2017 46th International Conference on Parallel Processing

Variable-Size Batched LU for Small Matrices and its Integration into Block-Jacobi Preconditioning

Hartwig Anzt[△⋆], Jack Dongarra[⋆†‡]
[△]*Karlsruhe Institute of Technology, Germany*
[⋆]*University of Tennessee, Knoxville, USA*
[†]*Oak Ridge National Laboratory, USA*
[‡]*University of Manchester, United Kingdom*
Emails: {hanzt,dongarra}@icl.utk.edu

Goran Flegar, Enrique S. Quintana-Ortí
Depto. de Ingeniería y Ciencia de Computadores
Universitat Jaume I, Castellón, Spain
Emails: {flegar,quintana}@uji.es

***Abstract*—We present a set of new batched CUDA kernels for the LU factorization of a large collection of independent problems of different size, and the subsequent triangular solves. All kernels heavily exploit the registers of the graphics processing unit (GPU) in order to deliver high performance for small problems. The development of these kernels is motivated by the need for tackling this embarrasingly-parallel scenario in the context of block-Jacobi preconditioning that is relevant for the iterative solution of sparse linear systems.**

***Keywords*-variable-size batched LU, GPU, block-Jacobi**

I. Introduction

The development of batched routines for linear algebra operations has received considerable interest in the past few years because of the hardware concurrency often exceeding the degree of parallelism present in the algorithms. At the same time, many problems arising in astrophysics, quantum chemistry, hydrodynamics, and hyperspectral image processing, among others, require the application of the same computational kernel not only to one but to a large number of data instances. Batched routines are optimized to tackle this embarrassingly-parallel scenario that comprises a large collection of independent problems, each of small dimension. Compared with conventional multi-threaded implementations of the *Basic Linear Algebra Subprograms* (BLAS) [1], optimized for moderate- and large-scale individual problem instances, batched routines may also exploit the parallelism available within the computational kernel, but mainly target the parallelism in-between the distinct data items.

With the increasing core-per-node ratio, there is an urging demand for batched routines, that are eventually expected to cover a significant fraction of the functionality currently supported by dense linear algebra libraries such as BLAS and the *Linear Algebra PACKage* (LAPACK) [2]. In addition, batched kernels are becoming a key ingredient for the solution of sparse linear systems via direct multifrontal solvers as well as for the efficient preconditioning of iterative solvers based on Krylov subspaces.

Preconditioning via a block-Jacobi scheme is a particularly simple technique that produces an effective acceleration of the iterative solve for some problem instances [3]. One option to realize block-Jacobi preconditioning requires to:

1) Extract multiple small-sized diagonal blocks of the sparse coefficient matrix of the linear system and factorize this collection of blocks during the preconditioner computation (setup).
2) Solve the resulting triangular systems during the preconditioner application (once per step of the iterative solve).

As the diagonal blocks are all pairwise independent, preconditioning via block-Jacobi naturally leads to a batched scenario. Our effort in this work is oriented toward elaborating efficient batched routines for these two steps, preconditioner setup and preconditioner application, yielding the following contributions:

- A variable-size batched LU factorization routine on GPUs tuned for small problems.
- A complementary variable-size batched lower and upper triangular solve routine on GPUs tuned for small sizes.
- An implicit pivoting technique that preserves the stability of the factorization without explicitly swapping the matrix elements in memory.
- A routine for efficiently extracting the diagonal blocks from a sparse matrix layout that ensures a good workload balance even for matrices with an unbalanced nonzero distribution.
- A complete block-Jacobi preconditioner ecosystem based on batched LU factorization and batched triangular solves that improves time-to-solution of iterative Krylov solvers for a large set of test problems.

At this point, we note that the term "batched" has been frequently used to refer to a large collection of small-size problems. However, the definition of *large* and *small* are at best blurry. Here we target a realistic case study for block-Jacobi preconditioning, where *small* can be defined as the diagonal blocks that participate in steps 1)–2) above being in the range 4×4 to 32×32, while *large* refers to thousands

2332-5690/17 $31.00 © 2017 IEEE
DOI 10.1109/ICPP.2017.18

or even tens of thousands of independent problems. We consider this as scientifically relevant as the block-Jacobi preconditioner aims at reflecting the sparsity block structure of a finite element discretization.

Due to their small size, processing the problem instances usually involves memory-bound operations, which can question the utilization of discrete accelerators (with a memory detached from that of the host memory). For the particular case of block-Jacobi preconditioning, the use of a discrete hardware accelerator device is justified because the Krylov solvers require the coefficient matrix to reside in the device memory in order to build up the Krylov subspace via the Krylov iterations. Therefore, the cost of transferring this matrix from the host to the accelerator is quickly amortized, and the on-device generation of the batched block-Jacobi preconditioner incurs no additional host-to-device communication.

The rest of the paper is structured as follows. In Section II we review a few related works and offer a brief survey of (batched) factorization for the solution of linear systems. In Section III we describe the implementation of our CUDA kernels for batched LU factorization (paying special attention to the introduction of implicit pivoting), batched triangular system solves, and the extraction procedure. In Section IV we assess the performance of the new batched kernels on an NVIDIA Tesla P100 (Pascal) GPU, and in Section V we close the paper with a discussion of remarks and future work.

II. Background and Related Work

A. Block-Jacobi preconditioning

For a coefficient matrix $A \in \mathbb{R}^{n \times n}$, the block-Jacobi method can be regarded as a straight-forward extension of its (scalar) Jacobi counterpart. Concretely, instead of splitting the coefficient matrix as $A = L + D + U$ (with diagonal $D = (\{a_{ii}\})$, lower triangular $L = (\{a_{ij} \quad : \quad i > j\})$ and upper triangular $U = (\{a_{ij} \quad : \quad i < j\})$), the block-Jacobi variant gathers the diagonal blocks of A into $D = (D_1, D_2, \ldots, D_N)$, $D_i \in \mathbb{R}^{m_i \times m_i}$, $i = 1, 2, \ldots, N$, with $n = \sum_{i=1}^{N} m_i$. (For simplicity, hereafter we assume that all blocks have the same size m, and n is an integer multiple of m.) The remaining elements of A are then partitioned into matrices L and U such that L contains the elements below the diagonal blocks while U comprises those above them [4]. The block-Jacobi method is well-defined if all diagonal blocks are non-singular, and the resulting preconditioner is expected to work effectively if the blocks succeed in reflecting the nonzero structure of the coefficient matrix A.

Fortunately, many linear systems exhibit some inherent block structure, for example because they arise from a finite element discretization of a partial differential equation (PDE), with multiple variables associated to each element [4]. The variables belonging to the same element usually share the same column sparsity pattern, and the set of variables is often referred to as a *supervariable*. *Supervariable blocking* [5] aims to identify variables sharing the same column-nonzero-pattern, and turns this information into a block-structure that can be used, for example, in block-Jacobi preconditioning. Depending on the pre-defined upper bound for the size of the diagonal blocks, multiple supervariables adjacent in the coefficient matrix can be clustered within the same diagonal block [5]. This is particularly efficient if the supervariables accumulated into the same Jacobi block are tightly coupled, which is the case if the variables ordered close-by in the matrix representation belong to elements that are nearby in the PDE mesh. Some reordering techniques such as reverse Cuthill-McKee or natural orderings preserve this locality [5].

Although different strategies exist to integrate a block-Jacobi preconditioner into an iterative solver setting, in this paper we focus on an approach that factorizes the diagonal blocks in the preconditioner setup, and then applies the preconditioner in terms of triangular solves. Alternatively, it is possible to explicitly compute the block-inverse before the iterative solution phase, and apply the preconditioner as a matrix-vector multiplication. These two strategies primarily differ in the workload size, and how this work is distributed between the preconditioner setup and the preconditioner application. Additionally, the factorization-based approach might exhibit more favorable numerical stability as it avoids the explicit inversion of the blocks in D.

B. Solution of linear systems via the LU factorization

The standard procedure to solve a dense linear system $D_i x = b$, for a square block D_i of order m and vectors x, b, each with m entries, consists of the following four steps [6]:

1) The computation of the LU factorization (with partial pivoting) $PD_i = LU$, where L is unit lower triangular, U is upper triangular, P is a permutation matrix, and all three matrices L, U, P are of the same order as D_i;
2) the application of the permutation P to the right-hand side b; i.e., $b := Pb$;
3) the solution of the unit lower triangular system $Ly = b$ for y; and
4) the solution of the upper triangular system $Ux = y$ to obtain the sought-after vector x.

The computational cost of this four-step solution process is $\frac{2}{3}m^3 + \mathcal{O}(m^2)$ flops (floating-point arithmetic operations), where the dominating term $\frac{2}{3}m^3$ comes from the factorization step. Neglecting the pivoting process associated with the permutation matrix P can result in the triangular factors becoming singular and the break-down of the algorithm [6]. Partial pivoting limits the process to row exchanges only; it is numerically stable in practice, and has become the norm in standard implementations of the LU factorization.

C. Batched solution of small linear systems

Batched routines for small-size problems play an important role in the context of preconditioning iterative methods for sparse linear systems. One example is the technique of block-Jacobi preconditioning, where the sparse coefficient matrix is scaled with its block-inverse [3]. This type of scaling requires the solution of a set of small linear systems induced by the diagonal blocks in D, which can be addressed via a factorization-based method. As argued earlier, for each block D_i, of dimension m, the cost of its LU factorization is $\frac{2}{3}m^3$ flops, while solving the triangular block system for a single block and right-hand side requires $2m^2$ flops. Alternatively, as the block-diagonal scaling is applied at each iteration of the solver, it may pay off to explicitly compute the block-inverse before the iteration process, at a cost of $2m^3$ flops (per block). With this approach, the preconditioner application can then be cast in terms of the matrix-vector product, with a cost of $2m^2$ flops (per block), but a much faster execution than a triangular block solve.

These two approaches, factorization-based and inversion-based, differ in the computational cost, numerical stability, and how they distribute the workload between preconditioner setup and preconditioner application. Which strategy is preferrable depends on how often the preconditioner is applied and the size of the distinct diagonal blocks.

In block-Jacobi preconditioning, the diagonal blocks are typically chosen to be of small size, for example when reflecting the block structure of a system matrix coming from a finite element discretization. At the same time, the number of these small blocks is typically large, which motivates the use of batched routines. For GPU architectures, we showed in [4] how to realize an inversion-based block-Jacobi preconditioner efficiently using Gauss-Jordan elimination (GJE). As the explicit inversion may be questionable in terms of numerical stability, in [7] we compared this solution with a block-Jacobi preconditioning procedure based on the factorization of diagonal blocks. The factorization method we used in that comparison was the Gauss-Huard (GH) algorithm [8], which is algorithmically similar to GJE.

In this paper we extend our survey on using batched routines for block-Jacobi preconditioning by addressing the factorization of the diagonal blocks via the mainstream LU factorization. From the numerical perspective, the LU factorization (with partial pivoting) and the GH algorithm (with column pivoting) present the same properties. However, they build upon distinct algorithms, resulting in different implementations, and, consequently, distinguishable computational performance.

III. Design of CUDA Kernels

This section describes the implementation of efficient CUDA kernels for both batched LU factorization as well as triangular system solves specifically tuned for small problem sizes where the system matrix (corresponding to a diagonal block in block-Jacobi preconditioning) contains at most 32×32 elements. This is consistent with the batched implementation of GH in [7], which is also designed for small problems of the type arising in block-Jacobi preconditioning.

```
1  % Input  : m x m nonsingular matrix block Di
2  % Output : Di overwritten by its L, U factors
3  p = [1:m];
4  for k = 1 : m
5    % Pivoting
6    [abs_ipiv, ipiv]      = max(abs(Di(k:m,k)));
7    ipiv                  = ipiv+k-1;
8    [Di(k,:), Di(ipiv,:)] = swap(Di(ipiv,:), Di(k,:));
9    [p(k),   p(ipiv)]     = swap(p(ipiv), p(k));
10
11   % Gauss transformation
12   d                = Di(k,k);                % Pivot
13   Di(k+1:m,k)      = Di(k+1:m,k) / d;        % SCAL
14   Di(k+1:m,k+1:m) = Di(k+1:m,k+1:m) ...
15                    - Di(k+1:m,k) * Di(k,k+1:m); % GER
16 end
```

```
1  % Input  : m x m nonsingular matrix block Di
2  % Output : Di overwritten by its L, U factors
3  p = zeros(1, m);
4  for k = 1 : m
5    % Implicit pivoting
6    abs_vals          = abs(Di(:,k));
7    abs_vals(p>0)     = -1; % exclude pivoted rows
8    [abs_ipiv, ipiv] = max(abs_vals);
9    p(ipiv)           = k;
10
11   % Gauss transformation
12   d                = Di(ipiv,k);             % Pivot
13   Di(p==0,k)       = Di(p==0,k) / d;         % SCAL
14   Di(p==0,k+1:m)  = Di(p==0,k+1:m) ...
15                    - Di(p==0,k) * Di(ipiv,k+1:m); %
                         GER
16 end
17 % Combined row swaps
18 p(p) = 1:m;       % Invert the permutation
19 Di   = Di(p,:);
```

Figure 1. Loop-body of the basic LU factorization in Matlab notation using explicit and implicit partial pivoting (top and bottom, respectively).

A. Batched LU factorization (GETRF)

A simplified MATLAB routine for the (right-looking) LU factorization of a square block `Di` is shown in Figure 1 (top). This algorithmic variant lies in the foundations of our batched CUDA kernel for this factorization that we discuss next.

Recent GPU architectures from NVIDIA feature a large amount of registers per thread, which makes it possible to assign problems of size up to 32×32 to a single warp. Each thread then stores one row of the system matrix `Di` into the local registers, while warp shuffle instructions allow access to elements from other rows (e.g., when performing the updates in lines 13–15). Using this technique, it is possible to read the system matrix only once, and perform the whole factorization process in the registers, avoiding the latency of memory and caches, as well as additional load and store instructions.

A complementary optimization, also important, addresses the pivoting procedure ensuring the practical stability of the

LU factorization. Even though the selection of the pivot row `ipiv` (lines 6–7) in step `k` of the factorization can be realized efficiently using a parallel reduction, the actual exchange of rows `k` and `ipiv` (lines 8–9) on a GPU is costly, as it involves only the two threads holding these rows, while the remaining threads stay idle. To tackle this issue, in [4] and [7] we proposed an implicit pivoting procedure for GJE and GH, which avoids the explicit row swaps and combines them into a single, easily parallelizable permutation, which is performed after the main loop. This technique can also be applied to LU by observing the following:

- During step `k` of the factorization, the operation performed on each row of the matrix (lines 13–15) depends only on the elements in this row and the pivot row `ipiv` (which was exchanged with row `k` when using standard, explicit pivoting as in Figure 1).
- The type of operation to perform on each row can be derived without knowing its position in the matrix: if the row has already been pivoted, then no operation is required (in Figure 1 such row was exchanged with one of the first `k` rows); otherwise the `k`-th element of the row has to be scaled (line 13), and an AXPY needs to be performed on the trailing vector (lines 14–15).

These observations turn the implicit pivoting procedure for LU even more efficient than its counterpart applied in GH, as the operations performed by each thread do not depend on the previously selected pivot rows. Conversely, for GH a list of their indices has to be replicated in each thread [7]. As an illustration, the bottom factorization code in Figure 1 shows the implicit pivoting approach in the LU factorization. A final optimization can be made by combining the row swap with the off-load of L and U to main memory, hereby eliminating all inter-thread communication induced by row swaps. In addition to writing the triangular factors, the pivoting information also has to be stored in main memory for the subsequent triangular solves.

B. Batched triangular system solves (TRSV)

Unlike the LU factorization, the triangular system solves offer only a limited amount of data reuse. Each element of the triangular factors is needed only once, so explicitly keeping this value in registers does not offer any advantage. In contrast, the right-hand side vector b, which is overwritten with the solution vector, is reused. Hence, reading and distributing this vector across the registers of the threads in the warp (one element per thread) is beneficial.

The permutation $b := Pb$ coming from pivoting is performed while reading b into the registers: Each element is stored to the registers of the correct threads. This step is followed by a unit lower triangular solve, and finally an upper triangular solve. As these operations are similar, we will for brevity only discuss the lower triangular solve (the solution of $Ly = b(= Pb)$) in detail. Different strategies

```
1 % Input  : m x m matrix L, rhs vector b
2 % Output : Vector b overwritten by the solution y
3 %          of Ly = b
4 for k = 2 : m
5     b(k) = b(k) - L(k,1:k-1) * b(1:k-1); % DOT
6 end
```

```
1 % Input  : m x m matrix L, rhs vector b
2 % Output : Vector b overwritten by the solution y
3 %          of Ly = b
4 for k = 1 : m-1
5     b(k+1:m) = b(k+1:m) - L(k+1:m,k) * b(k); % AXPY
6 end
```

Figure 2. Loop-body of the "lazy" and "eager" algorithmic variants (top and bottom, respectively) for the solution of a unit lower triangular system in Matlab notation.

exist for realizing the triangular solve: the "lazy" variant (code in Figure 2, top) relies on an inner (DOT) to compute the final value of y_k at step `k`, while the "eager" one (code in Figure 2, bottom) leverages an AXPY to update the trailing vector $y_{k+1:m}$. In this case, the latter variant is more convenient, as the parallelization of AXPY is straightforward, while the DOT product requires a reduction. The memory accesses to the system matrix are also different: the "lazy" variant reads one row per step, while the "eager" one reads one column. Therefore, assuming standard column-major storage, the "eager" variant also has the benefit of coalesced memory access.

C. Block-Jacobi preconditioning using batched LU

Using batched factorization routines for block-Jacobi preconditioning requires the extraction of the diagonal blocks from the sparse system matrix. This is a non-trivial step as accessing a (dense) diagonal block embedded in a sparse data structure (such as those typically used for storing the system matrix, e.g., CSR [3]) can be quite elaborate. Furthermore, exploiting the fine-grained parallelism provided by the GPU hardware in the extraction step makes this operation challenging for problems with an unbalanced sparsity pattern: Assigning the parallel resources to the distinct rows will inevitably result in severe work imbalance for problems with a very unbalanced nonzero distribution, like for example those arising in circuit simulation. Additionally, accessing the distinct rows in the row-major-based CSR layout in parallel results in non-coalescent data access. In [4] we proposed a strategy to overcome the latter drawback while simultaneously diminishing the effects of the former one by means of an intermediate step that stores the diagonal blocks in shared memory. Although we refrain from showing a comparison between the standard approach and the shared-memory-based strategy in the experimental section, we recall the central ideas of this *shared memory extraction* for convenience:

Instead of assigning the distinct threads within the warp to the distinct rows corresponding to the diagonal block, all

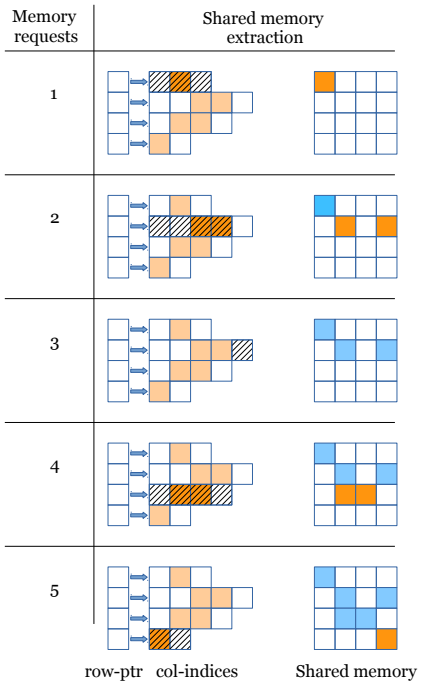

Figure 3. Illustration of the memory requests for the shared memory extraction. The elements part of the diagonal block are colored in light orange, the elements that have already been extracted in light blue. We assume warps of 4 threads, and visualize the data read by the distinct threads at each iteration with dashed cells. If an element part of the diagonal block is currently accessed (dark orange) it is extracted and stored in the correct location in shared memory (dark orange also in the shared memory). We only show the accesses to the vector storing the col-indices of the CSR matrix structure [3]; the access to the actual values induces far less overhead, as these memory locations are accessed only if a location belonging to a diagonal block is found. In that case, the access pattern is equivalent to the one used for col-indices.

threads of the warp collaborate to process each row. The threads accessing an element that is part of the diagonal block extract the respective value and store it into shared memory. This allows for coalescent access to the elements stored in CSR format, and avoids the load imbalance up to a level where load imbalance only occurs between threads of the same warp. After extracting the elements that are part of the diagonal block, they are copied into registers of the thread that will handle the respective row in the factorization process. Figure 3 visualizes this diagonal block extraction strategy [4].

IV. Numerical Experiments

In this section we evaluate the performance of the batched LU factorization and triangular solve kernels tuned for small-size problems by benchmarking them against alternative kernels offering similar functionality. Concretely, our experimental analysis includes the following kernels:

- *Small-size LU:* The batched LU factorization and triangular solve kernels developed as part of this work.
- *Gauss-Huard:* The batched factorization and triangular solve kernels based on GH [7].
- *Gauss-Huard-T:* The factorization in this routine is identical to the GH kernels, except in that the triangular systems are stored in a transpose access-friendly mode to accelerate the triangular solves [7].
- *cuBLAS LU:* The batched LU factorization and triangular solve kernels available in NVIDIA's cuBLAS package (version 8.0).

We point out that the first three implementations are part of the same software stack, the kernel implementations are similar in design, and received the same level of tuning. cuBLAS is a vendor implementation, optimized specifically for the targeted architecture, but its source is not available. As variable block size is not supported by the batched kernels in cuBLAS, the experiments involving them were conducted using fixed block size for the entire batch. This ensures a fair comparison and credible conclusions.

In addition to the evaluation of the LU factorization and triangular solve kernels, we assess the effectiveness of the computed preconditioner integrated into the iterative IDR(4) solver for sparse linear systems [3].

A. Hardware and software framework

We employed an NVIDIA Tesla P100 GPU with full double precision support in the experimentation together with NVIDIA's GPU compilers that are shipped with the CUDA toolkit 8.0. Except for the GETRF and GETRS[1] routines taken from NVIDIA's cuBLAS library [9], all kernels were designed to be integrated into the MAGMA-sparse library [10]. MAGMA-sparse was also leveraged to provide a testing environment, the block-pattern generation, and the sparse solvers. Since the complete algorithm is executed on the GPU, the details of the CPU are not relevant.

B. Performance of batched factorization routines

Figure 4 compares the performance of the four batched factorization routines in terms of GFLOPS (billions of flops per second). The left-hand side plots in the figure report the performance for a batch of matrices of size 16×16. The reference implementation for batched LU-based GETRF taken from NVIDIA's cuBLAS library achieves about 110 GFLOPS in single precision (top row). In comparison, the small-size LU, Gauss-Huard and Gauss-Huard-T all achieve about 130 GFLOPS for this case. In double precision (bottom row), the performance of the small-size LU is about 35% lower than that of the GH-based factorization routines, with the latter delivering about 100 GFLOPS. The scenario is different when the problem dimension is 32×32 (plots in the right-hand side of Figure 4): The performance of Gauss-Huard-T is then about 5% below that of Gauss-Huard, and

[1]Routine GETRS applies the sequence of permutations computed by the LU factorization routine to the right-hand side vector, followed by two triangular solves (TRSV).

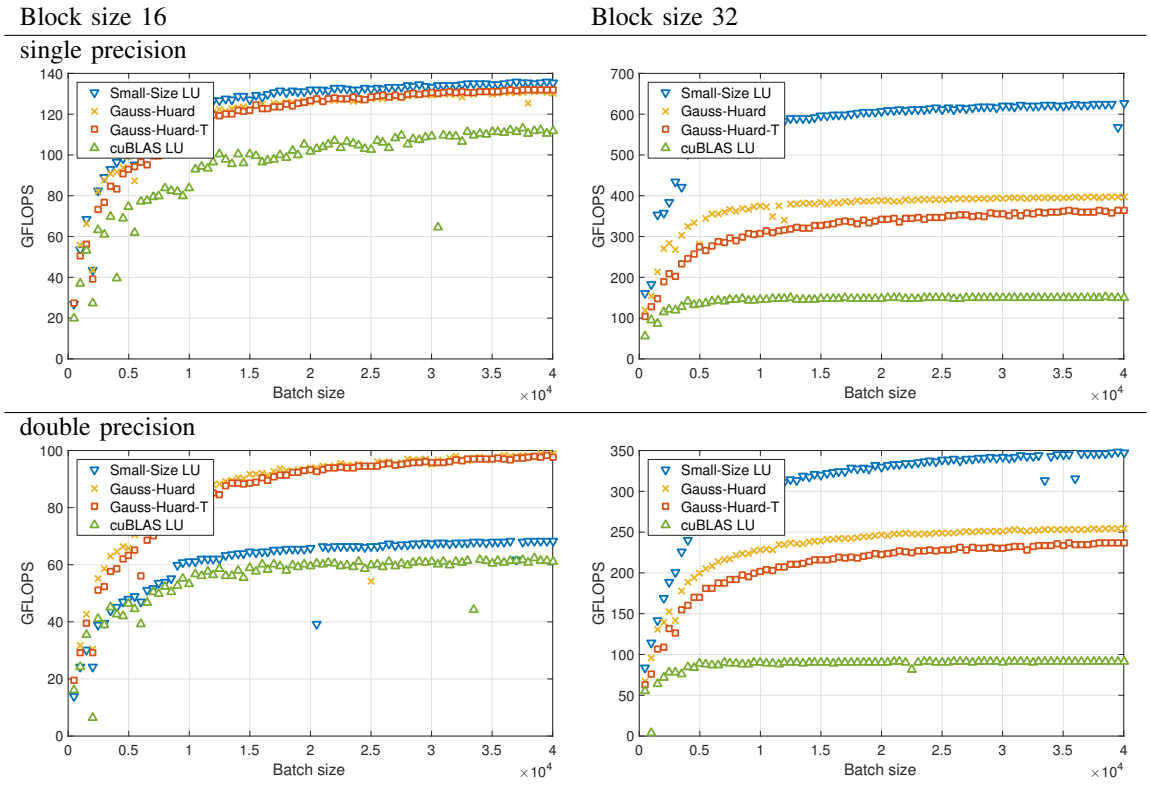

Figure 4. Performance of batched factorization routines depending on the batch size.

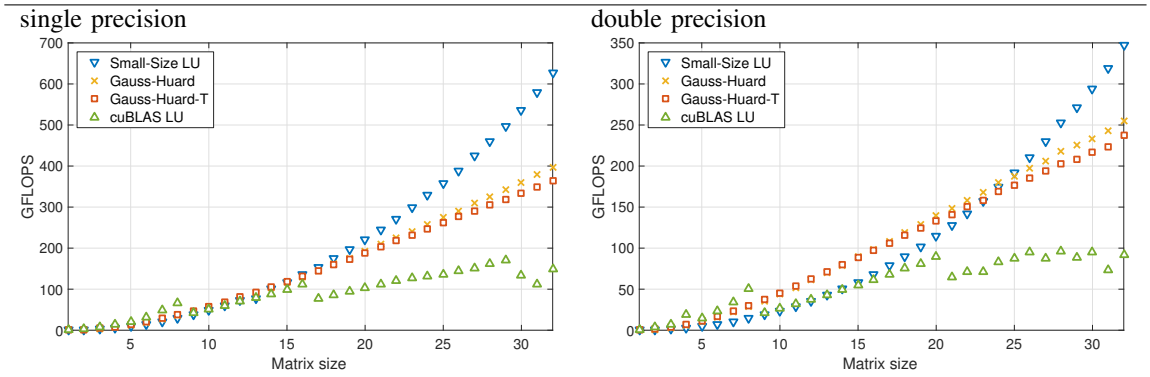

Figure 5. Performance of batched factorization routines depending on the size of the matrices. The batch size is fixed to 40,000 systems.

the small-size LU outperforms both routines by a significant margin, achieving up to 600 GFLOPS in single precision and 350 GFLOPS in double precision. The cuBLAS counterpart providing the same functionality is 3.5× slower, delivering about 100 GFLOPS only. The explanation for this block-size-dependent behavior is an implementation detail, which will be corrected as part of future work. Concretely, for block size $k < 32$, both the small-size LU and GH routines operate with a matrix of size 32×32, padding the input with zeros, but performing only the first k steps of the factorization. This benefits GH, since it implements a "lazy" factorization, while the "eager" (right-looking) algorithmic variant selected for the LU factorization performs more flops than its GH counterpart for block size $k < 32$. By optimizing the algorithms specifically for smaller block sizes, we expect to observe the same behavior as that obtained for block size 32.

Figure 5 reports the performance as a function of the problem size. The results indicate that the non-coalescent writes in Gauss-Huard-T play a significant role only for

problems of dimension larger than 16×16. For single precision, this also corresponds to the threshold from which the small-size LU starts to outperform the GH-type factorizations. In double precision, the small-size LU is slower than the GH-based factorizations for problems smaller than 23×23. The size-dependent results for cuBLAS LU reveal the system-specific optimizations: local performance peaks can be identified for sizes 8, 16, and 29 in single precision arithmetic, and for dimensions 8 and 20 in double precision arithmetic. Although we do not tune for specific sizes by handling multiple problems per warp, the small-size LU outperforms the cuBLAS LU for almost all sizes.

C. Performance of batched triangular solves

We next employ the same notation in order to distinguish the different batched implementations of the triangular solves that complement the factorization routines. On the left-hand side plot in Figure 6, we assess the performance of the triangular solves for a batch of matrices with size 16×16. Unlike the factorization step, the performance for both GH variants and the small-size LU are almost identical in single- as well as double precision arithmetic (44 GFLOPS and 37 GFLOPS, respectively).

For problems of size 32×32 (right-hand side plots in Figure 6), the more expensive Gauss-Huard-T factorization pays off by accelerating the triangular solve from 47 GFLOPS (for Gauss-Huard) to 80+ GFLOPS when using single precision. In double precision the triangular solves of Gauss-Huard-T are also about twice faster (70 GFLOPS) than those associated with the Gauss-Huard kernel (35 GFLOPS). The small-size LU achieves 90+ GFLOPS in single precision and close to 80 GFLOPS in double precision. This implies speed-up factors of $4.5\times$ and $4\times$ over cuBLAS, respectively.

Figure 7 analyzes the performance depending on the problem size. Conversely to the factorization step, the non-coalescent reads in the Gauss-Huard triangular solves harm the performance for problems larger than 16×16. For Gauss-Huard-T, the price of non-coalescent access was payed in the factorization step. As a result, the Gauss-Huard-T triangular solves remain competitive with the small-size LU triangular solves. As in the factorization step, NVIDIA's GETRS seems to be optimized for problem of dimension smaller than 16; nonetheless, this option achieves only a fraction of the performance of our small-size LU for all dimensions.

D. Analysis of block-Jacobi preconditioning

In this section we assess the efficiency of batched factorization routines in the context of block-Jacobi preconditioning. For this purpose we enhance an IDR(4) Krylov solver (taken from the MAGMA-sparse open source software package [11]) with a block-Jacobi preconditioner that is generated via batched factorization routines based on LU or GH, and applied in terms of triangular solves. The diagonal block structure is generated via the supervariable blocking routines available in MAGMA-sparse, and we only vary the upper bound for the size of the diagonal blocks. (At this point, we note that we do not include the cuBLAS batched LU in this comparison as it does not support variable problem size, which is needed for block-Jacobi preconditioning based on supervariable blocking.) We perform our tests using a set of 48 selected matrices from the SuiteSparse matrix collection [12] (see the column labeled as "Matrix in Table I, and http://www.cise.ufl.edu/research/sparse/matrices). The test problems are listed along with some key characteristics in Table I, and all carry some inherent block structure that makes them attractive targets for block-Jacobi preconditioning. We initialize the right-hand side vector with all its entries set to one, start the iterative solver with an initial guess of zero, and stop once the relative residual norm is decreased by six orders of magnitude. We allow for up to 10,000 iterations.

Although both the LU-based and GH-based factorizations present the same practical stability [13], we acknowledge the possibility of rounding effects. At this point, we note that rounding errors can have significant effect on the convergence rate of the Krylov solver, and a more accurate factorization (preconditioner setup) does not inevitably result in faster convergence of the preconditioned iterative solver. Figure 8 displays the convergence difference of IDR(4) depending on whether the block-Jacobi preconditioner is based on LU or GH. The x-axis of the histogram reflects the iteration overhead, while the y-axis shows the number of test cases for which LU provided a "better" preconditioner (bars left of center) or GH did (bars right of center). For all block sizes, the majority of the problems is located in the center, corresponding to those problem instances where both methods resulted in the same iteration count. Furthermore, the histogram exposes a remarkable level of symmetry, suggesting that, although rounding effects do occur, none of the factorization strategies is generally superior.

In addition to the convergence rate (iteration count), we are also interested in comparing the practical performance of the two factorization strategies, in terms of execution time, in a block-Jacobi setting. In Figure 9 we compare the total execution time (preconditioner setup time + iterative solver runtime) of the IDR(4) solver using a block-Jacobi preconditioner based on either LU, GH, or GH-T. For this experiment, we use an upper bound of 32 for the supervariable agglomeration. In most cases, the performance differences between the three options are negligible. Only due to rounding-error, and derived iteration count differences, one of the methods becomes superior. The differences between GH and GH-T come from the faster preconditioner generation combined with a potentially faster application of the latter. The matrices are ordered in the x-axis according to the total execution time of the solver and can be identified by the corresponding index in Table I (see column labelled as "ID"). The four missing cases correspond to matrices for

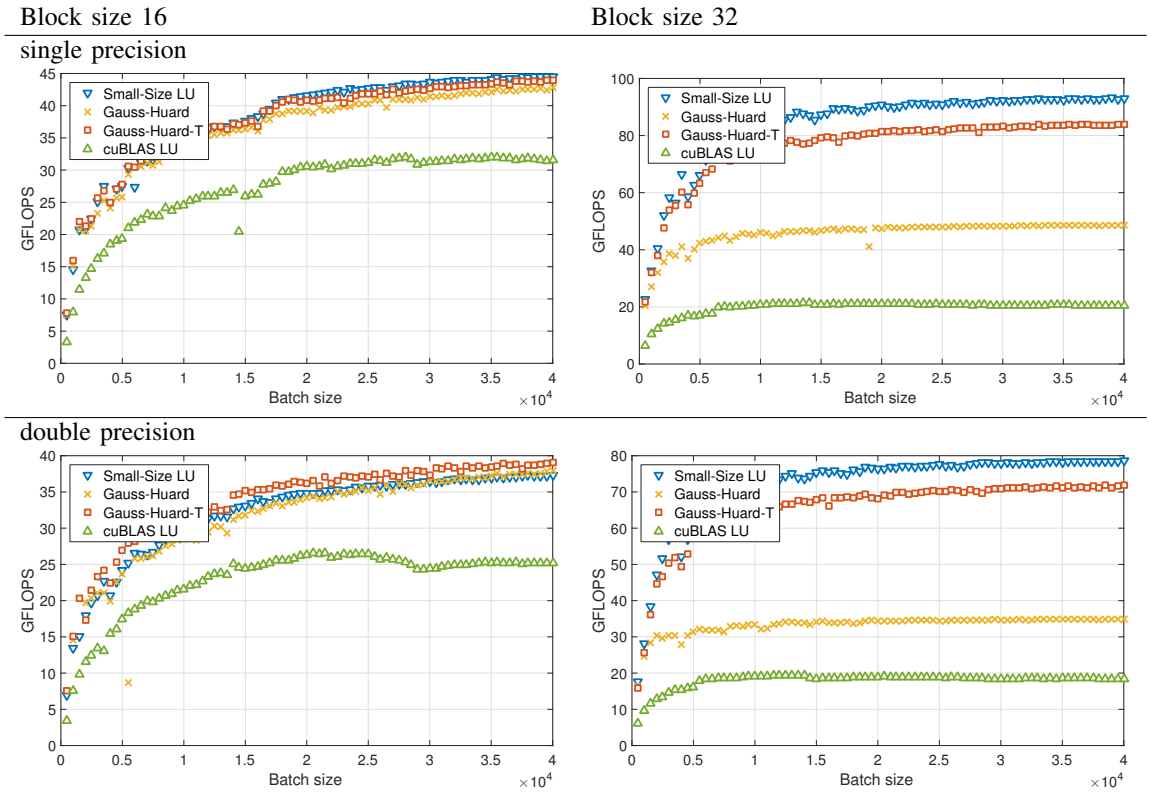

Figure 6. Performance of batched triangular solve routines depending on the batch size.

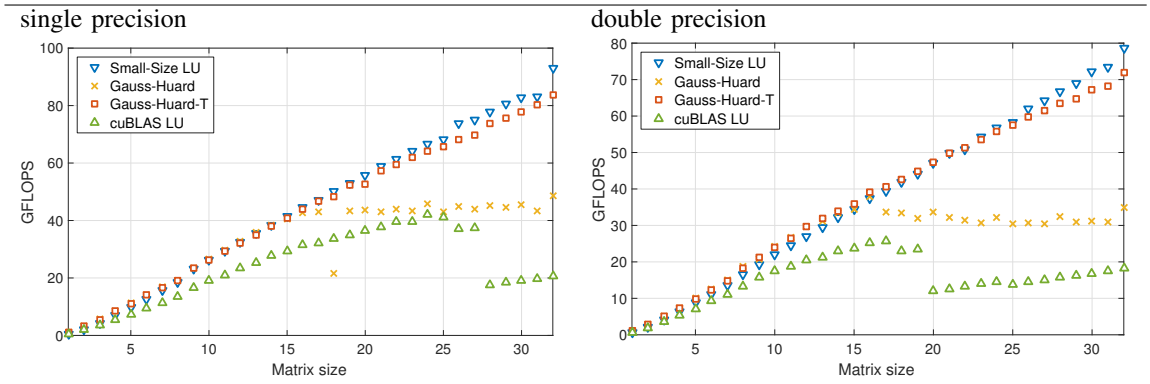

Figure 7. Performance of batched triangular solve routines depending on the size of the matrices. The batch size is fixed to 40,000 systems.

which the solver did not attain convergence.

To close this section, Table I lists all test matrices along with the convergence behavior and execution time when using different upper bounds for the Jacobi blocks in an IDR(4) solver preconditioned with the small-size LU-based block-Jacobi. The results suggest that larger block sizes typically improve the solver convergence with respect to both iteration count and time-to-solution.

V. Concluding Remarks and Future Work

We have presented variable-size batched CUDA kernels for the solution of linear systems via the LU factorization that are optimized for small-size problems and outperform existing counterparts offering the same functionality by a large margin. This performance is achieved by extensive use of the GPU registers, and the integration of an implicit pivoting technique that preserves numerical stability while removing the costly data movements due to the row exchanges.

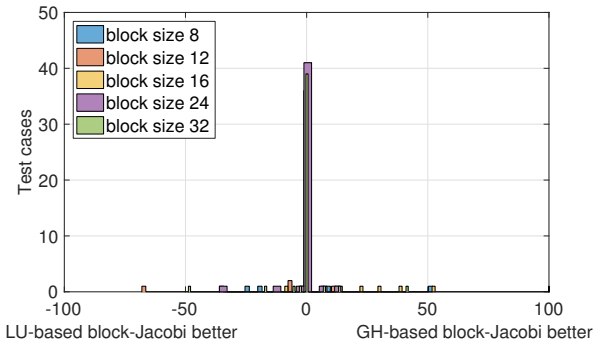

Figure 8. IDR(4) convergence using block-Jacobi preconditioning based on LU factorization or GH-factorization: Iteration overhead in % if LU provides the better preconditioner (left of center) or GH does (right of center).

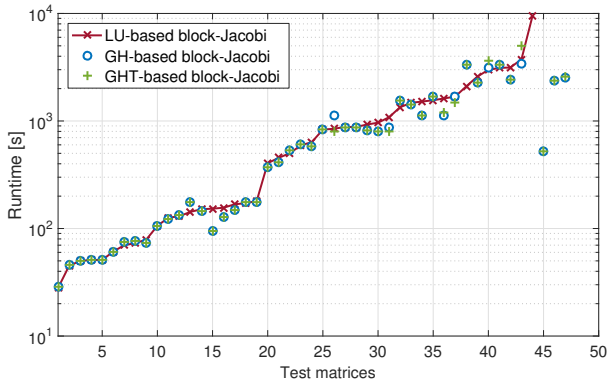

Figure 9. Total execution time (setup+solve) for IDR(4) enhanced with block-Jacobi preconditioning based on either LU or GH factorization. The size of the distinct diagonal blocks is adapted to the system matrix via supervariable blocking with 32 as upper bound. The matrix indices correspond to the values in the column labeled as "ID" in Table I.

Combined with an efficient strategy for the extraction of the diagonal blocks from a sparse data structure, we have presented an ecosystem of a factorization-based block-Jacobi preconditioner that succeeds in reducing the time-to-solution of the iterative IDR(4) Krylov method for a large range of problems.

Future work will address the development of a Cholesky-based variant for symmetric positive definite problems and the optimization of the batched kernels for any problem size.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC-0010042. H. Anzt was supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association. G. Flegar and E. S. Quintana-Ortí were supported by project TIN2014-53495-R of the MINECO, FEDER, and the EU H2020 project 732631"OPRECOMP. Open Transprecision Computing". The authors would also like to thank the Swiss National Computing Centre (CSCS) for granting computing resources in the Small Development Project entitled "Energy-Efficient preconditioning for iterative linear solvers" (d65).

References

[1] L. S. Blackford *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/567806.567807

[2] E. Anderson *et al.*, *LAPACK Users' guide*, 3rd ed. SIAM, 1999.

[3] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. SIAM, 2003.

[4] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí, "Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs," in *8th Int. Workshop Programming Models & Appl. for Multicores & Manycores*, ser. PMAM, 2017, pp. 1–10.

[5] E. Chow and J. Scott, "On the use of iterative methods and blocking for solving sparse triangular systems in incomplete factorization preconditioning," Rutherford Appleton Laboratory, Tech. Rep. Technical Report RAL-P-2016-006, 2016.

[6] G. Golub and C. V. Loan, *Matrix Computations*, 3rd ed. Baltimore: The Johns Hopkins University Press, 1996.

[7] H. Anzt, J. Dongarra, G. Flegar, E. S. Quintana-Ortí, and A. E. Tomás, "Variable-size batched gauss-huard for block-jacobi preconditioning," *Procedia Computer Science*, vol. 108, pp. 1783 – 1792, 2017, International Conference on Computational Science (ICCS 2017).

[8] P. Huard, "La méthode simplex sans inverse explicite," *EDB Bull, Direction Études Rech. Sér. C Math. Inform. 2*, pp. 79–98, 1979.

[9] *CUDA Toolkit v8.0*, NVIDIA Corporation, March 2017.

[10] Innovative Computing Lab, "Software distribution of MAGMA version 2.0," http://icl.cs.utk.edu/magma/, 2016.

[11] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, and M. Koehler, "Efficiency of General Krylov Methods on GPUs – An Experimental Study," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 683–691.

[12] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.

[13] T. J. Dekker, W. Hoffmann, and K. Potma, "Stability of the Gauss-Huard algorithm with partial pivoting," *Computing*, vol. 58, pp. 225–244, 1997.

Matrix	size	#nnz	ID	Jacobi		Block-Jacobi (8)		Block-Jacobi (12)		Block-Jacobi (16)		Block-Jacobi (24)		Block-Jacobi (32)	
				#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]
ABACUS_shell_ud	23,412	218,484	7	2074	3.38	2086	3.65	1717	3.07	1958	3.47	1889	3.48	**1560**	**2.96**
af_shell3	504,855	17,562,051	14	1443	8.91	1117	8.85	1148	9.09	**953**	**7.66**	968	7.95	1084	8.77
bcsstk17	10,974	428,650	13	1584	2.69	1524	2.73	797	1.50	736	1.38	**612**	**1.13**	927	1.79
bcsstk18	11,948	149,090	10	781	1.27	562	1.02	**377**	**0.69**	529	0.97	351	0.69	455	0.89
bcsstk38	8,032	355,460	48	–	–	–	–	–	–	7722	15.45	2875	5.86	**2101**	**4.44**
cbuckle	13,681	676,515	24	212	0.43	96	0.24	**19**	**0.04**	69	0.16	18	0.05	70	0.19
Chebyshev2	2,053	18,447	6	–	–	175	0.52	57	0.18	45	0.13	34	0.10	**28**	**0.08**
Chebyshev3	4,101	36,879	37	–	–	897	3.37	327	1.26	133	0.53	88	0.34	**79**	**0.31**
crankseg_1	52,804	10,614,210	36	180	0.61	125	0.48	**123**	**0.47**	144	0.54	126	0.48	126	0.51
CurlCurl_0	11,083	113,343	45	95	0.16	64	0.12	57	0.12	51	0.11	**49**	**0.10**	45	0.10
CurlCurl_1	226,451	2,472,071	35	262	0.73	191	0.72	182	0.67	132	0.50	117	0.45	**106**	**0.43**
dc3	116,835	766,396	8	183	18.92	116	12.10	142	14.76	131	13.68	**109**	**11.36**	168	17.52
dw1024	2,048	10,114	39	–	–	133	0.23	136	0.28	128	0.27	72	0.15	**51**	**0.10**
dw2048	2,048	10,114	25	–	–	133	0.22	136	0.22	128	0.25	72	0.15	**51**	**0.09**
dw4096	8,192	41,746	41	–	–	–	–	–	–	5035	7.94	1565	2.57	**879**	**1.53**
dw8192	8,192	41,746	40	–	–	–	–	–	–	5035	7.93	1565	2.57	**879**	**1.56**
ecology2	999,999	4,995,991	11	–	–	**2826**	**27.58**	3919	38.06	3539	34.92	3358	34.56	2990	30.17
F1	343,791	26,837,113	9	2906	19.17	1944	15.26	**1422**	**11.16**	1930	15.20	2065	16.63	1688	13.52
G3_circuit	1,585,478	7,660,826	30	**1030**	**9.47**	1079	15.82	1105	16.12	1070	15.85	1260	19.50	847	12.93
gas_sensor	66,917	1,703,365	4	–	–	–	–	–	–	–	–	–	–	**9480**	**22.33**
gridgena	48,962	512,084	42	1322	2.37	669	1.37	791	1.62	**657**	**1.34**	730	1.52	637	1.35
Hook_1498	1,498,023	59,374,451	38	7215	120.43	5017	109.84	**2731**	**59.64**	5358	117.96	3486	78.96	3747	83.84
ibm_matrix_2	51,448	537,038	21	–	–	–	–	**211**	**0.54**	–	–	300	0.76	–	–
Kuu	7,102	340,200	15	101	0.19	66	0.15	67	0.13	71	0.22	66	0.15	**61**	**0.13**
LeGresley_2508	2,508	16,727	16	266	0.49	189	0.36	166	0.30	171	0.34	**172**	**0.30**	151	0.32
linverse	11,999	95,977	43	2069	3.16	–	–	–	–	5981	10.19	2010	3.56	**964**	**1.80**
matrix_9	103,430	1,205,518	3	1425	2.96	–	–	**469**	**1.19**	–	–	508	1.34	–	–
matrix-new_3	125,329	893,984	26	–	–	–	–	**408**	**1.09**	–	–	452	1.24	–	–
ML_Geer	1,504,002	110,686,677	1	2092	44.06	1844	48.63	1917	50.43	1700	45.09	2111	57.35	**1621**	**43.58**
nasa2910	2,910	174,296	46	486	0.83	575	0.93	386	0.67	**340**	**0.62**	440	0.74	405	0.71
nd12k	36,000	14,220,946	5	–	–	3902	13.38	2863	9.81	2191	7.60	**1487**	**5.21**	1476	5.23
nd24k	72,000	28,715,634	31	–	–	3970	21.01	2542	13.46	2129	11.31	1691	9.04	**1516**	**8.12**
nd3k	9,000	3,279,690	34	–	–	–	–	3875	8.08	2354	4.94	**2066**	**4.45**	3145	6.87
nd6k	18,000	6,897,316	33	–	–	–	–	4173	10.39	3635	9.08	**1945**	**5.01**	2585	6.80
nemeth15	9,506	539,802	32	**47**	**0.09**	–	–	1298	2.28	392	0.73	280	0.53	156	0.31
olm5000	5,000	19,996	47	–	–	224	0.46	256	0.48	**119**	**0.23**	114	0.24	153	0.26
Pres_Poisson	14,822	715,804	2	177	0.32	136	0.29	97	0.19	88	0.21	**81**	**0.17**	73	0.26
rail_79841	79,841	553,921	19	915	1.70	987	2.20	825	1.91	658	1.55	801	1.88	**593**	**1.34**
s1rmt3m1	5,489	217,651	18	206	0.33	166	0.32	134	0.28	**140**	**0.27**	134	0.28	130	0.32
s2rmq4m1	5,489	263,351	12	722	1.13	**190**	**0.35**	197	0.37	200	0.38	188	0.38	172	0.39
s2rmt3m1	5,489	217,681	20	1131	1.75	268	0.52	217	0.40	206	0.38	200	0.40	**180**	**0.36**
s3rmq4m1	5,489	262,943	17	–	–	595	1.03	693	1.26	**494**	**0.86**	491	0.90	501	0.95
s3rmt3m1	5,489	217,669	29	–	–	1303	2.07	769	1.29	733	1.24	**472**	**0.96**	831	1.58
s3rmt3m3	5,357	207,123	22	–	–	–	–	–	–	–	–	**9676**	**16.40**	–	–
saylr4	3,564	22,316	27	1118	1.77	249	0.48	182	0.34	171	0.32	**160**	**0.28**	143	0.30
ship_003	121,728	3,777,036	23	–	–	2731	8.12	2488	7.37	3042	9.19	1739	5.38	**1343**	**4.18**
sme3Dc	42,930	3,148,656	28	3311	7.87	2945	7.58	3178	8.19	**2397**	**6.22**	2578	6.79	3113	8.32
sts4098	4,098	72,356	44	108	0.23	77	0.17	83	0.16	60	0.14	60	0.12	**50**	**0.10**

Table I

ITERATIONS AND EXECUTION TIME OF IDR(4) ENHANCED WITH SCALAR JACOBI PRECONDITIONING OR BLOCK-JACOBI PRECONDITIONING. THE RUNTIME COMBINES THE PRECONDITIONER SETUP TIME AND THE ITERATIVE SOLVER EXECUTION TIME.