the matrix. The subsequent symbolic operations are, depending on the

matrix' nonzero structure, expensive. In this section, we propose a simple and effective semi-implicit scheme for block-pivoting in the frontend. We first give details on the data structure involved and then outline the operations that are required to permute a block and its row respective column to the end of the matrix.

6.4.1 Data Structure

After initially locating the nonzero elements in the input matrix A, we create the following management data structure: Similar to the CSR and CSC formats for conventional sparse matrices, we use a blocked dual CSR/CSC format. For each block row, we list all blocks as array R_i with index pairs $(i, \texttt{ptr})$ containing the original block column as well as a pointer to the dense block in memory. The same is kept for all block columns as lists C_j; in case of symmetric matrices, we only keep the row-based structure. An example, containing only the row index pairs, is given in Figure 6.3a (the green arrows represent C_2, C_3 and the black arrows R_2, R_3). In addition, we save the index of the block on the diagonal in R_i/C_i in arrays $\hat{R}_i, \hat{C}_i$ for convenient access to the lower and upper triangular part of the block matrix. Assuming we chose a large enough block size $k \times k$ and a suitable reordering during preprocessing, the nonzero structure (i.e., the location of nonzero blocks inside the matrix), is significantly smaller than that of the input matrix A.

6.4.2 Pivoting

With this structure in mind, we turn to the implementation of outer pivoting. In order to create an efficient, parallel pivoting approach, we restrict ourselves to 'push-back' types of permutations, i.e., each operation can pivot a set of rows and columns to the end of the matrix; yet the relative order among them stays constant. Secondly, we only perform symmetric permutations which do not affect the blocks on the block diagonal. This scheme leads to a preconditioner approach that is often also classified as multilevel [Bollhöfer and Saad 2006] (not to be confused with the level scheduling for parallelization), an a posteriori pivoting scheme. During the factorization, blocks with small pivots are recorded and marked for push back; this, however, requires additional storage as such blocks must be rolled back before the pivoting operation. Note that this does not, in any way, result in a loss of generality - we can still generate arbitrary (symmetric) permutations, just at the price of more steps. In our implementation, we maintain 3 arrays:

- P_i which saves the current diagonal block on position i, i.e., the first row and column of the matrix,
- its inverse P_i^{-1}, pointing to the current position of (initial) diagonal block i and
- X_i which store a diagonal block's level.

These levels in X_i are an upper bound to levels resulting from level scheduling based preconditioners. All pivots pushed back in one level will stay together in that level, so no additional dependency analysis is necessary.

A 'push-back' operation executes the following steps:

1. Increment the X entries for diagonal blocks to be permuted,
2. execute a stable sort on the diagonal block ids by their respective level; this results in the updated array for P.
3. Invert P into P^{-1}.
4. Sort all index pairs $(i, \texttt{ptr})$ in R, C by comparing diagonal blocks by their entries in P^{-1}.
5. Lastly, update arrays $\hat{R}, \hat{C}$.

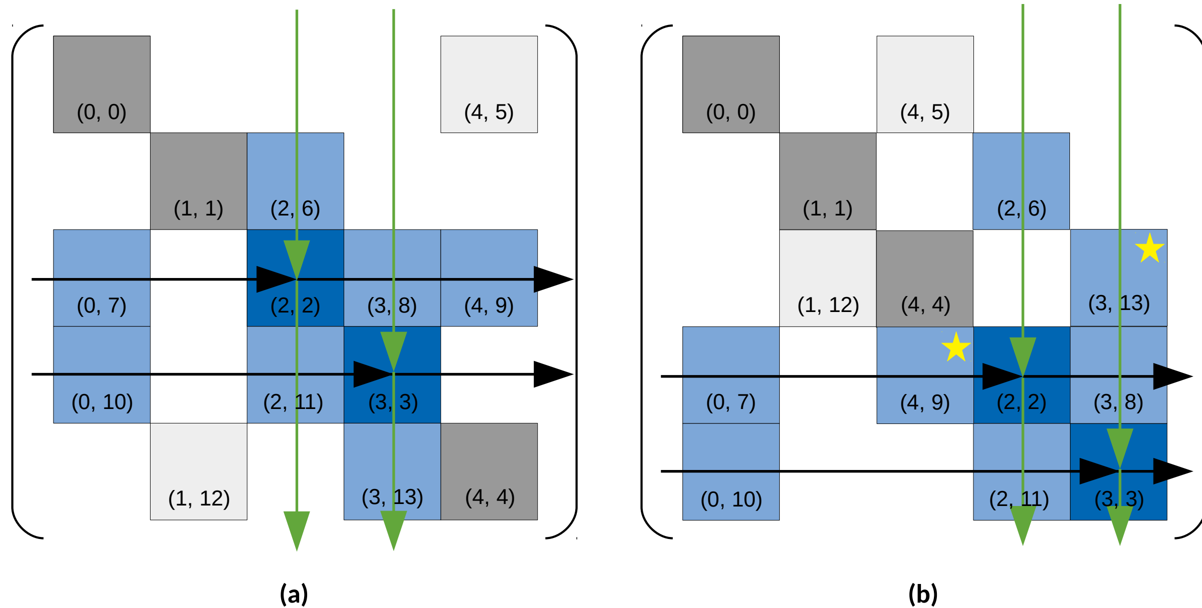


Figure 6.3: Example of a 'push-back' type pivoting operation: we permute rows and columns 2, 3 symmetrically to the rear of the matrix and sort the block references according to their new value in `piv_position` (see Section 6.4). Note: only the index pairs for R. are shown. The star signals transposed blocks.

An example of a 5×5 matrix pushing diagonal blocks 2 and 3 is given in Figure 6.3. Following the steps above, we ask the reader to draw his attention to the following points: (a) the most expensive step (4.) operates on all diagonal blocks independently, parallelized over diagonal blocks; (b) no actual permutation of diagonal blocks in memory is performed, we solely operate on indexes. After a permutation, the index pairs do not represent valid column respective row coordinates any more. Instead, the frontend always needs to map those via P^{-1}; in this sense, the index pairs are always sorted. Although indirecting through these indices seems like a hassle, it can be a benefit for the symmetric case: Consider the two blocks marked with yellow stars in Figure 6.3b: these have switched their sides of the block-diagonal, i.e., a block from $L^\top$ moved into L and vice versa. Since we do not store the blocks of L, $L^\top$ twice in memory, such blocks need to transparently transposed in the backend. Due to our pivoting scheme, we already imply this marker: if we encounter an index pair in L whose first component is larger that the initial column of this diagonal blocks' diagonal block, the block should be transposed (and vice versa for $L^\top$).

After such pivoting operations, the job schedules for subsequent levels need to be regenerated. To avoid additional latency, scheduling higher levels can be executed in a separate host thread while the backend starts working on the first, newly scheduled levels.

6.5 A Modified Jacobi-$LDL^\top$ Algorithm

The fixed point equations of Equation 6.2 are well suited for (massively-) parallel systems: each equation is indepedent from others, enabling the exploitation of coarse grained parallelism in the hardware. A sufficient condition for convergence of the fixed point iteration is diagonal dominance of the matrix A; for symmetric indefinite matrices, that is rarely the case. Pivoting operations during numerical factorizations are designed to move large elements to the matrix' diagonal. Therefore, we hypothesize that embedding

Algorithm 11 Embedding pivoting and blocking into the Jacobi $LDL^\top$ algorithm. Returns L'', D'' that minimizes $||L''D''L''^\top - A||_\infty / ||A||_\infty$ for a symmetric, indefinite matrix A. For convenience, we refer to block (i,j) in the blocked version of A as $A(i,j)$. p are permutation vectors for inner pivoting.

1: $L'' = L' = L \leftarrow$ tril$(A), D'' = D' = D \leftarrow I, p'' = p' = p = I$
2: **for** $sweep = 1, \ldots$ **do**
3: $\quad L' \leftarrow L'', D' \leftarrow D'', p' \leftarrow p''$
4: $\quad$ **for** $i = 1 \ldots m$ **do**
5: $\quad\quad B \leftarrow L(i,i)(p'(i), p'(i)) - \sum_{k<i} L'(i,k)D'(k)L'(i,k)^\top$
6: $\quad\quad [L''(i,i), D''(i), q] \leftarrow$ LDL$(B, \epsilon||A||_1)$
7: $\quad\quad p''(i) \leftarrow p'(i)(q)$
8: $\quad$ **end for**
9: $\quad$ **for** (i,j) where $j < i$ **do**
10: $\quad\quad C \leftarrow L(i,j)(p'(i), p'(j)) - \sum_{k<i} L'(i,k)D'(k)L'(j,k)^\top$
11: $\quad\quad L''(i,j) \leftarrow C(p''(i), p''(j))L''(j,j)^{-T}D''(j)^{-1}$
12: $\quad$ **end for**
13: $\quad \epsilon \leftarrow \epsilon\delta$
14: **end for**

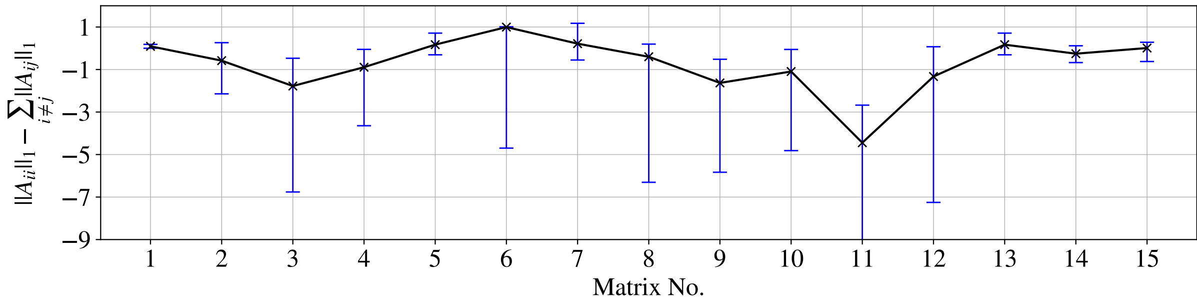
pivoting in Jacobi (fixed-point) factorization should help with convergence. To that end, we adapt Equation 6.2 for the $LDL^\top$ case and track the local permutations from inner pivoting in Algorithm 11. Two points are the deciding factors for convergence even in the symmetric indefinite case: (1) local permutations must be updated in sync with L', D', i.e., whenever accessing L', D', we also have to access p' to adapt the initial guess to the current iterate's permutation; (2) due to the lack of diagonal dominance, we frequently encountered small pivots, leading to large values in the off-diagonal blocks and, in consequence, divergence. As a remedy, we found aggressive pivot perturbations especially helpful: we set the ϵ in Algorithm 11 to large values such as 0.1 or 0.01. This usually stabilizes the iterates, such that it can be relaxed after each sweep by multiplication with $\delta < 1$. In our experiments, we use $\epsilon = 0.1, \delta = 0.95$. Going one level further, outer pivoting can be included as well: after the first sweep, we detect and push problematic diagonal blocks back. Then, all block accesses in Algorithm 11 need to go through P^{-1}. Lastly, we remark the following: When processing positive definite, but not diagonal dominant matrices, some diagonal blocks turn indefinite during the factorization; thus, the $LDL^\top$-variant also covers that case.

6.6 System Evaluation

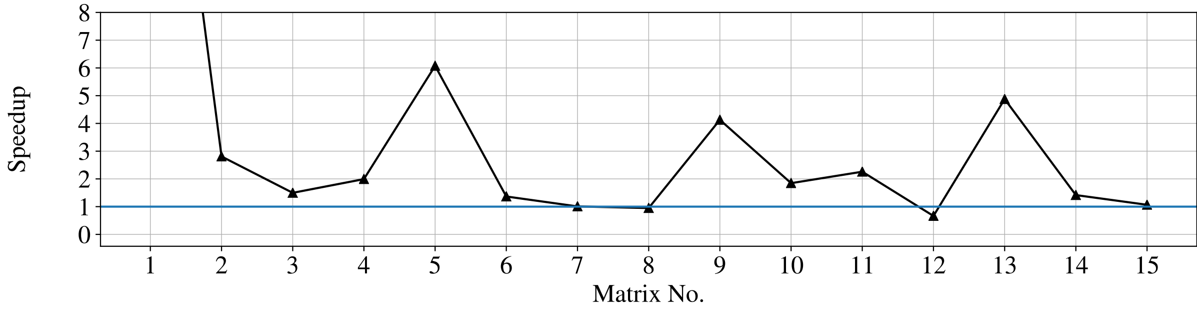
We implemented the modified Jacobi algorithm and the pivoting data structures in C++ with GPU parts in CUDA. Since this chapter is mostly concerned with enabling the use of Jacobi-factorizations (in comparison with the conventional level scheduling parallelization), we leave a highly-tuned implementation for future work. Despite that fact, Table 6.1 gives some timings for level scheduling factorizations. All experiments were executed on the same machine as those in Section 6.3 with 5 Jacobi sweeps.

We evaluate our method on a set of 15 matrices from the previous chapter in the same order; all matrices are symmetric indefinite and sparse. Every metric is permuted and scaled with MC64 and then overlapped with a regular grid. Since the original Jacobi method is sensitive to diagonal dominance, Figure 6.4a lists the minima, average and maxima of the quantity $||A_{ii}||_1 - \sum_{i \neq j} ||A_{ij}||_1$, a measure of the degree of diagonal dominance, over block rows. We notice a clear separation: while matrices 1, 5, 6, 7, 13, 14, and 15 are close to diagonal dominance, the other matrices are far from it. Applying a regular Jacobi method

(a) Average diagonal dominance (minimum/maximum values are logarithmic)



(b) Speedup of our Jacobi-method vs. level set scheduling



(c) Approximation error over Jacobi sweeps

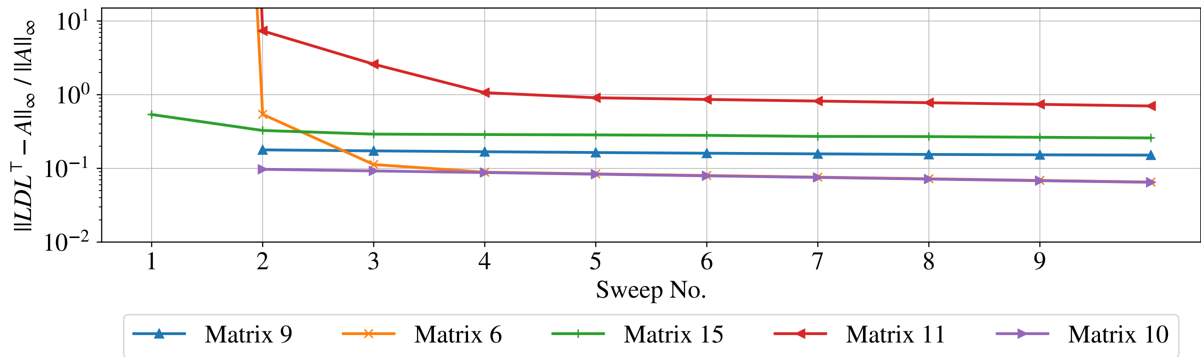


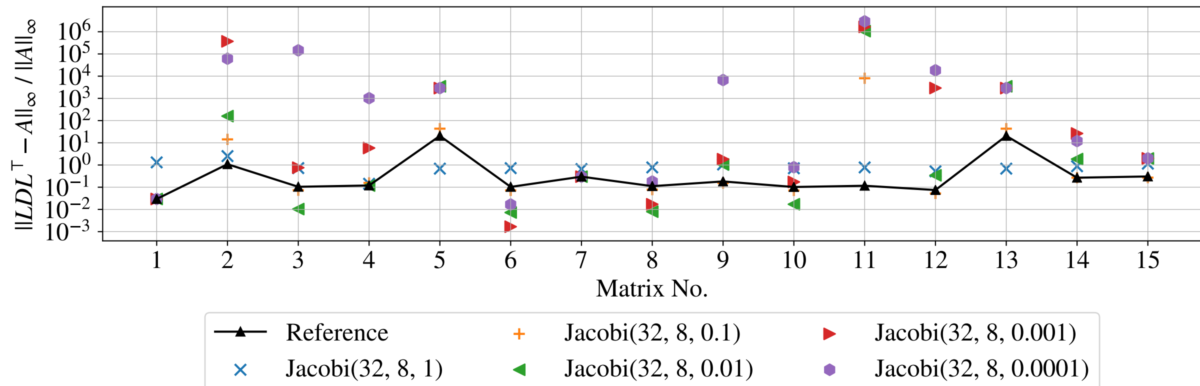
Figure 6.4: (a) Characterizes the matrices in our test set by their rows' diagonal dominance. **(b)** and **(c)** visualize the performance of the modified Jacobi algorithm; it exhibits high speedups after around 5 sweeps of preconditioner computation. Taken together, these results show that our modifications make the Jacobi method much more robust.

to the latter leads to strong divergence. No fill-in is added to the initial block structure on input matrices.

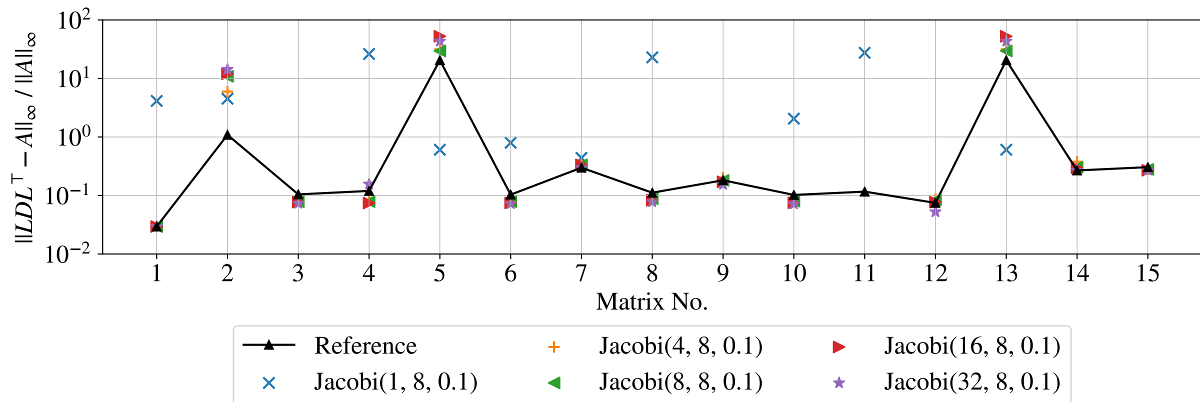
In Figure 6.5a - 6.5c, we present ablation studies for (a) the threshold ϵ, (b) the block size k as well as the the effectiveness of both inner and outer pivoting in (c). In all cases, we use the reconstruction error $||LDL^\top - A||_\infty / ||A||_\infty$; in our experience, this error behaves similar to the backward error when solving a linear system with the computed factorization as preconditioner. Results for the backwards error and relative residual in that case are given in Table 6.1; the Jacobi results were generated with the best setting extracted from Figure 6.5a per matrix. In all experiments, our reference is a level scheduling factorization (same backend, but different frontend) with otherwise the same parameters.

Additionally, we also used Magma's ParILU and ParILUT [Anzt et al. 2019b] and applied it to all systems as a reference. However, we did not get a single success - no system ever came beyond a residual of 0.1;

(a) Pivot perturbation tolerance ϵ



(b) Block size k



(c) Inner (i) and outer pivoting (o levels)

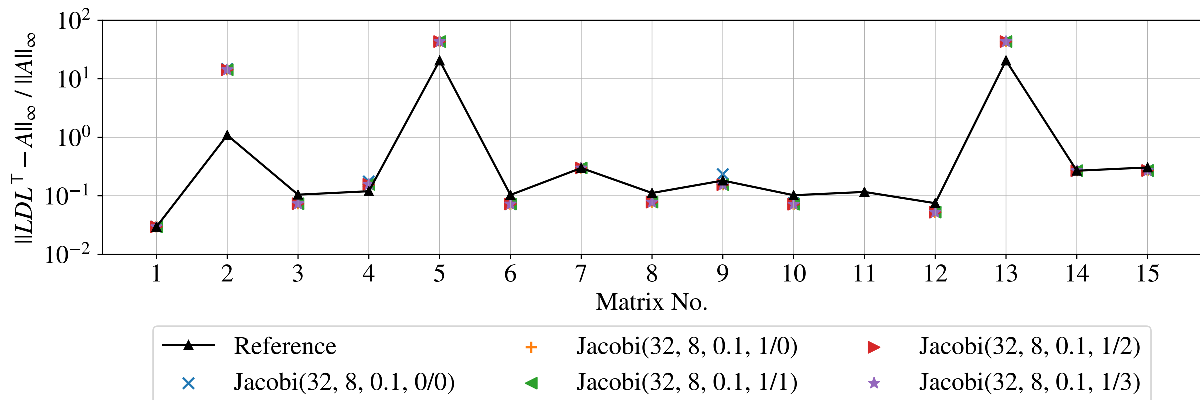


Figure 6.5: Plots for ablation experiments. Jacobi(K, S, ϵ, i/o) refers to the blocked Jacobi method with block size K, using S sweeps with pivot tolerance ϵ. Inner pivoting is enabled for i = 1, o gives the number of levels permitted for outer pivoting.

		Jacobi(32, 0.1)			Multilevel(32, 0.1)		
No.	Matrix	B	R	t	B	R	t
1	qpband	-6	-8	0.1 s	-6	-8	1.6 s
2	mario001	-1	-1	0.2 s	-1	-1	0.6 s
3	bab5	-4	-1	0.6 s	-4	-1	0.9 s
4	c-72	-3	-1	0.8 s	-3	-1	1.6 s
5	Si10H16	-4	-3	0.3 s	-4	-3	1.9 s
6	boyd1	-5	-4	1.1 s	-5	-4	1.5 s
7	Lin	-4	-3	0.5 s	-4	-3	0.5 s
8	map06	-4	-2	3.2 s	-4	-2	3.0 s
9	bley_xl1	-3	-2	1.7 s	-3	-2	6.9 s
10	c-big	-4	-2	2.0 s	-4	-2	3.8 s
11	co-100	-3	-2	1.7 s	-3	-2	3.9 s
12	bab3	-3	-2	20.1 s	-4	-2	13.4 s
13	G3_circuit	-4	-3	0.3 s	-4	-3	1.6 s
14	BenElechi1	-5	-5	1.0 s	-5	-6	1.4 s
15	af_shell7	-5	-4	1.8 s	-5	-4	1.9 s

Table 6.1: Using the computed preconditioner to solve matrices using 4 runs of GMRES(25). **B** refers to the backward error $\log_{10}(||b - Ax||_\infty / (||A||_\infty ||x||_\infty + ||b||_\infty))$; **R** is the unpreconditioned relative residual $\log_{10}(||b - Ax||_2 / ||b||_2)$ - **t** marks the computation times for both preconditioners.

most did not finish the computation of the preconditioner. This lines up with our earlier, independently produced results (see Chapter 5). In the interest of a fair comparison, we implemented a scalar variant of Jacobi-$LDL^\top$, see Figure 6.5b with $k = 1$.

In Figure 6.5a, we notice the importance of a strong ϵ for non-diagonal dominant matrices: small thresholds such as 0.001 or 0.0001 often lead to reconstruction errors magnitudes over the reference. On the contrary, a large threshold seems to contain numeric artifacts and help with convergence. For almost all matrices, we can find a parameter setting such that the Jacobi preconditioner matches or outperforms the level schedules one in its error. Furthermore, such a high ϵ also dampens the sensitivity with regard to block size k: As Figure 6.5b reports, the results of most block sizes over $k = 4$ show only minor differences. The scalar Jacobi algorithm, however, diverges strongly from that: its error is mostly more than two orders of magnitude higher, since it is restricted to its initial nonzero structure. Especially when dealing with indefinite matrices, capturing relevant fill-in is crucial.

During most of the Jacobi sweeps, the high ϵ keeps all pivots in such a range that inner pivoting and, surprisingly, outer pivoting, do not play a striking role in the final error of the computation (see Figure 6.5c). Beyond the Jacobi method, inner pivoting can still make or break a factorization in traditional schedules.

Lastly, we investigated the optimal number of sweeps and estimated the performance benefits of using the Jacobi method. As visible in Figure 6.4b, the factorization phase can be accelerated in this way; integrated implementations of frontend and backend could offer even more speedup. Figure 6.4c resembles the curves in Chow and Patel's [Chow and Patel 2015] original paper: Saturation often occurs after the fifth sweep, including for numerically tough matrices.

6.7 Conclusion

We presented blocking and a two-level pivoting scheme as an approach to use massively-parallel Jacobi-schemes for preconditioning even for numerically tough matrices. We found that aggressive pivot thresholding presents a useful remedy to a lack of diagonal dominance. Our results show that paired with the right ingredients, these methods can close the gap to traditional preconditioners.

The techniques presented in this chapter improve upon or replace the approaches from Chapter 5. Taken together, we now have a viable toolbox of multiple components that we can mix-and-match to create suitable solvers for parallel systems. However, both chapters are tightly coupled to CUDA-capable GPUs as a platform. The modified Schur complement, for instance, only works due to the hardware support for prediciation. In order to support other platforms, we need to simplify these kernels and consider moving more complexity into the shared host part.

Chapter 7

Cross-Accelerator Programming for Batched Kernels

Contents

7.1 Background 87

7.2 PIRCH - A Common Virtual Accelerator Architecture 88

7.3 borG 91

7.4 Evaluation 100

7.5 Related Work 108

7.6 Conclusion and Future Work 110

The inability to further scale single processor performance has triggered significant developments in widely parallel processor designs [Diaz et al. 2012]. Unfortunately, upcoming generations of processors are likely to include significant architectural changes to overcome the emerging technological limitations of silicon-based computers [Hennessy and Patterson 2019]. While these innovations help processors achieve higher performance, they usually introduce software stack modifications, including changes to programming models and languages, instruction sets and Software Development Kits (SDKs). In HPC environments, where software often requires a high degree of specialization and hand-tuning to achieve maximum performance on a given system, the cost of introducing new processors (e.g., GPUs) is therefore significant, since it requires changing the application software.

A second complication comes from the appearance of new diverse applications that have quickly changing requirements, such as machine/deep learning applications. In the ML field, new developments outpace the ability of system researchers to provide efficient software implementations, as also pointed out by a recent report [Barham and Isard 2019]. In fact, ML application frameworks, such as TensorFlow and PyTorch, make it easy to define deep learning computations in Python and then map to highly tuned, vendor-provided accelerator libraries, e.g., cuDNN, cuBLAS or OneDNN. Nonetheless, such frameworks mainly support common computations and hardware, lagging behind cutting edge developments, which therefore often need custom compute kernels for the target accelerators.

Together, these two combined hardware and software trends significantly complicate the situation for programmers and researchers alike, since implementing an optimized computation requires building custom kernels for each different architecture. In practice, this can be performed either by hand-coding or using architecture-specific native compilers, such as ISPC [Pharr and Mark 2012]. In both cases, due to the different platforms' programming models and idioms, this effectively requires the development of different platform-specific compute kernels, for any different computation and target architecture combination. The final result is a codebase that quickly diverges over time, as new applications and architectures appear. Ideally, at least for compute kernels that handle runtime hotspots, a single maintainable codebase without modifications across devices would be preferable.

In the context of sparse linear algebra, Chapters 5 and 6 introduced incomplete factorization algorithms that relied heavily on the availability of fast kernels for dense linear algebra on small ($\leq 32 \times 32$) blocks.

The provided, hand-implemented kernels achieved high performance results on CUDA GPUs, but their development required considerable effort. Similarly, they are tied to some aspects of the GPU's architectures. The future development of HPC demands for software that adapts to novel, heterogeneous systems. Therefore, we motivate this chapter's development of a cross-platform approach that makes the production of suitable kernels for, but not limited to, sparse linear algebra, easier and portable.

With this in mind, we address the issue presenting an abstraction to program high-performance compute kernels, and a compiler that is capable of mapping them to different target architectures. Avoiding the temptation of defining yet another programming model, our approach is instead targeted at extending existing abstractions that are a potential candidate for high-performance cross-architecture programming. In particular, we focus our attention on the "warp register cache" idiom, as commonly used in the CUDA community [Zhao et al. 2019; Li et al. 2018; Chen et al. 2019]. The idiom combines descriptions of computations that are performed by a small set of parallel executors (warp-centric, bulk-synchronous programming) with the explicit use of executors' registers as memory caches (cf. Section 7.1). While adhering to these two principles limits the expressiveness of CUDA code, the resulting compute kernels can offer a 2-3x speedup over other implementations, e.g., those that use a shared memory approach (see Chapter 5). Interestingly, we discover that these restrictions also make the implemented compute kernels easier to port to different architectures, while maintaining high-performance.

Starting from this observation, we therefore generalize the warp register idiom into the *collective register cache* (CoRe) by treating the number of parallel executors as a compiler-time constant. To abstract the target architecture, we then define a virtual architecture – PIRCH – which captures the information provided by a compute kernel that adheres to CoRe into an IR that subsumes representations for Single Instruction Multiple Thread (SIMT) and Single Instruction Multiple Data (SIMD) compute models (cf. Section 7.2).

Building on this basis, we design borG, a hybrid source-to-source compiler that compiles a subset of OpenCL code that follows the CoRe idiom to implement batched, general-purpose kernels to different architectures (cf. Section 7.3) via PIRCH's IR. The current borG implementation supports three different devices: Intel CPUs' AVX512 subsystem (SIMD); NVIDIA GPUs (SIMT); and NEC SX-Aurora vector processors (wide SIMD) as well as an LLVM IR backend. We show that borG can build compute kernels that compete in performance with hand-optimized ones, and in some cases with vendor-tuned libraries (cf. Section 7.4) using 7 kernels from various applications and areas.

In summary, our contributions are:

- we introduce PIRCH, a virtual architecture and its associated "InfReg-IR" to capture the semantics of programs that follow the CoRe idiom, and to abstract away the specific properties of target SIMD and SIMT architectures;
- we present borG, a source-to-source compiler for batched OpenCL programs, which generates high-performance compute kernels targeting three different accelerator architectures (AVX512, CUDA and NEC SX-Aurora), as well as LLVM IR;
- we provide details on specific tuning steps implemented in each of the accelerator backends and discuss their optimized primitives;
- we implement 7 different computational kernels from linear algebra, machine learning and numerical optimization using the CoRe idiom, and we evaluate borG for such implementations versus

reference kernels and implementations for each of the three supported architectures.

We provide a discussion about related work in Section 7.5, and give an outlook on future directions and open research questions in Section 7.6. For the remainder of this paper, we assume that the reader is familiar with OpenCL (or CUDA) and its vocabulary.

7.1 Background

When discussing "accelerators", we usually refer to PCIe cards such as GPUs or Intel's discontinued Xeon Phi series; however, here we include AVX512-enabled CPUs as well. Otherwise, we limit our scope to devices that can still run general-purpose code; this excludes FPGAs and Google's TPUs. While some accelerators, notably NEC's SX-Aurora, can run code as a host, we focus on the use of those accelerators as offload targets for hot spots in the code. We previously gave details on three such architectures in Chapter 4. Table 4.1 serves as a brief comparison of the accelerators' capabilities.

7.1.1 The Warp Register Cache Idiom

CUDA[1] offers a three-level structure for structuring the computation: grids, blocks and threads. Each block consists of a customizable number of threads and is located in a 3 dimensional index grid at runtime. A popular approach to porting applications onto GPUs is to structure the computation following the task-inherent data parallelism, e.g., a block of a matrix inside a General Matrix-Vector Multiplication (GEMV) kernel. As a second step, the programmer decides whether a piece of data would best be processed inside a single thread or by a whole block. Blocks, however, are virtual concepts without correspondence in hardware. For recent NVIDIA GPUs, the smallest unit for the GPU's scheduler is a warp, a set of 32 threads operating in lockstep[2]. With their Kepler architecture, NVIDIA introduced "shuffle" instructions as a means to communicate directly between threads inside the same warp. Given that recent NVIDIA GPUs offer 65,536 registers per SM, this allows programmers to deal with warps as small, but interlocked units with a large set of registers and very cheap communication within the warp.

This development popularized a new idiom for CUDA kernels: the "warp register cache" idiom, a combination of exploiting the lockstep execution inside a warp and the large register file as a user-managed cache to limit global or shared memory access. Strictly adhering to this idiom can result in high speedups compared to "conventional", block-and-shared-memory base implementations, especially when there is no need for inter-warp communication. The prime use case for this idiom are batched computations on small portions of data, e.g., batched GEMM or batched matrix factorizations [Anzt et al. 2017a; Anzt et al. 2017b; Thuerck 2019]. In those examples, each thread inside a warp loads, e.g., a row of the warps' matrix and uses shuffle whenever it needs access to the row stored by another thread. Since CUDA's language does not expose registers to the language level, developers commonly try to adhere to a few restrictions in order to "motivate" the compiler into mapping variables uniquely to registers:

1. Do not use thread-private arrays. Such arrays trigger allocations in global memory and there is no guarantee that NVCC caches accesses in registers. Instead, express every single scalar in an array as a separate variable.

[1] We did not find any literature discussing similar techniques w.r.t OpenCL, hence we stick to CUDA in this section.

[2] With the Volta architecture, NVIDIA has recently added Thread Independent Scheduling that adds a program counter to each thread, relaxing the lockstep model. In the remainder of this chapter, we still consider only the pre-Volta mode and use synchronous `shuffle`s and `syncwarp`s to enforce synchronization.

2. When dynamic selection is necessary, partition the data in a way such that shuffles may be used. If, e.g., each thread holds a row of the matrix (with each column as a single variable), we may use a shuffle to fetch data from a row determined at runtime. This is a GPU-specific hardware feature.
3. Try to balance the workload between warps – since the number of blocks scheduled onto an SM is limited, this could result in long-running warps that hog SMs while leaving resources unused.

In general, the first restriction is the most important one and also the one complicating manual implementations the most. It may lead to bloated kernel code (see e.g., the source code for Thuerck et al. [2018]) that requires some kind of code generation in order to be maintainable. Although research into the use of SIMD registers for register caching on CPUs [Shin et al. 2002] exists, its embedding into a SIMT context with fast shuffle instructions has not been investigated as a means to implement performance-portable kernels across architectures. In the remainder of this chapter, we propose exactly that. Simply put, we answer the question: How (and if) can code following the warp register cache idiom run fast on other accelerators than just GPUs?

7.2 PIRCH - A Common Virtual Accelerator Architecture

The warp register idiom binds the programmer to warps with 32 threads, which is clearly specific to the NVIDIA's architecture, limiting the portability of the idiom to different architectures. We therefore define the "collective register cache" (CoRe) idiom by simply removing this limitation, and instead allowing a configurable size for the number of threads. For instance, for code, e.g., processing $3x3$ matrices, one would select 3 threads and have the threads share the input matrix collectively. With the idiom defined, we now need a way to map it to different architectural models, i.e., SIMD and SIMT.

While the two models share similar concepts, their native programming models are different. For instance, in CUDA, we implement algorithms from the perspective of a single thread and we may use primitives to synchronize inside a block or warp. While the program is implemented as a scalar program, on GPUs the warp-based execution model provides implicit vectorization. For SIMD systems, both assembly and C with intrinsics use fixed-width vectors as a data type (i.e., explicit vectorization). Thus, their respective intermediate representations are also interpreted in different contexts. A prime example is `gpucc` [Wu et al. 2016] where host (x86, incl. SIMD) and device (GPU) code each have their own respective IR.

To build a common abstraction over the two models, we propose "Partitioned Infinite Register maCHine" (PIRCH) , a virtual architecture and its "InfReg IR" that offers a well-defined representation from which both SIMD and SIMT codes may be extracted. In a nutshell, PIRCH can be described as SIMT-on-SIMD (the distinction to ISPC's [Pharr and Mark 2012] SPMD-on-SIMD is outlined in Section 7.5). PIRCH is defined by three components: a compute/execution model and a (virtual) register format and, optionally, an IR for expressing programs.

We now turn to the three components that make up our PIRCH design.

7.2.1 Compute and Execution Models

PIRCH adopts the OpenCL compute model, which is, up to terms, the same as CUDA: a kernel is executed on a set of indexed work items (short: WIs, CUDA: threads) which are partitioned into equal-sized work groups (WGs, blocks). A certain number of work groups is launched with a kernel call into a queue (grid). Apart from access to device (global) memory, each WG has exclusive access to a defined amount of

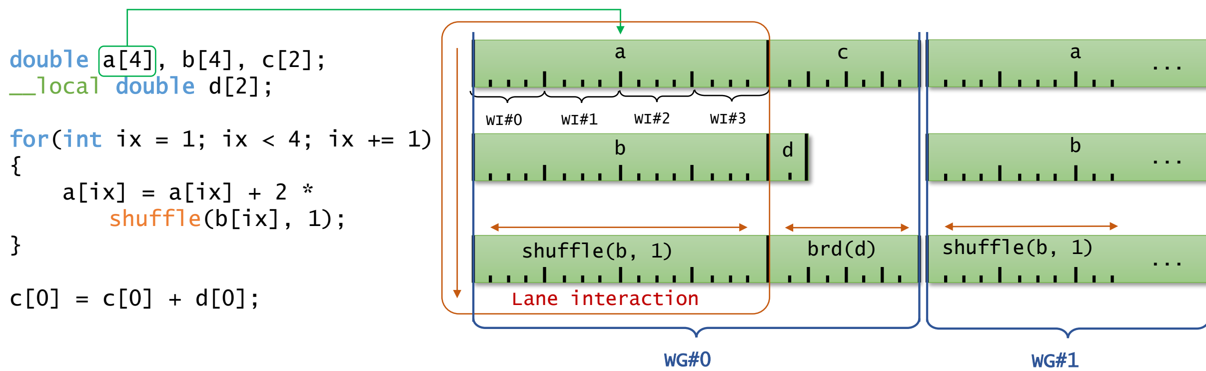


Figure 7.1: Mapping an OpenCL program to PIRCH with its WG-centered execution model: private variables are WI-wise concatenated into an InfReg (infinite register) and annotations with the corresponding WI are added to all lanes. Interacting variables are aligned vertically to avoid slow cross-lane operations. When WGs are completely independent, multiple instance of this kernel may be packed into the InfReg transparently by concatenating copies of the InfReg. This results in a single coherent representation for SIMT and SIMD programs.

local (shared) memory that may only be accessed by all WIs in the same WG. WGs are all launched and processed independently from each other.

In the execution model, however, we deviate from OpenCL: The standard does not define any specific way of executing the individual work items. While a scalar CPU may execute them sequentially, GPUs break up each WG into wavefronts (respective warps for CUDA) of a hardware-dependent size. Inside these wavefronts, the work items are, up to branching and predication, executed in lockstep. Wavefronts in the same WG are independent and must be manually synchronized by `barrier(...)`. In PIRCH, we execute the whole WG as if it was one wavefront. All work items in one WG execute in lockstep without explicit synchronization and we transfer CUDA's `shuffle` instructions (due to the lack of a similar instruction in the OpenCL standard) to exchange data between WIs with low latency. All work items in a WG share the same program counter (PC); on GPUs, this execution is similar to calling `barrier(...)` after each instruction. In a nutshell, PIRCH executes "wavefronts of software-defined size" instead of hardware sizes (64 for AMD, 32 for NVIDIA). As a consequence, we set aside the term "wavefront" for the remainder of this paper and use the now equivalent "WG"; second, we denote the compile time-constant wavefront size as `WG_SIZE`. Mapping this execution model to the actual hardware at hand and writing code that follows this model can be quite complicated. This is where borG (cf. next section) comes in: for batched kernels written in a limited subset of OpenCL C, it automatizes this mapping, resulting in cross-platform translation of PIRCH.

7.2.2 Register Format

PIRCH is defined as a vector architecture with registers of varying – theoretically infinite – width; we call those "INFinite REGisters" (InfRegs). Unlike conventional SIMD architectures, InfRegs have a layer of metadata associated with each lane (see Figure 7.1): a work item's offset in the work group and the work group's id, conveying all necessary information from the execution model in order to extract SIMT code. This effectively constitutes a layer of SIMT on top of SIMD-like vectors and implies access rights: only the WI identified by the metadata of a lane may access that lane in SIMT code; cross-WI operations require the explicit primitive `shuffle`. Lanes representing WG-wide local memory have multiple work item indices in their metadata. While in theory, every lane in an InfReg may have different scalar types,

we limit this to one common scalar type per InfReg in order to facilitate the mapping to actual hardware.

PIRCH's mapping of variables to InfRegs draws on the experience from current SIMD systems where interactions between different lanes are costly, but interactions between vectors within the same lanes are cheap. Mapping a WG's variables follows these rules:

1. For each variable with dimension n, allocate `WG_SIZE` $\times$ n and assign them similar to variable `a[4]` in Figure 7.1.
2. Variables interacting through computations are placed into different InfRegs, but are aligned to the same lanes.
3. In case two variables have nonintersecting liveness ranges and are compatible in type and dimension, they can be assigned to the same register and lanes.

An example is given in Figure 7.1. Notably, there is only a unique association from program variables to InfRegs, but not vice versa.

7.2.3 InfReg-IR

Putting both the InfReg format and the execution model together, we obtain a single representation that subsumes SIMT and SIMD representations. In practice, many typed IRs can be used to encode programs using PIRCH. We extend a broadly adopted IR in this paper, rather than adding another one to the mix. One of the most popular IRs in HPC these days is LLVM IR [Lattner and Adve 2004], the basis for many modern high-level programming language compilers. LLVM IR already supports both metadata and the definition of custom, aggregate types, but not both in combination or as type aliases. Thus, we propose to add type aliases and include the ability to associate typed metadata tuples.

Defining an InfReg type for the OpenCL declaration `float test[2]` with compile parameter `WG_SIZE = 3` leads to the following vector type alias definition:

```
%IR0 = type <6 x f32> metadata {i32, i32, i32} {ix, local_id, local_size}
       <{0,0,3},{1,0,3},{0,1,3},{1,1,3},{0,2,3},{1,2,3}>
```

This defines the type `%IR0`, an InfReg with 6 lanes where each WI's private arrays are concatenated. Each lane is, furthermore, associated with an metadata tuple `(ix, local_id, local_size)`, i.e., the array index represented by the lane, the associated WI index and the WG size. We propose to access metadata via a `%type!metaname` (similar to the LLVM IR meta operator) syntax, e.g., to create a boolean mask that activates only the lanes corresponding to WI with id 1, we issue: `%mask = icmp eq i32 %IR0!local_id, 1`. Apart from the metadata, `%IR0` may be used exactly like any other LLVM type: `%val2 = fadd %IR0 %val0, %val1` adds two InfRegs. Mapping the so-modified LLVM IR to SIMD processors is straightforward. However, for SIMT systems, we have to enforce location constraints: two lanes of vector types may interact if they share the `local_id`. The only notable exception here is the LLVM IR instruction `shufflevector` to which PIRCH's `shuffle` maps:

```
%IR1 = type <3 x f32> metadata {i32, i32, i32} {ix, local_id, local_size}
       <{0,0,3},{0,1,3},{0,2,3}>>
%val2 = shufflevector %IR0 %val0, %IR0 %val0, <3 x i32> <i32 2, i32 2, i32 2>
        to %IR1
```

corresponds to `float val2; val2 = shuffle(val0[0], 1);`. While the arguments for `shuffle` are deduced from the index vector, the output type must be explicitly named by adding `to...` as metadata

prevents a complex type inference. Figure 7.2 shows the interplay of these ideas.

In a nutshell, we demonstrated that PIRCH can be represented as an IR by careful modification of LLVM IR as a community standard. Alternatively, the modified LLVM IR may be implemented as a dialect of MLIR [Lattner et al. 2020], a lightweight framework for defining IRs. In the following, we refer to this modified LLVM IR as “Infreg-IR”.

7.3 borG

Porting programs written using the CoRe idiom to native programming models is usually hard; this is where borG comes in: it maps certain programs that follow the CoRe idiom automatically onto the InfReg-IR following the PIRCH execution model. borG is meant for implementing *batched* general-purpose kernels, e.g., kernels that run the same code on a batch of problems without any form of synchronization between the items in the batch. Some of those kernels, like batched GEMM on small matrices, have started to appear in popular numerical libraries Wang et al. [2014] and Gennady and Evgenii [2017] and found their way into, e.g., popular deep neural networks such as BERT [Devlin et al. 2018]. borG's frontend consumes a subset of OpenCL C as defined by the grammar in Table 7.1; we currently support all standard primitive C types and the OpenCL functions $\mathcal{B} = \{$`get_local_id`, `get_group_id`, `get_local_size`, `sqrt`, `max`, `min`, `abs`, `rcp`$\}$. Using multiple backends, borG can then generate code for CUDA, SX-Aurora and AVX512 as well as emit LLVM-IR. In short: While InfReg-IR suits general-purpose code, borG only processes a subset of those programs: batched (general-purpose) kernels.

Packing Kernels. When dealing with very small problems (e.g., $4x4$ matrices), a popular technique is packing multiple items into one WG, potentially further complicating the program code. This is where InfReg-IR shines: once a kernel is translated into InfReg-IR, we clone each InfReg and concatenate all instances of the same InfReg horizontally. Additionally, we add local offsets of a WG in its pack to the InfReg's metadata. After running this packed IR through the same backend, the resulting code processes multiple WGs' work items simultaneously (see the right hand side of Figure 7.1). Inside borG the variable `WG_PACK` determines how many WGs are packed in this way. Packing is the crucial method that allows the generated code to run well on both short and long SIMD architectures.

Advantages and Limitations. Besides the potential performance benefit from packing, borG offers several advantages: For GPU developers, it eases the pain when implementing warp cache kernels, through enabling the use of arrays. If the kernel uses `WG_SIZE` as a parameter (e.g., processing matrices of size `WG_SIZE` $\times$ `WG_SIZE`), borG also acts as a code generator for different problem sizes. For AVX512 and SX-Aurora developers, it offers the possibility to generate highly efficient code from a single codebase and programming model that is popular within the HPC community. Especially for the rather exotic SX-Aurora hardware with its very long vector lanes, borG can prove useful. For applications that are mainly based on batched kernels, borG immediately bumps up the number of supported architectures.

borG is based on PyCParser [Bendersky 2010] as it is easier to extend than LLVM. Our implementation does not allow any communication between different WGs, including atomic memory accesses. However, this limitation is not systemic; while within the restrictions of the collective warp cache idiom, borG could be extended to support it. An overview of borG's pipeline is given in Figure 7.3.

```
float t_mat_lu[get_local_size()];
float t_vec_b;

[...]

float tmp;
for(int j = 0; j < get_local_size(); j += 1)    {

    /* update each row's value */
    tmp = shuffle(t_vec_b, j);


    if(get_local_id() > j) {

        t_vec_b = t_vec_b - t_mat_lu[j] * tmp;

    }

    [...]
}
```

```
%IR0 = type <32 x float> metadata {'ix', '__wi_ix', '__wg_size', '__wg_off'} <{0,
    <0>, 4, 0}, {1, <0>, 4, 0}, {2, <0>, 4, 0}, {3, <0>, 4, 0}, […],
    {1, <3>, 4, 1}, {2, <3>, 4, 1}, {3, <3>, 4, 1}>

%IR1 = type <8 x float> metadata {'ix', '__wi_ix', '__wg_size', '__wg_off'} <{0,
    <0>, 4, 0}, […], {0, <3>, 4, 1}>

[...]

%ir00003.00000 = shufflevector %IR2 %ir00002.00000, %IR2 %ir00002.00000, <8 x
    i32> <i32 0, i32 0, i32 0, i32 0, i32 4, i32 4, i32 4, i32 4> to %IR3


%ir00004.00000 = icmp gt %IR1!__wi_ix, 0


%ir00001.00000 = shufflevector %IR0 %ir00000.00000, %IR0 %ir00000.00000, <8 x
    i32> <i32 0, i32 4, i32 8, i32 12, i32 4, i32 20, i32 24, i32 28> to %IR1

%ir00002.00001 = fneg %IR2 %ir00001.00000

%ir00002.00002 = tail call %IR2 @llvm.fma.v8f32(%IR2 %ir00002.00001, %IR3
    %ir00003.00000, %IR2 %ir00002.00000)

%ir00002.00003 = select %IR4 %ir00004.00000, %IR2 %ir00002.00002, %IR2
    %ir00002.00000


[...]
```

Figure 7.2: IR translation for a part of the TRSV (triangular solve) kernel according to the description in Section 7.2.3.

s	::=		*Statements:*
		;	skip
	|	$\{\hat{s}\}$	scope
	|	e;	expression statement
	|	t `id`;	declaration
	|	__local t `id`;	local variable declaration
	|	$e_1 = e_2$;	assignment
	|	e_1 `op=` e_2; `op` $\in \mathcal{O}$	(shorthand)
	|	`if` s_1 `else` s_2	if-else
	|	`while`(e) s	while
	|	`for`(s_1; s_2; s_3) s_4	for
f	::=		*Function:*
		ϵ	empty
	|	$f_1\ f_2$	sequence
	|	__kernel t `id`($\overline{t\ \texttt{id}}$) $\{\hat{s}\}$	function
e	::=		*Expressions:*
		c_α	constant of type $\alpha \in \mathcal{A}$
	|	`id`	variable
	|	`id`[e]	array indexing
	|	e_1 `op` e_2; `op` $\in \mathcal{O}$	binary operation
	|	$b(\overline{e}) \in \mathcal{B}$	call to build-in function
	|	`shuffle`($\overline{e}$)	WG shuffle
$\hat{s}$	::=		*Scoped Statements:*
		s	statement
	|	$\hat{s}\ s$	sequence
t	::=		*Type:*
		$\alpha \in \mathcal{T}$	base

Table 7.1: Grammar of the supported C subset by borG's frontend. The start symbol is f and we use the following descriptive sets: $\mathcal{T}$ for basic types, $\mathcal{B}$ for built-in functions and $\mathcal{O}$ for operators in the C standard (adapted from Leißa et al. [2012]).

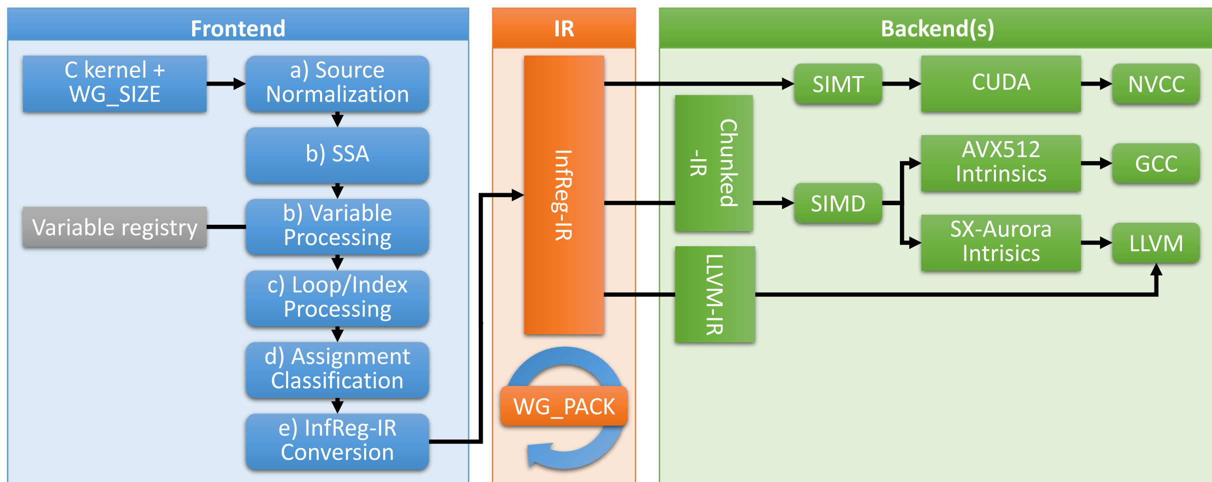


Figure 7.3: System overview of the borG source-to-source compiler: the system consumes a CUDA-like kernel written following the warp register cache idiom (Section 7.1.1) and maps it onto InfReg-IR, based on the virtual architecture PIRCH. From there, either C + intrinsics or LLVM-IR may be generated, effectively generating code for three popular accelerator architectures.

7.3.1 Frontend

borG's frontend consumes the kernel code and a WG configuration (`WG_SIZE`, `WG_PACK`) and generates a kernel description in InfReg-IR. Given a OpenCL C-file, the frontend parses all functions with a `__kernel` annotation. As depicted in Figure 7.3, borG's frontend executes a sequence of code transformations on the abstract syntax tree (AST). Listing 3 shows the different steps of the code transformations; Listing 3 (a) represents the input code.

Source Normalization. In order to facilitate source code processing downstream, we promote all scalar variables to array variables of size 1 and add an index `[0]` to all accesses; furthermore, we expand all combined assignments (e.g., `+=`) into their full expressions and move the conditions in ternary operations into if-blocks. Lastly, we collect all defined variables with their respective (scalar) types and dimensions. All collected variables are classified as "explicit" variables.

SSA Decomposition. Similar to most compilers, we perform a conventional Single Static Assignment (SSA) decomposition on each assignment in the code, leaving only elementary operations on the right-hand side; all resulting assignments are placed in a single code block. For each additional assignment, we instance a copy of the variable on the left hand side of the original assignment that is local to the generated block. We mark those local variables as "implicit". Due to PIRCH's requirements on cross-lane operations and the restriction to compile-time array indices, we extend the SSA decomposition: Array indices for the variables on the left and right hand side are compared symbolically using Sympy Joyner et al. [2012]. If we encounter symbolically different indices, this results in an additional assignment for variable reordering being generated. Similarly, any calls to `shuffle`, another type of cross-lane operations in the InfReg down the road, imply an additional assignment. For each `if`- and `else` block we create boolean array-valued variables that are used as masks. All masks – one instance per compatible IR-format of variables that appear in the conditional's body – are then propagated to assignments as conditionals in ternary operations. Lastly, whenever we encounter a matching pattern in the AST, we insert FMA-type operations automatically. After this step, the code of our running example has been transformed into

Listing 3 Example of code transformations undergone in borG's frontend up to the InfReg-IR conversion. For details, please refer to Section 7.3.1.

a)

```
float a[4], b[4], c[4];
for(int ix = 0; ix < 4; ++ix)
{
        a[ix] = a[ix] + 2 * (b[4 - ix - 1] + shuffle(c[ix], s));
        if(get_local_id() > s)
                b[ix / 2] += c[ix / 2];
}
```

b)

```
float a[4], b[4], c[4];
for(int ix = 0; ix < 4; ++ix)
{
        a_0[ix] = b[4 - ix - 1]
        a_1[ix] = shuffle(c[ix], s);
        a_2[ix] = a_0[ix] + a_1[ix]
        a[ix] = __fma(2, a_2[ix], a[ix]);
        if_0[ix / 2] = static_resolve(get_local_id() > s);
        b[ix / 2] = if_0[ix / 2] ? (b[ix / 2] + c[ix / 2]) : b[ix / 2];
}
```

c)

```
float a[4], b[4], c[4];

/* ix = 0 */
a_0[0] = b[3]
a_1[0] = shuffle(c[0], s);
a_2[0] = a_0[0] + a_1[0]
a[0] = __fma(2, a_2[0], a[0]);
if_0[0] = static_resolve(get_local_id() > s);
b[0] = if_0[0] ? (b[0] + c[0]) : b[0];

/* ix = 1, 2 */
for(int ix = 2; ix < 3; ++ix)
{
        a_0[ix] = b[4 - ix - 1]
        a_1[ix] = shuffle(c[ix], s);
        a_2[ix] = a_0[ix] + a_1[ix]
        a[ix] = __fma(2, a_2[ix], a[ix]);
        if_0[ix / 2] = static_resolve(
                get_local_id() > s);
        b[ix / 2] = if_0[ix / 2] ? (b[ix / 2] +
                c[ix / 2]) : b[ix / 2];
}

/* ix = 3 */
[...]
```

.

Listing 4 Static pivoting LDU-Kernel implementation for borG.

```
1  __kernel
2  void ldus_generic(double * mat_a) {
3          double t_mat_a[get_local_size()];
4          double t_piv_row[get_local_size()];
5
6          /* load input matrix A */
7          for(int ix = 0; ix < get_local_size(); ix += 1) {
8                  t_mat_a[ix] = mat_a[get_group_id() * (get_local_size() *
9                          get_local_size()) + get_local_id() * get_local_size()
10                         + ix];
11         }
12
13         /* decompose matrix (with fused Schur complement!) */
14         for(int s = 0; s < get_local_size(); s += 1) {
15
16                 /* copy pivot row to PE-local storage */
17                 for(int ix = s; ix < get_local_size(); ix += 1)
18                         t_piv_row[ix] = shuffle(t_mat_a[ix], s);
19
20                 /* scale pivot column */
21                 if(get_local_id() > s)
22                         t_mat_a[s] = t_mat_a[s] / t_piv_row[s];
23
24                 /* scale pivot row and perform Schur downdate */
25                 for(int ix = s + 1; ix < get_local_size(); ix += 1) {
26                         if(get_local_id() >= s) {
27                                 t_mat_a[ix] = (get_local_id() == s ? (t_mat_a[ix] /
28                                         t_piv_row[s]) : (t_mat_a[ix] - t_mat_a[s] *
29                                         t_piv_row[ix]));
30                         }
31                 }
32         }
33
34         /* save decomposed matrix A = LDU */
35         for(int ix = 0; ix < get_local_size(); ix += 1) {
36                 mat_a[get_group_id() * (get_local_size() * get_local_size()) +
37                         get_local_id() * get_local_size() + ix] = t_mat_a[ix];
38         }
39 }
```

listing (b) in Listing 3.

Variable Processing. We extract all implicit and explicit variables and assign unique global IDs. Then, we gather all variables that interact with each other, i.e., appear in the same assignment, into a matrix. By interpreting this matrix as a graph and extracting connected components, we find independent sets of variables. Two variables from different sets may be packed into the same lanes in an InfReg; variables inside the same set must be packed into different InfRegs, but aligned to the same lanes. We pack all variables into InfRegs in a greedy manner, packing the independent sets horizontally into each InfReg. Lastly, we annotate all lanes with their respective WI ids.

Loop and Index Processing. Kernels following CoRe can contain WI-local arrays, e.g., `t_mat_a` in Listing 4. We map each lane of an array to registers, hence their indices must be resolved at compile time. When arrays are accesses in the body of a for-loop, we first classify that loop into one of three categories: unroll, vectorize and runtime. If the loop index depends on a runtime variable, all instructions inside the loop's body are predicated with a mask that deactivates all lanes of WIs that have already left the loop. At runtime, the loop is executed until all WIs evaluate the loop condition to false. In case of nested for-loops, all outer loops are unrolled. Otherwise, we evaluate their indices in borG and propagate the values to array accesses in the loop body. Here, we iterate over the loop's counter variable sequentially and determine index sets that could be processed within a single instruction ("vectorize"). As an example, compare lines 6 and 8 in Listing 3's (b): for `ix = [0, 1, 2, 3]`, line 6 leads to indices `[0, 1, 2, 3]` on the left hand side, so all lanes used in this loops may be processed in a single instruction. Line 8 leads to `[0, 0, 1, 1]`. To avoid write conflicts, we split the loop progressively into index sets `[[0], [1, 2], [3]]`. Without this type of analysis, SIMD units would be dramatically underutilized. The resulting code after splitting is given in listing (c) of Listing 3. In doubt, e.g., when an array index depends on the loop variable of a runtime-for, we execute a loop sequentially, i.e., with one array member per WI processed per iteration instead of the whole array in a single iteration. When array indices cannot be determined at compile time, the whole array is temporarily stored in memory and accesses for the current instruction are mapped to gather/scatter instructions, leading to drastic performance losses.

InfReg-IR Conversion. Remaining loops in the transformed C code are unrolled into separate IR nodes at this stage, resulting in a doubly-linked list of nodes, that are then converted into InfReg-IR. As last step, we pack multiple WG instances into the InfRegs as described before, which then automatically expands all IR instructions to processing multiple WGs. The resulting IR is then handed off to one of our backends.

7.3.2 SIMT Backend

borG currently offers two classes of backends: SIMD and SIMT. The SIMT backend generates CUDA-C++ (since OpenCL does not expose all features of NVIDIA GPUs, e.g., `shuffle`) that follows the warp register cache idiom from InfReg-IR. We restrict `WG_SIZE` to $\leq$ `32` to leverage the GPU's hardware shuffle functions and warp-lockstep execution. The crucial point in this backend is the decomposition of InfReg-IR computations into execution groups, i.e., groups of CUDA threads inside a warp that execute the same code (possibly of multiple WGs depending on the `WG_SIZE` as at this point, individual WGs have been abstracted away). We partition the WIs into active sets according to the InfReg lanes' element indices that they access. Each groups' code is predicated using a condition such as `((1 « get_local_id()) & 0x02)` with `get_local_id()` as the WIs index from an InfReg's metadata. Groups' code is serialized in case of diverging accesses, as in the case for `b` in `a[0] = a[0] / b[get_local_id()]`. Here, `b`'s index varies per WI, thus every WI is its own execution group. Consequently, all WIs are serialized (for `WG_SIZE` = 2), resulting in code resembling the following extract:

```
if(get_local_id() == 0) a_0 = a_0 / b_0;
if(get_local_id() == 1) a_0 = a_0 / b_1;
```

If `WG_PACK` is set to a value $>$ `1`, the whole WG code is replicated and references to the work group index are modified accordingly with the appropriate `WG_PACK`. Reordering, shuffles and ternary operations all map natively to the scalar CUDA language. The shuffle and reordering patterns are inferred from the InfReg-IR by analyzing the lane permutation vectors. While WI-local variables are mapped to registers,

WD-wide shared variables are put into the GPU's local memory. Furthermore, we use the cooperative groups-API offered by NVIDIA to handle `WG_SIZEs` ≤ 32.

Memory Accesses and Optimizations. In an optional frontend pass, we analyze index expressions for memory accesses using SymPy [Joyner et al. 2012]. We try to express the expression as a polynomial on a work item's local and global id. Based on that analysis, we determine whether an access is contiguous and extract an scalar offset for contiguous loads and stores. An example is line 8 in Algorithm 4: PIRCH's InfReg format, where the private arrays `t_mat_a` are concatenated WI-wise, matches the row-major memory format: iterating over the local id and then `ix` leads to contiguous indices into the InfReg. Hence, borG can forego the vectorized index computation and directly issue contiguous load/store calls for SIMD systems. However, on GPUs, all work items load one member of `t_mat_a` per instruction, leading to non-coalesced loads with a stride of `WG_SIZE` doubles. For ideal register-cached kernels, this does not matter much given that they ideally only read and write their data once from/to memory and otherwise use registers exclusively. For the future, we plan to include an automatic transposition pass that reads the data using coalesced loads and then transposes it through `shuffles`. Until then, developers can change the storage format of `mat_a` to, e.g., col-major.

Except for dead code elimination and memory analysis, borG's backend does currently not perform any optimizations on the IR or native code level. Since these optimizations can be very architecture-dependent and we only compile to the source (or IR) level, we rely on the optimizations performed in the native compilers.

7.3.3 SIMD Backends

The initial mapping from InfReg-IR to SIMD-based ISAs proceeds by splitting arbitrary-length IR vectors into pieces in the native vector length, which we call "chunks". This is possible due to the packing constraints enforced by the frontend after SSA'ing the input code: all interacting data in InfRegs are aligned to the same lanes; reorderings have been processed in an earlier assignment and the resulting implicit variable is aligned in its InfReg as well. We implemented a common "SIMD" backend for AVX512 and SX-Aurora and specialized only the configuration of scalar type-specific native vector lengths and primitive mappings from IR operations to vendor-specific C intrinsics.

Specialized Reorderings for Cross-lane Operations. CUDA kernels in the warp register cache idiom often follow an approach where each thread stores only parts of the problem's input data in its registers and uses shuffles to access other threads' data. Using borG, this approach leads to a high number of undesirable cross-lane operations. By default, such operation default to issuing a store and a gather call to permute the lanes contents. Nevertheless, analyzing the overlap in the AVX512 and SX-Aurora instruction set has resulted in three techniques mirroring useful shuffle patterns in CUDA (see Figure 7.4, all instructions here embedded in a contiguous `for`-loop with loop counter `ix`):

- Scalar broadcast: `a[ix] = shuffle(a[0], 2)`,
- Vector broadcast: `a[ix] = shuffle(a[ix], 2)` and
- Segmented scalar broadcast: `a[ix] = a[0]`.

Each of these methods works within one native register. Interestingly, the length of chunks covering an InfReg can make a striking difference regarding which method is selected, as evident from Figure 7.5. In practice, shorter vectors see more scalar broadcasts, longer vectors more vector or segmented scalar

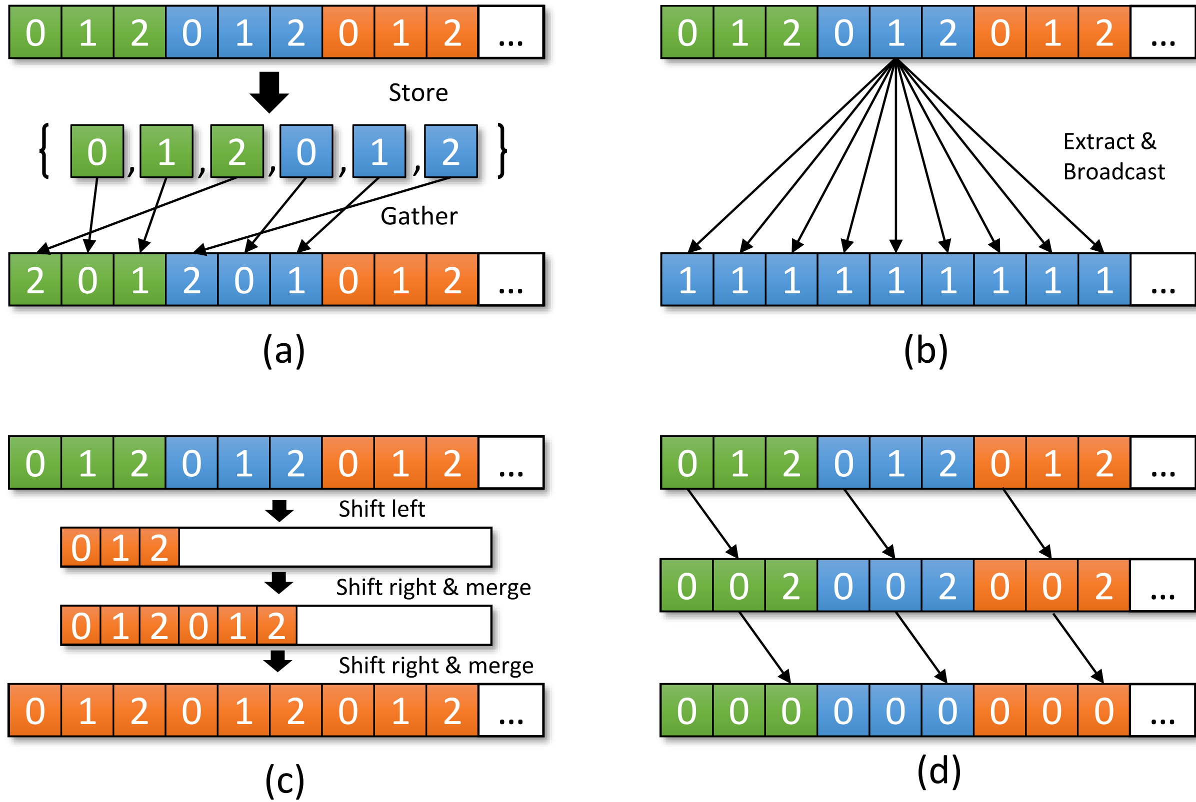


Figure 7.4: Normal store/gather and our three cross-lane shuffle/reorder implementations for SIMD architectures to speed-up borG's kernels (from left to right): **(a)** Store/gather for arbitrary permutations, **(b)** Scalar Broadcast as well as **(c)** Vector Broadcast and **(d)** Segmented Scalar Broadcast, the latter being implemented by repeated logarithmic lane shifting and merging.

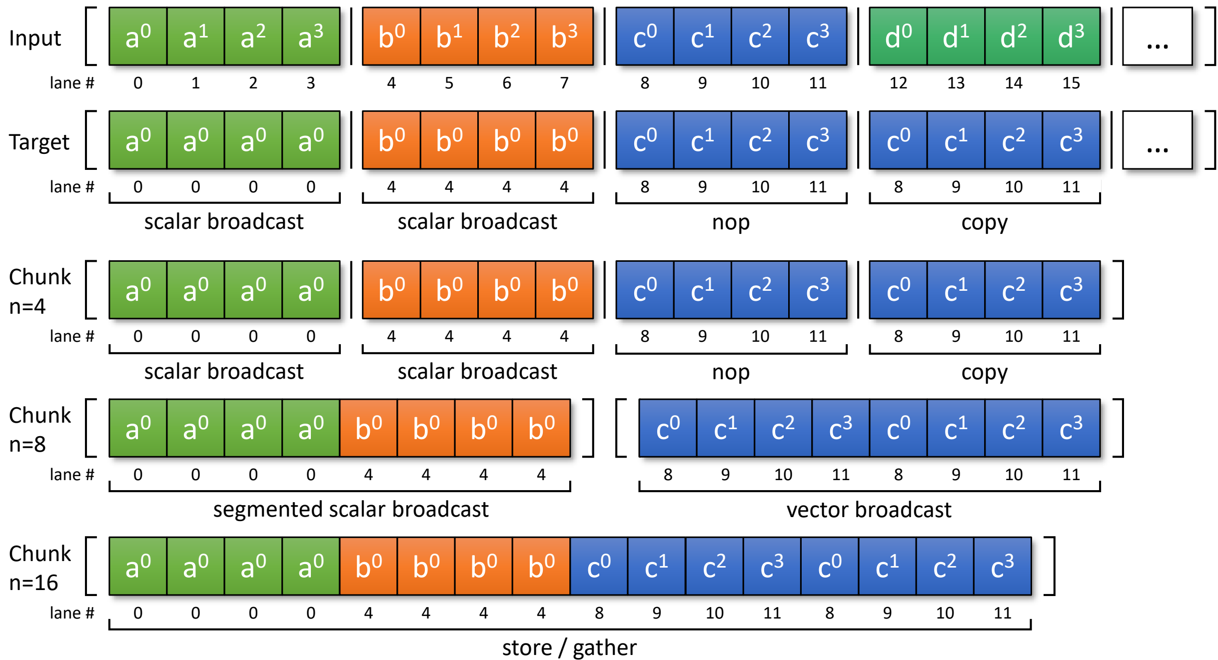


Figure 7.5: When an InfReg is split into pieces of native SIMD width (chunks), the length of each of those chunks can change the selected implementation for reorderings. Within a general permutation vector, there may be chunk-aligned sub-sequences that fit one of our patterns.

broadcasts. Additionally, when the same permutation vector appears multiple times, we just copy the register in question.

7.3.4 LLVM backend

The simplest among the three backends is the LLVM backend, which technically belongs to the SIMD class. Given InfReg-IR, it outputs LLVM-IR that may then be compiled for all architectures supported by LLVM, such as x86, ARM or, as with our chunked backend, AVX512 and SX-Aurora. Converting from InfReg-IR to LLVM-IR consequently only means dropping all metadata and expressing all InfRegs as vectors with the respective number of lanes. With the exception of masked scatter and gather, we avoid LLVM intrinsics (as they are not supported across most architectures) and just use standard LLVM-IR. We model masks by vectors of type `<n x i1>` which are applied through the `select` instruction. Cross-lane permutation vectors are also stored directly in the IR. Chunking and implementation selection is left to LLVM's backends.

7.4 Evaluation

We evaluate borG on 7 batched kernels from linear algebra (4), numerical optimization (2) and machine learning (1) with $n = \texttt{WG_SIZE}$ and matrices $A \in \mathbb{N}^{n \times n}, B \in \mathbb{N}^{n \times n}, C \in \mathbb{N}^{n \times n}$:

1. **GEMA**: matrix-addition $C = A + B$;
2. **GEMM**: matrix-matrix multiplication $C = AB$ in outer product formulation;
3. **TRSV**: 2 subsequent $n \times n$ triangular solves of an n-vector b;
4. **LDUS**: $n \times n$ LDU factorization $A = LDU$ where D is a diagonal matrix with 1×1 diagonal blocks with static pivoting;
5. **PADE**: backward pass for the recently proposed, polynomial Padé Activation Unit Molina et al. [2019], a kind of polynomial, learnable activation function for deep learning;
6. **ENUM**: on-the-fly enumeration of small integer programs of the form $\max c^\top x$ s.t. $Dx \leq b, x \in \mathbb{Z}^{10}$ and $b \in \mathbb{R}^8$;
7. **FRCG**: Fletcher-Reeves CG Fletcher and Reeves [1964] optimization on the Rosenbrook family of functions $f(x, y) = (a - x)^2 + b(y - x^2)^2$ with minimum line search and a fixed number of steps.

All matrices are randomly initialized with random numbers from $[0, 1]$.

These kernels were selected with different aspects in mind: most of the kernel choices were application driven. For very large scientific simulations, the Jacobi method has recently been re-established as a method of choice, especially when considered with mixed precision [Anzt et al. 2019a; Anzt et al. 2017a]. Therein, computing inverse of a block-diagonal matrix requires application of the LDUS and TRSV kernels; furthermore, if we expand the block-matrix concept to incomplete factorizations, we require a GEMM kernel as well as a GEMA kernel for Schur downdates. Very large sparse matrices beyond a few million rows quickly lead to batch sizes of $100,000$ for $\texttt{WG_SIZE} = 32$. PADE is taken directly from Molina et al. [2019] and ENUM is used in a research project based on Buchheim et al. [2008] and Applegate et al. [2001] in which we repeatedly enumerate the whole solution space of integer programs with k variables – there, the batch size is k. Using a single GPU, we often set $k \approx 12$, leading to a batch size in the billions. Second, we selected the kernels such that they cover different reordering patterns and functionalities: GEMM uses both segmented scalar and vector broadcast, TRSV relies on implicit reorderings to convert

between IR formats, LDUS uses masking and ENUM (base) type conversions. Lastly, we try to cover both a range of standard kernel that are available in vendors' libraries (GEMA, GEMM, LDUS) as well as custom kernels to back our claims.

We group our kernels into two categories: "group-wise" kernels (1-4) and "item-wise" kernels (5-7). An example of group-wise kernels is GEMM, where each batch item corresponds to a matrix and each work item only holds a part (e.g., a row) of the matrix. In such kernels, work items must share their data extensively via `shuffle` operations, leading to the maximum InfReg width being $\texttt{WG_SIZE}^2 \times \texttt{WG_PACK}$. There, the data-parallelism is on the level of the work group. In item-wise kernels, all variables are scalar and all WIs are independent, i.e., there is no data sharing via `shuffle` instructions. The InfReg widths for such a kernel are uniformly $\texttt{WG_SIZE} \cdot \texttt{WG_PACK}$, which is a similar mapping as ISPC with one work item per SIMD lane – so the granularity of data-parallelism is in the work items.

Since borG's purpose is supporting rapid cross-architecture development of batched general-purpose computing kernels, we compare against hand-written, hand-tuned **references** for each accelerator. All reference kernels were written targeting the architectures' native programming models and run through their native compilers. For CUDA, we use NVIDIA's **NVCC** and used `WG_SIZE`, as a template parameter. All data is stored in private arrays and as all array indices can be resolved at compile time, NVCC can cache array accesses in registers. Whenever possible, we also used (sub-)warp shuffles using NVIDIA's cooperative groups API. For SX-Aurora, we used NEC's auto-vectorizing **NCC** compiler v3.0.28. We embed scalar kernels into OpenMP loops over all batch items, allowing the compiler to vectorize beyond the boundaries of single problems (similar to WG packing). AVX512 code is handled by **ISPC**, which has proven to generate code that often surpasses hand-written intrinsics.

In addition to our references, we compare linear algebra kernels to vendors' BLAS libraries if they contain batched kernels as well. In the following, we list the equivalent libraries and functions therein for the GEMA/GEMM/LDUS kernels for all three architectures:

- Aurora: veBLAS with `{s,d}axpy_/{s,d}gemm_/-`
- AVX512: MKL with `cblas_{s,d}axpy/{S,D}GEMM_BATCH/{S,D}GETRF`
- CUDA: cuBLAS with `cublas{S,D}axpy/cublas{S,D}gemmStridedBatched/`
  `cublas{S,D}getrfBatched`

Where necessary, we have added parallelizing for-loops around BLAS to support batched workloads.

Since BLAS only covers standard kernels, we also compare to Tensor Comprehensions (TC) [Vasilache et al. 2019], another generator for custom kernels for NVIDIA GPUs. However, we were only able to create a subset of our kernels with TC. TC infers iteration variables and their ranges automatically from the source code. Partial iterations, like in line 22 in Algorithm 4 are not supported – they must be emulated through the manual definition of "views" on the matrix in a user-provided wrapper. Even then, iterative computations on such views must be unrolled into separate kernels and reduced in a separate kernel at the end. Since the resulting performance is far from the range that borG's kernels lie in, we only include GEMA and GEMM which are implemented as a single TC function each.

borG's generated intrinsic code is compiled by GCC 8 (AVX512), the experimental VE-extension to LLVM (LLVM-VE v1.13) and NVCC (CUDA). Since the LLVM extension for SX-Aurora with arbitrary vector lengths (LLVM-VE-VL) failed to compile most of the generated LLVM IR that its AVX512 counterpart successfully

Kernel	CUDA	AVX512	AVX512/LLVM	VE
GEMA	0.71x	1.00x	1.02x	0.98x
GEMM	1.30x	2.26x	2.92x	0.22x
TRSV	2.00x	1.70x	1.60x	1.87x
LDUS	3.31x	2.38x	2.86x	1.72x
PADE	0.96x	0.91x	-	3.82x
ENUM	0.91x	1.57x	-	1.29x
FRCG	0.82x	1.26x	2.63x	0.28x
geom. Mean	1.23x	1.49x	2.05x	1.00x

Table 7.2: Geometric mean speedups for borG vs. the reference kernels, computed over all `WG_SIZE`s.

builds (clang 8), we exclude it from our benchmarks. We run both AVX512 and SX-Aurora experiments on a server featuring an Intel Xeon 6126 Gold processsor, 96 GB RAM and 2 NEC SX-Aurora TSUBASA 10B cards running on CentOS 7. CUDA experiments are executed on a PC with an Intel i7-9700K CPU, 32 GB RAM and a NVIDIA TITAN V GPU. We use CUDA 10.1 and the nVIDIA driver version 430.50. All compilers may use the flags `-O3 -march=native` for optimization. We execute all kernels with a batch size of $100,000$ and the determine the minimum execution time in 10 benchmark rounds – after running 2 warmup rounds in order to compensate for OS, scheduling and cache effects. We measure execution time using C++11's chrono API. Double and single precision are plotted separately. The correctness of all kernels was checked by automatically generated tests, comparing the results against a CPU reference implementation.

7.4.1 Comparison with Reference Kernels

We build all kernels for all three architectures with parameters `WG_PACK` $\in \{1, 2, 4\}$ and `WG_SIZE` $\in \{4, 8, 12, 16, 20, 24, 28, 32\}$ for "group"-kernels and up to the native vector width for item-wise kernels – the limit of `WG_SIZE` ≤ 32 is a result of the CUDA-backend mapping WGs to warps; all other backends can exceed that threshold. In the interest of comparisons between architectures, we thus limit the range of benchmarks to suit CUDA. We run benchmarks for both single and double precision and select the result with the lowest runtime per precision and `WG_SIZE` (see table in the supplementary material for selected `WG_PACK`s). We plot the resulting runtime in logarithmic scale (compensating for different orders of runtime between $n = 4$ and $n = 32$) in Figures 7.6 and 7.7. Per kernel and device, we compute geometric mean speedups for borG-generated kernels over native kernels, see Table 7.2. Averaged over all architectures, borG's generated code achieves speedups of $1.23x$ for CUDA, $2.05x$ for AVX512 (via LLVM) and comes to a draw for the SX-Aurora. In most cases, (near-) optimal parameters may be determined by adhering to the following guideline: `data per WI` $\times$ `WG_SIZE` $\times$ `WG_PACK` should be less or equal to 2 times the native vector length on SIMD systems.

Group-type Kernels. borG's GEMA kernel on CUDA is 39% slower than its reference. The reference implementation reads two values from memory, adds them up and immediately stores them back to memory instead of caching the whole matrix in registers first, significantly reducing the number of registers required compared to a full-blown register cache. Since there is no data reuse in GEMA, the extra effort for caching both input matrices does not pay off here. This is in stark contrast to the LDUS kernel on CUDA: with a high degree of data reuse directly from registers, borG achieves a speedup of

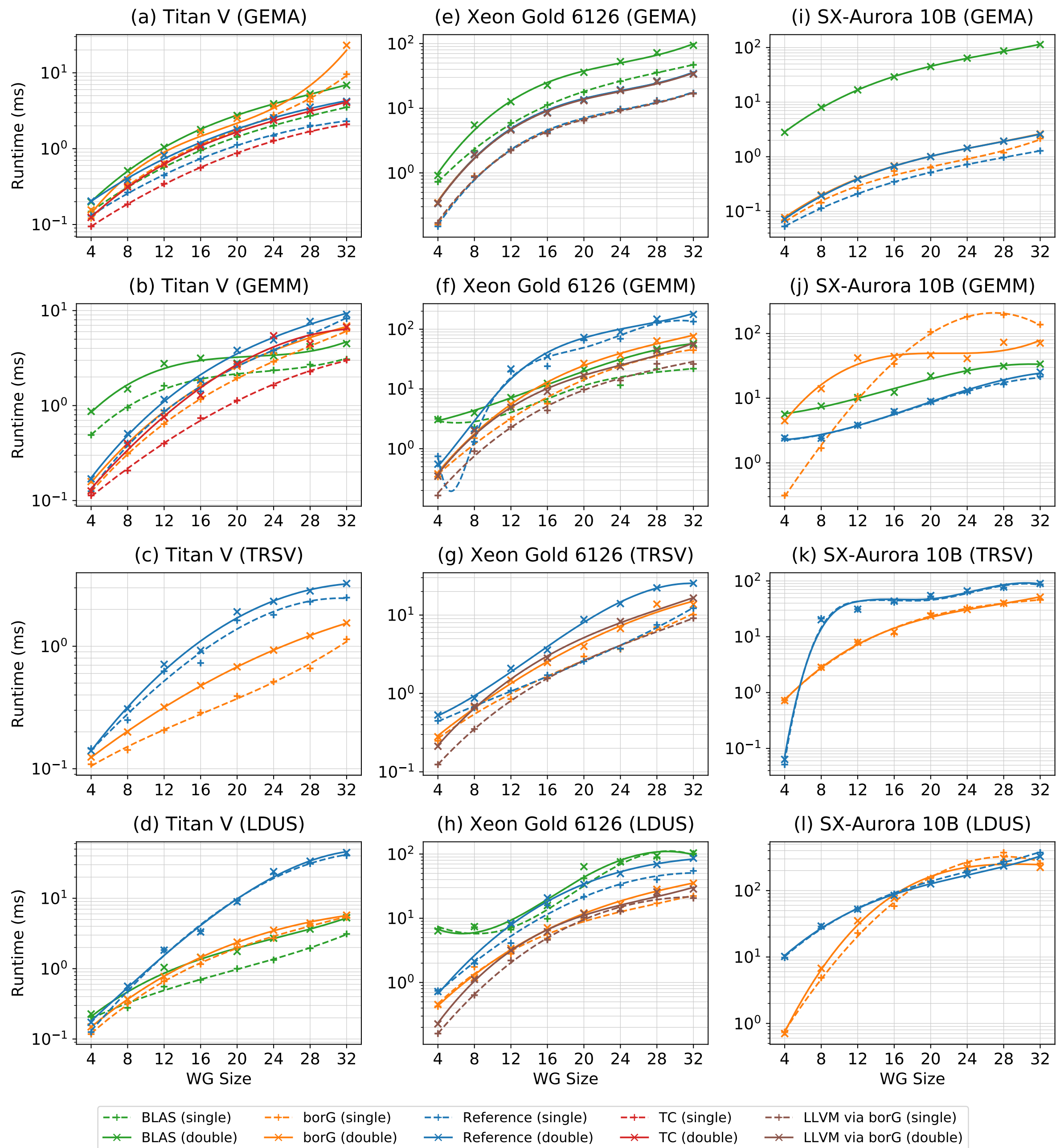


Figure 7.6: Runtimes (mind the **logarithmic scale**) of the 4 "group-type" benchmark kernels, with one kernel per row and one device per column. We plot both single and double precision separately. For each `WG_SIZE`, we pick the `PG_PACK` leading to the lowest runtime. Table 7.2 summarizes some of this data. Overall, borG compares favorably to native compilers and vendor libraries, reaching mean speedups up to 3.31x for CUDA and 3.82x for the SIMD backends.

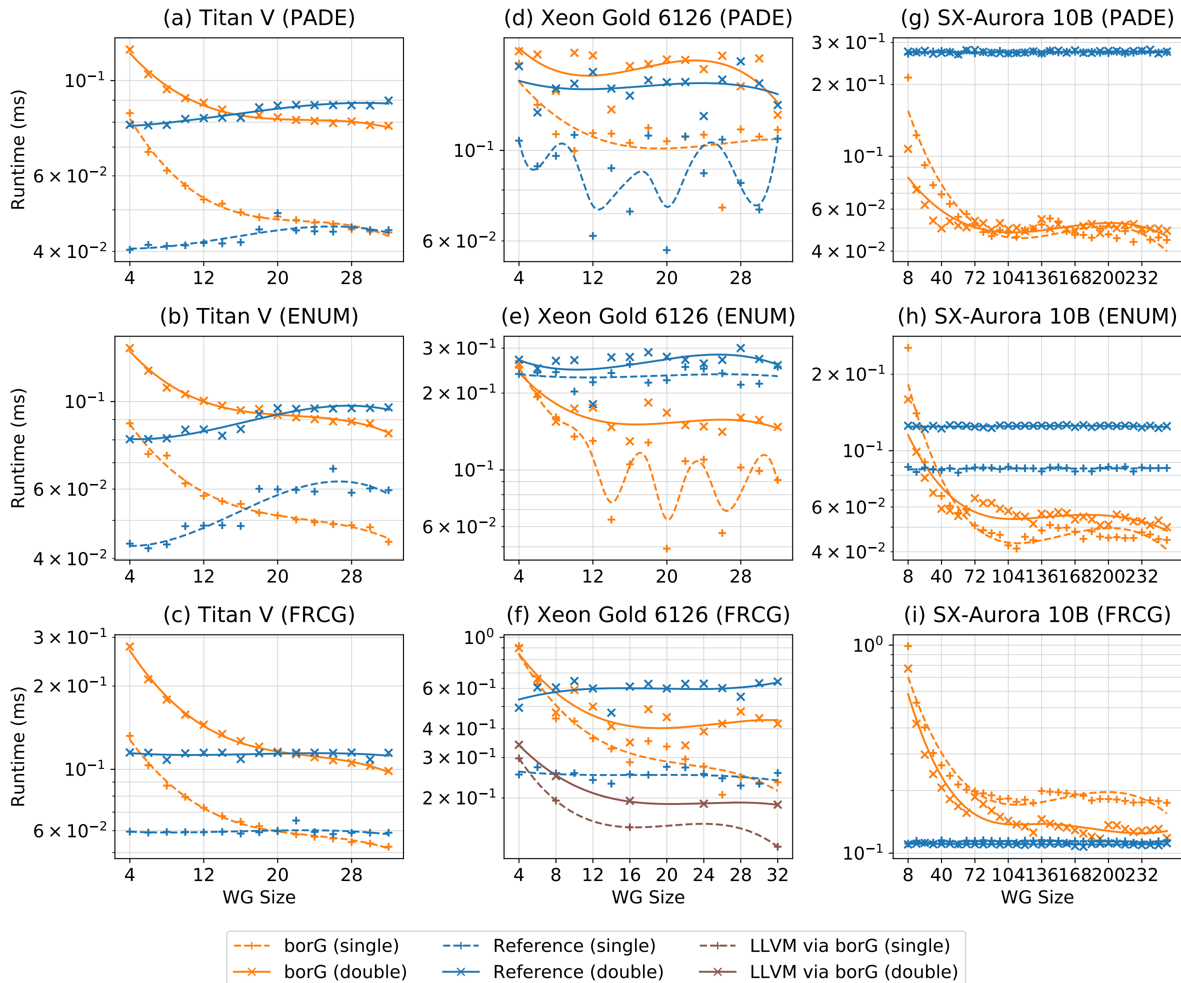


Figure 7.7: Runtimes (mind the **logarithmic scale**) of the 3 "item-type" benchmark kernels, following the same methodology as Figure 7.6. borG's kernels can keep up with CUDA and outperform ISPC, even though both are designated for this type of parallelization.

$3.31x$. Particularly for double precision, registers (holding 3 matrices) are the limiting factors for WG_SIZEs close to 32. For double precision, TC generates kernels that have a comparable runtime to borG, although borG is a one-shot system whereas TC starts from a higher-level description and finds better kernel variants through loop optimization and a polyhedral compiler. However, TC has a clear advantage for single precision kernels. At the moment, the reason is not clear to us; we will investigate this further as part of our future work.

On AVX512, both of our own backends and the LLVM-IR backend outperform ISPC on almost all kernels. ISPCs SPMD-on-SIMD approach, putting each kernel instance into individual register lanes versus borG's PIRCH-based approach leads to a significantly higher amount of spills, slowing down its kernels' execution time. LLVM's own backend for AVX512 outperforms our own chunked AVX512 backend; the heavy use of intrinsics in our code prevents the compiler from optimizations. Moreover, LLVM's AVX512 backend is a mature compiler, able to access the whole range of the AVX512 ISA, whereas our research compiler only offers a limited selection of intrinsics. However, since borG simplifies InfReg-IR back to LLVM IR, this reinforces our belief that such a novel IR can be useful for general purpose computing.

On SX-Aurora, GEMM is the worst case for borG: due to the wide SIMD vectors, most reordering operations fall back to the store/gather approach. Our kernel's outer product approach to GEMM roughly uses one FMA per reordering, i.e., gather operation. We note that the SX-Aurora is the only architecture where single precision kernels are slower than double precision kernels. Aurora's ISA offers packed float operations only for a subset of instructions; in all other cases, we resort to splitting a float vector into double vectors and processing both parts individually – effectively more than doubling the number of compute instructions. Moreover, the imature LLVM-VE compiler stack was unable to compile some kernel instances due to the lack of register spilling slots during register allocation. We did not find any systematic to this and it varied over LLVM-VE versions – hence, the results for VE are to be taken with a grain of salt.

For GEMM, GEMA and LDUS, we include runtimes for batched kernels from the vendors' BLAS libraries as an absolute reference point. While we have the lead in the GEMA kernels (where we use DAXPY/SAXPY and a memcopy operation to express $C = A + B$), borG's code does compare poorly on GEMM on larger matrices - a code that is classically ill-suited for the register cache approach since it needs to cache 3 full matrices, leading to massive register spills. An exception to this is CUDA, where for larger `WG_SIZE`s, the high degree of data reuse and the massive register file compensate for the ill-suited outer-product algorithm (compared to a GPU-friendly inner-product approach that would otherwise hurt SIMD performance as a result from many cross-lane operations). For kernels that benefit from a register cache, borG is able to keep up, as evidenced by LDUS. At least the AVX512 BLAS comparisons, however, should be taken with a grain of salt: lacking some batched kernels in the MKL Wang et al. [2014], we used OpenMP to parallelize BLAS kernels over the batch size. Additionally, BLAS kernels are more flexible, accepting matrices of arbitrary size, while borG specializes for one fixed size per compile run, saving a lot of overhead in the process. On the other hand, these kernels have often been hand-tuned by vendors on the assembly level.

Item-wise Kernels. The next three kernels (results in Figure 7.7) do not use cross-lane operations, making them prime examples for ISPC and CUDA kernels. Given the kernels' short runtimes, results for AVX512 benchmarks are noisy due to OS and scheduler influences. However, we can still see an edge for borG on average – it comes close or exceeds ISPC's performance. On CUDA, we are second to NVCC on smaller `WG_SIZE`s, but consistently have an (albeit small) advantage for larger `WG_SIZE`s. This trend continues on SX-Aurora – where borG catches up once the wide vectors are filled to a sufficiently degree. NCC is able to vectorize over all batch items directly, explaining its consistent performance. At this point, we point out that NCC seemingly operates with a reduced precision even with optimization level `-O3`; both in FRCG and LDUS, it lacks precision compared to all the other backends and compilers. As Figure 7.8 (f) shows, this can make a considerable difference on SX-Aurora.

7.4.2 Ablation Studies

We study effects of the crucial `WG_PACK` parameter and the optimizations in borG's SIMD backends as discussed in Section 7.3.3. The LDUS (double precision) kernel is used in both experiments. First, we plot the effects of `WG_PACK` in Figure 7.9 by varying it between `1` and `8` for all `WG_SIZE`s. For CUDA, higher packing leads to higher register pressure and thus limits the SM's ability to hide latencies. The SIMD backends show a different characteristic: As long as there are empty lanes in vector registers, packing improves performance. On AVX512, due to its shorter vector length, this tipping point comes much earlier than for SX-Aurora. Besides the higher vector length, each core of the SX-Aurora has a higher number

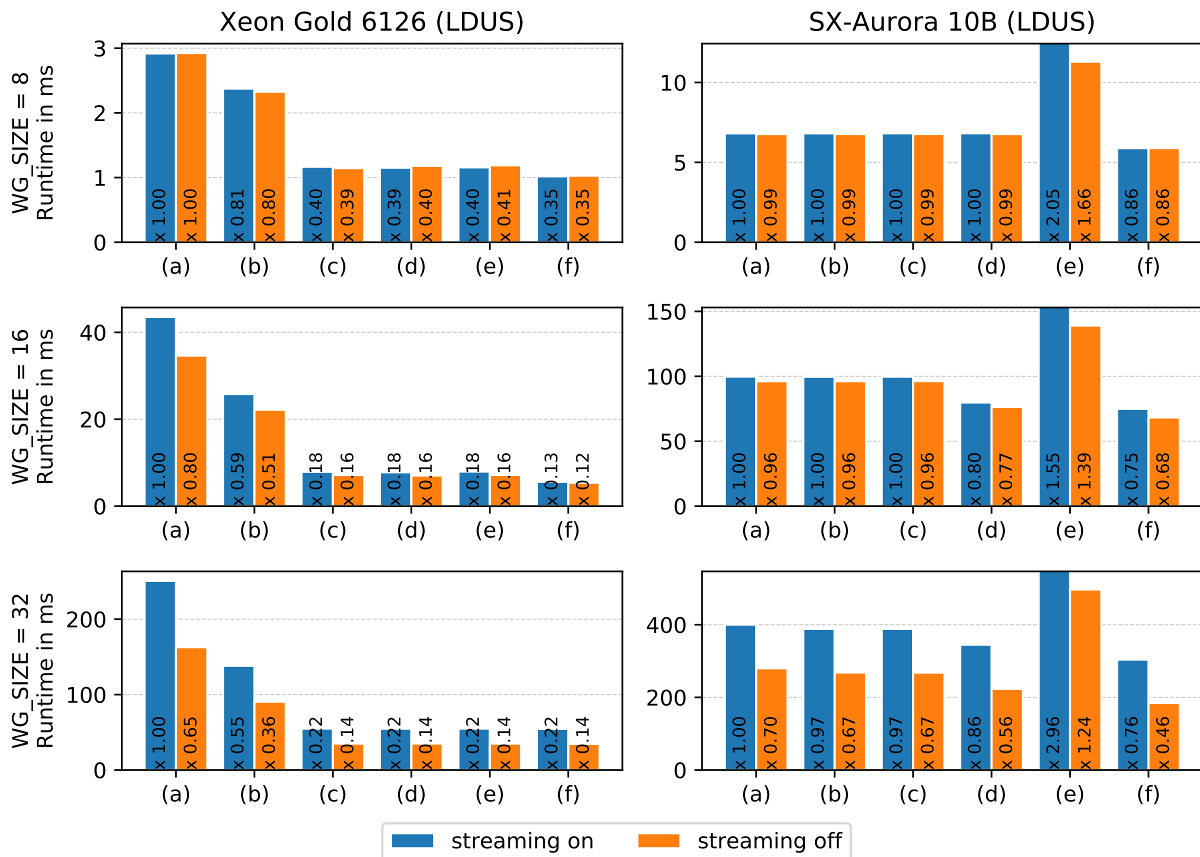


Figure 7.8: Ablation study for specialized reordering functions on the LDUS kernel. As a baseline, we use store/gather (a) for each reordering and then continue to enable more specialized functions ((b) reuse, (c) scalar broadcast, (d) vector broadcasts, (e) segmented scalar broadcasts and (f) RCP approximation for division). Except for (f), all specialized implementations improve runtimes over a store/gather approach when discovered by borG.

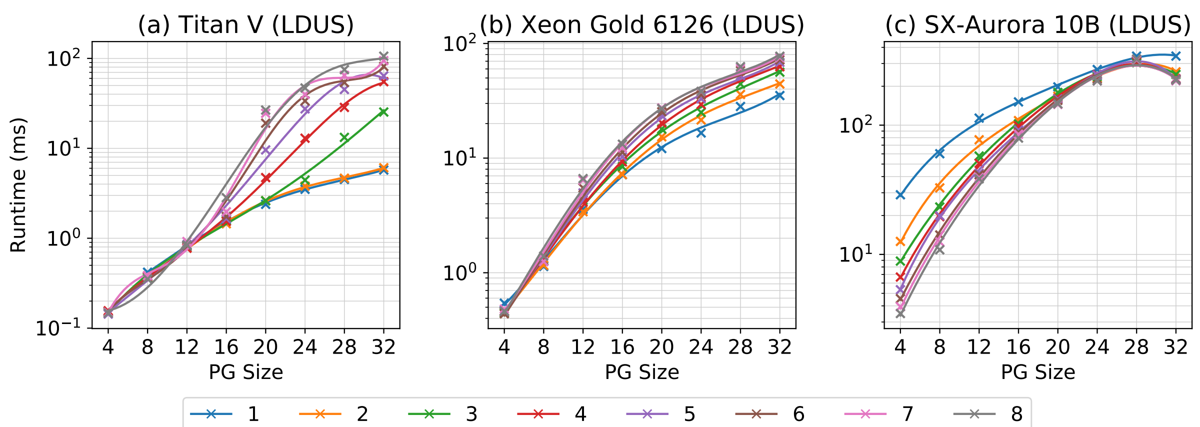


Figure 7.9: Parameter studies for the influence of the WG packing factor: while packing has an adverse effect in CUDA, the parameter proves to be crucial to fill vectors of wide SIMD systems such as SX Aurora.

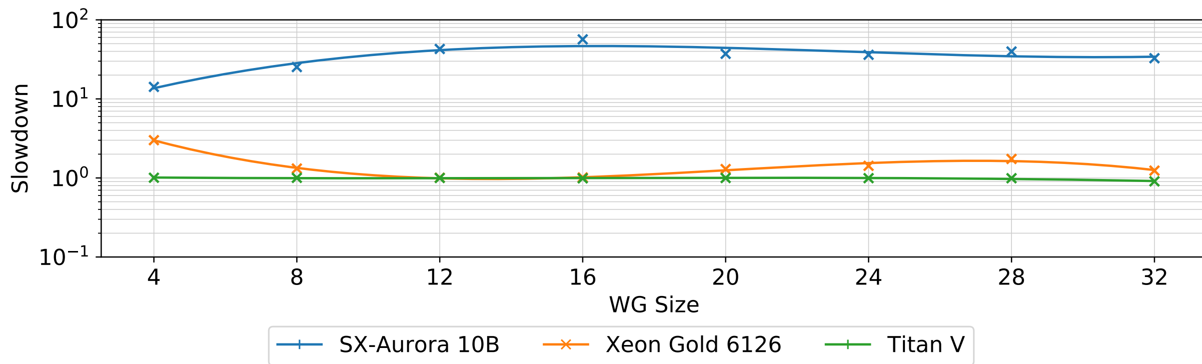


Figure 7.10: Ablation study for the effect of masking due to runtime-bounded loops. We loop over a GEMA kernel 5 times and plot the slowdown factor when we use runtime loops vs. compiler-time, unrolled loops. Note that all WIs are still executing the same code all the time, i.e., there is no divergence.

of 96 vector registers vs. 32 for AVX512. We consider `WG_PACK` to be the crucial parameter to scale up borG's kernels for both short and long SIMD systems.

Second, we perform an ablation study on the specialized reordering instructions in both SIMD backends, see Figure 7.8. Starting with configuration (a), where all cross-lane operations gathers, we enable the following optimized reorderings: (b) reuse a previous chunk, (c) scalar broadcast, (d) vector broadcast and (e) segmented scalar broadcast. Additionally, we extend (d) to use an approximation for floating point division (RCP14) as configuration (f). We also measure all configurations additionally with a deactivated register cache (streaming on), streaming all vectors from/to memory at every access, illustrating the register cache's beneficial effect on performance. The labels in the bars are the configurations' runtimes relative to configuration (a) - streaming enabled. In general, these results confirm that specialized reordering significantly improves performance over store/gather for all `WG_SIZE`s. As illustrated in Figure 7.5, depending on the chunk size and WG configuration, the same reordering pattern may be decomposed into different specialized reorderings. AVX512 has no instruction that would enable any of the logarithmic shuffle patterns; however, due to its shorter SIMD length, borG can often use scalar broadcasts instead or reuse (i.e., copy) other chunks – therefore, those two optimizations improve performance the most. For SX-Aurora, the longer SIMD width shifts the balance towards vector and segmented scalar broadcasts. While the latter's move pattern can result in a speed-up of 20%, using logarithmic shifts for segmented scalar broadcasts results in a heavy performance penalty. The documentation of the SX-Aurora's architecture does not provide an explanation for this behavior. Streaming turns out to not be of use; dealing with register oversubscription is best left to the compiler itself.

Lastly, we investigate the effects of using runtime-bound instead of unrolled loops in borG-generated kernels. For this experiment, we wrap the GEMA-kernel's addition in a for-loop, repeating it 5 times, i.e., $C = 5 \cdot (A + B)$. We compare a version ("loop") of the kernel where the loop's bounds are set through a runtime variable and one where bounds are set at compile time ("unroll") for unrolling. Figure 7.10 plots the slowdown factor for all three architectures over `WG_SIZE`s; we use `WG_PACK` $= 1$ for all builds. When encountering a runtime-bound loop, borG applies masks to all instructions in the body and requires a `popcount()` to determine when the execution of the whole WG can move beyond the loop. For group-wise kernels with their InfReg formats, creating masks involves implicit reorderings (segmented scalar broadcasts, see Figure 7.5) and cross-lane communication. Aurora's performance is hit hard (up to

30x loss) by these two requirements; in addition, every time a mask is used, the hardware copies the processed register, doubling the register requirements. On AVX512, most chunks of the reordering resolve to simple assignments and cross-lane communication is much cheaper, especially when the `WG_SIZE` matches multiples of the hardware SIMD width. As a result, we see a less staggering drop in performance (up to 2.5x). On GPUs, there is almost no noticeable slowdown: all cross-lane communication maps to fast `shuffle` calls and branches as well as predication are supported by the hardware. From this experiment, we conclude that one should prefer compile-time constant loops and minimize mask usage in order to achieve the best possible performance also on SIMD platforms.

7.5 Related Work

The warp register cache idiom has become mainstream in the CUDA community; it is especially popular for batched dense or sparse linear algebra [Anzt et al. 2017a; Anzt et al. 2017b] or deep learning [Chetlur et al. 2014]. For chemical combustion, Bauer et al. [2014] generalize the mentioned techniques to "task-parallel programming for warps".

7.5.1 Architectures and Programming Models

SIMD and SIMT are not the end of the architectural developments; researchers are exploring extensions to both. Tino et al. [2020] design and simulate a SIMT micro-architecture that includes speculative execution and out-of-order execution with register renaming. Likewise, Fung et al. [2009a] investigate the dynamic re-building of warps on the fly in case of branching code, which is beyond the current-generation SIMT model. On the software side, several warp-explicit programming models have been proposed in the past. Warp consolidation [Li et al. 2018] makes the implicit warp-centric programming explicit by a series of (manual) code transformations, which launch only one warp per CUDA block. Chen et al. [2019] draw inspiration from hardware systolic arrays and map them onto CUDA warps, which is very effective for kernels with completely static control flow that can be expressed as a sequence of unary transformations and warp reductions. Similarly, warps are used as the smallest executable units that offer the benefit of being able to alternate between SIMD and SISD phases in [Li et al. 2018]. All these models share the trait of mapping their computation to warps, minimizing global communication. However, none of these works comes with a compiler or code generator; they are merely guidelines how to implement a kernel.

7.5.2 DSLs and Code Generation

While borG is tailored towards a programming idiom, Domain-Specific Languagess (DSLs) evolved as a means to rapidly implement applications in a limited area. DSLs can either generate high-level code in a more general language or directly go to an IR level such as LLVM-IR. For batched Cholesky factorization and Kalman filters, Lemaitre et al. [2018] propose a template system. Rodrigues et al. [2018] specify a small DSL for static tensor multiplications – even parallelizing error correction in 5G base stations [Cassagne et al. 2018] warrants a DSL. Likewise, there is a DSL for stencil operations, prime examples for memory-bound kernels and the importance of minimizing memory transformations through registers, in CUDA [Zhao et al. 2019]. Code generation is not limited to DSLs, though: several approaches transform scalar and sequential into parallel code, either through an IR [Beckingsale et al. 2019] and different backends or directly through a reinforcement learning approach, that learns when and how to parallelize and chunk for-loops [Haj-Ali et al. 2020]. Ben-Nun et al. [2019] propose a dataflow multi-graph as IR that allows to split domain science and performance engineering in the development process and generates

code for different platforms. On the CPU side, Bertolacci et al. [2016] identify loop chains as a code construct that defines data sharing and parallel schedules that allow automatic OpenMP code generation. A recent system specializing on tensor kernels for deep learning is Tensor Comprehensions [Vasilache et al. 2019] which integrates polyhedral code generation, auto-tuning and a DSL for tensor expressions in Einstein notation.

7.5.3 Vectorizing Compilers

The most general approach for accelerator programming are vectorizing and parallelizing compilers. Such compilers mainly work by analyzing execution patterns within (possibly nested) loops, including function calls. [Shin et al. 2002] detects computations in a program that may be performed simultaneously in SIMD lanes, and optimizes for vector register usage in order to minimize memory traffic. The Region Vectorizer [Karrenberg 2015] is the first open source vectorizer that works on the whole-function level. After converting a whole function to an SSA based representation, execution flows are traced, masks generated and loops reordered. Its latest extension, TensorRV [Moll et al. 2019] extends its capabilities to nested, multidimensional loops It is also used as a part of a larger system PACXXv2p [Haidl et al. 2017], which – driven by a single coherent programming model resembling the data-parallel and memory abstractions of CUDA – can build native kernels for CPU and GPU. Scalar computation is wrapped into three levels of for-loops which are then vectorized treated by RV. The classical approach to vectorization is the polyhedral model for code generation [Griebl et al. 1998]. One could say that auto-vectorizing compilers offer a common representation for SIMD and SIMT as well: C code. From the same C code, they generate programs for SIMD-CPUs and GPUs. However, we argue that SIMD auto-vectorization is somewhat opaque, in contrast to PIRCH and borG, where a well-defined execution model allows the developer to stay close to the actual hardware – the transformation is very transparent.

The closest competitor to borG is ISPC [Pharr and Mark 2012]. Inspired by graphic shaders, ISPC works with a SPMD-on-SIMD model, which is similar in concept to our SIMT-on-SIMD model. There are two main differences: first, ISPC maps every kernel to one SIMD lane; e.g., a batched, 32×32 GEMM requires $3 \times 32 \times 32 = 3096$ SIMD registers, far exceeding the capacities of any SIMD processor - but only 12 vector registers on the SX-Aurora using borG. Thus, SPMD-on-SIMD is not well-suited for CoRe based kernels. Second, ISPC arranges threads in gangs, similar to our WGs, but with a fixed size that is dictated by hardware. On the other side, its longer development history and support by Intel makes it a much more general and production-oriented system that goes beyond borG's capabilities when it comes to match data structures to computation. The co-developed Sierra [Leißa et al. 2014] is a C/C++ extension that shares some ideas such as automatic blocking and conversion of data structures.

A lot of compilers for high-level languages are based on the compiler infrastructure LLVM [Lattner and Adve 2004] with its IR in SSA-form that is both hardware and language-independent, but permits the inclusion of platform-specific intrinsics. LLVM IR offers native support for fixed-length vectors of simple datatypes but, so far, lacks the support for vector features such as an active vector length (e.g., SX-Aurora) or masked instructions. Furthermore, LLVM IR does not offer a mapping from SIMD lanes to SIMT threads for, e.g., compilation for GPUs. Instead, the internal GPU support works on scalar code that is run on each thread on the SIMT system. An extension with SIMT support, especially regarding the semantics of thread divergence, was proposed [Hähnle 2019]. Over the last decade, LLVM IR has become sort of the "lingua franca" for the compiler community; it is, however, relatively low-level and lacks support

for custom extensions. In order to overcome these weaknesses, Lattner et al. [2020] propose MLIR, a framework for the definition of customized SSA-based IR dialects. MLIR allows developers to implement generalized compiler passes on the control flow graph, enabling a progressive lowering of such dialects to vanilla LLVM IR. Another IR that is used mainly in the graphics area is Khronos' SPIR Kessenich et al. [2018]. SPIR-V, its latest version, is a portable binary format that describes compute kernels and graphics shaders for GPUs and is used as an IR in popular graphics Application Programming Interfaces (APIs) OpenGL, Vulkan but also offers support for OpenCL 2.1.

7.5.4 Portability

Vectorizing compilers convert scalar code into code for vectorized computation. Since there are many platforms that can deal with vectors, a mapping from vectorized code to the platform is the next step to a final binary. Thus, another category of works target program portability, i.e., the ability to use a program on different platforms without re-implementing it. For SIMD platforms, this entails the ability to work with different hardware SIMD widths, ideally without recompiling. Vapour SIMD Nuzman et al. [2011] solves this problem by explicitly embedding the vector idioms that a compilers auto-vectorization pass detects in scalar code in a custom bytecode format. A recent proposal for an extension to LLVM [Moll 2019] also lowers arbitrary-length vectors in LLVM IR to the underlying SIMD platform at compiler time. At runtime, a Just-In-Time compiler is used to lower the bytecode to the SIMD instruction set in use, often comparable in performance to native vectorization. Liquid SIMD Clark et al. [2007] proposes a lower-level approach: mapping a baseline scalar instruction set to vector instructions at runtime in hardware. By encoding induction variables for scalar loops, a deterministic finite automaton may be synthesized for this task.

A stricter variant of program portability is performance portability, which was the goal of the OpenCL [Stone et al. 2010] standard. Du et al. [2012] evaluated its effectiveness on a GEMM kernel, concluding that it is vital to tune parameters, such as blocking or unrolling, per platform. A similar evaluation has been done for the competing OpenACC standard [Lopez et al. 2016] with similar results. Building on this portability, Steuwer et al. [2017] propose an IR for higher-level languages that encodes some of OpenCL's concepts, allowing tuning and optimization on a lower level and outputting transformed OpenCL code. The performance and versatility of OpenCL depends on the quality of drivers and compilers implemented on each platform. For a SIMT-to-SIMD style transformation, CPU-based OpenCL drivers apply ISPC's approach, sharing weaknesses and advantages. While borG inevitably shares some concepts with the referenced works, its hybrid approach, integrating a code generator for flexible WG parameterization, an abstract IR for both SIMD and SIMT representation and a cross-architecture source-to-source compiler makes it a unique approach.

7.6 Conclusion and Future Work

We observed that programs following the collective register cache idiom are good candidates for performance-portable, cross-architecture computational kernels. We therefore defined a generalization of the idiom – the collective register cache – and a virtual architecture – PIRCH – to abstract SIMD and SIMT architectures. We finally presented the borG compiler, to automatically map the programs to PIRCH, and generate optimized compute kernels for three different architectures.

The provided kernels provide similar performance to hand-tuned implementation on all three evaluated

architectures, and in some cases, borG even produced code that is comparable to vendor-tuned libraries, significantly reducing the burden for programmers in need of custom kernels. The generated kernels can immediately used inside the frameworks from Chapters 5 and 6 to create a cross-architecture sparse linear algebra package.

Beyond the current, static control flow-centric abilities of borG, we plan to extend abstractions to add more dynamic control flow and runtime-indexed shuffles. Furthermore, we would like to couple borG with stencil optimizers to address the generation of custom deep learning kernels. Based on the ideas in this chapter, we propose a hardware extension in Chapter 9 that realizes a variant of PIRCH on a slightly modified vector architecture. We plan to investigate this even further in order to drive future accelerator architectures.

With a limited control flow, our approach actually generalizes beyond the three types of accelerators that we currently support. CoRe turns out to be a superset of the systolic array paradigm in Chen et al. [2019], from which ports to novel systolic accelerators such as Google's TPUs or even FPGAs become feasible.

Chapter 8

Customized Sparse Numerical Factorizations

Contents

8.1 Related Work 113
8.2 A Factorization Meta-algorithm 116
8.3 System Architecture 119
8.4 Inclusion of Domain-specific Accelerators 126
8.5 Feasibility Studies and Discussion 131
8.6 Conclusion 135

The previous three chapters have each dealt with issues from the realm of sparse factorization individually: block accelerator-friendly global pivoting, warp register cache-based factorization kernels, Jacobi parallelization and cross-architecture kernel generation. The resulting systems, including block-iLDL$^\top$ and borG implemented proposed solutions to those issues and were able to outperform various competitors from the state of the art. In this chapter, we go beyond the individual systems and discuss a concept for their integration into a larger system which we call "METAPACK". By taking ideas from each of the previous chapters, we construct a sparse matrix factorization generator that targets the rapid synthesis of customized factorization codes supporting various types of compute accelerators.

Chapters 1 and 4 identified compute accelerators and heterogeneous systems as *the* driver for HPC performance increases during the last decade. The use of GPUs and vector processors required changes in programming models, but code from various areas was successfully accelerated. Recently, the rise of deep neural networks generated demand for compute power at lower cost. Considering that training a huge neural network can cost milions of dollars in power alone [Brown et al. 2020], power efficiency has become an important consideration. The consolidation of the community around the PyTorch [Paszke et al. 2019] and TensorFlow [Abadi et al. 2016] environments with their focus on computational building blocks such as convolution, pooling or activation functions sparked an arms race for machine-learning focused accelerators. These accelerators fall under the umbrella of "domain-specific hardware accelerators" [Dally et al. 2020] and often use very different architectures than CPUs or GPUs, e.g., systolic arrays. They offer limited (if any) programmability, but offer orders of magnitude higher efficiency and performance for supported operations. Furthermore, we see the integration of general-purpose and fixed-function computing. NVIDIA's "Tensor cores" are essentially small systolic arrays that have been added to GPU-SMs in order to accelerate GEMM-like operations.

The compute power offered by these accelerators would be a most welcome opportunity for sparse matrix factorizations. State of the art methods map sparse factorizations to a sequence of BLAS calls of which, e.g., GEMMs are what novel accelerators were build for. However, a successful integration would require vendors to provide BLAS kernels for variable matrix sizes. Due to their origins in machine learning, the accelerator's software stacks focus mostly on popular machine learning frameworks. A prime example

here is Google's TPU [He et al. 2020] which exposes its capabilities through TensorFlow. Furthermore, we require a multifrontal or supernodal frontend that is flexible enough to be combined with a new BLAS-backend. As most sparse matrix software packages are developed with a single architecture in mind, there are currently no frameworks for the rapid development of sparse factorization codes that support these novel accelerators.

As the parameter studies in Chapters 5 and 6 indicate, choosing the right solver for a sparse linear system requires careful algorithmic configurations and extensive parameter tuning. Furthermore, we currently lack the software to benchmark a sparse linear algorithm or a feature thereof in isolation. Linear algebra software packages combine different techniques and few let the user set the involved parameters (e.g., fill-in type, parallelization, dropping, ...) manually. Instead, we can only benchmark certain combinations of features in the form of a packaged implementation. Furthermore, these packages are often hand-written and tied to a certain architecture. Large codebases make it hard, if not impossible, to modify or even port the package to another architecture. As a consequence, manual effort is required to reimplement parts of an algorithm. Program synthesis approaches and domain-specific languages could present a welcome relief. They automate the process of generating highly-performant code that is tailored to a specific architecture. However, the relevant synthesis systems for linear algebra are limited to static sparse matrix patterns.

The lack of configurable solver software that can be rapidly deployed on novel hardware architecture motivated us to create the concept for METAPACK. We simplify the device-side requirements of our basic, block-sparse approach from Chapters 5 and 6 up to the point that only a handful of simple kernels are required on an accelerator. Through a refined level set scheduling code, we avoid write conflicts and arrive at a meta-algorithm for sparse factorization that covers most factorization invariants and hardware accelerators. This combines device-code generation with a reusable frontend in form of a configurable, host-side template system. The functional requirements for accelerator architectures are kept intentionally minimal, even FPGAs and systolic arrays may be employed as backends. In summary, this chapter makes the following contributions:

- We formulate a meta-algorithm for sparse matrix factorizations that combines a configurable, host-side frontend with a device-side backend.
- In order to resolve write conflicts and further simplify the device kernels, we extend the well-known level set technique by a second pass, creating conflict-free level subsets.
- An extension to borG is considered that generates pipeline schematics for OpenCL kernels with static control flow which is ideal for FPGA implementations.
- We evaluate the preprocessing strategies from Section 5.2 in conjunction with the fill-in generation in full factorizations and develop an improved strategy based on the results.

At the time of submission of this thesis, the METAPACK system is not fully implemented. Therefore, this chapter uses a prototype for experiments and assesses the potential of our concept on the basis of our experiences from the previous chapters.

8.1 Related Work

As emphasized by the three motivating examples in Chapter 2, solving sparse linear systems of equations is an important computational primitive. Since preconditioned iterative methods, have been covered in

Chapters 5 and 6 this overview over related work focuses on full factorizations (in the remainder of this chapter: just “factorizations”).

8.1.1 Implementation Techniques

Linear algebra kernels, both sparse and dense, are a part of almost every set of hardware-specific vendor libraries. Supplying a BLAS for a novel piece of hardware immediately increases its potential use cases. Consequently, there is an ample body of research on many issues on the spectrum from hand-optimized kernels to high-level program synthesis. Considerable effort is spent on improving the performance of SpGEMM and SpMV as the workhorses of sparse linear algebra kernels. Jiang et al. [2020] uses a similar reordering and blocking scheme to block-iLDL$^\top$'s BR and RB methods for SpMM. They efficiently compute row similarity through locality-sensitive hashing, but remark that within the SuiteSparse collection [Davis and Hu 2011], about a third of the matrices capture less than 1% of their nonzeros in the generated dense blocks. Still, they achieve a speedup of 19% over state of the art implementations. Interestingly, the blocked approach can also lead to improvements for dense linear algebra code. Buttari et al. [2007] propose an asynchronous block-execution model for tiled dense linear algebra, leading to a twofold increase in FLOPs for a QR factorization in a multi-core CPU setting. Beyond single-architecture systems, heterogeneous systems (e.g., CPU and GPU on a die) have the potential to accelerate these computations, particularly in the sparse case, even more. The irregular part of the computation is executed on the CPU and more regular parts are handled by the GPU. Tightly integrating CPU and GPU on a single die with shared memory and coherent caches further remove latencies from memory transfers. FinePar [Zhang et al. 2017] automatically creates an experiment-driven workload partitioning for sparse kernels like SpMV, evaluating their approach on AMD's integrated ‘Kaveri” SoCs.

The complexity of these rather specialized improvements for core computational kernels motivate the desire to find programming models and idioms that help to improve code agnostic of the application. Similar to PIRCH from the previous chapter, they too would be cross-architecture. For platforms that support asynchronous scheduling, Zuckerman et al. [2011] proposes the “codelet” model, a hierarchical system of kernel execution. In this context, a codelet is a part of a kernel. Kernels are compositional and thus allow pipelining data through their codelets, offering an additional level of parallelism beyond batch execution. Lastly, we want to mention the software-systolic array concept by Chen et al. [2019]. Inspired by hardware systolic arrays, they propose an abstraction over these arrays that. When implemented as CUDA kernels, they outperform hand-tuned, state of the art stencil kernels. This work adopts the same warp register cache idiom as borG and represents a nice bridge between novel hardware architectures and the “classical” von-Neumann world.

8.1.2 Linear Algebra Program Synthesis

Despite their relatively high level of abstraction, software-systolic arrays still require manual implementation by experts. Program synthesis represents an alternative, automating the implementation process. Starting from simple task descriptions or high-level programs, these approaches generate highly efficient low level code, possibly matching a given architecture.

Dense. As central kernels in machine learning and linear algebra, BLAS-like kernels are frequently picked for synthesis. Since BLAS and LAPACK only cover primitive operations such as GEMM, Barthels et al. [2020] proposes an automated framework to map complex computations, e.g., $(A^\top BC^{-1})$ to a sequence

of BLAS calls. Transpositions and dynamic programming-based matrix chaining are applied automatically, leading to speed-ups of over 10× over MATLAB's execution engine. SLinGen [Spampinato et al. 2018] compiler uses a similar approach, but applies it to a whole program. This scope enables to re-use partial results, even going so far as to lower parts of the program into a loop-based representation for auto-vectorization. GEMM-like operations are a frequent use case for synthesis approaches. Through hierarchic tiling, fixed-size GEMM-kernels are tailored towards a system's caches and cores. Tiling may be done on the data structure holding a dense matrix, leading to a dataflow graph [Yang et al. 2019] or on the compiler level, performing low-level optimizations on LLVM IR [Su et al. 2017].

Sparse. Indirect, data-dependent memory accesses and poor locality characterize sparse linear algebra workloads. Sparse data structures such as CSR matrices require indexing, leading to indirect memory accesses (e.g., `csr_val[csr_row][i]`). Therefore, sparse code synthesis can be regarded as an additional layer on top of dense synthesis that tries to mitigate or remove such accesses. In cases where the nonzero layout of the matrix in question is known at compile time, the Sympiler [Cheshmi et al. 2017] instances sparse algorithms, completely removing the symbolic processing stage from e.g., a Cholesky factorization. Similar to our approach, Augustine et al. [2019] tries to detect dense substructures in sparse problems. Other than e.g. block-iLDL$^\top$, they detect the existence of formalized patterns that can be substituted by closed forms in the program code. In both works, the generated code is optimized by a polyhedral compiler framework.

If we consider arbitrary sparse matrix patterns, it becomes mandatory to include runtime information into the synthesis approach. A common approach here is the inspector/executor [Saltz et al. 1990] pattern. An *inspector* program analyzes the pattern at runtime and transforms it into a data structure that obeys certain constraints (e.g., sorting of the entries) which the *executor* code can use to optimize the numerical code. An example is the parallelization of sparse matrix code through level sets (or *wavefronts*). In this case, the inspector detects wavefronts in matrix structures and the executor can exploit the guarantee that there are no dependencies between nodes in the same level set. Mohammadi et al. [2019] embed constraints like monotonicity or triangularity on index arrays directly into the polyhedral code generation framework. Symbolic logic further simplifies the inspector code, leading to inspectors that compete with hand-written symbolic analysis phases. Venkat et al. [2015] complement this work by a set of transformations that replaces the inspector by closed-form expressions for e.g., blocked matrix layouts, resulting in automatically generated SpMV codes that exceed the performance of NVIDIA's sparse matrix libraries. Pizzuti et al. [2020] express data structures in a high-level, functional language. Combined with a polyhedral framework for code generation, we arrive at a system that creates efficient code for *immutable* sparse matrices from simple descriptions of both the algorithm and data structure.

8.1.3 Sparse Matrix Factorizations Packages

A key limitation of synthesis approaches is the lack of support for *dynamic* data structures as are required for dynamic pivoting. Hence, sparse matrix factorization software is mainly still implemented by hand and tuned for a certain architecture and certain BLAS backends. Particularly intensive research went into the parallelization of sparse factorizations. Over the years, parallelism has been exploited at multiple scales, e.g., on the level of nodes in an elimination tree or on dense submatrices [Geist and Ng 1989; Liu 1986b]. Typically, parallel factorizations follow either the multifrontal or supernodal method in order to structure the computation along BLAS/LAPACK calls.

Multifrontal. Multifrontal methods use temporary, dense matrices that contain both a node's data and collect all its contributions to its ancestors. After processing a node of the elimination tree, these frontal matrices are passed to its parents. Frontal matrices are rectangular for LU decompositions [Davis and Duff 1997] and square for symmetric factorizations. A typical, production-grade Cholesky package using the multifrontal approach is CHOLMOD [Chen et al. 2008], which is also used within MATLAB. Amestoy et al. [2001] implement parallelism on three levels within the algorithm: subtrees of the elimination tree are sent to different processors where amalgamated nodes may be further split up locally to multithreading. Lastly, large frontal matrices are processed through hierarchical tiling in LAPACK. At first sight, the multifrontal method is a good candidate for heterogeneous systems, considering that messages are only exchanged along the edges of the elimination tree. However, most implementations manage frontal matrices on the CPU and decide which of these matrices are processed on the accelerator based on a cost model. In order to avoid repeated memory transfers, Hogg et al. [2016] propose to process all frontal matrices on a GPU, achieving speed-ups of up to 2× even for pivoting-intensive indefinite matrices.

Supernodal. In general, supernodal approaches do not hold a supernode's contribution to its ancestors in the elimination tree locally, but rather send out their factorized columns and rows to the ancestors. Updates are then computed at the processing site of the respective ancestor. Compared to the multifrontal approach, this allows scheduling operations such as updates with finer granularity. PARDISO [Schenk et al. 2001; Schenk et al. 2001] uses this advantage, queuing updates and subtrees for asynchronous processing on multi-core platforms. Since pivoting is restricted to supernodes, tough matrices require post-processing with iterative refinement. PARDISO supports unsymmetric LU factorizations as well. As Demmel [Demmel et al. 1999a; Demmel et al. 1999b] shows, the scheduling of tasks for unsymmetric matrices leads to processing bipartite graphs. Hogg et al. [2010] further extends the toolbox for scheduling through DAGs. Instead of all updates eminating from a node in the elimination tree as one job, they consider each individual update, resulting in a dependency DAG.

Two previous supernodal approaches share some traits with METAPACK: Gilbert and Schreiber [1992] propose to overlay a regular grid over a sparse matrix and map the resulting mesh of dense blocks to a connection machine, an early spatial architecture of a distributed, mesh-like computer. More recently, PaStiX [Hénon et al. 2002] divides matrices into block-columns and centers their scheduling around this abstraction. In order to regularize workloads, block-columns are locally split. PaStiX operates on an SMP system and assigns block-columns to individual processors. However, it is limited to Cholesky factorizations without pivoting.

Of the mentioned packages, only CHOLMOD [Chen et al. 2008], SSIDS [Hogg et al. 2016] and MUMPS [Amestoy et al. 2001] support GPUs as accelerators.

8.2 A Factorization Meta-algorithm

As is evident from the related work, there is a lack of highly configurable, cross-platform software for sparse factorization. Adapting or even implementing state of the art approaches such as the multifrontal method require expertise both in the sparse (or symbolic) and the dense part, considering the difficulty of tuning BLAS kernels for a platform. For the realization of such a system, we pursue the following two design goals.

8.2.1 Goal #1: Degrees of Freedom

Determining the best solver for a sparse linear system given a set of constraints (e.g., power, runtime, ...) is an open problem in research. In the last years, data-driven approaches have shown some progress on that front [Yeom et al. 2016; Bhowmick et al. 2006; Götz and Anzt 2018] but as supervised techniques, their success crucially depends on the availability of training data. To separate the algorithm from the implementation, we propose to follow the motto “caeteris paribus” and only change one of the two. Particularly, we suggest using a single software package that is configurable enough to cover a large range of (in-) complete sparse factorization variants. Therefore, we suggest to expose the following algorithmic parameters to the user:

- Type of factorization: $LDU, LL^\top$ or $LDL^\top$;
- Block format: fixed-size dense blocks or compressed formats with custom load and store procedures;
- Fill-in: either perform a full factorization or add level-based fill-in as in block-iLDL$^\top$;
- Parallelism scheme: use level set scheduling or a Jacobi iteration as in Chapter 6;
- Pivoting schemes: use inner, outer or a combination of pivoting routines as in Chapter 6 and
- Numerical thresholds for small pivot replacement or the drop tolerance for fill in.

Additionally, several preprocessing options (see Sections 3.3 and 5.2) should be included in the consideration. All of these aspects have already been included in block-iLDL$^\top$ and the frontend in Chapter 6. Therefore, we use combination of these ideas to build METAPACK. We are aware that the flexibility may come with a performance penalty compared to highly-integrated sparse packages which have received man-years of development. We leave this point for future work and the evaluation of a pending implementation of METAPACK.

8.2.2 Goal #2: Portability

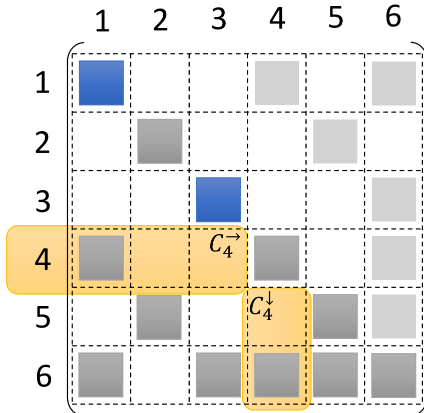
METAPACK is supposed to be more than a sparse solver, it is a *framework* for the *generation* of sparse solver software. We propose to carefully design the operations that are performed on the accelerator to be as simple as possible, ideally restricting them to static control flow, an ideal target for borG. Thanks to the performance portability of such kernels across architectures, manually tuning BLAS kernels for variable input sizes becomes unnecessary. Furthermore, we abstract over the different kinds of parallelism offered by accelerators (see Figure 8.5) by submitting only batched jobs. In particular, these jobs require no synchronization within a batch, just between multiple batches.

Following these design goals leads to a package that separates the device-side batched, regular kernels from the irregular, host-side code. The latter can be reused over different systems and combined with device-specific backends.

8.2.3 The METAPACK Algorithm

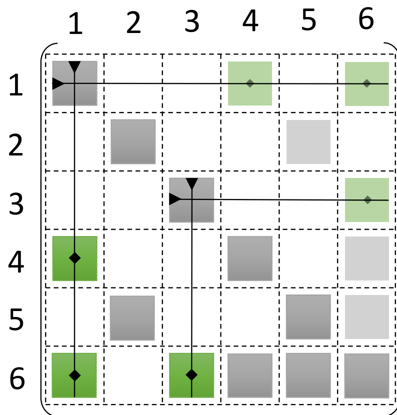
We base METAPACK on the indexed blocked CSR (BCSR) data structure from Chapter 6. Other than the left-right looking method derived from Equation (5.1), we settle on a purely right-looking variant for METAPACK. The left-looking leads to update kernels that use runtime loops, similar to Algorithm 9. In the context of borG, the ablation study in Figure 7.10 showed heavy slowdowns on some architectures due to runtime loops. Instead, we circumvent the issue by a change to the conventional level set scheduling.

(a) Phase I: Dense $LDL^\top$ factorization of pivot blocks



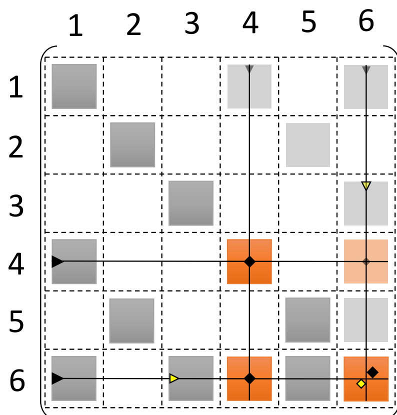
procedure ldlt_phase_i(A) ▷ "Factorize"
  for $p \in S$ **do**
    $A_{pp} \leftarrow L_{pp} D_{pp} L_{pp}^\top = A_{pp}$
  end for
end procedure

(b) Phase II: Triangular solve with respective pivot blocks



procedure ldlt_phase_ii(A) ▷ "Solve"
  for $p \in S$ **do**
    for $i \in C_p$ **do**
      $A_{ip} \leftarrow A_{ip} L_{pp}^{-\top} D_{pp}^{-1}$
    end for
  end for
end procedure

(c) Phase III: Schur downdate (note the 2 consecutive updates in block $(6, 6)$!) - the block products $ADA^\top$ are often called "contributions"



procedure ldlt_phase_iii(A) ▷ "Schur downdate"
  for $p \in S$ **do**
    for $i \in C_p^{\downarrow}$ **do**
      $A_{ii} \leftarrow A_{ii} - A_{ip} D_{pp} A_{ip}^\top$
      for $f \in C_p^{\downarrow}, j > i$ **do**
        $A_{ij} \leftarrow A_{ij} - A_{ip} D_{pp} A_{jp}^\top$ **if** $(i, j) \in \mathcal{L}$
      end for
    end for
  end for
end procedure

Figure 8.1: A blocked factorization executes three phases, each with a different operation, for each pivot p in the current level set $\mathcal{S}$ (see Algorithm 12). While the each phase must be completed before entering the next phase, all block operations inside a phase may be batched. This is, of course, unless multiple downdates are applied to the same block. for instance, this is the case for block $(6, 6)$ in **(c)**. $\mathcal{L}$ denotes the nonzero block-pattern of the factor L. For symmetric matrices, the upper triangular part is usually omitted, though we draw it here to convey an idea of how unsymmetric matrices would be handled.

We recap the three basic steps when executing a block-sparse $LDL^\top$ factorization in Figure 8.1. As in block-iLDL$^\top$, we parallelize in two dimensions. On the outer level, all pivots inside a level set may be processed simultaneously. Additionally, within a level set, all block-operations can be treated as batched operations. As in Chapter 6, we allow global pivoting by permutation operations in the BCSR structure. Within one level set, pivot blocks which encounter a small enough pivot entry during their dense factorization are considered as “rejected pivots”. Instead of permuting them to the rear of the matrix, we skip all block operations within the current level set that depend on this pivot. Only after all level set have been fully factorized, we collect all rejected pivots and push them to the rear in a single phase. Bollhöfer and Saad [2006] use this pattern as well, calling it “multilevel”. In order to avoid confusion, we use the term “stage” instead, leading to the “multistage” factorization. Thus, each stage of the factorization potentially includes multiple level sets which in turn contain multiple block-columns. In the absence of pivoting, a matrix factorization only uses a single stage. With dynamic pivoting, we do not know the number of the stages ahead of time. We note that this strategy is known as “delayed pivoting”, first proposed by Duff and Reid [1983] and used for a GPU-based factorization by Hogg et al. [2016]. Combining all of the above yields Algorithm 12, a meta-algorithm for numerical factorizations that offers enough flexibility to fulfill our design goals.

Algorithm 12 Meta-algorithm for sparse numerical factorization. Any concrete assignment for all parameters (marked by $< \dots >$) yields a sparse factorization algorithm, covering both incomplete and full factorizations of different kinds.

1: Set up the initial <block type> structure
2: `stage` $\leftarrow 0$
3: **while** `stage` $<$ `num_stages` **do**
4: $\quad P \leftarrow \emptyset$
5: $\quad$ Compute <fill-in> blocks, merge into block structure and apply deferred Schur downdates
6: $\quad$ Discover level sets S (Figure 8.3a right)
7: $\quad$ <Discover level subsets $S' \subseteq S$ through 2-hop-coloring on a bipartite subgraph (Figure 8.3b)>
8: $\quad$ **for** $T \in S'$ **do**
9: $\qquad$ Schedule <factorize kernel> calls for pivot blocks (Phase I in Figure 8.1a)
10: $\qquad$ <Permute block-rows and column according to the factorized pivotes>
11: $\qquad$ Schedule <solve kernel> calls (Phase II in Figure 8.1b)
12: $\qquad$ <Drop fill-in according to treshold ρ in all blocks of phase II>
13: $\qquad$ Schedule <Schur downdate kernel> calls (Phase III in Figure 8.1c)
14: $\quad$ **end for**
15: $\quad P \leftarrow \{$Rejected pivots in $S\}$
16: $\quad$ Create new stage from P and permute to rear as in Figure 8.2
17: $\quad$ `stage` $\leftarrow$ `stage` $+ 1$, `num_stages` $\leftarrow$ `num_stages` $+ 1$
18: **end while**

8.3 System Architecture

In the following sections, we motivate and discuss the details of Meta-Algorithm 12 and propose possible insertions for all the meta-parameters enclosed with $< \dots >$. When referring to line X within the meta-algorithm, we use the shorthand “(ll. X)”.

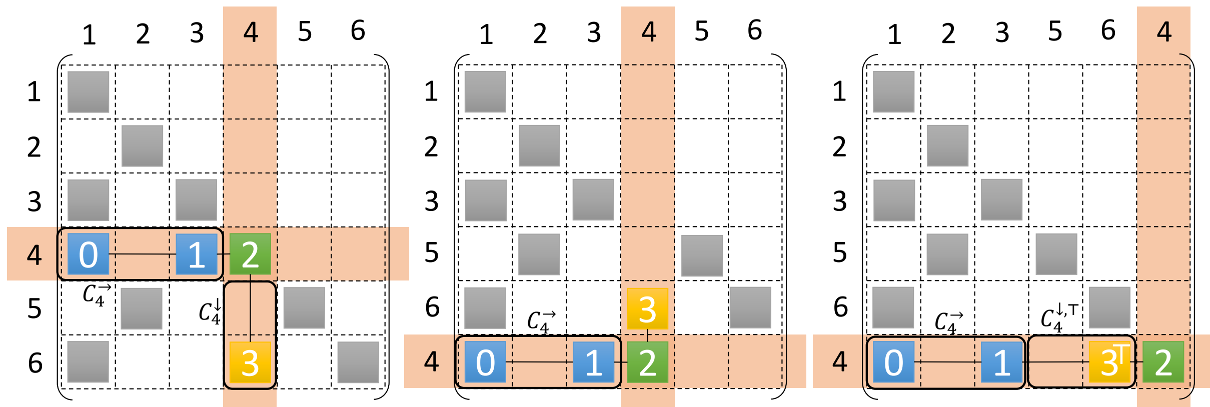


Figure 8.2: We use a delayed pivoting strategy similar to multilevel or multistage preconditioners. Assume that the pivot block A_{44} of the left matrix (only lower triangular part drawn for convenience) exceeds the permitted error within its block factorization. In this case, the operation is rejected an all operations involving blocks in the reddish shaded area are skipped. After completion, frame 4 is permuted to the end. This is equivalent to first pushing row 4, then column 4 to the end of the matrix. This operation leads to the matrix on the right hand side. Note that block 3 is transposed in the process.

8.3.1 Indexed BCSR and Pivoting

As basis for our meta-algorithm, we use the block-sparse structure from Chapter 6 which we denote by *iBCSR*. In favor of a simpler notation, we consider only the case of symmetric matrices where sets R, C from Section 6.4 simplify to

$$
\begin{aligned}
C_i &= C_i^{\rightarrow} \cup \{i\} \cup C_i^{\downarrow}, \text{ where} \\
C_i^{\rightarrow} &= \{j \mid (i,j) \in \mathcal{L}\} \\
C_i^{\downarrow} &= \{j \mid (j,i) \in \mathcal{L}\}.
\end{aligned}
$$

For a pivot i (i.e., block (i,i)), $C_i^{\rightarrow}$ lists all nonzero blocks in row i left of the pivot and $C_i^{\downarrow}$ all nonzero blocks in column i below the pivot in the block-pattern $\mathcal{L}$ of the factor. Both sets are visualized for pivot 4 in Figure 8.1a as yellow shapes.

So far, the proposed data structure is limited to static fill-in, i.e., only blocks that were generated before the first pivoting operation. For full factorizations, we extend the data structure to support the generation of new blocks within the matrix (cf. below). We keep indexing data on the host, but hold the actual numerical data for the blocks in device memory. The index itself contains the current coordinate, i.e., row in $C^{\downarrow}$ and column in $C^{\rightarrow}$, as well as the device pointer to the blocks' data. Every array $C_{\cdot}$ is sorted by the coordinate. Furthermore, we allow the user to allocate additional buffers to use within the factorizations. These buffers could hold, e.g., pivot block permutations for inner pivoting and also act as a data exchange between kernels. Each kernel call has access to all these buffers in Structure of Arrays (SoA) format.

Section 6.4 contains an algorithm for outer pivoting. In many multifrontal codes, pivots that fail a certain test against a threshold are pushed to the rear of the frontal matrix, possibly even to the rear of a ancestor's frontal matrix. This strategy is commonly knows as "threshold-delayed pivoting" [Duff and Reid 1983] and is considered to be as powerful as partial pivoting [Hogg et al. 2016]. Since frontal matrices are dense, these permutations are simple. In our case, every pivoting operation requires a modification

to the block structure, leading to a synchronization code between host and device. To avoid this, we propose the following strategy: While level sets are factorized, any pivot that fails the threshold test is marked as “rejected”. Then, all block-operations that involve a block from a rejected row or column are skipped or, alternatively, their results are not committed to memory. We give an example in Figure 8.2 (left): pivot block $(4, 4)$ fails the threshold test and consequently, all operations involving blocks within the reddish area are ignored.

Only after all level sets of a stage have been fully processed, we perform the actual modification to the iBCSR in Figure 8.2 (center and right). The black frames in the rightmost matrix there reveal that on an indexing level, the permutation operation only performs two modifications to C_4: replace every element $i \in C_4$ by $P^{-1}(i)$ and reorder the subsets to $C_i = C_i^{\rightarrow} \cup C_i^{\downarrow,\top} \cup \{P^{-1}(i)\}$. For symmetric matrices, blocks in $C_i^{\downarrow}$ mandate transposition (denoted by $\top$). The set of rejected rows and columns thus form the next stage of the factorization at the rear of the matrix.

Skipping rejected pivots delays downdates from factorized rows to the rejected blocks. Hence, after the next stage is formed, these contributions must be applied before starting with the second stage. To derive a method for these delayed contributions, we consider a permutation of the following 3×3 block matrix:

$$\begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \Rightarrow \left(\begin{array}{cc|c} L_{11} & & \\ L_{31} & L_{32} & \\ \hline L_{21} & L_{32}^{\top} & L_{22} \end{array}\right) =: \left(\begin{array}{c|c} \overline{L}_1 & \\ \hline \overline{L}_{21} & \overline{L}_2 \end{array}\right) \tag{8.1}$$

Assuming that all rejected pivots are clustered in L_{22}, this permutation follows the strategy outlined above. The resulting matix is then interpreted as the 2×2 - block matrix on the right. $\overline{L}_1$ contains the fully factorized part of the matrix (i.e., stage 1) and $\overline{L}_2$ its Schur complement, i.e., stage 2. Drilling further down, we notice that the right-looking strategy results in different stati for L_{21} and $L_{32}^{\top}$. Since L_{21} contained the parts in $C^{\rightarrow}$ of the rejected columns, all contributions there have already been applied. $L_{32}^{\top}$ represents the set $C^{\downarrow}$ of the rejected pivots which have not received any downdates so far. This mandates the following sequence of steps before we treat $\overline{L}_2$ in Meta-Algorithm 12:

1. Apply downdates from L_{21} to $L_{32}^{\top}$ and $\overline{L}_{22}$ in parallel,
2. apply downdates and execute triangular solves within $L_{32}^{\top}$ and finally
3. apply downdates from $L_{32}^{\top}$ to $\overline{L}_2^{\top}$.

Conflicting updates may be scheduled through graph coloring (cf. below). In practice, only $\overline{L}_1, \overline{L}_2$ require an iBCSR layout, for the inter-stage block $\overline{L}_{21}$, a standard BCSR layout suffices.

In summary, this pivoting strategy leads to alternating phases of iBSCR modification on the host and uninterrupted execution of block-kernels on the device (cf. below). During the latter, we additionally allow the user to do *inner* pivoting as well, which require the following specializations of Meta-Algorithm 12:

- a factorization kernel (ll. 9) that supports inner pivoting and outputs the resulting permutation into an additional buffer and
- an additional factorization step (ll. 10), permuting all rows and columns of the current level set. Alternatively, this step may be fused into the solve kernel (ll. 11).

If there are no pivoting kernels for the chosen accelerator architecture, we use Butterfly transformations

[Baboulin et al. 2014] as an alternative. At the begin of all stages after the initial stage, we simply apply a new Butterfly transformation to the rejected pivot blocks. As these transformations require only static control flow, such kernels could be created by, e.g., borG.

8.3.2 Fill-In Generation

block-iLDL$^\top$ and our modified Jacobi algorithm (Algorithm 11) both support level-based fill-in with the option of element dropping inside the blocks in (ll. 12). Since the locations of level-based fill-in blocks are determined through the algorithm by Hysom and Pothen [2002], the resulting block structure contains an upper bound of the scalar preconditioner with the same level of fill. If the meta-algorithm is to be instanced for a full factorization, we use the sequential elimination tree-based algorithm [Liu 1986a] due to its lower memory footprint.

Other than the static fill-in from before, we re-compute the location of fill-in blocks for every stage in (ll. 5). The resulting fill-in blocks are then merged into the corresponding iBCSR rows and columns. If the permutation renders some fill-in blocks from the previous layout useless, these are repurposed and zero'd out first. Otherwise, we allocate additional memory on the device and include the resulting device points into the iBCSR structure. Together with the original nonzero blocks in the permuted blocks-rows and columns, deferred updates must also be applied to the fill-in blocks. In the block factorization of the previous section, fill-in blocks for both $L_{32}^\top$ and $\overline{L}_{22}$ must be recomputed before proceeding to the second stage.

8.3.3 Level Subset Scheduling

Fostering our objective of a simplistic device kernels, we decided to use a right-looking factorization strategy. In a batched kernel setting, however, phase III of the factorization (ll. 13) may lead to write conflicts, requiring synchronization between batch items. We present an example in Figure 8.3a. Here, block $(6,6)$ receives multiple updates: tracking the black and yellow triangles along the black lines reveal that both blocks $(6,1)$ and $(6,4)$ push downdates to block $(6,6)$. The associated pivots 1 and 4 are in the same level set (see Figure 8.2 right). Thus, blindly following the level sets would result in these updates being included in the same batch of operations. Consequently, there are write conflicts for the downdates to block $(6,6)$. While left-looking approaches would use a loop inside the downdate kernel, we try to avoid any complex control structures inside kernels. Runtime-length loops can lead to divergent code and excessive masking for SIMD systems, leading to significant slowdowns, as Figure 7.10 demonstrates.

Instead, we propose to solve this issue on the host. As indicated in (ll. 7), we further partition the pivots within the same level set into conflict-free subsets. The location of potential conflicts can be determined from the algorithm for phase III in Figure 8.1c. For general matrices and sets $R_\cdot, C_\cdot$ from Chapter 6, we find:

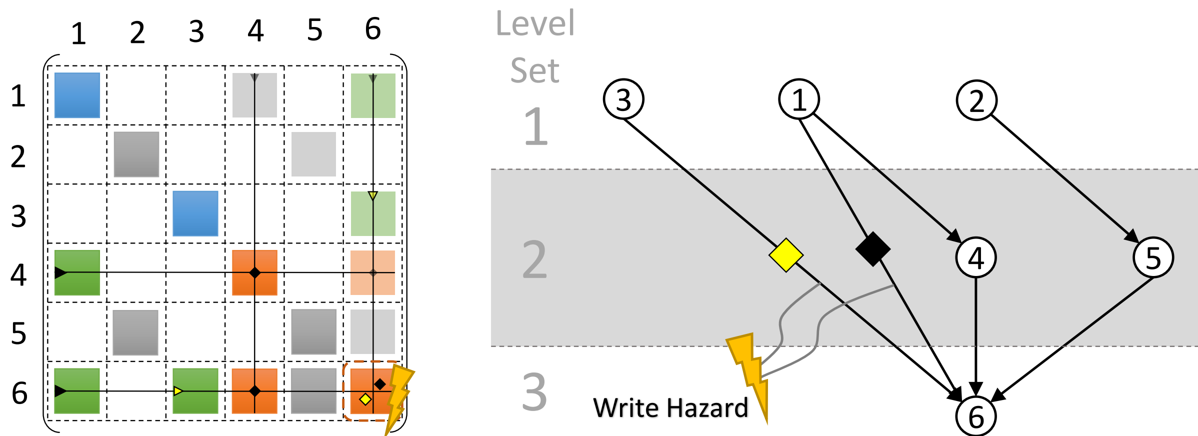
Theorem 4. *A write conflict occurs in block (i,j) of the Schur complement if and only if there are pivots $\{p,q\} \subseteq \mathcal{S}$ (ll. 6) such that*

i) $(i,p) \in C_p, (p,j) \in R_p$ *and*

ii) $(i,q) \in C_q, (q,j) \in R_q$.

A naïve implementation of a conflict resolution based on this strategy would be to symbolically compute

(a) Write conflicts occur between pivots in the level set that share an immediate successor in the adjancency matrix



(b) Write conflicts may be resolved by a distance-2 coloring of nodes in the current level set

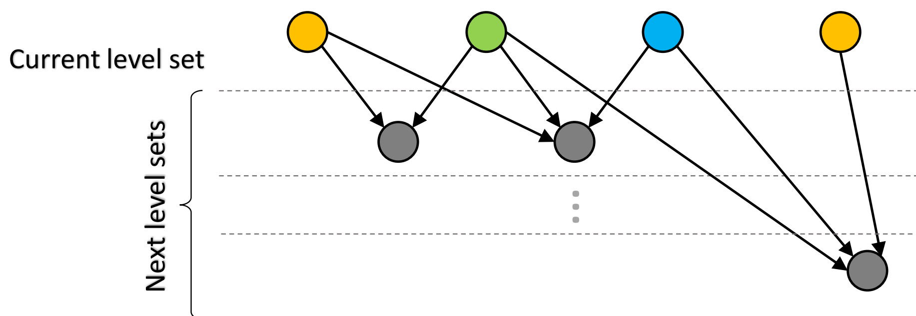


Figure 8.3: Within the same level set, some blocks in the Schur complement may receive multiple downdates that lead to write hazards (see Figure 8.1c and **(a)**). We propose to avoid these by further splitting level sets into subsets through a 2-hop coloring on a bipartite graph.

the sets of blocks affected by (ll. 13) for a level set, collect them (similar to (ll. 13)) and, if there are repeated updates, put these into multiple queues. To detect repeated updates, a hash-set like data structure is required, which hampers the parallelization of this task. Instead, we derive an alternative way that works solely on the level set DAG of L.

Symmetric case. For symmetric matrices, we have $(i, j) \in \mathcal{L} \Leftrightarrow (j, i) \in \mathcal{L}$. Assume that there is a conflict in block (i, j) between pivots p and q in the same level set. Theorem 4 tells us that there are nonzero blocks $(i, p), (p, j), (i, q), (q, j)$ in $\mathcal{L}$ respective $\mathcal{L}^\top$. Due to symmetry, the same holds for blocks $(p, i), (j, p), (q, i), (j, q)$. By using Theorem 4's other direction, we find a conflict in (i, i) resulting from blocks $(i, p), (p, i), (i, q), (q, i)$ (and (j, j)) as well. To summarize, we derived the following corollary to the theorem:

Corollary 1. *Let A be symmetric. Then, if there is a conflict between a pair of pivots p, q through a block (i, j), there are also write conflicts through (i, i) and (j, j).*

Finally, the juxtaposition to Corollary 1 leads us to an efficient subset algorithm: unless two pivots do not have nonzero blocks within the same block row, there are no conflicts. That is a necessary condition. Conveniently, the adjacency DAG of L captures exactly that information. In Figure 8.3a (right), the

conflicted blocks lies within the row and column of pivot 6 which happens to be a successor of both conflicting pivots 1 and 4. With this in mind, we fomally state the the conflict-free level subset problem:

Problem 2. *Find a partitioning S' of S such that for each $s \in S'$ no pair $i, j \in s, i \neq j$ shares a successor node in the adjacency DAG.*

A straightforward solution to this problem would be: create a graph on S, connect all pairs of nodes whose pivots share a successor in the adjacency DAG and find independent sets via graph multicoloring [Saad and Zhang 1999; Lukarski et al. 2014; Naumov et al. 2015]. Unfortunately, this method requires the formation of this graph, which can have up to $|S|(|S|-1)/2$ edges. The properties of the level set graph offer us an opportunity to represent this graph implicitly. To that end, we consider the undirected version of a subgraph of the adjacency DAG of L. It contains all pivots of the level set S as well as all their successors in the adjacency DAG. We include all edges eminating from pivots in S and their successors, but not edges between those successors. By construction, this graph is bipartite as there are neither connections within S (due to the nature of the level set) nor between the successors (by construction). Hence, graph coloring algorithms can find conflicting nodes by "jumping" over successor nodes. Furthermore, the latter nodes can be ignored in the coloring by, e.g., setting them all to a single color. Formally, this way is equivalent to computing a distance-2 coloring [Lloyd and Ramanathan 1992; Bozdağ et al. 2010] on the bipartite graph (see Figure 8.3b). Since graph coloring for general graphs is NP-hard, we use parallel heuristics as in Sistla and Nandivada [2019]. We note that, in lieu of the 2-step procedure for level subset partitioning, it is possible to apply a distance-2 coloring to the whole adjacency DAG at once. However, since this takes into account edges between successor nodes of a particular level set und cannot benefit from bipartiteness, the coloring leads to more level sets. In order to benefit from inherent parallelism, we strive to minimize the number of level sets.

General Case. Unfortunately, nonsymmetric matrices do not satisfy Corollary 1. Hence, the described procedure does not apply in the general case. In order to avoid conflict discovery through enumeration, we propose following the same method as above, but using the symmetric version $\mathcal{L} + \mathcal{U} + \mathcal{L}^\top + \mathcal{U}^\top$. When building the bipartite graph as above, this symmetric version can be represented implicitly by just considering both successors in R_p and C_p for a pivot $p \in S$. On the downside, this method may result in more level subsets than necessary, depending on how strong the nonzero patterns of $\mathcal{L}$ and $\mathcal{U}^\top$ differ.

Atomic Operations. Lastly, we point out we can skip the level subset partitioning if the accelerator supports atomic operations. In that case, conflicting downdates are prepared as batched tasks in parallel and applied via atomic subtractions to the Schur complement. In that case, developers set $S' = S$ and supply an appropriate downdate kernel. Naturally, we lose the deterministic ordering on the updates and can expect variances between factorization runs in the output.

8.3.4 Interface to Accelerators and Scheduling

We use a batched, regular computation model to abstract over possible accelerators. Similar to borG, we always submit tasks in *batches*, where each batch item is independent from all others. This simplistic model allows using hardware without any synchronization between tasks. Furthermore, we assume that job execution works synchronously and blocks the calling thread. If the accelerator's runtime allows asynchronous memory transfers, we make use of that.

Overall, we use two main processes (or threads) on the host side: the *inspector* and the *executor*. Much

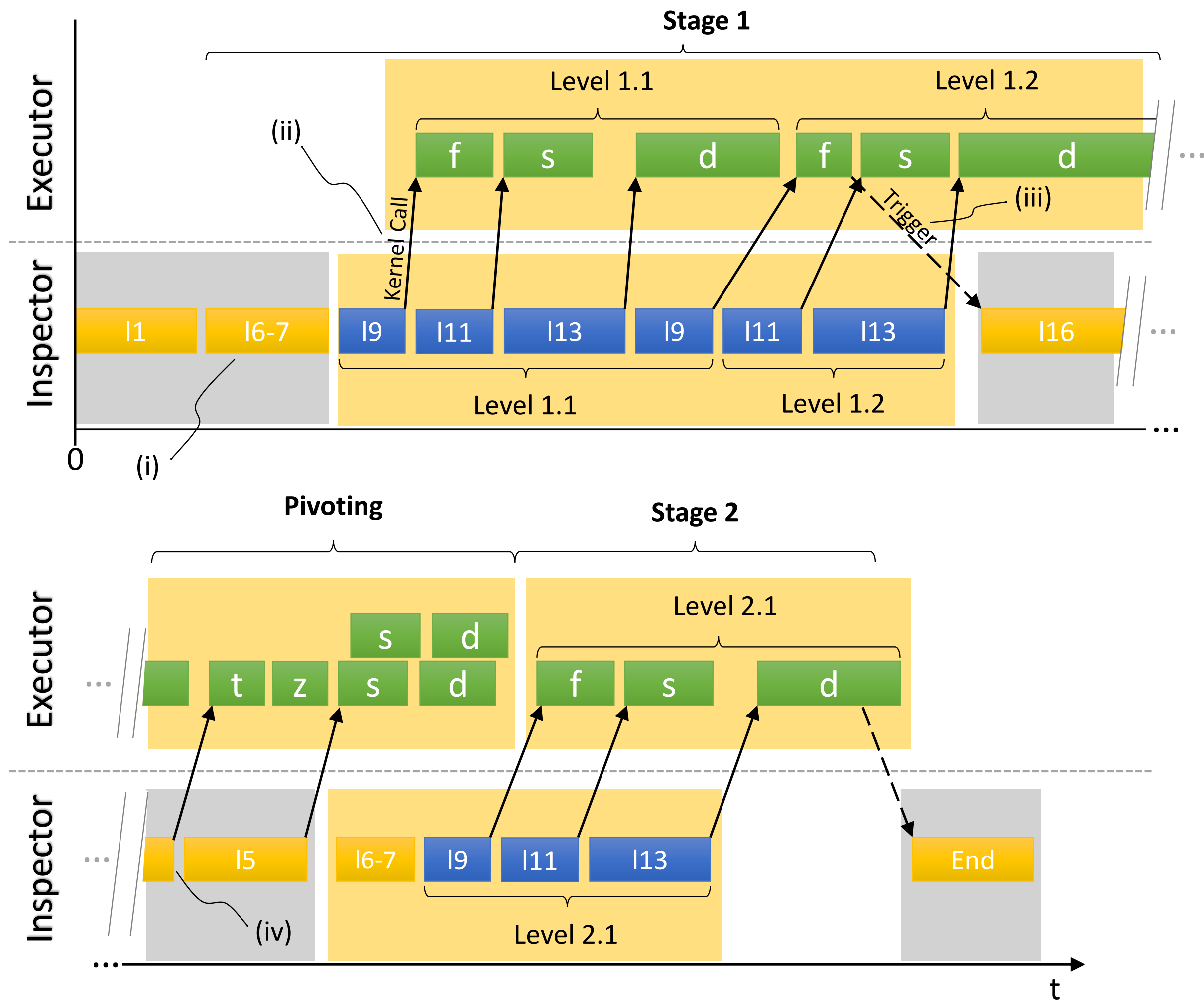


Figure 8.4: Interactions between host and device in a 2-stage factorization setting. "IX" refers to line X in Meta-Algorithm 12 and the letters in green bars represent the following kernels: **f**actorize, **s**olve, **d**owndate, **t**ranspose, **z**ero out. In our scheduling concept, we use an inspector and an executor thread in order to use pipelining. Some symbolic computation, even at the boundary between two stages, can be hidden behind numerical computations. Modifications to the iBCSR structure are shaded in gray whereas numerical computations and their scheduling are shaded in yellow.

like in sparse program synthesis, the inspector follows Meta-Algorithm 12. "Schedule" in this context means to generate a list of all block operations that must be executed in this particular step. Tuples of pointers to the interacting blocks are generated on the host and then passed to the executor, who puts the kernel call and the parameters and block pointer tuples into its queue. One by one, the executor proceeds to execute the kernel code on the device by calling an appropriate wrapper library. If possible, parameter arrays are copied to the device asynchronously.

Figure 8.4 shows an example of this communication in form of a timeline for a factorization with 2 stages. The figure highlights the following points of the scheduling approach:

(i) Initially, the iBCSR structure is set up and arrays $C^{\rightarrow}, C^{\downarrow}$ are generated. Our example assumes that the matrix' block data already resides on the device.

(ii) Once the setup has finished, we can begin listing tuples for all three phases in Figure 8.1 and store them in arrays. Level by level, this data (for the first stage only) is passed to the executor thread.

The executor thread then uploads the tuples and other parameters and calls the desired kernel on the device as soon as it is idle. Due to our decision to skip operations with rejected pivots, all levels for the whole stage can be scheduled consecutively, without synchronization.

(iii) After the last factorization kernel for a stage terminates, all rejected pivots are known. Additionally, since the remaining kernels consume tuples of pointers as input data, their execution is independent from the host's iBCSR structure. Therefore, we may already begin the permutation procedure on the iBCSR. To signal this event, the executor thread may send certain triggers to the inspector.

(iv) The delayed downdates to $L_{32}^{\top}$ and $\overline{L}_{22}$ (see above) do not influence $\overline{L}_{22}$'s iBCSR structure. While downdates are still running, the inspector may already start with the level set analysis for stage 2.

In summary, the combination of threshold pivoting and delayed updates with our inspector-executor structure results in the ability of pipelining all symbolic analysis and numerical computation steps. The only parts of the meta-algorithm that cause both a latency and a synchronization in the computation are modifications to the iBCSR structure, which should motivate us to try and avoid them by avoiding global pivoting. The critical path of our computation thus depends on both the number of stages as well as the number of level sets within each stage. Trying to optimize both closes a circle as it leads back to the study of reorderings and parallelism for block-iLDL$^{\top}$ in Section 5.2.

8.4 Inclusion of Domain-specific Accelerators

The numerical workload in METAPACK on the executor's side are multiple types of batched kernels working on fixed-sized matrices. This is similar to training neural network with mini-batches, where we execute the same operation within a computational graph simultaneously on multiple data points. Therefore, we suggest exploiting this similarity by using the recent wave of machine learning-focused domain-specific accelerators. Machine learning systems are used in many products nowadays at, e.g., Google, Amazon, Facebook. Facebook [Hazelwood et al. 2018] delivered an overview over its AI-based applications in 2018 that covers, among others, its news feed, face detection and translation service. Training was done primarily on GPUs, but also included CPUs when there were enough spare cores. However, as Dally et al. [2020] point out, loading and interpreting an instruction from memory often costs $10\times$ to $4000\times$ more energy than actually executing a simple ADD operation. Considering that Facebook re-trains some models hourly [Hazelwood et al. 2018] and that GPUs often lack in performance for inference tasks [Jouppi et al. 2017], many companies have presented dedicated hardware accelerators for this problem.

Google's TPU (Tensor processor unit, [Jouppi et al. 2020]), Amazon's AWS Inferentia [Amazon 2020] and Habana/Intel's GOYA and GAUDI [Habana 2019] architectures are tailored towards GEMM-like tasks based on systolic arrays. In particular, these involve convolutions and matrix multiplications and are often optimized for reduced precision computations with 8 bit or 16 bit datatypes. The first generation of TPUs (released in 2017) was reported to improve upon an NVIDIA's K80 (released in 2014) by $15-30\times$ in terms of performance and by $30-80\times$ in terms of energy efficiency [Jouppi et al. 2017]. Later benchmarks for the third generation of TPUs by the same authors [Jouppi et al. 2020] result in a comparable performance to a NVIDIA V100 GPU at roughly $5-10\times$ less energy per FLOP. At their core, these accelerators all use a systolic array (cf. below), as do NVIDIA's Tensor Cores.

These systolic arrays are only one instance for an architectural principle other than the classical von-Neumann "fetch-interpret-execute-store": spatial computing. Here, the computation is mapped into

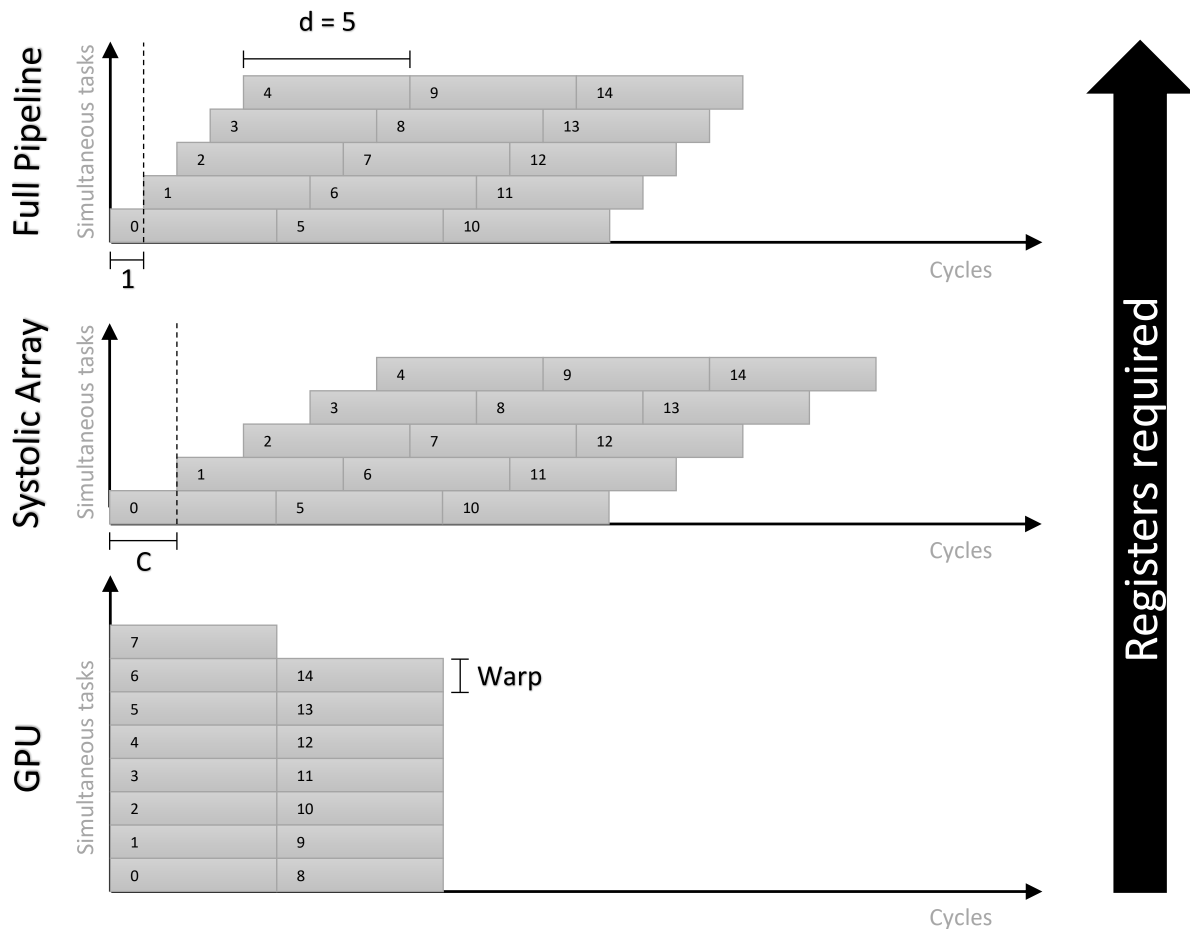


Figure 8.5: Parallelism concepts in modern accelerators: Pipelines (**top**) spatially replicate the state of the computation after each operation. Therefore, they require the most registers, but are able to input a new job every cycle. GPUs (**bottom**), on the other hand, use multiple warps to process a fixed number of jobs to termination before accepting new ones without any pipelining support from the hardware. Since a state may be modified frequently, this concept requires the fewest registers. Systolic arrays (**middle**) are somewhere between the two ends. They modify states within their execution units multiple times (i.e., reusing registers), but may use pipelining to interleave the execution of multiple jobs, depending on the algorithm's input interval C.

space and/or time, not just exclusively over time. Each step of the computation implemented through separate hardware and multiple data points can travel through these models in a pipelined manner. An educational illustration is given in Figure 8.5. From top to bottom, we show the parallelism model for archetypical parallel architectures. While the ideal pipeline has a latency of 1, meaning it can accept one new data input per cycle but needs to replicate the whole state of the computation after each step. Systolic arrays enable the re-use of operands, leading to a larger number of cycles (there: C) before the next point is consumed. GPUs, as classical von-Neumann machines, follow the well-known batch model. Architectures that do not conform to the von-Neumann pattern are colloquially called "non-von Neumann".

Apart from systolic arrays, there are also tiled processors, i.e., processors with more than 1000 cores and local storage that exchange data via a NoC. Examples are GraphCore's IPU Graphcore Ltd [2018] and Cerebras CS-1 [Cerebras Systems 2019], a wafer-scale processor. At the same time, FPGAs have resurfaced

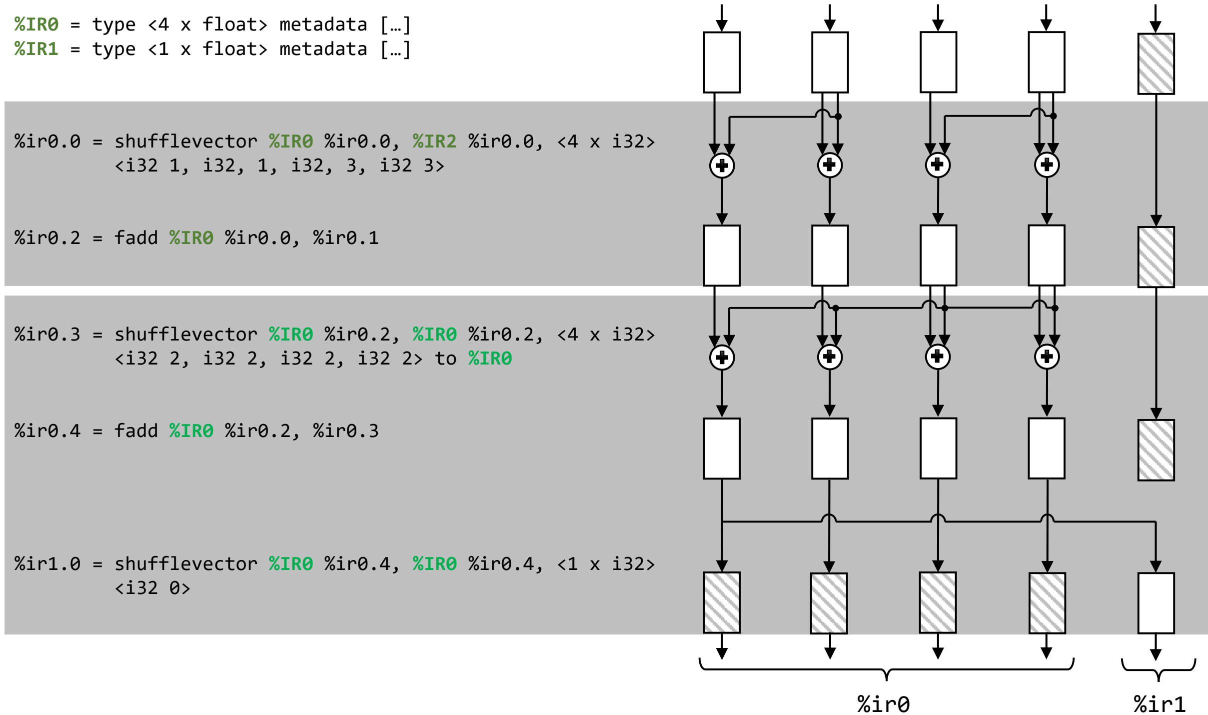


Figure 8.6: For FPGAs, we derive a pipeline concept from InfReg IR output by borG. With purely static control flow (except conditionals), all reorderings are precomputed into permutation on a SIMD vector. Concatenating all InfRegs to which are not read-only yields the *state* of the kernel. We replicate the state spatially and add functional units for computations, where shuffles are translated to wiring in the pipeline. In a subsequent optimization step, we remove the shaded registers.

an option for reconfigurable domain-specific accelerators [Microsoft Research 2010; Matteis et al. 2020; Lambert et al. 2018], fueled by progress of high-level synthesis. Recently, the industry has moved towards SoCs that combine a regular processor with multiple domain-specific accelerators or FPGAs. An example is Apples's M1 architecture, which includes a systolic array in form of the "Neural Engine" co-processor.

For a full survey of the machine learning-accelerator landscape, we refer the reader to Reuther et al. [2019] and their update in Reuther et al. [2020]. These developments now pose the following question: can we exploit these recent advances for our, non machine-learning workloads?

8.4.1 Pipelined Implementation

With the tools that we described in Chapter 7, we are able to describe a mapping from InfReg IR to a fully-functional, pipelined version for borG-kernels with purely static control flow. Figure 8.6 conveys the main idea. First. we define the state of the application as the union of all InfRegs that are modified through computation over the course of a kernel. This state is replicated for every instruction in the kernel as a set of registers. Second, each instruction is added as an execution unit between two stages. In order so save registers, we can directly merge shuffles with the succeeding computations into a single stage by setting the wiring accordingly (see Figure 8.6). Moreover, superfluous registers, such as registers that hold no useful data (shaded in), may be eliminated in an optimization pass. This algorithm allows a quick transformation from InfReg IR to, e.g., the pragma-annotated C code that may then be used for high-level synthesis, running borG kernels on an FPGA.

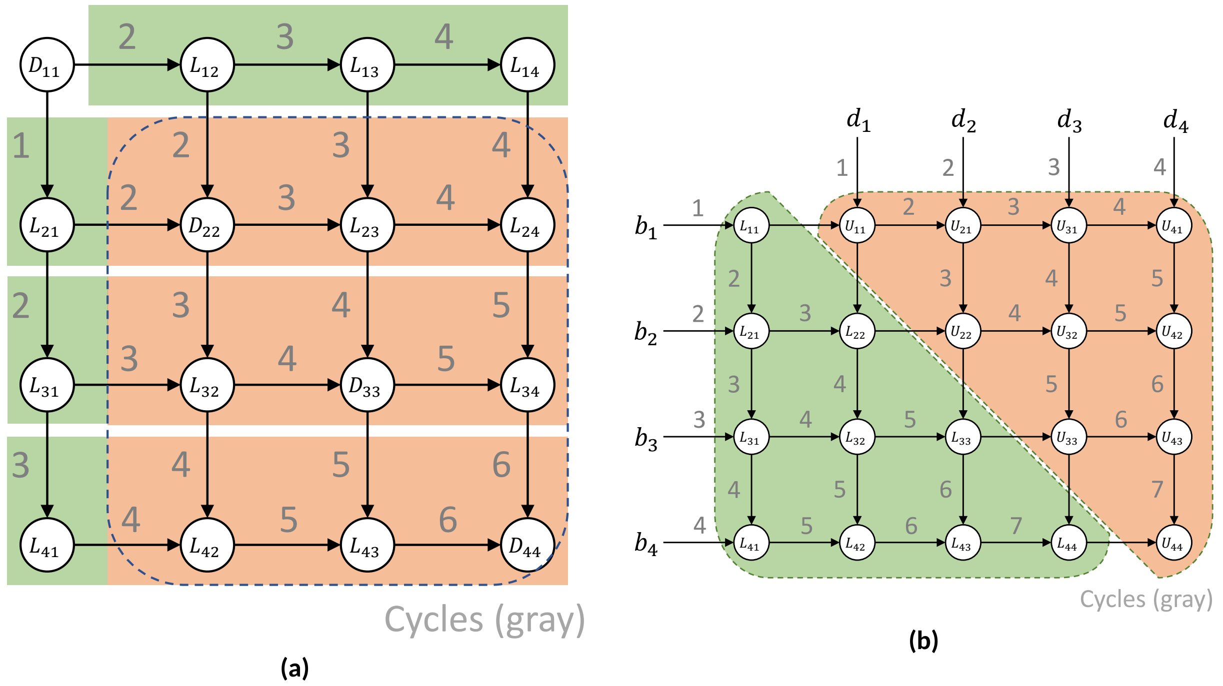


Figure 8.7: Proposed implementations of the block operations required in Figure 8.1 on a TPU-like, rectangular systolic array: **(a)** an $LDL^\top$ factorization and **(b)** a double triangular solve. Numbers in gray determine the cycle in which a connection is used for propagation.

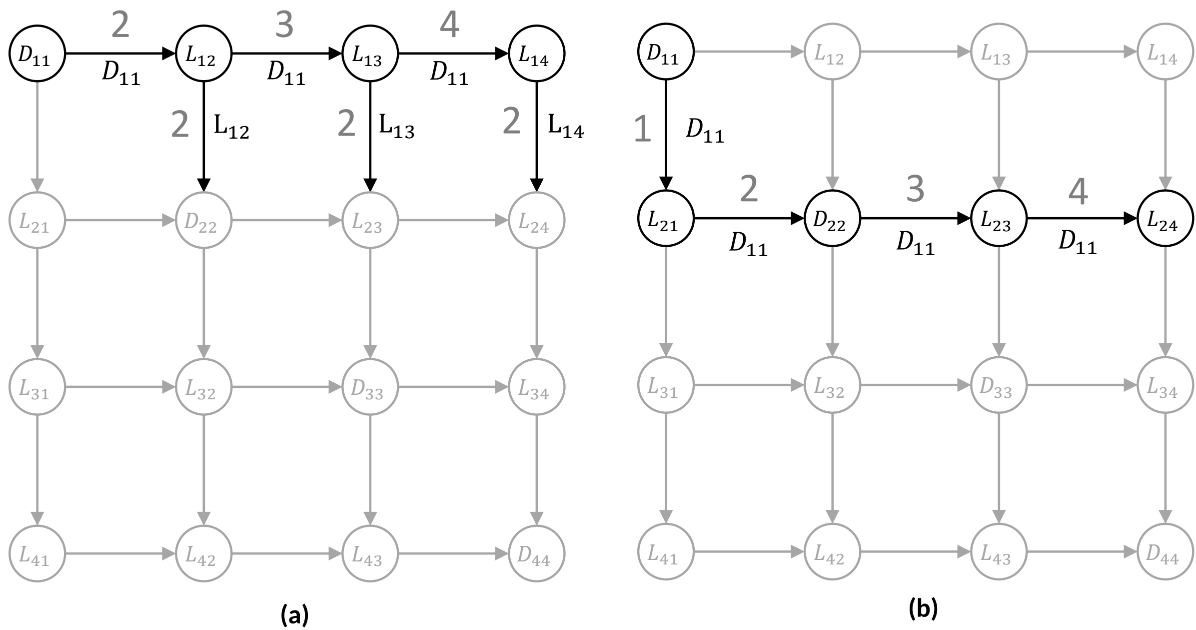


Figure 8.8: The factorization algorithm from Figure 8.7 is the result of a superposition of two phases: **(a)** passing the right factor of the Schur complement downwards and **(b)** passing the pre-multiplied factor to the right. Concatenating these phases 3 times finishes the first pivot of the $LDL^\top$ factorization. After the first phase, the next matrix may be fed in.

8.4.2 Systolic Implementations

Systolic arrays [Sun-Yuan Kung 1984] are another sort of pipelined architecture, however here, functional units are arranged spatially according to a given task. In addition, each functional unit usually performs only a single function and communicates with only a few neighbors. In Google's TPU, all systolic arrays have GEMM-compatible, square 128×128 layouts where each unit propagates its results to its right and lower neighbor. Only units at the boundary of the array can communicate with the outside world. Data is propagated between neighboring cells with a fixed clock as analogue to the heartbeat (coining the term "systolic"). Depending on the computation, a next data point may be input into the array while the current data point is propagated out of the array. Therefore, systolic arrays represent an entity between GPUs and pure pipelines (cf. above) in terms of register requirements and parallelism.

Until the resurgence of systolic arrays in the form of TPUs, implementing an algorithm required the implementation of a fully customized array structure, like Rajopadhye [1988] propose for a dense LU decomposition. In order to use already available machine learning-focused accelerators, we instead map the $LDL^\top$-factorization (without pivoting) and the triangular solve operation on the TPU's square arrays – GEMM is already included in most software development kits and libraries. Here, TRSV is the straightforward case. Figure 8.7b uses a 4×5 array to solve with 2 matrices at the same time for $Lx = b$ and $Uy = d$. We denote each unit (or "cell") with the output variable that it is holding. If we follow the flow of the data through the array, we see that the cells must execute the usual operations for TRSV:

- $L_{i1} = b_i - L_{i1}L_{11},\ i \leq n$
- $L_{ij} = L_{i(j-1)} - L_{ij}L_{jj},\ j > 1$
- $L_{ii} = L_{i(i-1)}/L_{ii}$

Thus, the right-hand side vector flows from left to right, then solution components from top to bottom. At the end, $(x_1\ x_2\ x_3\ x_4)^\top = (L_{11}\ L_{22}\ L_{33}\ L_{44})$ and y accordingly. Here, we assume that matrix L has been stored inside the array in an initialization phase.

The implementation for an $LDL^\top$ factorization is slightly more involved. For the variant in Figure 8.7a, we use the fact that for $LDL^\top$, step c) of Equation 3.18 can be re-written as:

$$L_{22}D_{22}L_{22}^\top = A_{22} - L_{21}D_{11}L_{21}^\top \tag{8.2}$$

$$= A_{22} - A_{21}D_{11}^{-1}D_{11}D_{11}^{-1}A_{21}^\top \tag{8.3}$$

$$= A_{22} - A_{21}D_{11}^{-1}A_{21}^\top \tag{8.4}$$

According to this, the factorization results from the superposition of two data flows. For once, we pass the pivot to the right neighbors of D_{11} while, at the same time, they pass their original data A_{1j} to the cells below (see Figure 8.8a). Simultaneously, the pivot is passed down to the second row and then propagated right (see Figure 8.8b). On receiving the value from the leftmost column, which is passed through to the right, each cell can compute its Schur downdate as above. Repeating the pattern two more times vertically leads to the first Schur downdate of the matrix. In the next step, we apply this idea recursively to the trailing submatrix, i.e., the reddish part of Figure 8.7a. Here, as in TRSV, the matrix A must be loaded into the cells in an initialization phase.

The systolic array also serves as the model for algorithms implemented in software. From there, the

resulting kernels may be mapped on an FPGA [Matteis et al. 2020] or a GPU [Chen et al. 2019].

8.5 Feasibility Studies and Discussion

Meta-algorithm 8.2 can be specialized into many forms of factorizations. With level-based fill, it computes preconditioners in a similar fashion to block-iLDL$^\top$ from Chapter 5. block-iLDL$^\top$'s success is therefore an indicator that, at least for incomplete factorizations, METAPACK could be an effective and competitive approach. However, it remains to be proven that its advantages extend to full factorizations[1] as well. Specifically, we have to evaluate the system against established multifrontal and supernodal implementations.

Factorizations pose two main challenges for accelerated computing: global pivoting and potentially high amounts of fill-in. Other than in the preconditioners from Chapter 6, factorizations leave no room for approximations. After each pivoting operation, the fill-in pattern changes, demanding a dynamic data structure. As outlined for iBCSR-based pivoting above, these operations lead to a stall in the computational pipeline in Figure 8.4. Hence, the critical path in a factorization is determined by both the number of stages (respectively outer pivoting operations) as well as the number of level sets within these stages. Since we limit ourselves to batched kernel calls, the number of batch items for a kernel call often contributes less to the overall runtime than the number of subsequent calls. At the same time, accelerator memory is often limited and can usually not just be increased at will. On the other hand, accelerators are used in exclusive mode, where only the kernel process is the only process on the device. Therefore, we only require to fit our problem into device memory, but it is not necessary to reduce memory requirements further than that.

From these requirements, we derive the following necessary conditions for a matrix in order to match METAPACK's concept well:

1. Minimize the number of blocks to shorten scheduling and pivoting phases by capturing as much fill as possible in as few blocks as possible.
2. Form the matrix such that the elimination tree over the matrix' block structure is shallow and wide, as this minimizes the number of level sets.
3. Minimize the number of stages through preprocessing and, e.g., cheap inner pivoting similar to in block-iLDL$^\top$.

How well our ideas from the previous chapters work with regards to the third point can only be determined once METAPACK is fully implemented. However, the first two points are the classical tasks of preprocessing a sparse matrix, minimizing fill-in and extracting parallelism – all in the context of the block decomposition. Chapter 5 compared two strategies: BR and RB. For METAPACK's regular-sized blocks, we drop their hierarchical optimal block size search. In order to satisfy the first two conditions above, any preprocessing pipeline must consider two levels of hierarchy: fill-in on a scalar level (i.e.,within the blocks) and on the block level. In the end, METAPACK's scheduler operates on a DAG on top of the block structure. Hence, we try to make the latter as shallow as possible.

Preprocessing and Fill-in Experiments. To investigate the effectiveness of both strategies in the presence of (full) fill-in, we perform the following experiment: we select a set of the 200 largest numerically

[1] in the remainder: factorizations

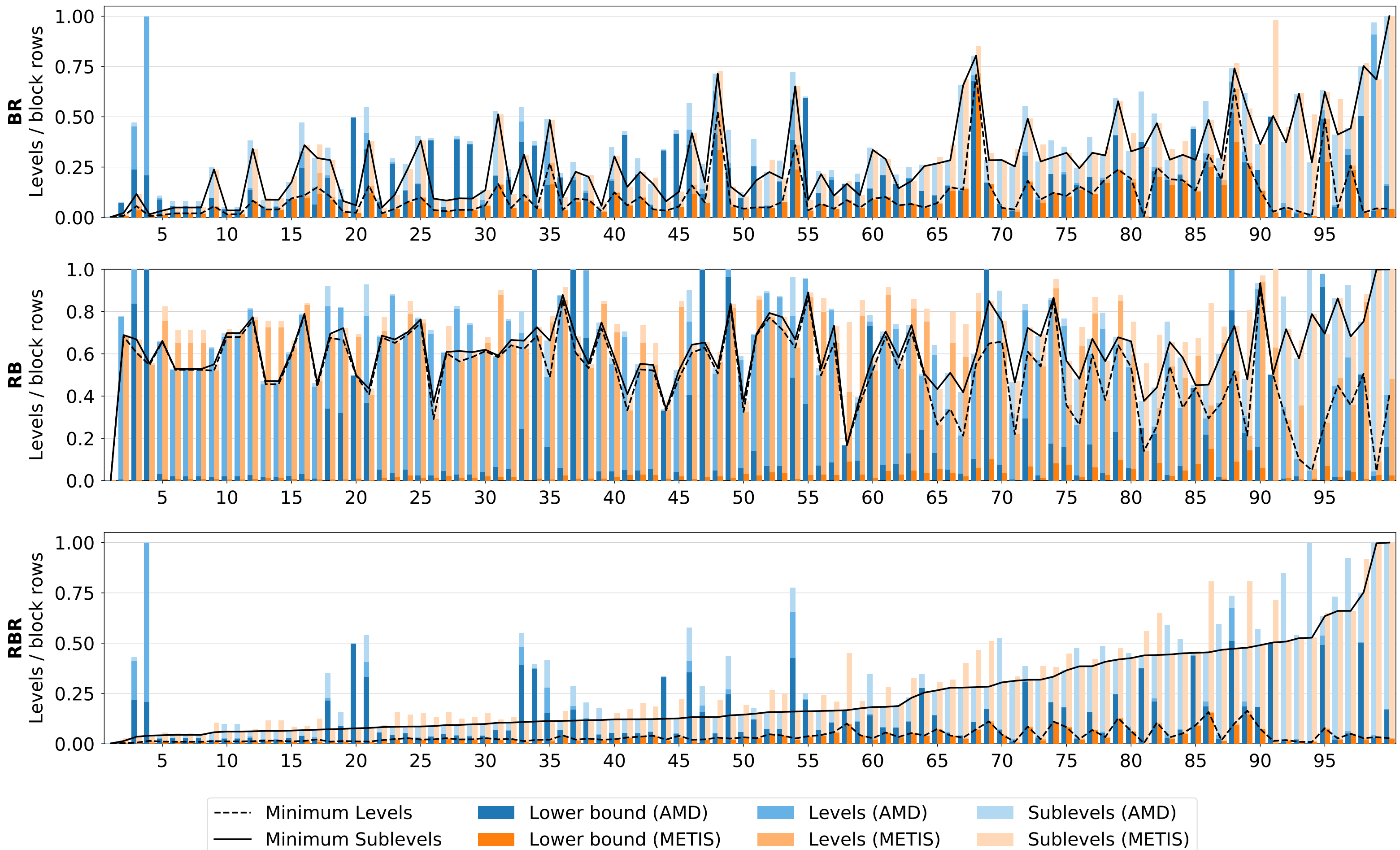


Figure 8.9: The effectiveness of METAPACK's blocked approach for full factorizations critically depends on the degree of parallelism that is extracted in preprocessing. In this study, we use the 100 largest symmetric sparse matrices from the SuiteSparse matrix collection [Davis and Hu 2011] and determine their potential for parallelism with $N = 16$ (tipping point in borG). We measure this through the number of level sets w.r.t to the number of block rows, where **lower is better**. Results are sorted by their number of sublevels with the RBR approach. This approach outperforms the others by large margins, often coming close to the "lower-bound", a supernodal factorization with arbitrary block sizes. However, the spike in sublevels compared to the number of levels raises potential issues on systems which lack atomic instructions.

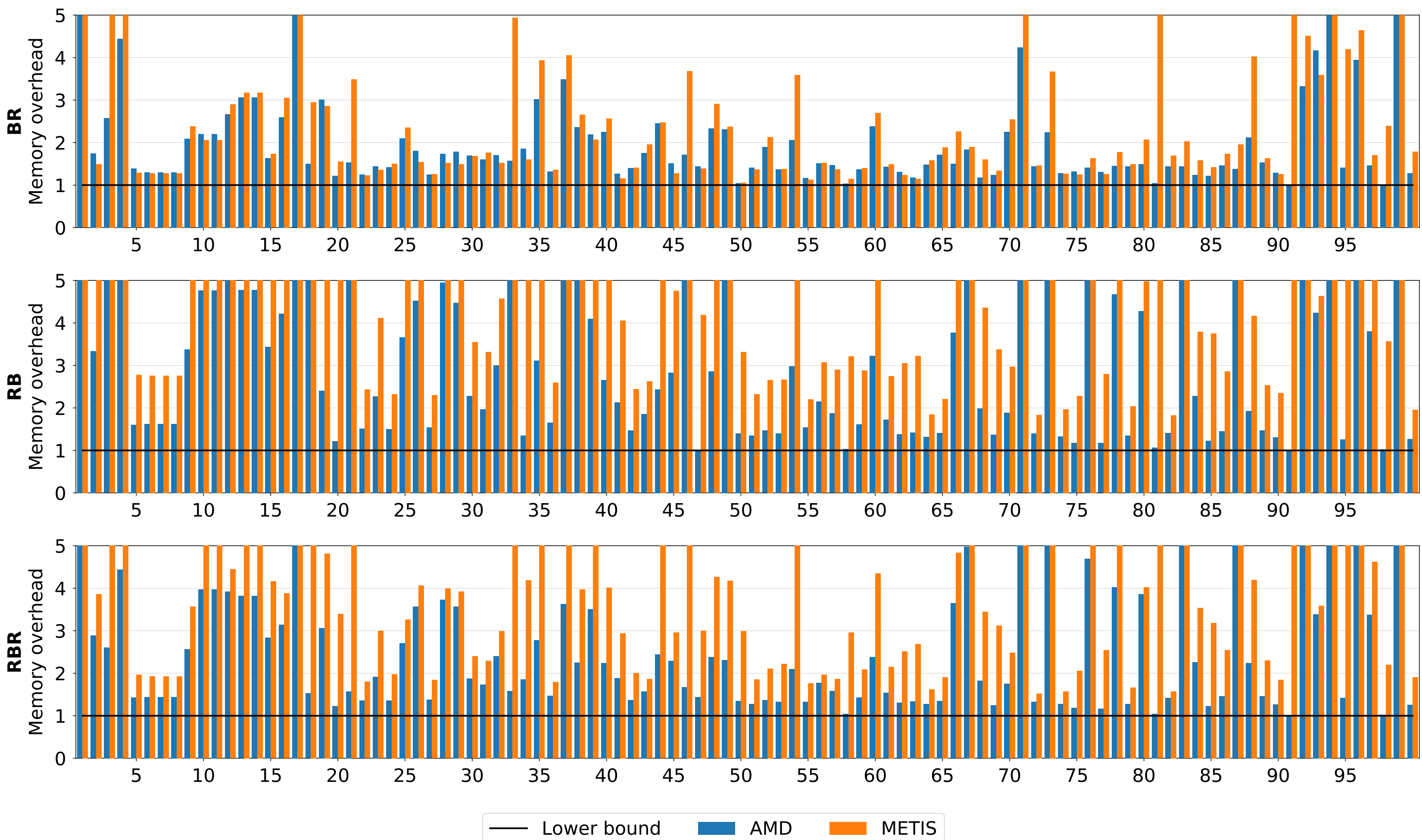


Figure 8.10: This experiment follows the setup of Figure 8.9, including the order of matrices. By storing data in dense blocks, we accumulate explicitly stored zeros as well. In this case, the lower bound yields the number of blocks that is needed to contain all scalar nonzero elements hypothetically, without any regards to matrix structure. Therefore, overheads of ~2x to 3x are satisfactory results. While BR's performance here may look good here, the absolute number of scalar fill-in often exceeds RB's by a factor of 5 or more. RBR turns out to be a compromise between fill-in and parallelism.

symmetric matrices from the SuiteSparse collection [Davis and Hu 2011] and randomly pick 100 of them for visualization purpose. On each of the matrices, we execute a preprocessing following BR and RB strategies. As reordering algorithms, we use both AMD [Amestoy et al. 2004] and METIS [Karypis 1998] and used block sizes $N \in \{4, 8, 12, 16, 20, 24, 28, 32\}$. For each permuted matrix, we log the number off nonzeros including fill-in on both the scalar and the block level. We capture the depth of the block-elimination tree, which equals the number of level sets and additionally, we record the number of sublevels due to write conflicts within level sets. As references, we derived the following lower bounds on the number of levels and the number of blocks required. For a scalar elimination tree of depth d, the block elimination tree has at least a depth of $\lceil d/N \rceil$. A scalar matrix of nz nonzero elements then hypothetically, ignoring structure, requires nz/N^2 blocks.

The first two plots in Figures 8.9 and 8.10 contain the results of these experiments. Each point on the x-axis corresponds to one matrix. The order of the matrices follows the third plots in both figures (cf below). First, Figure 8.9 visualizes the number of (sub-)levels per matrix when preprocessed with BR/RB strategy and AMD (blue) or METIS (orange) ordering as a faction of the number of block rows. The black lines connects the minima of AMD and METIS' results per matrix and help to get a quick visual "feel" for the results. The bars are read as sums where the upper limit of the lightest shade yields the number of sublevels, followed below by the number of levels and the mentioned lower bound for this matrix. Figure 8.10 augments these results with information about memory usage. We list the overhead required as the capacity of all nonzero blocks divided by the actual number of scalar nonzero elements in the reordered matrix. We note that all quantities are relative to the permuted matrix, hence each bar uses a different absolute reference. Finally, Figure 8.11 contains absolute numbers for 4 selected, representative matrices.

Discussion. We summarize Figures 8.9 and 8.10 quantitatively by geometric means for each strategy/re-ordering algorithm pair in Table 8.1. When looking at the relative values in both Figures 8.9 and 8.10, we notice a clear advantage of strategy BR over RB in both level sets and memory demands. BR-ordered matrices seem to parallelize better at lower storage costs. Even matrices in Figure 8.11 shows that the number of blocks for RB and BR are often within close proximity. The critical difference here is the number of *scalar* nonzeros. Since BR does not reorder scalar rows within block rows, there is considerable scalar fill within the blocks. As a result, BR factors often have $3\times - 5\times$ more scalar nonzeros than RB factors, which may pose a problem for the triangular solve operation after the factorizations. RB, on the other hand, does minimize the fill within blocks, but it lacks the awareness of blocks and works purely on the scalar pattern. Thus, blocks often include rows from independent subtrees of the elimination tree, leading to many more level sets. With level set ratios of over 0.6, RB disqualifies as a strategy for METAPACK's factorization, even though it results in the sparsest factors.

All experiments so far have used $N = 16$. For some representative matrices, Figure 8.11 presents absolute numbers of blocks and level sets for other block sizes N as well. Overall, they confirm the prior findings: BR minimizes blocks and levels, RB can keep up in terms of memory requirements but serializes the computation to a large extent. As one would expect it, increasing the block size uniformly leads to less nonzero blocks and levels. In summary, if we can live with a high number of scalar fill-in elements, BR is the strategy of choice. Going forward, we would ideally like to combine the fill-minimizing property of RB and the parallelizing ability of BR in a single technique.

RBR strategy. Minimizing fill is one of the jobs that reorderings such as AMD and METIS perform. The

	BR			RB			RBR		
	AMD	**METIS**	**LB**	**AMD**	**METIS**	**LB**	**AMD**	**METIS**	**LB**
Levels	0.17	0.07	0.07	0.53	0.52	0.45	0.09	0.03	0.03
Sublevels	0.29	0.21	0.20	0.64	0.67	0.57	0.22	0.20	0.16
Memory	1.82	2.18	1.0	2.75	5.14	1.0	2.24	3.65	1.0
Memory†	1.01	1.05	1.0	1.0	1.82	1.0	1.01	1.52	1.0
Memory‡	15.83	21.79	1.0	21.68	18.1	1.0	15.79	15.80	1.0

Table 8.1: Geometric means for level and sublevel ratios from Figure 8.9 and memory overhead from Figure 8.10. For the latter, we also report the minimum (†) and maximum (‡) values.

effects of the RB/BR strategy are a consequence of the level on which these reorderings are applied. In RB, they operate on the scalar matrix, minimizing scalar fill while in BR, they operate on the block level, blind to what's going on inside these blocks. Therefore, we propose to combine the two. First, we use the reordering on the scalar level. Then, after cutting blocks, the same reordering is used another time, however this time on the blocks structure, resulting in a "'RBR"' strategy. The basic idea is that each reordering run will minimize fill-in on one level, first within the blocks, then between the blocks. We repeat all prior experiments for the RBR strategy, resulting in a superior ordering: RBR beats both BR and RB on average according to Table 8.1 both in terms of levels and blocks. At the same time, it often is within a factor of 1.5× of RB's scalar number of nonzero levels. Since the second run of the reordering happens on the block level, its runtime overhead versus RB or BR is often negligible. RBR's only downside is the creation of excessive write conflicts within level sets, leading to increased numbers of sublevels. In this case, we might prefer atomic operations where possible over our 2-step scheduling or revert to the BR strategy. All in all, RBR proves to be a superior strategy, resulting levels and blocks that are so close to the lower bound (which, in turn, roughly approximates a supernodal method) that it offers the potential for METAPACK to be competitive with the established supernodal and multifrontal methods.

8.6 Conclusion

This chapter outlined the concept of METAPACK, a framework for customized sparse factorizations using various classes of compute accelerators. From a proposed factorization "meta-algorithm", we can instance both incomplete and full factorizations with various features, e.g., two-stage, delayed pivoting. METAPACK builds upon our ideas from earlier chapters: the block-based idea from Chapter 5, the inner-outer pivoting data structure from Chapter 6 and, for the backend kernels, borG as a pathway to supporting various accelerator architectures. In the process, we continued to simplify the frontend structure on the host in order to support various accelerator architectures and parallelism models, including the latest domain specific accelerators and FPGA. For the latter and systolic arrays, we suggested possible implementations for factorization kernels.

Although METAPACK is not fully implemented at the time of writing of this thesis, our experience with block-iLDL$^\top$ suggests that the concept can lead to efficient software packages. Experiments with preprocessing additionally resulted in the birth of the strategy "RBR", yielding high-quality blockings for matrices that compare to supernodal approaches. How close the full pipelines' performance come to established solvers remains to be seen.

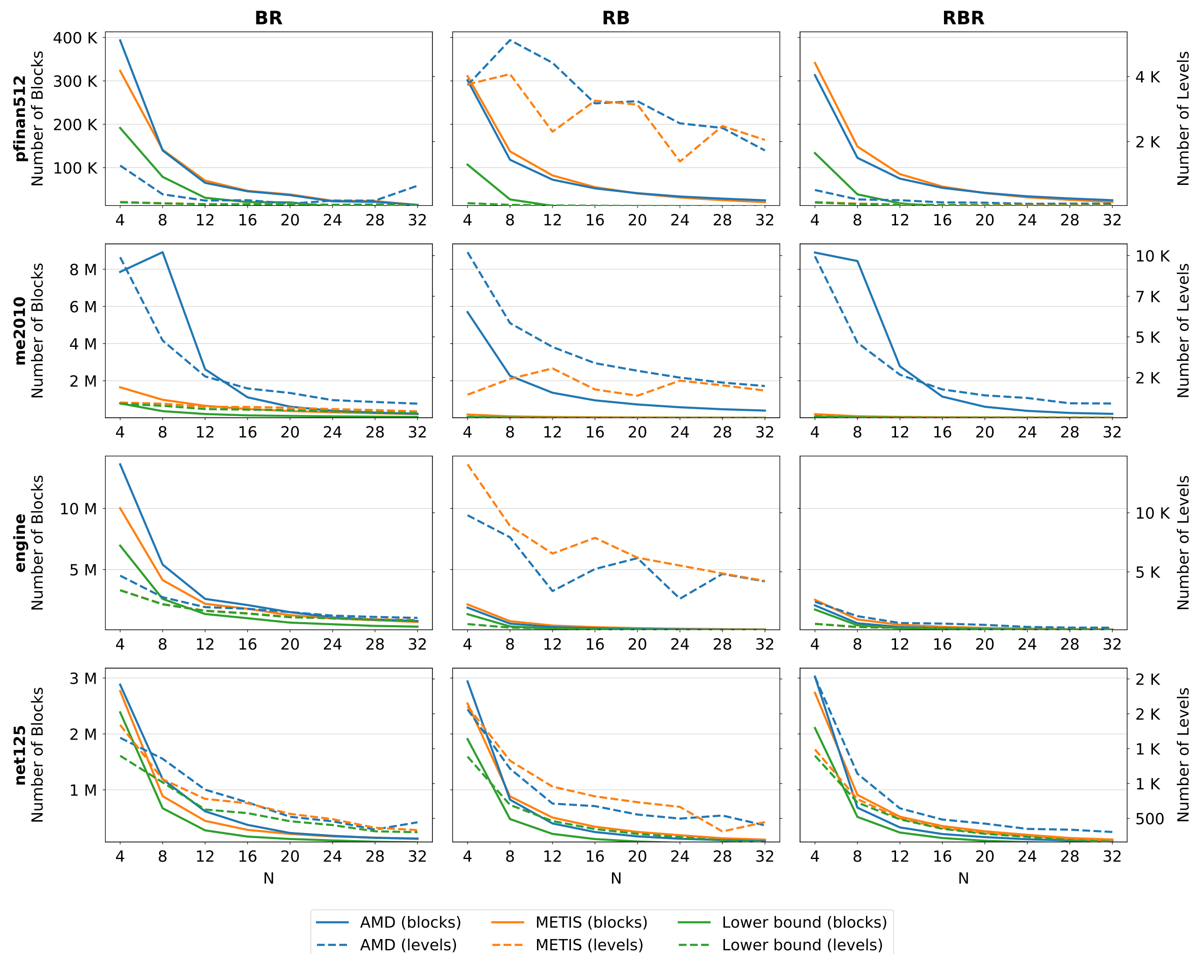


Figure 8.11: Figures 8.9 and 8.10 gave an overview over statistics for $N = 16$ over a set of 100 matrices. This experiment augments these results by investigating the effect of N on the resulting number of blocks (left axes, solid lines) and the number of level sets (right axes, dashed lines). As in Figure 5.2, BR usually yields a smaller number of levels at the cost of a higher number of blocks and, in turn, scalar elements. RB minimizes the number of blocks but often leads to a high number of level sets. RBR combines the strengths of the two methods. The choice of AMD or METIS reorderings depends on the matrix at hand, there is no clear overall winner.

Our next step is the full implementation of the concepts we have discussed in detail within this chapter. In the end, we plan to offer a DSL-like interface that, when combined with borG-code, offers domain scientists the option to create efficient sparse matrix software in minutes that is compiled for the hardware at hand. Besides the drastically shorter ramp-up time for software stacks of novel accelerator hardware, METAPACK allows tailoring the solver algorithm to the matrix at hand. We hope that this customization process can be automatized as a next step, leading to an automatic solver generator for scientific computation and optimization problems.

Chapter 9
Towards Architectural Support for Irregular Tasks

Contents

9.1 Related Work 138
9.2 Programming Model 138
9.3 Implementation 139
9.4 Simulation Results 142
9.5 Conclusion 144

So far in this thesis, we have tried to mitigate data and control flow irregularity that occurs in sparse matrix factorization by conceptual or algorithmic approaches. In Chapter 5, we used dense blocks to allow collective processing of parts of the matrix, including local pivoting. Chapter 6 expanded upon that technique, enabling global pivoting in a 2-stage concept while minimizing pivoting-related divergence. Both approaches relied on the availability of highly efficient accelerator kernels for processing such dense blocks; our compiler borG (see Chapter 7) has proven that such kernels can easily be implemented and are performance portable across multiple platforms.

At this point, it is clear, that a lot of time and work went into our setup. However, the solutions are specific to factorization and do not necessarily help with other tasks – hence, the discrepancy between today's accelerator hardware – built for regular computing loads – and demands of irregular workloads should, in our opinion, attacked on a much deeper level, i.e., in hardware. This chapter outlines an early idea that attempts to bridge the discussed gap.

As we noted in Chapter 1, the last decade has seen a massive increase in raw compute power due to the inclusion of GPUs in HPC systems. GPUs augment the previously dominant multi-core CPU setups by Throughput Oriented Computing (TOC). Garland and Kirk [2010] mention three important design principles for such systems: an abundance of rather simple processing units, SIMD-style execution and hardware multithreading. In this spirit, the scheduling hardware on GPUs is kept simple in favor of more compute, leading them to perform poorly in case of control flow irregularity (e.g., branch divergence, fine-grained synchronization) or data irregularity (e.g., differing resource requirements between work items). Modern multi-core CPUs, on the other hand, are optimized for latency and can use multithreading (SMT) to swap between executing applications. CPU cores operate independently and request resources as needed. Even though the concepts of latency and throughput-oriented architectures are diametrical opposites, researchers have tried to transplant features between them to speed up the execution of programs in their native programming models.

Classical domains where TOC proves effective are now facing the increasing use of irregular applications: sparse matrices, graph neural networks and sum-product networks frequently appear in, e.g., the machine learning community. In order to extend the success and simplicity of CUDA's programming model to these applications, we propose software and hardware modifications that are quickly realizable and add native support for data and control flow irregularity.

This chapter makes the following contributions:

- We extend CUDA's PTX ISA to support an irregular compute model via a mapping to metadata registers in wide-SIMD processors (handling data irregularity).
- We describe a method to use register renaming as a tool for SMT-like processing on a simple vector core, avoiding costly context switches (handling control flow irregularity).
- Lastly, we integrate both ideas into a modified wide-SIMD architecture and validate the effectiveness of our ideas using a model-based software simulation.

In the following, we focus on ideas and their integration; realizing a full system is an objective for future work. We back our claims and support the viability of our ideas using a model-based simulation (cf. below).

9.1 Related Work

The architectural continuum between throughput-oriented SIMT/SIMD designs and latency-based (e.g., SMT) designs has been explored to great depths. Multiple extensions to SIMT designs have been proposed: Scalar co-processors for GPU cores [Chen and Kaeli 2016; Yang et al. 2014; Stanic et al. 2017] that avoid repeated scalar computations; central schedulers share the GPU between host threads [Kim et al. 2019] or dynamic re-grouping of threads from a warp or block into convergent subgroups [Fung et al. 2009b; Fung and Aamodt 2011]. Similar to SMT context switches, Frey et al. [Frey et al. 2012] propose a model for oversubscription of tasks to SIMT cores and a method for faster context switches using the cores' L1 cache. Similarly, work-stealing between warps has been explored [Huzaifa et al. 2020].

On the other end of the spectrum, multiple works have investigated layering a SIMT scheduler on top of arrays of in-order CPU cores [Chen and Chen 2018]. These arrays switch between MIMD mode (each processor operates independently) and SIMT mode (control logic is shared by all processors) to save power. Subsequent works extend this idea into a form of "hardware auto-vectorization" [Tino et al. 2020; Collange 2017] of scalar code. In order to group similar instructions on cores of the array, expensive crossbars are required.

Liquid SIMD [Clark et al. 2007] and Vapour SIMD [Nuzman et al. 2011] on the software side and ARM's SVE hardware extensions [Armejach et al. 2018] improve SIMD systems for irregular applications: they offer a convenient way to set the SIMD vector length at runtime, adapting to tasks with varying resource requirements.

9.2 Programming Model

We now describe our proposed generalization of CUDA's grid-based compute model to irregular workloads. Traditionally, CUDA kernels are parameterized over a grid of blocks, e.g.

```
kernel<<<m, n>>>(...);
```

which launches `m` blocks of `n` threads. Each block is scheduled onto a streaming multiprocessor (SM) which executes warps of 32 threads. All threads within a warp operate in lockstep[1] and branches or conditionals are implemented by predication in recent GPU architectures, all threads in a warp share

[1]With thread-independent scheduling as in the Pascal microarchitecture, the lockstep model has been somewhat relaxed.

access to the SM's register file and communicate through it with low latency. Blocks of variable sizes m_i have to be emulated by setting the block size to $m = \max\{m_i\}_i$ and masking out threads in each block at runtime.

We build our proposed programming model with this issue in mind: First, we get rid of the block abstraction and directly expose warps to users. In order to handle data irregularity, we make the individual warps' sizes a runtime parameter. Instead of passing parameters m, n to the kernel, this requires passing an explicit list of warp sizes:

```
const int w_list[] = {4, 11, 3, 2, 8, 3};
kernel<<<m, w_list>>>(...);
```

As on GPUs, warps are assigned statically to SMs at initialization. Implementing kernels follows the same principle as for warp-centric [Hong et al. 2011] models: all threads in a warp execute in a bulk-synchronous manner and threads communicate through shuffle instructions. To distinguish code for our model from traditional CUDA code, we use the keywords `tid` for a thread's index in a warp and `wid`, `ntids` for a warp's index and size, respectively. As a poster child example, we use the SpMV-kernel in Figure 9.1 (left), where each warp handles one row of a sparse CSR matrix (arrays `csr_row, csr_col, csr_val`). Therein, each thread handles one nonzero entry of the row and the results are accumulated by a warp-wide logarithmic reduction (lines 8 through 12). This simple kernel exhibits both data and light control flow irregularity: First, each warp uses a varying amount of threads and thus the share of the SM's register file depends on a runtime parameter where the classical CUDA execution model requires the register count at compiler time. Second, the number of reduction steps depends on the warp size, leading to different execution paths.

9.3 Implementation

A little unconventional, we propose executing programs in the programming model from above on traditional SIMD hardware. Specifically, we consider wide-SIMD (i.e., vector) hardware due to their ability to set a vector size at runtime. The proposed system introduces additions to CUDA's C-to-PTX compiler and modifications to vector instruction buffers and register renaming units in hardware. Such buffers and renaming units are frequently found in SIMD microarchitectures (e.g., Intel CPUs with AVX). The fundamental ideas of our system are to translate SIMT code (with SIMD-friendly additions) into SIMD code at runtime (as opposed to, e.g., binary translation [Diamos et al. n.d.]) and use register renaming tables to simultaneously execute multiple warps.

9.3.1 Front-End

Executing SIMT code efficiently requires hardware support for predicated execution and branch as well as reconvergence handling. In SIMT models, each thread inside a warp executes the same (scalar) code, but is parameterized by its index inside the warp (CUDA: `lane_id`). SIMD code, on the other hand, operates only on whole vectors at once. Thus, we propose modifications to CUDA's virtual PTX code in order to make it more SIMD-friendly, simplifying processing in the back-end. Changes are visualized using the SpMV example in Figure 9.1.

Metadata registers. We follow the SIMT-on-SIMD paradigm of ISPC Pharr and Mark [2012] by mapping the corresponding registers of threads in a warp to lanes in SIMD registers. We find that it is possible to

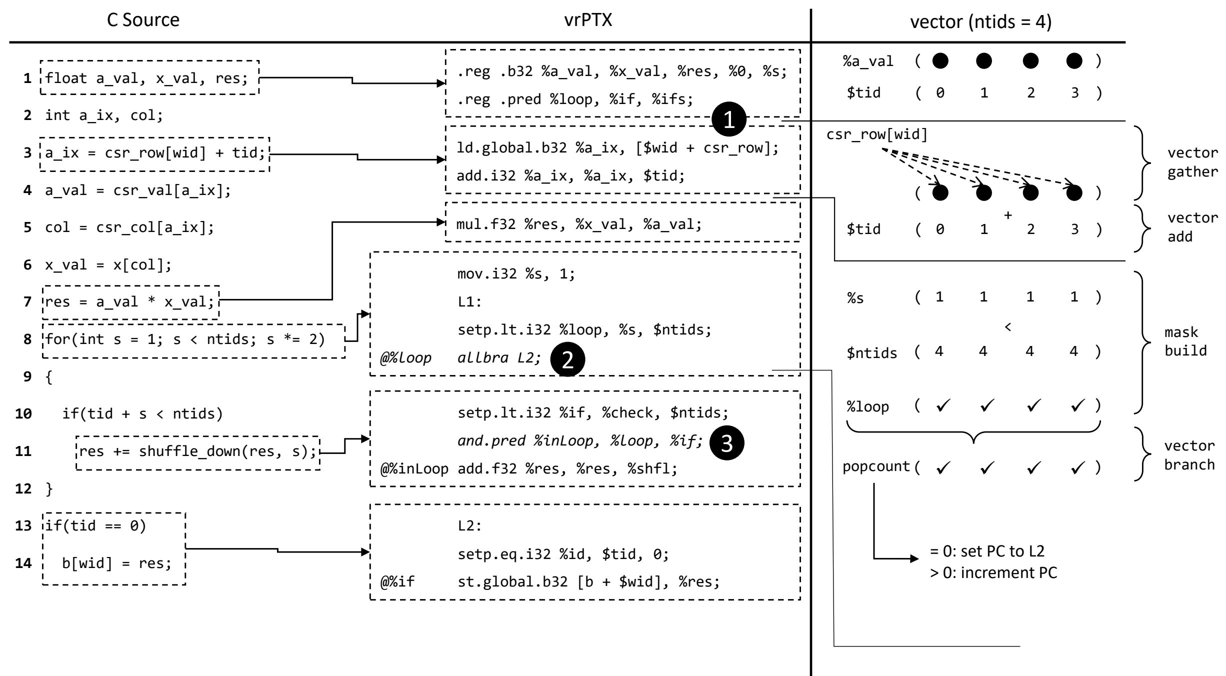


Figure 9.1: Our solution integrates a modified compilation pass from CUDA to *vector-ready PTX* that enables execution of SIMT code on SIMD architectures by concatenating threads' registers into SIMD registers. There, we add special registers for warp and thread indices (❶), insert synchronized branch instructions to handle after thread divergence (❷) and turn predicated instructions into masked instructions (❸).

emulate SIMT execution by explicitly associating keywords such as `$tid` in metadata registers to each lane (parameterized SIMD execution). Appearances of those keywords in the code are then translated to registers with suitable execution, as marked by (1) in Figure 9.1. Unless there is branching involved, SIMT code translates 1:1 into SIMD code; predicated instruction are realized through masked SIMD registers. One more data register holds a candidate PC per thread.

Scalar branch control. Without such per-lane PCs, SIMD hardware is unable to track execution paths of different threads. Instead, all threads must follow the same path of execution, inactive threads' lanes are masked out. In order to handle divergence and lane masks, we use the technique by Lorie and Strong Jr [1984] that inserts JOIN nodes into the code at which the activity of all lanes is tested; if (through, e.g., branching) no active threads remain, the PC of an inactive thread is picked up as the warps' sole PC. In the following, we refer to PTX code with these two additions as Vector-Ready PTX (vrPTX).

9.3.2 Back-End

With parameterized SIMD execution, differently-sized warps all execute the same vrPTX code. Nevertheless, executing each warp on its own SIMD core would often underutilize the hardware and prevent us from hiding latencies, one of the bedrocks of TOC. Therefore, we propose two hardware modifications in SIMD systems:

Partitioning. Following our execution models, executing a single warp of size less than the number of SIMD lanes would leave many SIMD lanes in wide-SIMD systems unoccupied. Since a lane's execution only depends on its metadata, we use this fact to pack multiple warps' data into the same SIMD registers, increasing vector length as necessary and refer to the packed warps as one "partition". Since warps in a

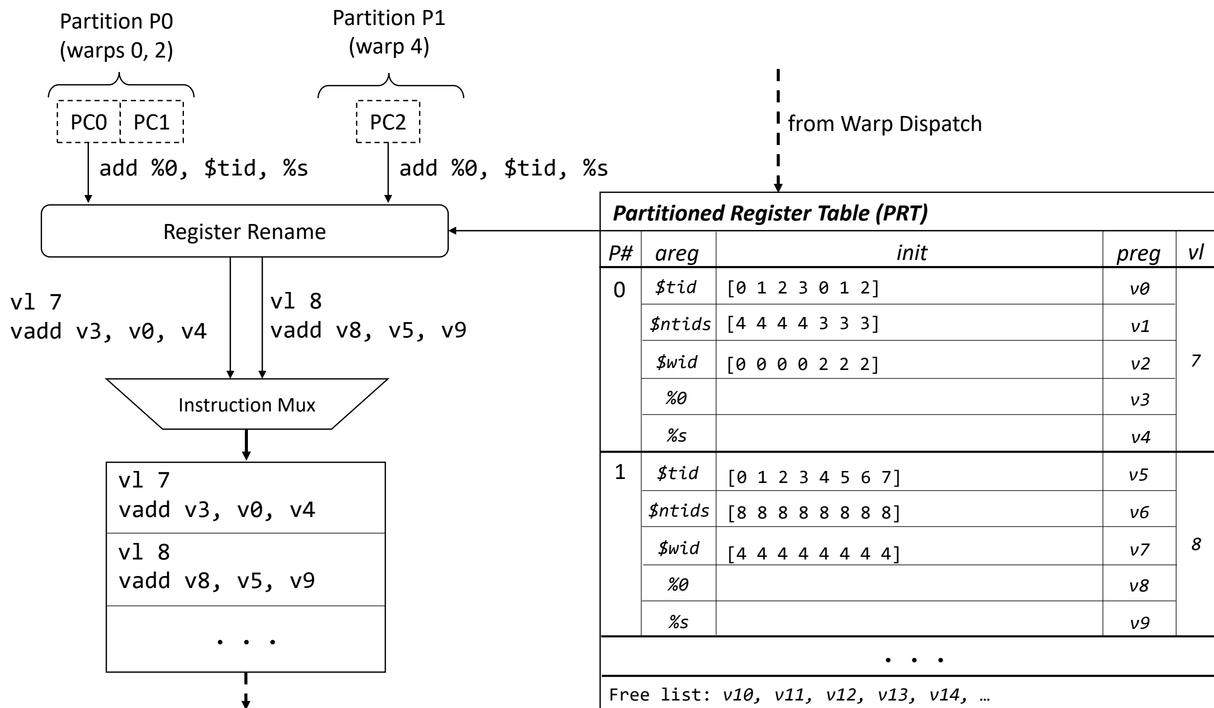


Figure 9.2: We propose to re-purpose the register renaming unit to multiplex vrPTX instruction streams from multiple warps into one single stream of vector instructions.

partition may diverge at branches, we propose the following: in the compile phase, we unroll the vrPTX source for multiple values of `ntids` and count the resulting number of instructions. We then group the possible values of `ntids` into buckets according to the difference in their number of instructions; warps that fall into the same bucket may be put into the same partition, hoping that they would behave similarly. We may repeat that experiment for random outcomes of branch instructions to improve our estimate.

Vector Code Issue and Multiplexing. Even with packing, we still need a method to handle warps of drastically different sizes and diverging control flow. Wrapping warps into SMT threads is not an option: with larger SIMD registers, context switches become prohibitively expensive. Instead, we propose a static "partition multiplex" scheme that uses a register renaming unit to execute multiple partitions at once. We visualize our idea in Figure 9.2: As long as `vl` is less than the number of SIMD lanes, all partitions require the same number of SIMD registers. Hence, we can proceed analogous to SIMT processors and divide the register file according to the partitions. In our example in Figure 9.2, partition 0 (packing warps 0 and 2) uses physical SIMD registers `v0` through `v4`. Using this Partitioned Register Table (PRT), the incoming vrPTX instructions can be mapped conflict-free to physical SIMD registers. After the mapping, a lookup table performs the 1:1 translation from vrPTX to SIMD vector instructions and sets the runtime `vl` accordingly. After renaming, there are no conflicts between streams from different partitions, so all are multiplexed into the same instruction buffer. Through a vector instruction scheduler, this method results automatically hides latencies.

9.3.3 Integration into SX-Aurora

As a practical example, we consider the modification of a vector processor design that is already on the market: NEC's SX-Aurora TSUBASA. As we have briefly stated in Section 4.4, Aurora's PCIe cards offers up to 10 cores at 1.6 GHz with up to 3.07 TFLOPs in double precision mode. All cores share 16 MB LLC

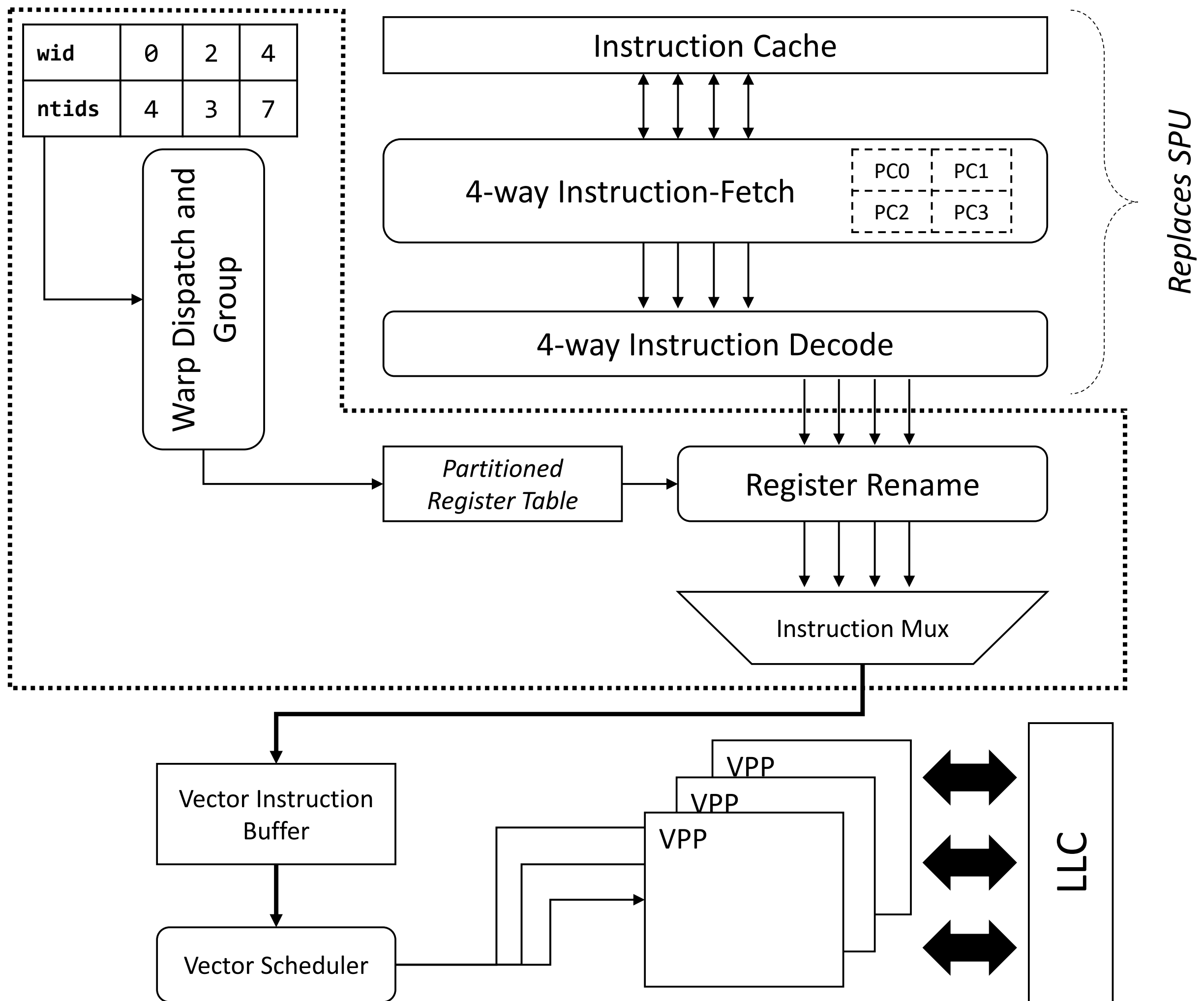


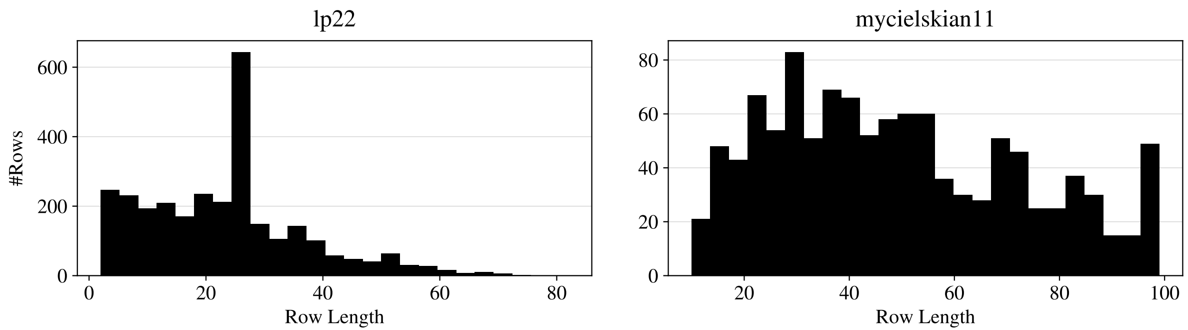
Figure 9.3: Integration of our proposed warp multiplexer (Figure 9.2) into SX-Aurora's vector core. We move the instruction buffer behind it and remove the SPU, except its instruction fetch unit.

and 48 GB HBM2 memory with a peak bandwidth of 1.53 TB/s. Each Aurora core includes a SPU and a VPU (with several Pector Pipeline Processorss (VPPs)) as in Figure 4.8. The VPU uses register renaming to execute vector instructions out-of-order by dispatching them to vector data and mask registers as well as FMA/ALU execution units. In our design, we focus on the VPU exclusively and use only the instruction fetching capabilities of the SPU. Figure 9.3 depicts our modifications: Compared to the original VPU, we pull the register renaming unit before the vector instruction buffer, since we do not use out of order execution within partitions. Instead, vrPTX instructions are loaded for multiple partitions and multiplexed into a single vector instruction stream. Hence, we offer a comparatively cheap way to leverage existing designs for efficient irregular processing.

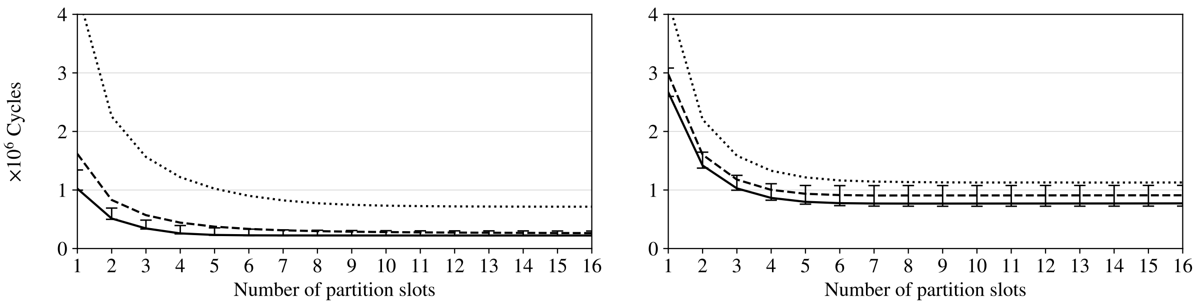
9.4 Simulation Results

Due to a lack of details regarding NEC's SX-Aurora, we used the available ISA documentation in order to build a model following Figure 4.8 and simulate it using the SimPy framework [Müller and Vignaux n.d.]. We express all latencies in terms of multiples of a simple arithmetic vector operation that takes 1 cycle, any complex respective store operation's latency is the active vector length. We simulate one core of

(a) Ranges of nonzero lengths for the test matrices as a measure of data irregularity



(b) Influence of partition multiplexing and packing on SpMV runtimes



(c) Resulting VPP utilization in (b)

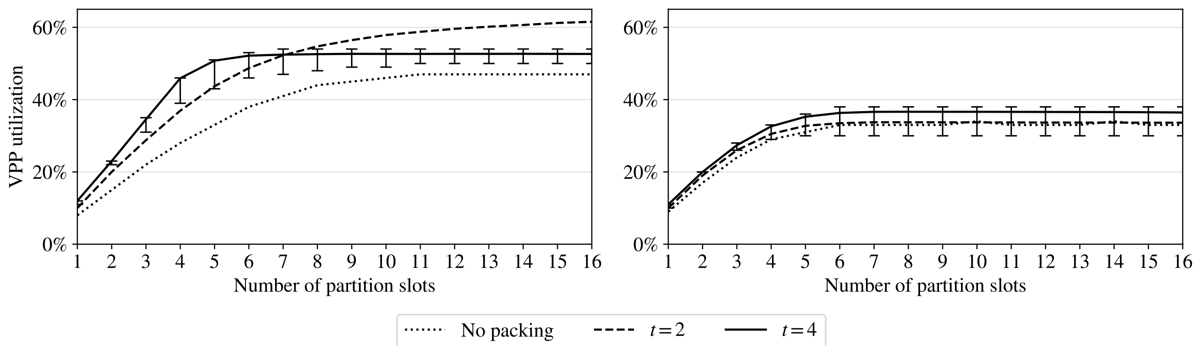


Figure 9.4: Results from a model-driven simulation of our setup running an SpMV kernel for two sparse matrices (left column: lp22, right column: mycielskian11) with one order of magnitude irregularity. The results show that our architecture modifications help to make efficient use of the execution units and hide latencies.

the vector processor with the same number of memory controllers and ports as Aurora. Our simulation consumes the generated PTX from our SPMV example code (see Figure 9.1). We input multiple matrices from the SuiteSparse Matrix Collection's [Davis and Hu 2011] linear programming category, since these matrices often suffer from data irregularity (i.e., different row lengths). This test is meant to showcase two things: First, allowing more partitions (t – also the number of required instruction fetch units) per core results in a larger vector instruction buffer which leads to better utilization of the execution units and thus less empty cycles. Second, packing can save instructions by batching warps – again, we expect less time to termination.

Figures 9.4 presents simulation results for two matrices that are representative for the test set: lp22

($2,958{\times}16,392; 68,512$ nz – first row) and mycielskian11 ($1,535{\times}1,535; 134,710$ nz – second row). Their row length distributions, and thus warp size distributions, are visualized in Figures 9.4a. Figures 9.4b support our first hypothesis: Independent from the packing setup, more slots result in consistently less cycles being used. More slots lead to more and potentially different simultaneous instructions in the instruction buffer which in turn may be executed in parallel (pending execution unit availability). Furthermore, a looser threshold for packing (permitting higher warp size variations inside a partition) further reduces the total number of cycles spent. In Figures 9.4c, we visualize the execution unit utilization in the same experiments: Again, both more partitions as well as looser packing thresholds increase the utilization until reaching a plateau.

Lastly, we point to the error bars for $t = 4$ (Figures 9.4b, 9.4c): for each parameter setting, we ran the simulation 100 times, every time with a random order of the input matrix' rows, and plot the resulting error bars in both cycle and utilization plot. We point out that although the variation is relatively large, at times negating the benefit of packing entirely, the average line (black) tends strongly towards the better region (lower cycles, higher utilization). This indicates that a smaller number of outliers is responsible for such failures. Since we currently do not support work stealing or dynamic allocation, these outliers directly correspond to certain row orders. We leave a closer investigation for future work.

9.5 Conclusion

In this chapter, we briefly discussed how existing vector accelerators could potentially be improved: Embedded in a full stack of programming model, compiler and architecture changes, they can mark a first step towards throughput oriented computing with full support for irregular workloads while retaining the familiar CUDA programming model. Having so far only validated our ideas using a model-based simulation, we plan to follow that up with a cycle-accurate simulation of the proposed system.

Chapter 10
Conclusion

Contents

10.1 Summary 145
10.2 Limitations 146
10.3 Lessons Learnt 149
10.4 Open Problems and Remaining Challenges 151

We conclude this thesis with a discussion of the work presented so far. We summarize our contributions, examine its limitations and reflect upon what we have learnt. Finally, we establish a roadmap on possible future extensions and improvements of our work.

10.1 Summary

Pertaining to our goals for this thesis outlined in Section 1.2, we proposed our central abstraction, the use of a block-based CSR structure in Chapter 5. We use it as a basis for an incomplete $LDL^\top$ preconditioner with support for fill-in on the block- and scalar level as a way to tame the irregular data structure of sparse matrices. In the idiom of collective programming, we found a viable candidate for implementing the required block operations on GPUs. The use of the BCSR format enabled local pivoting operations inside the GPU kernels without any irregular control flow. The resulting software package, block-iLDL$^\top$, solved a wide range of tough, indefinite matrices and outperforms a full pivoting based CPU preconditioner. We complement this engineering effort with an analysis on how crucial sparse reordering heuristics are best applied in conjunction with the block-building process. The analysis put an emphasis on the ability to preserve the inherent parallelism within sparse matrix patterns. Surprisingly, our results also showed that the much-touted asynchronous and regular Jacobi method was unable to solve most of our test matrices.

Chapter 6 saw an extension of Chapter 5's block abstraction in two areas: first, we demonstrated the viability of implementing GPU kernels following the warp-register cache idiom. While common wisdom dictates that these kernels must adhere to a static control flow, we were able to show that it is indeed possible to use full, dynamic pivoting. Here, we chose to pay the penalty for a dynamic register access, but hide it inside the Schur downgrade using ternary operations. This results in a ~2x speedup over shared memory $LL^\top$ and LDU kernels and up to ~3x higher FLOPs. Second, we introduced a flexible index layer on top of the BCSR structure. A global pivoting operation for both symmetric as well as unsymmetric BCSR matrices was reduced to parallel index operations on the hosts' side. No data is moved on the device side. Combined with the pivoting GPU kernels, this approach allows to replace the irregular control flow associated with conventional pivoting by a sequence of regular, parallel operations. Both techniques were applied to an asynchronous block-Jacobi algorithm for incomplete factorizations. The resulting method was able to outperform the conventional level-set approach by large margins. However, as a comprehensive ablation study showed, the determining factors there were both the pivot dampening by an $\epsilon > 0$ as well as the blocking and less the inclusion of global pivoting.

The blocked abstraction, as presented, relies heavily on the availability of batched BLAS-like kernels. Following the warp register cache idiom as suggested in Chapter 6 makes such kernels hard to write, even for experts. In Chapter 7, we offered our source-to-source compiler borG as a remedy. In addition to automatically generating source code for the desired kernels from array-centric, convenient OpenCL code snippets, borG extends far beyond the GPU realm we focused on up to here. Through a virtual architecture, we developed a common IR abstracting over SIMD and SIMT code and discuss the compilation process in-detail. As a result, the combination of our IR and borG enabled developers to write parameterized code for three different compute accelerator architectures from a single code base. Developers can continue to benefit from knowledge and practice that they have acquired with CUDA over the years. The resulting kernels' performance compared favorably and sometimes outperforms code that is written in the native environments, even coming close to heavily autotuned code. As a by-product, this approach showed that a mapping from SIMT-style to various other programmings model is relatively straightforward, putting SIMT ahead in the race to find a common programming model in a time where the zoo of compute architectures is becoming more and more diverse by the year.

In Chapter 8, we combined all our efforts and prior abstractions into a single coherent concept for a system: METAPACK. In accordance with the current development of heterogeneous compute architectures, we proposed to consider sparse matrix factorizations as instances of a common meta-algorithm based on the data structures from Chapters 5 and 6. Using a novel scheduling strategy, we strove to exploit both pipeline-based parallelism and batch parallelism fitting both traditional and "non-Neumann" accelerator architectures. We significantly reduced the complexity of the block operations that make up the computational part of this factorization. Using a prototype implementation, we showed that the METAPACK concept has the potential to generate efficient sparse linear algebra packages for various emerging hardware architectures with minimal user interaction. The results warrant further investigation and encourage a fully fleshed-out and tuned implementation.

We wrap up this thesis with Chapter 9. Having invested a considerable amount of effort into dealing with data and control flow irregularities over the course of our work, we took a step back and discussed how future accelerators can offer better hardware support to deal with irregularity. In an unconventional move, we followed up on our abstraction from borG and proposed to execute warp-centric SIMT code on SIMD accelerators through runtime-configurable metadata registers. While this enables the use of variable-sized SIMT warps on the software side, we lost a key driver of throughput on SIMT systems: latency hiding. Consequently, we proposed to invert the out-of-order execution logic on SIMD systems in order to arrive at a similar functionality. As our architectural simulation of an SpMV kernel shows, the resulting conceptual architecture is able to offer a significantly higher throughput than the original architecture with the added benefit of a convenient developer interface. We hope that such a low-stakes modification of existing SIMD IP can pave the way for native support of irregularity in throughput-oriented computing without resorting to problem-specific measures as we did in the first part of this thesis.

10.2 Limitations

Nothing's perfect - and neither is our work. In the following, we discuss several limitations of the ideas and systems we summarized above.

10.2.1 No "one size fits all"

All our approaches, be it METAPACK, borG or block-iLDL$^\top$ are systems that ultimately form a compromise between different features when solving a linear system. Smaller blocks mean memory savings, but could potentially lead to a longer critical path through level sets. Pivoting adds synchronization points into an otherwise autonomous sequence of instructions but may ultimately improve accuracy. More fill-in may be a necessity to achieve a desired accuracy while otherwise leading to a denser matrix. In order to cover as much ground as possible in the space of (incomplete) matrix factorizations, we open most of these decisions up to the user. Freedom always implies a responsibility - here, this responsibility manifests itself as the burden of choice. As the parameter sets in Chapter 5's evaluation and Chapter 6's ablations studies demonstrate, the optimal choice is highly specific to each matrix. In other words: *there is no "one size fits all"* solution.

Beyond the space of matrix factorizations, there are many more ways to solve sparse linear systems. Be it problem-specific preconditioners or symbolic algorithms – the optimal solver does not necessarily lie within the subspace of linear solvers covered in this thesis.

Every software system has two aspects: its concept and its implementation. Even if the optimal solver for, e.g., a sparse, indefinite matrix would be a direct $LDL^\top$ factorization amenable to block factorizations, deficiencies of our implementations might still put it behind other solvers. This holds even more when we take industry-grade code with many man-years of experience flowing into them (e.g., PARDISO [Schenk et al. 2001]) into account. Moreover, those supernodal approaches often rely on BLAS kernels that are vendor-tuned for a certain platform and make use of all available features of the platform in question. It is more than unlikely for cross-architecture approaches that only uses a subset of this platform's features to reach these levels of performance. In a nutshell: we cannot have a cake and eat it too – in our case, to have a configurable solver and still compare favorably to highly optimized products.

10.2.2 Mitigation, not Resolution of Irregularity

For some algorithms, irregularity is just inherent. Sparse matrix factorization is one of them: sparse data structures lead to data irregularity and pivoting to control flow irregularity. Highly-parallel, regular approaches such as Jacobi methods on distributed systems [Anzt et al. 2018] avoid these irregularities, sacrificing their applicability for tough matrices (see Chapter 6) in the process. Consequently, we have to learn to live with some degree of irregularity. For this reason, the strategies we discuss in this thesis are mitigation strategies only. They do not remove the irregular parts from the computation, but rather minimize their adverse effects.

Over the course of our work, we proposed a) to hide pivoting operations in dense kernels behind the regular Schur downdates in Chapter 6 and b) skipped operations depending on delayed pivots in Chapter 8. Since a) comes at the obvious costs of using more FLOPS for the same result and b) leads to wasted cycles, it becomes clear that there is a tipping point after which it is faster to just execute the irregular computations and accept latencies caused by, e.g., scatter/gather memory accesses. While hardware features such as latency hiding can help pushing the point further back, the key limitation here is the fact that all our proposed strategies can just compensate for the irregularity up to a certain degree. For sparse matrix factorizations, the limiting factor is usually the degree of pivoting: since every pivot operation in METAPACK stalls the block pipeline and requires changes in the index structure, matrices that require

heavy pivoting are better served by a sequential factorization package.

A similar limitation concerns the proposed architecture in Chapter 9: there, the ability to hide latencies crucially depends on the number of partitions and execution units that we offer. Each expansion here requires hardware that adds to the power demands and costs. While concatenating warps increases the used vector length, irregular intra-warp execution of concatenated warps might ultimately lead to slower execution as our proposal does not consider any sort of latency hiding within a partition.

10.2.3 Cross-Architecture Efforts

One of the ideas leading to the development of borG and METAPACK was the ability to generate sparse factorization software for various compute accelerators and emerging architectures without dedicated BLAS-like libraries from vendors. The resulting codes would ideally be performance portable, enabling cooperation between different types of devices. As we showed with borG, this flexibility comes at a price: abstracting over several different architectures leads to forming a very basic model. In our case, we just assume the ability to process over vectors in lockstep, a type of register and contiguous/random access to RAM. This ignores cache hierarchies and on-chip fast memory, e.g., shared memory for CUDA devices. In BLAS-like libraries, its usage is practically mandatory for achieving high sustained levels of performance. Furthermore, our simplistic, batch-like job submission model neglects the existence of asynchronous streams, synchronization between work groups and the availability of atomic operations on some architectures.

borG relies heavily on the symbolic evaluation of data flow at compile time. Only then can shuffle patterns be detected and mapped onto an optimized implementation in device-specific backends. For instance, we can concatenate sequential memory accesses into a single contiguous operation. This, together with the costly implementation of runtime loops on, e.g., SX-Aurora, currently restricts the applicability of borG mostly to parameterized kernels with static control flow. Features that would push this envelope, e.g., cheap runtime shuffles for CUDA GPUs, have no analogue on other architectures and thus fall outside of our performance portable approach. With METAPACK, developers can combine borG-generated kernels with hand-written code, allowing them to trade some development effort for performance boosts due to the use of architecture-specific features.

10.2.4 Consideration of Physical Constraints

This last category concerns our evaluation methodology more than it does our ideas: We only used runtimes respective cycles, memory usage or FLOPs as metrics to judge performance. Even though our approaches cover a diverse set of algorithms and architectures and thus use cases, we did not consider any other objectives than minimizing the time to solution. In particularly, we did not consider any *physical constraints* that may imposed upon a system running borG kernels or METAPACK software.

A simple example is the power usage: a 300W GPU might be faster at decomposing a matrix than a 50W SoC with FPGA, but consumes an order of magnitude more power in the process. This influences the algorithmic choices when deciding on a parameter set for METAPACK: our irregularity mitigation approaches use additional instructions to mask pivot operations in dense kernels, potentially leading to an increased power intake. Here, it might be cheaper, power-wise, to repeat the block factorization multiple times with various fixed pivot orders. Another example concerns programmable hardware: with limited Look-Up Table (LUT) space on an FPGA, complex pivoting kernels might not be a good fit. Thus

the feature set must be reduced in order to generate an METAPACK instance for the chip in question.

Such design considerations would require a careful evaluation of their effects onto physical quantities of e.g., space and power demands. We did not execute these experiments within the scope of this thesis; however, we would like to point out that it is just this flexibility of borG and METAPACK that allows developers to find variants that obey their constraints.

10.3 Lessons Learnt

In this section, we discuss the lessons we have learnt while working on our projects – both with relation to the contents of our research and our methodology. Many of the limitations that we discussed in the preceding section were initially not clear and thus fall under this part as well.

10.3.1 The Importance of Preprocessing

With block-based approaches, preprocessing of sparse matrices is more important than ever. As our studies in Chapters 5 and 8 show, the degree of parallelism varies significantly with the reordering scheme. While RBR emerged as a candidate for a go-to strategy, its effects on the numerics of the factorization remains to be determined. On the numerical side, it is surprising to see how well "classical" heuristics such as MC64 [Duff and Koster 2001] and AMD [Amestoy et al. 2004] perform even without explicitly considering parallelism. These results hint at a deeper connection between the degree of parallelism inherent in a matrix and its tendency to require pivoting, a show stopper for any parallel decomposition approach.

10.3.2 Restricted Applicability of Fixed-Point Methods

Based on our experiences in Chapter 6, we would like to underscore that it is absolutely required to test one's approaches on numerically *hard* cases. Over the last 5 years, many approaches based on relaxations and embarrassingly-parallel algorithms such as Jacobi's method [Chow et al. 2018; Anzt et al. 2019a] have resurfaced. The distributed nature of these approaches makes them likely candidates for the path to Exascale with a high density of local FLOPS, only minimal messaging between nodes and the ability to use reduced precision. Our experiments were able to show that these methods fail when tougher test problems, e.g., from optimization, are used. In many of the papers we have mentioned, the test set was limited to positive definite matrices with condition numbers $\ll 10^6$ and sometimes even diagonal dominance without explicitly mention. ParILUT [Anzt et al. 2019b] and extensions that are based on Jacobi approaches fail similarly. These methods are useful in the context of, e.g., finite element problems, but their applicability is restricted. Hence, we have learnt that considering these methods as a "gold standard" for parallel linear algebra is not helpful. Research into extending "classical", pivoting-based methods still has its merits, as is underscored by, e.g., the continuous improvement of the industry-leading solver PARDISO [Schenk et al. 2001].

10.3.3 Parallel Complexities of Matrices

A third lesson actually demonstrated to us the lack of knowledge in an unexpected area: the "parallel complexity" of a sparse matrix. We use this expression to describe the degree of performance gain that we experience for linear systems with a certain sparse matrix when solved with suitable algorithm on a parallel system. Or, in other terms: is there any solver technique that results in a meaningful speedup on a parallel computer compared to the optimal sequential solver? In our experiments, we found matrices

whose structure exhibits long chains of nonzero elements that lead to a tight coupling between rows yet work quite well with the Jacobi algorithm. However, we also encountered matrices with only a few level sets that require many pivoting operations, synchronizing computations to the point of falling behind a sequential solver. To the best of our knowledge, this concept has not been formally defined and investigated, but we regard it as highly relevant. The parallel complexity in conjunction with the numerical complexity would enable us to get an upper bound on the expected performance for parallel solvers. We note that some other authors' experiments hint at the complexity of the topic: While Bhowmick et al. [2006] and Yeom et al. [2016] have studied the correlations of some matrix properties with the runtime of iterative solvers and preconditioners, they did not consider either's parallelization. They identify the condition number, Frobenius norm and nonzero count of the matrix as the most relevant features in their setup, but stay limited to a class of PDE-matrices. They did not include cases where iterative solvers fail – an important, but unknown, tipping point. The only work considering parallelism, to the best of our knowledge, is Götz and Anzt [2018]. They use a Convolutional Neural Network (CNN) to find the optimal block sizes for the application of a block-Jacobi preconditioner. In this process, they require 9 million parameters just for this seemingly simple task.

10.3.4 Lack of Support for Irregular Workloads on Emerging Accelerators

Reviewing the development of compute accelerators leads to the conclusion that we actually see even less support for irregular support on the hardware side. Systolic arrays, FPGAs, AI accelerators and other "non-Neumann" types are more and more integrated into HPC setups, where they offer power-efficient acceleration of certain static control flows. This market is powered by a steady infusion of venture capital and moving at a steady pace towards even more exotic architectures in the realm of neuromorphic computing. Considering these developments, we realized that rather development-intensive irregularity mitigation strategies relying on special features of certain architecture are not the way to go. If we want to amortize development costs by supporting multiple such architectures, we may consider an alternative path: with integrated heterogeneous chips and SoCs, we can pair such regular hardware with a few CPU cores for irregular parts of the program. On the hardware side, we would tightly integrate two separate architectures, one for the regular and one for the irregular tasks, sharing their data. On the software side, this demands codes which clearly separate the regular from the irregular tasks and offer flexible interfaces. In a nutshell, this is exactly what we are trying to achieve with METAPACK: Handle complexities such as pivoting and scheduling on a CPU core, thus simplifying the actual kernels to the point that cross-device compilers, such as borG, are able to create highly performant kernels.

10.3.5 Top-Down vs. Bottom Up Systems Research

The last 5 years have not just deepened our knowledge in the fields of computational linear algebra and heterogeneous systems, but also allowed the author to improve his research methodology and learn some valuable lessons how research should be done. Thanks to feedback from others, we have learnt a vital lesson that especially applies to systems reach: consider working top-down on a system. In practice, this means starting with a rough proof of concept of a system, e.g., METAPACK. After measuring its potential using a set of quick experiments, we would be able to proceed drilling down into each component and improving them individually. Only at the very end, one should do a second pass and further integrate the individual parts. We often found ourselves working bottom-up where we should have been working in a top-down fashion. As a consequence, our initial project METAPACK has been

repeatedly pushed back in favor of tuning a part of it, such as borG.

10.3.6 Evaluating Research and Presenting Results

Following a bottom-up methodology also helps with another aspect: clearly communicating and motivating ideas. With METAPACK as a framework, the value of each improvement in one of its components can be quantified with respect to the whole setup. This especially helps with communication through publications. Instead of, e.g., combining two different aspects in the paper contained in Chapter 6 and evaluating them with a separate prototype (the improved Jacobi algorithm), every single idea could have been evaluated in isolation regarding their effects on METAPACK. For the future, we plan to restrict publications to one principal idea where possible, and concentrate on a crystal clear presentation instead.

10.3.7 Exchange of Ideas

Another lesson covers the benefits of communication and the importance of continuing exchange within the research community: many stepping stones in our research could have been avoided by discussions with fellow researchers. In addition, novel ideas are often born within these, sometimes spontaneous, communications. In this thesis, the idea for METAPACK was the result of a discussion with Prof. Matthias Bollhöfer. Lastly, we have learnt that similar research may be already on its way in another field, using other specific terms: for example, Sparse Tensor operations [Gale et al. 2020] in the “Systems for ML” community adopt many of the SpMM and SpMV optimization strategies that have been presented before for GPUs in the context of sparse linear matrix algebra [Anzt et al. 2020]. Thus, we conclude that it is vitally important to talk to researchers from fields outside of our own fields of research as well.

Self-realization is the first step to success. We hope that the lessons we have learnt through this work are not just useful for the author’s development as a researcher, but may also be of use to eventual readers.

10.4 Open Problems and Remaining Challenges

Overcoming each of the aforementioned limitations, where possible, are among our top priorities when it comes to the continuation of our work beyond this thesis. In addition, we see opportunities for extensions and improvements to our research in three areas.

10.4.1 Linear Algebra

As a first step, we plan to fully flesh out the implementation of METAPACK and tune it, resulting in a set of distributed software packages for typical compute platforms. A large-scale study on thousands of sparse matrices and parameter sets may hopefully help us figuring out the defining characteristics of matrices when it comes to solver selection. Ideally, we would further develop, possibly with the help of graph-based machine learning, a “fingerprinting” technique for sparse matrices that recommends suitable solvers. Given their size, matrices from production are orders of magnitude larger than what current-generation graph neural networks can handle. We suspect that these efforts may also aid the sparse machine learning community.

Furthermore, we would like to explore a tighter integration of METAPACK with flexible iterative methods, e.g., Flexible GMRES [Saad 1993]. These methods allow using multiple preconditioners over the course of their iterations; using METAPACK’s parameterization, we could monitor the solvers progress and refine the incomplete factorization as needed, avoiding unnecessary fill-in. Finding the decision points at which to

start improving the preconditioner given this moves' up-front cost, may be relegated to a reinforcement learning algorithm. Research into the usage of machine learning techniques for linear algebra has only begun, most recently with Luna and Blaschke [2020].

Using METAPACK's flexibility, we see a possible avenue for future research in exploring hybrid parallelizations of the factorization process. For example, we might use a classical delayed pivoting process on the outer loop, but replace the typical level set parallelization within each pivoting stage by a ϵ-dampened Jacobi-like iteration (see Chapter 6). Alternatively, we could attempt to replace our delayed pivoting strategy completely by static approaches to the likes of butterfly transformations [Baboulin et al. 2014]. These approaches may also help in further reducing the effect of pivoting-related irregularity.

Our flexible block format in METAPACK opens up new opportunities for the integration of reduced precision techniques: by chaining custom code to block-load and -store operations, we could adaptively reduce the accuracy of all values in a block in order to benefit from faster FP16-operations that come with e.g., modern Tensor Cores on NVIDIA GPUs. Furthermore, we could consider the accuracy of a whole block instead of the accuracy of each of its members in isolation. Following up on this view, we can consider using lossy compression on block levels for blocks that have a small magnitude compared to the pivot blocks, for example. On the software side, we should allow the user to chain arbitrary kernels into a form of block "serialization". All in all, this would turn METAPACK into a platform for research and development of future linear algebra algorithms.

Lastly, we need to extend an evaluation of METAPACK's design space into the physical realm. If we can create compatible software packages for various compute platforms, the issue of a platform selection arises. Given a set physical constraints and a sparse matrix, we would need to issue some recommendations about which platform could solve the problem within the limits set. Besides power, these limits may especially concern the price of buying hardware (or renting it in the cloud). Such research would be immensely valuable in the increasingly heterogeneous landscape of high performance computing.

10.4.2 Cross-Architecture Efforts

As of the time of submission, our prototype implementation of borG is the limiting factor. Its concept, overlaying SIMD and SIMT computation in order to support a broad set of architectures, presents a path to much greater impact. We believe that there are many roads to expand the applicability of borG to more scenarios. For METAPACK, we desperately need to find more abstractions over features of the target architectures. To mind comes the memory hierarchy beyond registers that almost all extend to some type of addressable local memory. Furthermore, some techniques that allow for irregular control and dataflow, such as runtime shuffles, require device-specific implementations. In short, we plan to improve borG step by step towards a production ready cross-architecture development approach. To that end, we consider a re-implementation based on the upcoming standard MLIR infrastructure [Lattner et al. 2020] and push our proposal for a modified LLVM IR to upstream. Given the similarity of the PIRCH programming model with the systolic array-based implementation strategy [Chen et al. 2019], we think the SIMT model is *the* way forward to achieve performance portable codebases over a large range of HPC architectures. borG's unique selling point is its ability to produce highly efficient code over many platforms with a transparent compilation process and no costly autotuning.

However, coding efficient SIMT-style kernels is still hard for developers that have no experience with GPU

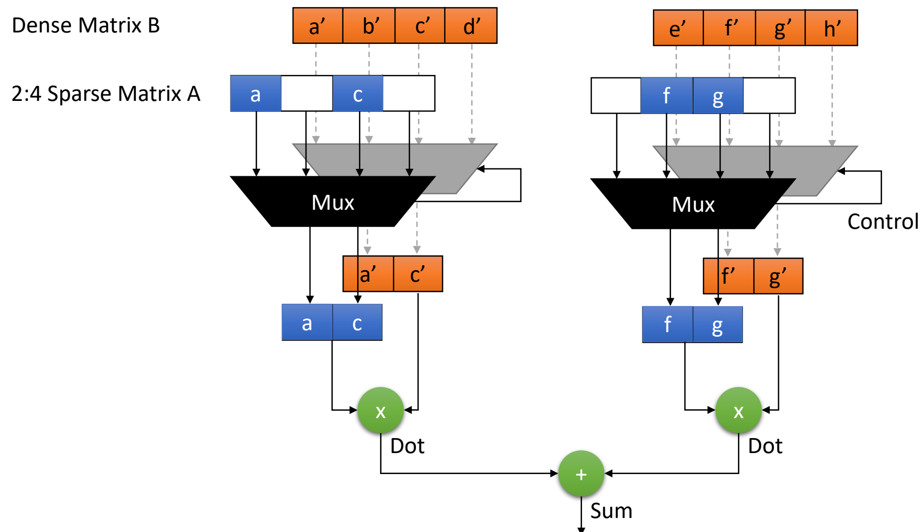


Figure 10.1: Principle of NVIDIA's third-generation Tensor Cores with sparse support: they compute a matrix product of a 2:4 sparse matrix A – 2 out of every 4 elements may be zero – with a dense matrix B. Hard zeros steer multiplexes that pack the A and B into dense matrices of half width. This figure visualizes the description of sparse Tensor Cores in NVIDIA Corporation [2020b].

development. In order to broaden the appeal of systems like borG and METAPACK, it is vital to simplify the programming model further. Through the similarity of SIMT execution and the model of systolic arrays, we see an opportunity in deriving SIMT code from dataflow graphs [Ben-Nun et al. 2019]. Ideally, SIMT code is created from scalar, loop-based representations that are far easier to use. This approach, however, requires a much more involved compilation process through polyhedral means [Vasilache et al. 2019]. If we are ready to sacrifice the general-purpose applicability of borG's programming model, borG may be used as a backend component to DSLs, e.g., via AnyDSL [Leißa et al. 2018]. Ultimately, we see borG and the SIMT approach as an important component in a universal pipeline that could one day help realizing program synthesis [Gulwani et al. 2017].

So far, we have compiled and optimized each kernel and parameter with borG in isolation. However, once we proceed to realize chained block functions in METAPACK, some storing and loading operations in each kernel of the chain may become redundant. Think of a chain of kernels that loads a compressed block, uncompresses it and executes some normalization on the result. Just executing these three kernels sequentially would mean that every kernels stores and loads the matrices from and to device memory. Instead, we could fuse these kernels, track the memory operations and only keep the first memory load and last memory store around. This change would retain the same performance levels as hand-written borG-"Uberkernels" but allow developers to create a library of basic building blocks, building complex kernels through recombination.

10.4.3 Emerging and Future Hardware

Fueled by the pace of the machine learning research community, there has been a proliferation of fixed-function accelerators in recent years. Beyond dense systolic arrays and their integration into NVIDIA GPUs in the form of "Tensor Cores", we start seeing the first wave of hardware accelerators that specifically target sparse data. With its "Ampêre" architecture, NVIDIA Corporation [2020b] recently presented the third generation of its Tensor Cores that adds native support for sparse tensors. Up to 2 out of 4

contiguous zero entries in a tensor can be exploited at runtime for performance benefits (see Figure 10.1). Beyond that, novel architectures such as GraphCore's Intelligence Processing Unit (IPU) [Jia et al. 2019] allows building a computational graph with embedded sparsity structures directly on the device. When fed with data, the IPU autonomously executes the whole model via its computational graph on hundreds of individual cores with a 6-fold round robin thread scheduler. Given that a Tensor is the basic unit of execution for both concepts, we could use them to accelerate METAPACK's kernels when the dense blocks exhibit some numerical zeros. This would limit memory overuse and achieve better performance for GEMM-like block operations.

In this thesis, we assumed a setup of a CPU-based computer with an additional compute accelerator card. We used borG to generate the device-side code and relegated the host CPU to perform pivoting operations on the index structure presented in Chapter 6. Thus, we have always considered the host CPU and accelerator card only individually, with communication limited to submitting job queues from the host to the device. However, system setups are moving more and more towards tight integration of heterogeneous (co-)processors, even on the same chip. Apple's M1 is a prime example of this integration, with several fixed-function chips on a die with a scalable ARM processor. Hence, in the future, we would like to investigate the opportunities presented by this change. For example, we could turn the setup around and put the accelerator in control. In this case, the accelerator may trigger data structure reorganizations (e.g., pivoting operations) and memory transfers based on its pipeline fill status. With that, we hope to avoid idle cycles on the accelerator and ultimately reduce the time-to-solution.

Lastly, as Chapter 9 teased, we have high hopes for the integration of different architectural concepts such as SIMT and SIMD on the same piece of hardware. In line with the recent re-resurgence of FP-GAs as reconfigurable (co-) processors, we envision that in the next decade, developers can produce cross-architectural code using their favorite programming and execution model. The hardware then automatically uses its configurable parts in order to execute the code in the most efficient way. We believe that this thesis has shown how we could take the first steps towards this goal – in the field of linear algebra.

Bibliography

[Abadi et al. 2016] Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard (2016). "Tensorflow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283.

[Abdelfattah et al. 2017] Abdelfattah, A., A. Haidar, S. Tomov, and J. Dongarra (May 2017). "Fast Cholesky Factorization on GPUs for Batch and Native Modes in MAGMA". en. In: *Journal of Computational Science* 20, pp. 85–93. issn: 18777503. doi: 10.1016/j.jocs.2016.12.009.

[Aliaga et al. 2011] Aliaga, J. I., M. Bollhöfer, A. F. Martı, and E. S. Quintana-Ortı (2011). "Exploiting Thread-Level Parallelism in the Iterative Solution of Sparse Linear Systems". In: *Parallel Computing* 37.3, pp. 183–202.

[Amazon 2020] Amazon (2020). *Amazon AWS Inferentia*.

[Amestoy et al. 2004] Amestoy, P. R., T. A. Davis, and I. S. Duff (2004). "Algorithm 837: AMD, an Approximate Minimum Degree Ordering Algorithm". In: *ACM Transactions on Mathematical Software (TOMS)* 30.3, pp. 381–388.

[Amestoy et al. 2001] Amestoy, P. R., I. S. Duff, J.-Y. L'Excellent, and J. Koster (2001). "A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling". In: *SIAM Journal on Matrix Analysis and Applications* 23.1, pp. 15–41.

[Anderson et al. 1999] Anderson, E., Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and A. McKenney (1999). *LAPACK Users' Guide*. SIAM.

[Anderson and Saad 1989] Anderson, E. and Y. Saad (1989). "Solving Sparse Triangular Linear Systems on Parallel Computers". In: *International Journal of High Speed Computing* 1.01, pp. 73–95.

[Anzt et al. 2015] Anzt, H., E. Chow, and J. Dongarra (2015). "Iterative Sparse Triangular Solves for Preconditioning". en. In: *Euro-Par 2015: Parallel Processing*. Ed. by J. L. Träff, S. Hunold, and F. Versaci. Vol. 9233. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 650–661. isbn: 978-3-662-48095-3 978-3-662-48096-0. doi: 10.1007/978-3-662-48096-0_50.

[Anzt et al. 2018] Anzt, H., E. Chow, and J. Dongarra (Jan. 2018). "ParILUT - A New Parallel Threshold ILU Factorization". en. In: *SIAM Journal on Scientific Computing* 40.4, pp. C503–C519. issn: 1064-8275, 1095-7197. doi: 10.1137/16M1079506.

[Anzt et al. 2020] Anzt, H., T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang (2020). "Load-Balancing Sparse Matrix Vector Product Kernels on Gpus". In: *ACM Transactions on Parallel Computing (TOPC)* 7.1, pp. 1–26.

[Anzt et al. 2019a] Anzt, H., J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana-Ortí (2019a). "Adaptive Precision in Block-Jacobi Preconditioning for Iterative Sparse Linear System Solvers". In: *Concurrency and Computation: Practice and Experience* 31.6, e4460.

[Anzt et al. 2017a] Anzt, H., J. Dongarra, G. Flegar, and E. S. Quintana-Ortí (2017a). "Batched Gauss-Jordan Elimination for Block-Jacobi Preconditioner Generation on Gpus". In: *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, pp. 1–10.

[Anzt et al. 2017b] Anzt, H., J. Dongarra, G. Flegar, and E. S. Quintana-Orti (Aug. 2017b). “Variable-Size Batched LU for Small Matrices and Its Integration into Block-Jacobi Preconditioning”. en. In: *2017 46th International Conference on Parallel Processing (ICPP)*. Bristol, United Kingdom: IEEE, pp. 91–100. isbn: 978-1-5386-1042-8. doi: `10.1109/ICPP.2017.18`.

[Anzt et al. 2019b] Anzt, H., T. Ribizel, G. Flegar, E. Chow, and J. Dongarra (May 2019b). “ParILUT - A Parallel Threshold ILU for GPUs”. en. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Rio de Janeiro, Brazil: IEEE, pp. 231–241. isbn: 978-1-72811-246-6. doi: `10.1109/IPDPS.2019.00033`.

[Applegate et al. 2001] Applegate, D., R. Bixby, V. Chvátal, and W. Cook (2001). “TSP Cuts Which Do Not Conform to the Template Paradigm”. In: *Computational Combinatorial Optimization*. Springer, pp. 261–303.

[Armejach et al. 2018] Armejach, A., H. Caminal, J. M. Cebrian, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó (Nov. 2018). “Stencil Codes on a Vector Length Agnostic Architecture”. In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’18. New York, NY, USA: Association for Computing Machinery, pp. 1–12. isbn: 978-1-4503-5986-3. doi: `10.1145/3243176.3243192`.

[Arnoldi 1951] Arnoldi, W. E. (1951). “The Principle of Minimized Iterations in the Solution of the Matrix Eigenvalue Problem”. In: *Quarterly of applied mathematics* 9.1, pp. 17–29.

[Asanović et al. 2006] Asanović, K., R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick (Dec. 2006). *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley.

[Augustine et al. 2019] Augustine, T., J. Sarma, L.-N. Pouchet, and G. Rodríguez (2019). “Generating Piecewise-Regular Code from Irregular Structures”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 625–639.

[Axelsson 1972] Axelsson, O. (1972). “A Generalized SSOR Method”. In: *BIT Numerical Mathematics* 12.4, pp. 443–467.

[Baboulin et al. 2014] Baboulin, M., X. S. Li, and F.-H. Rouet (2014). “Using Random Butterfly Transformations to Avoid Pivoting in Sparse Direct Methods”. In: *International Conference on High Performance Computing for Computational Science*. Springer, pp. 135–144.

[Barham and Isard 2019] Barham, P. and M. Isard (May 2019). “Machine Learning Systems Are Stuck in a Rut”. en. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. Bertinoro Italy: ACM, pp. 177–183. isbn: 978-1-4503-6727-1. doi: `10.1145/3317550.3321441`.

[Barthels et al. 2020] Barthels, H., C. Psarras, and P. Bientinesi (June 2020). “Automatic Generation of Efficient Linear Algebra Programs”. In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC ’20. New York, NY, USA: Association for Computing Machinery, pp. 1–11. isbn: 978-1-4503-7993-9. doi: `10.1145/3394277.3401836`.

[Batchelder 2004] Batchelder, N. (2004). *Cog - Code Generator*.

[Bathe 2006] Bathe, K.-J. (2006). *Finite Element Procedures*. Klaus-Jurgen Bathe.

[Bauer et al. 2014] Bauer, M., S. Treichler, and A. Aiken (2014). “Singe: Leveraging Warp Specialization for High Performance on GPUs”. In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 119–130.

[Becker et al. 2012] Becker, D., M. Baboulin, and J. Dongarra (2012). “Reducing the Amount of Pivoting in Symmetric Indefinite Systems”. en. In: *Parallel Processing and Applied Mathematics*. Ed. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski. Vol. 7203. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 133–142. isbn: 978-3-642-31463-6 978-3-642-31464-3. doi: 10.1007/978-3-642-31464-3_14.

[Beckingsale et al. 2019] Beckingsale, D. A., J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland (2019). “RAJA: Portable Performance for Large-Scale Scientific Applications”. In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, pp. 71–81.

[Ben-Nun et al. 2019] Ben-Nun, T., J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler (2019). “Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14.

[Bendersky 2010] Bendersky, E. (2010). *PyCParser*.

[Bertolacci et al. 2016] Bertolacci, I. J., M. M. Strout, S. Guzik, J. Riley, and C. Olschanowsky (2016). “Identifying and Scheduling Loop Chains Using Directives”. In: *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*. IEEE, pp. 57–67.

[Bhandarkar and Clark 1991] Bhandarkar, D. and D. W. Clark (1991). “Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization”. In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 310–319.

[Bhowmick et al. 2006] Bhowmick, S., V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes (2006). “Application of Machine Learning to the Selection of Sparse Linear Solvers”. In: *Int. J. High Perf. Comput. Appl*.

[Bian et al. 2020] Bian, H., J. Huang, R. Dong, L. Liu, and X. Wang (May 2020). “CSR2: A New Format for SIMD-Accelerated SpMV”. In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pp. 350–359. doi: 10.1109/CCGrid49817.2020.00-58.

[Bollhöfer and Saad 2006] Bollhöfer, M. and Y. Saad (Jan. 2006). “Multilevel Preconditioners Constructed From Inverse-Based ILUs”. en. In: *SIAM Journal on Scientific Computing* 27.5, pp. 1627–1650. issn: 1064-8275, 1095-7197. doi: 10.1137/040608374.

[Bollhöfer et al. 2019] Bollhöfer, M., O. Schenk, and F. Verbosio (2019). “High Performance Block Incomplete LU Factorization”. In: *arXiv preprint arXiv:1908.10169*. arXiv: 1908.10169.

[Bozdağ et al. 2010] Bozdağ, D., Ü. i. t. V. Çatalyürek, A. H. Gebremedhin, F. Manne, E. G. Boman, and F. Özgüner (2010). “Distributed-Memory Parallel Algorithms for Distance-2 Coloring and Related Problems in Derivative Computation”. In: *SIAM Journal on Scientific Computing* 32.4, pp. 2418–2446.

[Brown et al. 2020] Brown, T. B., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark,

C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei (July 2020). “Language Models Are Few-Shot Learners”. In: *arXiv:2005.14165 [cs]*. arXiv: 2005.14165 [cs].

[Buchheim et al. 2008] Buchheim, C., F. Liers, and M. Oswald (2008). “Local Cuts Revisited”. In: *Operations Research Letters* 36.4, pp. 430–433.

[Buck et al. 2004] Buck, I., T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan (2004). “Brook for GPUs: Stream Computing on Graphics Hardware”. In: *ACM transactions on graphics (TOG)* 23.3, pp. 777–786.

[Bunch et al. 1976] Bunch, J. R., L. Kaufman, and B. N. Parlett (1976). “Decomposition of a Symmetric Matrix”. In: *Numerische Mathematik* 27.1, pp. 95–109.

[Bunch and Parlett 1971] Bunch, J. R. and B. N. Parlett (1971). “Direct Methods for Solving Symmetric Indefinite Systems of Linear Equations”. In: *SIAM Journal on Numerical Analysis* 8.4, pp. 639–655.

[Buttari et al. 2007] Buttari, A., J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov (2007). “The Impact of Multicore on Math Software”. en. In: *Applied Parallel Computing. State of the Art in Scientific Computing*. Ed. by B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski. Vol. 4699. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–10. isbn: 978-3-540-75754-2. doi: 10.1007/978-3-540-75755-9_1.

[Cassagne et al. 2018] Cassagne, A., O. Aumage, D. Barthou, C. Leroux, and C. Jégo (2018). “MIPP: A Portable C++ SIMD Wrapper and Its Use for Error Correction Coding in 5G Standard”. In: *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, pp. 1–8.

[Cerebras Systems 2019] Cerebras Systems (2019). *CS-1*.

[Chen and Chen 2018] Chen, K.-C. and C.-H. Chen (Apr. 2018). “Enabling SIMT Execution Model on Homogeneous Multi-Core System”. en. In: *ACM Transactions on Architecture and Code Optimization* 15.1, pp. 1–26. issn: 1544-3566, 1544-3973. doi: 10.1145/3177960.

[Chen et al. 2019] Chen, P., M. Wahib, S. Takizawa, R. Takano, and S. Matsuoka (2019). “A Versatile Software Systolic Execution Model for GPU Memory-Bound Kernels”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–81.

[Chen et al. 2008] Chen, Y., T. A. Davis, W. W. Hager, and S. Rajamanickam (2008). “Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate”. In: *ACM Transactions on Mathematical Software (TOMS)* 35.3, pp. 1–14.

[Chen and Kaeli 2016] Chen, Z. and D. Kaeli (May 2016). “Balancing Scalar and Vector Execution on GPU Architectures”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 973–982. doi: 10.1109/IPDPS.2016.74.

[Cheshmi et al. 2017] Cheshmi, K., S. Kamil, M. M. Strout, and M. M. Dehnavi (Nov. 2017). “Sympiler: Transforming Sparse Matrix Codes by Decoupling Symbolic Analysis”. en. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver Colorado: ACM, pp. 1–13. isbn: 978-1-4503-5114-0. doi: 10.1145/3126908.3126936.

[Chetlur et al. 2014] Chetlur, S., C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer (2014). “Cudnn: Efficient Primitives for Deep Learning”. In: *arXiv preprint arXiv:1410.0759*. arXiv: 1410.0759.

[Chow and Scott 2016] Chow, E. and J. Scott (2016). “On the Use of Iterative Methods and Blocking for Solving Sparse Triangular Systems in Incomplete Factorization Preconditioning”. In: *Rutherford Appleton Laboratory, Tech. Rep. Technical Report RAL-P-2016-006*.

[Chow et al. 2018] Chow, E., H. Anzt, J. Scott, and J. Dongarra (Sept. 2018). “Using Jacobi Iterations and Blocking for Solving Sparse Triangular Systems in Incomplete Factorization Preconditioning”. en. In: *Journal of Parallel and Distributed Computing* 119, pp. 219–230. issn: 07437315. doi: 10.1016/j.jpdc.2018.04.017.

[Chow and Patel 2015] Chow, E. and A. Patel (Jan. 2015). “Fine-Grained Parallel Incomplete LU Factorization”. en. In: *SIAM Journal on Scientific Computing* 37.2, pp. C169–C193. issn: 1064-8275, 1095-7197. doi: 10.1137/140968896.

[Chow and Saad 1997] Chow, E. and Y. Saad (1997). “Experimental Study of ILU Preconditioners for Indefinite Matrices”. In: *Journal of computational and applied mathematics* 86.2, pp. 387–414.

[Clark et al. 2007] Clark, N., A. Hormati, S. Yehia, S. Mahlke, and K. Flautner (2007). “Liquid SIMD: Abstracting SIMD Hardware Using Lightweight Dynamic Mapping”. In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, pp. 216–227.

[Collange 2017] Collange, S. (2017). “Simty: Generalized SIMT Execution on RISC-V”. en. In: p. 7.

[Cui et al. 2019] Cui, Y., K. Morikuni, T. Tsuchiya, and K. Hayami (2019). “Implementation of Interior-Point Methods for LP Based on Krylov Subspace Iterative Solvers with Inner-Iteration Preconditioning”. In: *Computational Optimization and Applications* 74.1, pp. 143–176.

[Dally et al. 2020] Dally, W. J., Y. Turakhia, and S. Han (June 2020). “Domain-Specific Hardware Accelerators”. en. In: *Communications of the ACM* 63.7, pp. 48–57. issn: 0001-0782, 1557-7317. doi: 10.1145/3361682.

[Dalton et al. 2015] Dalton, S., L. Olson, and N. Bell (2015). “Optimizing Sparse Matrix—Matrix Multiplication for the Gpu”. In: *ACM Transactions on Mathematical Software (TOMS)* 41.4, pp. 1–20.

[Davis 2006] Davis, T. A. (2006). *Direct Methods for Sparse Linear Systems*. SIAM.

[Davis and Duff 1997] Davis, T. A. and I. S. Duff (1997). “An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization”. In: *SIAM Journal on Matrix Analysis and Applications* 18.1, pp. 140–158.

[Davis and Hu 2011] Davis, T. A. and Y. Hu (2011). “The University of Florida Sparse Matrix Collection”. In: *ACM Transactions on Mathematical Software (TOMS)* 38.1, pp. 1–25.

[Demmel et al. 1999a] Demmel, J. W., S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. Liu (1999a). “A Supernodal Approach to Sparse Partial Pivoting”. In: *SIAM Journal on Matrix Analysis and Applications* 20.3, pp. 720–755.

[Demmel et al. 1999b] Demmel, J. W., J. R. Gilbert, and X. S. Li (1999b). “An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination”. In: *SIAM Journal on Matrix Analysis and Applications* 20.4, pp. 915–952.

[Demmel 1991] Demmel, J. (1991). “LAPACK: A Portable Linear Algebra Library for High-Performance Computers”. en. In: *Concurrency: Practice and Experience* 3.6, pp. 655–666. issn: 1096-9128. doi: 10.1002/cpe.4330030610.

[Deng et al. 2009] Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei (2009). “Imagenet: A Large-Scale Hierarchical Image Database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. Ieee, pp. 248–255.

[Devlin et al. 2018] Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova (2018). “Bert: Pre-Training of Deep Bidirectional Transformers for Language Understanding”. In: *arXiv preprint arXiv:1810.04805*. arXiv: 1810.04805.

[Diamos et al. n.d.] Diamos, G., A. Kerr, and M. Kesavan (n.d.). “Translating GPU Binaries to Tiered SIMD Architectures with Ocelot”. en. In: (), p. 14.

[Diaz et al. 2012] Diaz, J., C. Munoz-Caro, and A. Nino (2012). “A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era”. In: *IEEE Transactions on parallel and distributed systems* 23.8, pp. 1369–1386.

[Dongarra et al. 1988] Dongarra, J. J., J. Du Croz, S. Hammarling, and R. J. Hanson (1988). “An Extended Set of FORTRAN Basic Linear Algebra Subprograms.” In: *ACM Trans. Math. Softw.* 14.1, pp. 1–17.

[Dongarra et al. 2003] Dongarra, J. J., P. Luszczek, and A. Petitet (2003). “The LINPACK Benchmark: Past, Present and Future”. In: *Concurrency and Computation: practice and experience* 15.9, pp. 803–820.

[Dongarra et al. 1979] Dongarra, J. J., C. B. Moler, J. R. Bunch, and G. W. Stewart (1979). *LINPACK Users' Guide*. SIAM.

[Du et al. 2012] Du, P., R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra (2012). “From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-Platform GPU Programming”. In: *Parallel Computing* 38.8, pp. 391–407.

[Duff and Koster 2001] Duff, I. S. and J. Koster (Jan. 2001). “On Algorithms For Permuting Large Entries to the Diagonal of a Sparse Matrix”. en. In: *SIAM Journal on Matrix Analysis and Applications* 22.4, pp. 973–996. issn: 0895-4798, 1095-7162. doi: 10.1137/S0895479899358443.

[Duff and Reid 1983] Duff, I. S. and J. K. Reid (1983). “The Multifrontal Solution of Indefinite Sparse Symmetric Linear”. In: *ACM Transactions on Mathematical Software (TOMS)* 9.3, pp. 302–325.

[Flegar and Anzt 2017] Flegar, G. and H. Anzt (2017). “Overcoming Load Imbalance for Irregular Sparse Matrices”. en. In: *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms - IA3'17*. Denver, CO, USA: ACM Press, pp. 1–8. isbn: 978-1-4503-5136-2. doi: 10.1145/3149704.3149767.

[Fletcher and Reeves 1964] Fletcher, R. and C. M. Reeves (1964). “Function Minimization by Conjugate Gradients”. In: *The computer journal* 7.2, pp. 149–154.

[Flynn 1972] Flynn, M. J. (1972). “Some Computer Organizations and Their Effectiveness”. In: *IEEE transactions on computers* 100.9, pp. 948–960.

[Foster 1997] Foster, L. V. (1997). “The Growth Factor and Efficiency of Gaussian Elimination with Rook Pivoting”. In: *Journal of Computational and Applied Mathematics* 86.1, pp. 177–194.

[Freund and Jarre 1997] Freund, R. W. and F. Jarre (1997). “A QMR-Based Interior-Point Algorithm for Solving Linear Programs”. In: *Mathematical Programming* 76.1, pp. 183–210.

[Freund and Nachtigal 1995] Freund, R. W. and N. M. Nachtigal (Dec. 1995). "Software for Simplified Lanczos and QMR Algorithms". en. In: *Applied Numerical Mathematics* 19.3, pp. 319–341. issn: 01689274. doi: 10.1016/0168-9274(95)00089-5.

[Frey et al. 2012] Frey, S., G. Reina, and T. Ertl (Feb. 2012). "SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms". en. In: *2012 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. Munich, Germany: IEEE, pp. 399–406. isbn: 978-1-4673-0226-5. doi: 10.1109/PDP.2012.62.

[Fung and Aamodt 2011] Fung, W. W. L. and T. M. Aamodt (Feb. 2011). "Thread Block Compaction for Efficient SIMT Control Flow". en. In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. San Antonio, TX, USA: IEEE, pp. 25–36. isbn: 978-1-4244-9432-3. doi: 10.1109/HPCA.2011.5749714.

[Fung et al. 2009a] Fung, W. W., I. Sham, G. Yuan, and T. M. Aamodt (2009a). "Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 6.2, pp. 1–37.

[Fung et al. 2009b] Fung, W. W., I. Sham, G. Yuan, and T. M. Aamodt (2009b). "Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 6.2, pp. 1–37.

[Gale et al. 2020] Gale, T., M. Zaharia, C. Young, and E. Elsen (Nov. 2020). "Sparse GPU Kernels for Deep Learning". In: *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 219–232. doi: 10.1109/SC41405.2020.00021.

[Garland and Kirk 2010] Garland, M. and D. B. Kirk (2010). "Understanding Throughput-Oriented Architectures". In: *Communications of the ACM* 53.11, pp. 58–66.

[Garland et al. 2008] Garland, M., S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov (2008). "Parallel Computing Experiences with CUDA". In: *IEEE micro* 28.4, pp. 13–27.

[Geist and Ng 1989] Geist, G. A. and E. Ng (1989). "Task Scheduling for Parallel Sparse Cholesky Factorization". In: *International Journal of Parallel Programming* 18.4, pp. 291–314.

[Gennady and Evgenii 2017] Gennady, F. and C. Evgenii (Aug. 2017). *Intel® Math Kernel Library - Introducing Vectorized Compact Routines*. https://software.intel.com/content/www/us/en/develop/articles/intelr-math-kernel-library-introducing-vectorized-compact-routines.html.

[George and Heath 1980] George, A. and M. T. Heath (1980). "Solution of Sparse Linear Least Squares Problems Using Givens Rotations". In: *Linear Algebra and its applications* 34, pp. 69–83.

[Gilbert and Schreiber 1992] Gilbert, J. R. and R. Schreiber (1992). "Highly Parallel Sparse Cholesky Factorization". In: *SIAM Journal on Scientific and Statistical Computing* 13.5, pp. 1151–1172.

[Golub and Ye 1999] Golub, G. H. and Q. Ye (Jan. 1999). "Inexact Preconditioned Conjugate Gradient Method with Inner-Outer Iteration". en. In: *SIAM Journal on Scientific Computing* 21.4, pp. 1305–1320. issn: 1064-8275, 1095-7197. doi: 10.1137/S1064827597323415.

[Goodfellow et al. 2016] Goodfellow, I., Y. Bengio, A. Courville, and Y. Bengio (2016). *Deep Learning*. Vol. 1. MIT press Cambridge.

[Götz and Anzt 2018] Götz, M. and H. Anzt (2018). “Machine Learning-Aided Numerical Linear Algebra: Convolutional Neural Networks for the Efficient Preconditioner Generation”. In: *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*. IEEE, pp. 49–56.

[Graphcore Ltd 2018] Graphcore Ltd (2018). *C2 IPU*.

[Greif et al. 2017] Greif, C., S. He, and P. Liu (July 2017). “SYM-ILDL: Incomplete LDLt Factorization of Symmetric Indefinite and Skew-Symmetric Matrices”. en. In: *ACM Transactions on Mathematical Software* 44.1, pp. 1–21. issn: 0098-3500, 1557-7295. doi: `10.1145/3054948`.

[Griebl et al. 1998] Griebl, M., C. Lengauer, and S. Wetzel (1998). “Code Generation in the Polytope Model”. In: *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*. IEEE, pp. 106–111.

[Grihon et al. 2009] Grihon, S., L. Krog, and D. Bassir (2009). “Numerical Optimization Applied to Structure Sizing at AIRBUS: A Multi-Step Process”. In: *International Journal for Simulation and Multidisciplinary Design Optimization* 3.4, pp. 432–442.

[Grützmacher et al. 2020] Grützmacher, T., T. Cojean, G. Flegar, F. Göbel, and H. Anzt (2020). “A Customized Precision Format Based on Mantissa Segmentation for Accelerating Sparse Linear Algebra”. In: *Concurrency and Computation: Practice and Experience* 32.15, e5418.

[Gulwani et al. 2017] Gulwani, S., O. Polozov, and R. Singh (2017). “Program Synthesis”. In: *Foundations and Trends® in Programming Languages* 4.1-2, pp. 1–119.

[Guthe and Thuerck 2020] Guthe, S. and D. Thuerck (2020). “Algorithm XXX: A Fast Scalable Solver for the Dense Linear (Sum) Assignment Problem”. In: *ACM Transactions on Mathematical Software (TOMS)* (Accepted).

[Habana 2019] Habana (2019). *GOYA and GAUDI Engines*.

[Hagemann and Schenk 2006] Hagemann, M. and O. Schenk (2006). “Weighted Matchings for Preconditioning Symmetric Indefinite Linear Systems”. In: *SIAM Journal on Scientific Computing* 28.2, pp. 403–420.

[Hähnle 2019] Hähnle, N. (Oct. 2019). *D68994 [RFC] Redefine ‘convergent‘ in Terms of Dynamic Instances*. https://reviews.llvm.org/D68994.

[Haidar et al. 2017] Haidar, A., A. Abdelfatah, S. Tomov, and J. Dongarra (2017). “High-Performance Cholesky Factorization for GPU-Only Execution”. en. In: *Proceedings of the General Purpose GPUs on - GPGPU-10*. Austin, TX, USA: ACM Press, pp. 42–52. isbn: 978-1-4503-4915-4. doi: `10.1145/3038228.3038237`.

[Haidl et al. 2017] Haidl, M., S. Moll, L. Klein, H. Sun, S. Hack, and S. Gorlatch (2017). “Pacxxv2+ RV: An LLVM-Based Portable High-Performance Programming Model”. In: *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–12.

[Haj-Ali et al. 2020] Haj-Ali, A., N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica (2020). “NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning”. In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pp. 242–255.

[Han et al. 2015] Han, S., J. Pool, J. Tran, and W. Dally (2015). “Learning Both Weights and Connections for Efficient Neural Network”. en. In: p. 9.

[Hazelwood et al. 2018] Hazelwood, K., S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang (Feb. 2018). "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 620–629. doi: 10.1109/HPCA.2018.00059.

[He et al. 2020] He, X., S. Pal, A. Amarnath, S. Feng, D.-H. Park, A. Rovinski, H. Ye, Y. Chen, R. Dreslinski, and T. Mudge (June 2020). "Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices". In: *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20. Barcelona, Spain: Association for Computing Machinery, pp. 1–12. isbn: 978-1-4503-7983-0. doi: 10.1145/3392717.3392751.

[Hennessy and Patterson 2019] Hennessy, J. L. and D. A. Patterson (Jan. 2019). "A New Golden Age for Computer Architecture". In: *Communications of the ACM* 62.2, pp. 48–60. issn: 0001-0782. doi: 10.1145/3282307.

[Hénon et al. 2002] Hénon, P., P. Ramet, and J. Roman (Feb. 2002). "PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Positive Definite Systems". en. In: *Parallel Computing* 28.2, pp. 301–321. issn: 0167-8191. doi: 10.1016/S0167-8191(01)00141-7.

[Hestenes 1956] Hestenes, M. R. (1956). "The Conjugate Gradient Method for Solving Linear Systems". In: *Proc. Symp. Appl. Math VI, American Mathematical Society*, pp. 83–102.

[Hogg et al. 2016] Hogg, J. D., E. Ovtchinnikov, and J. A. Scott (Mar. 2016). "A Sparse Symmetric Indefinite Direct Solver for GPU Architectures". en. In: *ACM Transactions on Mathematical Software* 42.1, pp. 1–25. issn: 0098-3500, 1557-7295. doi: 10.1145/2756548.

[Hogg et al. 2010] Hogg, J. D., J. K. Reid, and J. A. Scott (2010). "Design of a Multicore Sparse Cholesky Factorization Using DAGs". In: *SIAM Journal on Scientific Computing* 32.6, pp. 3627–3649.

[Hogg and Scott 2015] Hogg, J. and J. Scott (Aug. 2015). "On the Use of Suboptimal Matchings for Scaling and Ordering Sparse Symmetric Matrices: SCALING OF SPARSE SYMMETRIC MATRICES USING AN AUCTION ALGORITHM". en. In: *Numerical Linear Algebra with Applications* 22.4, pp. 648–663. issn: 10705325. doi: 10.1002/nla.1978.

[Hogg et al. 2017] Hogg, J., J. Scott, and S. Thorne (Oct. 2017). "Numerically Aware Orderings for Sparse Symmetric Indefinite Linear Systems". en. In: *ACM Transactions on Mathematical Software* 44.2, pp. 1–22. issn: 0098-3500, 1557-7295. doi: 10.1145/3104991.

[Hong et al. 2011] Hong, S., S. K. Kim, T. Oguntebi, and K. Olukotun (2011). "Accelerating CUDA Graph Algorithms at Maximum Warp". In: *Acm Sigplan Notices* 46.8, pp. 267–276.

[Hopf and Ertl 1999] Hopf, M. and T. Ertl (1999). "Accelerating 3D Convolution Using Graphics Hardware". In: *Proceedings Visualization'99 (Cat. No. 99CB37067)*. IEEE, pp. 471–564.

[Householder 1958] Householder, A. S. (1958). "Unitary Triangularization of a Nonsymmetric Matrix". In: *Journal of the ACM (JACM)* 5.4, pp. 339–342.

[Huzaifa et al. 2020] Huzaifa, M., J. Alsop, A. Mahmoud, G. Salvador, M. D. Sinclair, and S. V. Adve (Aug. 2020). "Inter-Kernel Reuse-Aware Thread Block Scheduling". en. In: *ACM Transactions on Architecture and Code Optimization* 17.3, pp. 1–27. issn: 1544-3566, 1544-3973. doi: 10.1145/3406538.

[Hysom and Pothen 2002] Hysom, D. and A. Pothen (2002). *Level-Based Incomplete LU Factorization: Graph Model and Algorithms*. Tech. rep.

[Jia et al. 2019] Jia, Z., B. Tillman, M. Maggioni, and D. P. Scarpazza (Dec. 2019). "Dissecting the Graphcore IPU Architecture via Microbenchmarking". In: *arXiv:1912.03413 [cs]*. arXiv: 1912.03413 [cs].

[Jiang et al. 2020] Jiang, P., C. Hong, and G. Agrawal (Feb. 2020). "A Novel Data Transformation and Execution Strategy for Accelerating Sparse Matrix Multiplication on GPUs". In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '20. New York, NY, USA: Association for Computing Machinery, pp. 376–388. isbn: 978-1-4503-6818-6. doi: 10.1145/3332466.3374546.

[Jouppi et al. 2020] Jouppi, N. P., D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson (June 2020). "A Domain-Specific Supercomputer for Training Deep Neural Networks". In: *Communications of the ACM* 63.7, pp. 67–78. issn: 0001-0782. doi: 10.1145/3360307.

[Jouppi et al. 2017] Jouppi, N. P., C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon (June 2017). "In-Datacenter Performance Analysis of a Tensor Processing Unit". en. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. Toronto ON Canada: ACM, pp. 1–12. isbn: 978-1-4503-4892-8. doi: 10.1145/3079856.3080246.

[Joyner et al. 2012] Joyner, D., O. Čertík, A. Meurer, and B. E. Granger (Jan. 2012). "Open Source Computer Algebra Systems: SymPy". In: *ACM Communications in Computer Algebra* 45.3/4, pp. 225–234. issn: 1932-2240. doi: 10.1145/2110170.2110185.

[Karmarkar 1984] Karmarkar, N. (1984). "A New Polynomial-Time Algorithm for Linear Programming". In: *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, pp. 302–311.

[Karrenberg 2015] Karrenberg, R. (2015). "Whole-Function Vectorization". In: *Automatic SIMD Vectorization of SSA-Based Control Flow Graphs*. Springer, pp. 85–125.

[Karypis 1998] Karypis, G. (1998). "METIS, a Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0". In: *http://glaros. dtc. umn. edu/gkhome/metis/metis/download*.

[Kessenich et al. 2018] Kessenich, J., B. Ouriel, and R. Krisch (2018). "SPIR-V Specification". In: *Khronos Group* 3.

[Khachiyan 1980] Khachiyan, L. G. (1980). "Polynomial Algorithms in Linear Programming". In: *USSR Computational Mathematics and Mathematical Physics* 20.1, pp. 53–72.

[Kim et al. 2019] Kim, J., J. Cha, J. J. K. Park, D. Jeon, and Y. Park (Jan. 2019). "Improving GPU Multitasking Efficiency Using Dynamic Resource Sharing". In: *IEEE Computer Architecture Letters* 18.1, pp. 1–5. issn: 1556-6064. doi: 10.1109/LCA.2018.2889042.

[Kingma and Ba 2017] Kingma, D. P. and J. Ba (Jan. 2017). "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]*. arXiv: 1412.6980 [cs].

[Koch et al. 2011] Koch, T., T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, and S. Heinz (2011). “MIPLIB 2010”. In: *Mathematical Programming Computation* 3.2, p. 103.

[Kocher et al. 2019] Kocher, P., J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, and T. Prescher (2019). “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 1–19.

[Koric et al. 2014] Koric, S., Q. Lu, and E. Guleryuz (2014). “Evaluation of Massively Parallel Linear Sparse Solvers on Unstructured Finite Element Meshes”. In: *Computers & Structures* 141, pp. 19–25.

[Krizhevsky et al. 2017] Krizhevsky, A., I. Sutskever, and G. E. Hinton (2017). “Imagenet Classification with Deep Convolutional Neural Networks”. In: *Communications of the ACM* 60.6, pp. 84–90.

[Lambert et al. 2018] Lambert, J., S. Lee, J. Kim, J. S. Vetter, and A. D. Malony (June 2018). “Directive-Based, High-Level Programming and Optimizations for High-Performance Computing with FPGAs”. In: *Proceedings of the 2018 International Conference on Supercomputing*. ICS ’18. New York, NY, USA: Association for Computing Machinery, pp. 160–171. isbn: 978-1-4503-5783-8. doi: `10.1145/3205289.3205324`.

[Lamiraux et al. 2005] Lamiraux, F., J.-P. Laumond, C. Van Geem, D. Boutonnet, and G. Raust (2005). “Trailer Truck Trajectory Optimization: The Transportation of Components for the Airbus A380”. In: *IEEE robotics & automation magazine* 12.1, pp. 14–21.

[Lanczos 1950] Lanczos, C. (1950). *An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators*. United States Governm. Press Office Los Angeles, CA.

[Larsen and McAllister 2001] Larsen, E. S. and D. McAllister (2001). “Fast Matrix Multiplies Using Graphics Hardware”. In: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pp. 55–55.

[Lattner and Adve 2004] Lattner, C. and V. Adve (2004). “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, pp. 75–86.

[Lattner et al. 2020] Lattner, C., M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko (Feb. 2020). “MLIR: A Compiler Infrastructure for the End of Moore’s Law”. In: *arXiv:2002.11054 [cs]*. arXiv: `2002.11054 [cs]`.

[Lawson et al. 1979] Lawson, C. L., R. J. Hanson, D. R. Kincaid, and F. T. Krogh (1979). “Basic Linear Algebra Subprograms for Fortran Usage”. In: *ACM Transactions on Mathematical Software (TOMS)* 5.3, pp. 308–323.

[LeCun et al. 2015] LeCun, Y., Y. Bengio, and G. Hinton (2015). “Deep Learning”. In: *nature* 521.7553, pp. 436–444.

[Leißa et al. 2018] Leißa, R., K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt (2018). “AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA, pp. 1–30.

[Leißa et al. 2012] Leißa, R., S. Hack, and I. Wald (2012). “Extending a C-like Language for Portable SIMD Programming”. In: *ACM SIGPLAN Notices* 47.8, pp. 65–74.

[Leißa et al. 2014] Leißa, R., I. Haffner, and S. Hack (2014). “Sierra: A SIMD Extension for C++”. In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, pp. 17–24.

[Lemaitre et al. 2018] Lemaitre, F., B. Couturier, and L. Lacassagne (2018). “Small SIMD Matrices for CERN High Throughput Computing”. In: *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, pp. 1–8.

[Li et al. 2018] Li, A., W. Liu, L. Wang, K. Barker, and S. L. Song (2018). “Warp-Consolidation: A Novel Execution Model for Gpus”. In: *Proceedings of the 2018 International Conference on Supercomputing*, pp. 53–64.

[Li and Shao 2011] Li, X. S. and M. Shao (Feb. 2011). “A Supernodal Approach to Incomplete LU Factorization with Partial Pivoting”. en. In: *ACM Transactions on Mathematical Software* 37.4, pp. 1–20. issn: 0098-3500, 1557-7295. doi: `10.1145/1916461.1916467`.

[Lipp et al. 2018] Lipp, M., M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, and D. Genkin (2018). “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*, pp. 973–990.

[Liu et al. 2018] Liu, C., B. Xie, X. Liu, W. Xue, H. Yang, and X. Liu (2018). “Towards Efficient SpMV on Sunway Manycore Architectures”. In: *Proceedings of the 2018 International Conference on Supercomputing*, pp. 363–373.

[Liu 1986a] Liu, J. W. (1986a). “A Compact Row Storage Scheme for Cholesky Factors Using Elimination Trees”. In: *ACM Transactions on Mathematical Software (TOMS)* 12.2, pp. 127–148.

[Liu 1986b] Liu, J. W. (1986b). “Computational Models and Task Scheduling for Parallel Sparse Cholesky Factorization”. In: *Parallel computing* 3.4, pp. 327–342.

[Liu and Sherman 1976] Liu, W.-H. and A. H. Sherman (1976). “Comparative Analysis of the Cuthill–McKee and the Reverse Cuthill–McKee Ordering Algorithms for Sparse Matrices”. In: *SIAM Journal on Numerical Analysis* 13.2, pp. 198–213.

[Lloyd and Ramanathan 1992] Lloyd, E. L. and S. Ramanathan (1992). “On the Complexity of Distance-2 Coloring”. In: *1992 Fourth International Conference on Computing and Information*. IEEE Computer Society, pp. 71–72.

[Lopez et al. 2016] Lopez, M. G., V. V. Larrea, W. Joubert, O. Hernandez, A. Haidar, S. Tomov, and J. Dongarra (2016). “Towards Achieving Performance Portability Using Directives for Accelerators”. In: *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*. IEEE, pp. 13–24.

[Lorie and Strong Jr 1984] Lorie, R. A. and H. R. Strong Jr (Mar. 1984). “Method for Conditional Branch Execution in SIMD Vector Processors”. Pat.

[Lukarski et al. 2014] Lukarski, D., H. Anzt, S. Tomov, and J. Dongarra (2014). “Hybrid Multi-Elimination ILU Preconditioners on GPUs”. In: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, pp. 7–16.

[Luna and Blaschke 2020] Luna, K. and J. Blaschke (Nov. 2020). “Accelerating GMRES with Deep Learning in Real-Time”. In: *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) - Posters*.

[Matteis et al. 2020] Matteis, T., J. Licht, and T. Hoefler (Nov. 2020). “FBLAS: Streaming Linear Algebra on FPGA”. In: *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 821–833. doi: `10.1109/SC41405.2020.00063`.

[Merrill and Garland 2016] Merrill, D. and M. Garland (2016). "Merge-Based Parallel Sparse Matrix-Vector Multiplication". In: *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, pp. 678–689.

[Microsoft Research 2010] Microsoft Research (2010). *Project Catapult*. en-US.

[Mohammadi et al. 2019] Mohammadi, M. S., T. Yuki, K. Cheshmi, E. C. Davis, M. Hall, M. M. Dehnavi, P. Nandy, C. Olschanowsky, A. Venkat, and M. M. Strout (2019). "Sparse Computation Data Dependence Simplification for Efficient Compiler-Generated Inspectors". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 594–609.

[Molina et al. 2019] Molina, A., P. Schramowski, and K. Kersting (2019). "Padé Activation Units: End-to-End Learning of Flexible Activation Functions in Deep Networks". In: *arXiv preprint arXiv:1907.06732*. arXiv: 1907.06732.

[Moll 2019] Moll, S. (Jan. 2019). *D57504 [RFC]: Prototype & Roadmap for Vector Predication in LLVM*. https://reviews.llvm.org/D57504.

[Moll et al. 2019] Moll, S., S. Sharma, M. Kurtenacker, and S. Hack (2019). "Multi-Dimensional Vectorization in LLVM". In: *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing*, pp. 1–8.

[Morton and Mayers 2005] Morton, K. W. and D. F. Mayers (2005). *Numerical Solution of Partial Differential Equations: An Introduction*. Cambridge university press.

[Moshfegh and Vouvakis 2017] Moshfegh, J. and M. N. Vouvakis (Sept. 2017). "Direct Solution of FEM Models: Are Sparse Direct Solvers the Best Strategy?" In: *2017 International Conference on Electromagnetics in Advanced Applications (ICEAA)*, pp. 1636–1638. doi: 10.1109/ICEAA.2017.8065603.

[Müller and Vignaux n.d.] Müller, K. G. and T. Vignaux (n.d.). *Simpy at Github*. https://github.com/cristiklein/simpy.

[NEC Corporation 2019] NEC Corporation (Mar. 2019). *SX-Aurora TSUBASA Architecture Guide (v1.1)*. en.

[NEC Corporation 2020] NEC Corporation (Oct. 2020). *SX-Aurora TSUBASA C/C++ Compiler User's Guide (V20)*.

[NVIDIA Corporation 2017] NVIDIA Corporation (Aug. 2017). *NVIDIA Tesla V100 GPU Architecture*. Tech. rep. WP-08608-001_v1.1.

[NVIDIA Corporation 2020a] NVIDIA Corporation (Oct. 2020a). *CUDA C++ Programming Guide (V11.1.1)*. en-us. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[NVIDIA Corporation 2020b] NVIDIA Corporation (2020b). *NVIDIA A100 Tensor Core GPU Architecture*. Tech. rep. v1.0.

[NVIDIA Corporation 2020c] NVIDIA Corporation (Oct. 2020c). *Parallel Thread Execution ISA 7.1 (V11.1.1)*. en-us. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Concept.

[Naumov 2011] Naumov, M. (2011). *Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU*. Tech. rep.

[Naumov et al. 2015] Naumov, M., P. Castonguay, and J. Cohen (2015). "Parallel Graph Coloring with Applications to the Incomplete-LU Factorization on the GPU". In: *Nvidia White Paper*.

[Neal and Poole 1992] Neal, L. and G. Poole (1992). “A Geometric Analysis of Gaussian Elimination. II”. In: *Linear algebra and its applications* 173, pp. 239–264.

[Nocedal and Wright 2006] Nocedal, J. and S. Wright (2006). *Numerical Optimization*. Springer Science & Business Media.

[Nuzman et al. 2011] Nuzman, D., S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks (2011). “Vapor SIMD: Auto-Vectorize Once, Run Everywhere”. In: *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, pp. 151–160.

[Nvidia 2014] Nvidia (2014). “Cusparse Library”. In: *NVIDIA Corporation, Santa Clara, California*.

[Nvidia 2008] Nvidia, C. (2008). “Cublas Library”. In: *NVIDIA Corporation, Santa Clara, California* 15.27, p. 31.

[Paige and Saunders 1975] Paige, C. C. and M. A. Saunders (1975). “Solution of Sparse Indefinite Systems of Linear Equations”. In: *SIAM journal on numerical analysis* 12.4, pp. 617–629.

[Parter 1961] Parter, S. (1961). “The Use of Linear Graphs in Gauss Elimination”. In: *SIAM review* 3.2, pp. 119–130.

[Paszke et al. 2019] Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, and L. Antiga (2019). “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems*, pp. 8024–8035.

[Peters and Wilkinson 1975] Peters, G. and J. H. Wilkinson (1975). “On the Stability of Gauss-Jordan Elimination with Pivoting”. In: *Communications of the ACM* 18.1, pp. 20–24.

[Pharr and Mark 2012] Pharr, M. and W. R. Mark (2012). “Ispc: A SPMD Compiler for High-Performance CPU Programming”. In: *2012 Innovative Parallel Computing (InPar)*. IEEE, pp. 1–13.

[Pizzuti et al. 2020] Pizzuti, F., M. Steuwer, and C. Dubach (Feb. 2020). “Generating Fast Sparse Matrix Vector Multiplication from a High Level Generic Functional IR”. In: *Proceedings of the 29th International Conference on Compiler Construction*. CC 2020. New York, NY, USA: Association for Computing Machinery, pp. 85–95. isbn: 978-1-4503-7120-9. doi: `10.1145/3377555.3377896`.

[Polok et al. 2013] Polok, L., V. Ila, and P. Smrz (2013). “Cache Efficient Implementation for Block Matrix Operations”. In: *Proceedings of the High Performance Computing Symposium*, pp. 1–8.

[Poole and Neal 1991] Poole, G. and L. Neal (1991). “A Geometric Analysis of Gaussian Elimination. I”. In: *Linear algebra and its applications* 149, pp. 249–272.

[Rajopadhye 1988] Rajopadhye, S. V. (June 1988). “Systolic Arrays for LU Decomposition”. In: *1988., IEEE International Symposium on Circuits and Systems*, 2513–2516 vol.3. doi: `10.1109/ISCAS.1988.15453`.

[Reuther et al. 2019] Reuther, A., P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner (Sept. 2019). “Survey and Benchmarking of Machine Learning Accelerators”. In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9. doi: `10.1109/HPEC.2019.8916327`.

[Reuther et al. 2020] Reuther, A., P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner (2020). “Survey of Machine Learning Accelerators”. In: *arXiv preprint arXiv:2009.00993*. arXiv: `2009.00993`.

[Rodrigues et al. 2018] Rodrigues, C., A. Phaosawasdi, and P. Wu (2018). "Simdization of Small Tensor Multiplication Kernels for Wide SIMD Vector Processors". In: *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, pp. 1–8.

[Rotman 2020] Rotman, D. (Jan. 2020). "Were Not Prepared for the End of Moore's Law". In: *MIT Technology Review*.

[Saad 1993] Saad, Y. (Mar. 1993). "A Flexible Inner-Outer Preconditioned GMRES Algorithm". en. In: *SIAM Journal on Scientific Computing* 14.2, pp. 461–469. issn: 1064-8275, 1095-7197. doi: 10.1137/0914028.

[Saad 1994] Saad, Y. (1994). "ILUT: A Dual Threshold Incomplete LU Factorization". In: *Numerical linear algebra with applications* 1.4, pp. 387–402.

[Saad 2003a] Saad, Y. (Jan. 2003a). "Finding Exact and Approximate Block Structures for ILU Preconditioning". en. In: *SIAM Journal on Scientific Computing* 24.4, pp. 1107–1123. issn: 1064-8275, 1095-7197. doi: 10.1137/S1064827501393393.

[Saad 2003b] Saad, Y. (2003b). *Iterative Methods for Sparse Linear Systems*. SIAM.

[Saad and Zhang 1999] Saad, Y. and J. Zhang (1999). "BILUM: Block Versions of Multielimination and Multilevel ILU Preconditioner for General Sparse Linear Systems". In: *SIAM Journal on Scientific Computing* 20.6, pp. 2103–2121.

[Saltz et al. 1990] Saltz, J., K. Crowley, R. Michandaney, and H. Berryman (1990). "Run-Time Scheduling and Execution of Loops on Message Passing Machines". In: *Journal of Parallel and Distributed Computing* 8.4, pp. 303–312.

[Schenk and Gärtner 2006] Schenk, O. and K. Gärtner (2006). "On Fast Factorization Pivoting Methods for Sparse Symmetric Indefinite Systems". In: *Electronic Transactions on Numerical Analysis* 23.1, pp. 158–179.

[Schenk et al. 2001] Schenk, O., K. Gärtner, W. Fichtner, and A. Stricker (2001). "PARDISO: A High-Performance Serial and Parallel Sparse Linear Solver in Semiconductor Device Simulation". In: *Future Generation Computer Systems* 18.1, pp. 69–78.

[Schork and Gondzio 2020] Schork, L. and J. Gondzio (Feb. 2020). "Implementation of an Interior Point Method with Basis Preconditioning". en. In: *Mathematical Programming Computation*. issn: 1867-2949, 1867-2957. doi: 10.1007/s12532-020-00181-8.

[Schreiber 1982] Schreiber, R. (1982). "A New Implementation of Sparse Gaussian Elimination". In: *ACM Transactions on Mathematical Software (TOMS)* 8.3, pp. 256–276.

[Schrijver 1998] Schrijver, A. (1998). *Theory of Linear and Integer Programming*. John Wiley & Sons.

[Scott and Tůma 2016] Scott, J. and M. Tůma (Jan. 2016). "Preconditioning of Linear Least Squares by Robust Incomplete Factorization for Implicitly Held Normal Equations". en. In: *SIAM Journal on Scientific Computing* 38.6, pp. C603–C623. issn: 1064-8275, 1095-7197. doi: 10.1137/16M105890X.

[Shar and Davidson 1974] Shar, L. E. and E. S. Davidson (1974). "A Multiminiprocessor System Implemented through Pipelining". In: *Computer* 7.2, pp. 42–51.

[Shen and Lipasti 2013] Shen, J. P. and M. H. Lipasti (2013). *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press.

[Shin et al. 2002] Shin, J., J. Chame, and M. Hall (Sept. 2002). “Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures”. In: *Proceedings.International Conference on Parallel Architectures and Compilation Techniques*, pp. 45–55. doi: 10.1109/PACT.2002.1106003.

[Simoncini and Szyld 2002] Simoncini, V. and D. B. Szyld (2002). “Flexible Inner-Outer Krylov Subspace Methods”. In: *SIAM Journal on Numerical Analysis* 40.6, pp. 2219–2239.

[Simoncini and Szyld 2013] Simoncini, V. and D. B. Szyld (2013). “On the Superlinear Convergence of MINRES”. In: *Numerical Mathematics and Advanced Applications 2011*. Springer, pp. 733–740.

[Sistla and Nandivada 2019] Sistla, M. A. and V. K. Nandivada (2019). “Graph Coloring Using GPUs”. In: *European Conference on Parallel Processing*. Springer, pp. 377–390.

[Spampinato et al. 2018] Spampinato, D. G., D. Fabregat-Traver, P. Bientinesi, and M. Püschel (Feb. 2018). “Program Generation for Small-Scale Linear Algebra Applications”. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. New York, NY, USA: Association for Computing Machinery, pp. 327–339. isbn: 978-1-4503-5617-6. doi: 10.1145/3168812.

[Stanic et al. 2017] Stanic, M., O. Palomar, T. Hayes, I. Ratkovic, A. Cristal, O. Unsal, and M. Valero (July 2017). “An Integrated Vector-Scalar Design on an In-Order ARM Core”. en. In: *ACM Transactions on Architecture and Code Optimization* 14.2, pp. 1–26. issn: 1544-3566, 1544-3973. doi: 10.1145/3075618.

[Steuwer et al. 2017] Steuwer, M., T. Remmelg, and C. Dubach (2017). “Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, pp. 74–85.

[Stone et al. 2010] Stone, J. E., D. Gohara, and G. Shi (2010). “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in science & engineering* 12.3, pp. 66–73.

[Strauss 2007] Strauss, W. A. (2007). *Partial Differential Equations: An Introduction*. John Wiley & Sons.

[Strikwerda 2004] Strikwerda, J. C. (2004). *Finite Difference Schemes and Partial Differential Equations*. SIAM.

[Su et al. 2017] Su, X., X. Liao, and J. Xue (2017). “Automatic Generation of Fast BLAS3-GEMM: A Portable Compiler Approach”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, pp. 122–133.

[Subgroup 2015] Subgroup, K. S. (2015). *SYCL Specification, SYCL Integrates OpenCL Devices with Modern C++.(May 2015)*.

[Sun-Yuan Kung 1984] Sun-Yuan Kung (1984). “On Supercomputing with Systolic/Wavefront Array Processors”. In: *Proceedings of the IEEE* 72.7, pp. 867–884. issn: 0018-9219. doi: 10.1109/PROC.1984.12944.

[TOP500 2020] TOP500 (Nov. 2020). *TOP 500*. https://www.top500.org/statistics/list/.

[Thuerck 2019] Thuerck, D. (2019). “Stretching Jacobi: Two-Stage Pivoting in Block-Based Factorization”. In: *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, pp. 51–58.

[Thuerck 2020] Thuerck, D. (2020). “Supporting Irregularity in Throughput-Oriented Computing by SIMT-SIMD Integration”. In: *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE.

[Thuerck and Goesele 2018] Thuerck, D. and M. Goesele (2018). *Lock-Free Parallel Feedback Vertex Set Selection*. Tech. rep. TU Darmstadt.

[Thuerck et al. 2018] Thuerck, D., M. Naumov, M. Garland, and M. Goesele (2018). "A Block-Oriented, Parallel and Collective Approach to Sparse Indefinite Preconditioning on GPUs". In: *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, pp. 1–10.

[Thuerck et al. 2016] Thuerck, D., M. Waechter, S. Widmer, M. von Bülow, P. Seemann, M. E. Pfetsch, and M. Goesele (2016). "A Fast, Massively Parallel Solver for Large, Irregular Pairwise Markov Random Fields." In: *High Performance Graphics*, pp. 173–183.

[Thuerck et al. 2020] Thuerck, D., N. Weber, and R. Bifulco (May 2020). "Flynn's Reconciliation: Automating the Register Cache Idiom for Cross-Accelerator Programming". In: *ACM Transactions on Architecture and Code Optimization (TACO)* (Accepted).

[Tino et al. 2020] Tino, A., C. Collange, and A. Seznec (May 2020). "SIMT-X: Extending Single-Instruction Multi-Threading to Out-of-Order Cores". In: *ACM Transactions on Architecture and Code Optimization* 17.2, 15:1–15:23. issn: 1544-3566. doi: `10.1145/3392032`.

[Tomov et al. 2011] Tomov, S., R. Nath, P. Du, and J. Dongarra (2011). "MAGMA Users' Guide". In: *ICL, UTK (November 2009)*.

[Van Loan and Golub 1983] Van Loan, C. F. and G. H. Golub (1983). *Matrix Computations*. Johns Hopkins University Press Baltimore.

[Vasilache et al. 2019] Vasilache, N., O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen (2019). "The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 16.4, pp. 1–26.

[Venkat et al. 2015] Venkat, A., M. Hall, and M. Strout (2015). "Loop and Data Transformations for Sparse Matrix Code". In: *ACM SIGPLAN Notices* 50.6, pp. 521–532.

[Wächter and Biegler 2006] Wächter, A. and L. T. Biegler (Mar. 2006). "On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming". en. In: *Mathematical Programming* 106.1, pp. 25–57. issn: 0025-5610, 1436-4646. doi: `10.1007/s10107-004-0559-y`.

[Wang et al. 2014] Wang, E., Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang (2014). "Intel Math Kernel Library". In: *High-Performance Computing on the Intel® Xeon Phi™*. Springer, pp. 167–188.

[Weber and Goesele 2017] Weber, N. and M. Goesele (2017). "MATOG: Array Layout Auto-Tuning for CUDA". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 14.3, pp. 1–26.

[Wienke et al. 2012] Wienke, S., P. Springer, C. Terboven, and D. an Mey (2012). "OpenACC—First Experiences with Real-World Applications". In: *European Conference on Parallel Processing*. Springer, pp. 859–870.

[Wilson 1989] Wilson, K. G. (Sept. 1989). "Grand Challenges to Computational Science". en. In: *Future Generation Computer Systems*. Grand Challenges to Computational Science 5.2, pp. 171–189. issn: 0167-739X. doi: `10.1016/0167-739X(89)90038-1`.

[Wu et al. 2016] Wu, J., A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt (2016). "Gpucc: An Open-Source GPGPU Compiler". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pp. 105–116.

[Yamada and Momose 2018] Yamada, Y. and S. Momose (2018). “Vector Engine Processor of NEC’s Brand-New Supercomputer SX-Aurora TSUBASA”. In: *Proceedings of A Symposium on High Performance Chips, Hot Chips*. Vol. 30, pp. 19–21.

[Yang et al. 2019] Yang, C.-C., J. C. Pichel, and D. A. Padua (2019). “Dataflow Execution of Hierarchically Tiled Arrays”. en. In: *Euro-Par 2019: Parallel Processing*. Ed. by R. Yahyapour. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 304–316. isbn: 978-3-030-29400-7. doi: 10.1007/978-3-030-29400-7_22.

[Yang et al. 2014] Yang, Y., P. Xiang, M. Mantor, N. Rubin, L. Hsu, Q. Dong, and H. Zhou (May 2014). “A Case for a Flexible Scalar Unit in SIMT Architecture”. en. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. Phoenix, AZ, USA: IEEE, pp. 93–102. isbn: 978-1-4799-3800-1 978-1-4799-3799-8. doi: 10.1109/IPDPS.2014.21.

[Yannakakis 1981] Yannakakis, M. (1981). “Computing the Minimum Fill-in Is NP-Complete”. In: *SIAM Journal on Algebraic Discrete Methods* 2.1, pp. 77–79.

[Yeom et al. 2016] Yeom, J.-S., J. J. Thiagarajan, A. Bhatele, G. Bronevetsky, and T. Kolev (Nov. 2016). “Data-Driven Performance Modeling of Linear Solvers for Sparse Matrices”. en. In: *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. Salt Lake, UT, USA: IEEE, pp. 32–42. isbn: 978-1-5090-5218-9. doi: 10.1109/PMBS.2016.009.

[Zhang et al. 2017] Zhang, F., B. Wu, J. Zhai, B. He, and W. Chen (2017). “Finepar: Irregularity-Aware Fine-Grained Workload Partitioning on Integrated Architectures”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, pp. 27–38.

[Zhang 1998] Zhang, Y. (1998). “Solving Large-Scale Linear Programs by Interior-Point Methods under the MATLAB Environment”. In: *Optimization Methods and Software* 10.1, pp. 1–31.

[Zhao et al. 2019] Zhao, T., P. Basu, S. Williams, M. Hall, and H. Johansen (2019). “Exploiting Reuse and Vectorization in Blocked Stencil Computations on CPUs and GPUs”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–44.

[Zuckerman et al. 2011] Zuckerman, S., J. Suetterlein, R. Knauerhase, and G. R. Gao (June 2011). “Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper”. In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. EXADAPT ’11. New York, NY, USA: Association for Computing Machinery, pp. 64–69. isbn: 978-1-4503-0708-6. doi: 10.1145/2000417.2000424.

(Co-)Authored Publications

Thuerck, D., N. Weber, and R. Bifulco (May 2020). "Flynn's Reconciliation: Automating the Register Cache Idiom for Cross-Accelerator Programming". In: *ACM Transactions on Architecture and Code Optimization (TACO)* (Accepted).

Guthe, S. and **D. Thuerck** (2020). "Algorithm 1052: A Fast Scalable Solver for the Dense Linear (Sum) Assignment Problem". In: *ACM Transactions on Mathematical Software (TOMS)* (Accepted).

Thuerck, D. (2020). "Supporting Irregularity in Throughput-Oriented Computing by SIMT- SIMD Integration". In: *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE

Thuerck, D. (2019). "Stretching Jacobi: Two-Stage Pivoting in Block-Based Factorization". In: *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE

Thuerck, D., M. Naumov, M. Garland, and M. Goesele (2018). "A Block-Oriented, Parallel and Collective Approach to Sparse Indefinite Preconditioning on GPUs". In: *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE

Thuerck, D. and M. Goesele (2018). Lock-Free Parallel Feedback Vertex Set Selection. Tech. rep. TU Darmstadt

Thuerck, D., M. Waechter, S. Widmer, M. von Bülow, P. Seemann, M. E. Pfetsch, and M. Goesele (2016). "A Fast, Massively Parallel Solver for Large, Irregular Pairwise Markov Random Fields." In: *High Performance Graphics*, pp. 173–183.

Curriculum Vitae

Daniel Thürck, M.Sc.

since 2020	Research Scientist at NEC Laboratories Europe, Heidelberg
2019 - 2020	Research associate and Ph.D. Student at the AIML Group at TU Darmstadt
2015 - 2019	Research associate and Ph.D. Student at the Graphics, Capture and Massively Parallel Computing group at TU Darmstadt
2018 - 2019	5 months research internship at Facebook Reality Labs, Pittsburgh, PA, US
2015 - 2016	10 months research internship at NVIDIA Research, Santa Clara, CA, US
2014 - 2020	Scholarship holder (2014 - 2015), later Associate of the Graduate School of Computational Engineering at TU Darmstadt
2013 - 2015	Master of Science in Computational Engineering at TU Darmstadt
2010 - 2013	Bachelor of Science in Computer Science at TU Darmstadt