IEICE TRANS. INF. & SYST., VOL.E96–D, NO.10 OCTOBER 2013

PAPER

Exploiting the Task-Pipelined Parallelism of Stream Programs on Many-Core GPUs

Shuai MU[†a)], ***Member***, **Dongdong LI**[†], **Yubei CHEN**[†], **Yangdong DENG**[†], ***and*** **Zhihua WANG**[†], ***Nonmembers***

SUMMARY By exploiting data-level parallelism, Graphics Processing Units (GPUs) have become a high-throughput, general purpose computing platform. Many real-world applications especially those following a stream processing pattern, however, feature interleaved task-pipelined and data parallelism. Current GPUs are ill equipped for such applications due to the insufficient usage of computing resources and/or the excessive off-chip memory traffic. In this paper, we focus on microarchitectural enhancements to enable task-pipelined execution of data-parallel kernels on GPUs. We propose an efficient adaptive dynamic scheduling mechanism and a moderately modified L2 design. With minor hardware overhead, our techniques orchestrate both task-pipeline and data parallelisms in a unified manner. Simulation results derived by a cycle-accurate simulator on real-world applications prove that the proposed GPU microarchitecture improves the computing throughput by 18% and reduces the overall accesses to off-chip GPU memory by 13%.

key words: *GPU, task-pipeline, dynamic scheduling, load balance, L2 cache*

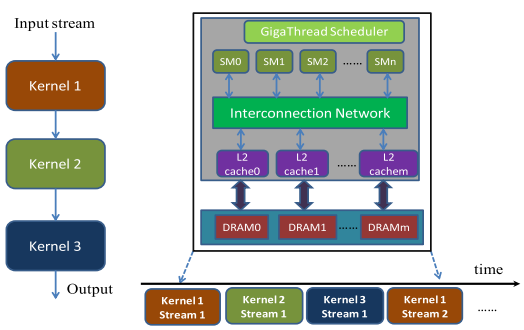
1. Introduction

Graphics Processing Units (GPUs) have evolved from a pure graphics rendering device with fixed functionality into a highly programmable, general-purpose computing platform. In the past a few years, GPUs have found wide applications in such domains as scientific computing, data mining, computer vision, geological exploration, and computational finance [1]. By offering unprecedented processing power (e.g., 4300GFLOPS by AMD's Radeon HD7900 GPU [2]) on a single chip, GPUs are dramatically changing the landscape of modern computing. Future GPUs will continue to offer ever-growing computing throughput and increasing flexibility by supporting more parallel processing patterns [3].

Unlike their CPU counterparts, GPUs are designed for efficient execution of data-parallel workloads. Today a typical GPU is equipped with hundreds of processing cores to run tens of thousands of concurrent threads. When executing a program, the processing cores on a GPU execute the same code but on different data sets by following a single program, multiple data (SPMD) style. Meanwhile, GPUs also adopt the multithreading with dedicated hardware for fast thread context switching to hide the memory latency. Programmers can take advantage of a software controlled on-chip shared memory and a memory coalescing mechanism to reduce the off-chip memory latency further [4]. With the above latency-hiding techniques, typical GPUs only deploy a relatively limited capacity cache on its silicon estate.

Despite the importance of data parallelism, real-world applications are far more complex. Many applications, especially those following a stream-processing pattern [5], have their data parallelism interleaved with task-pipelined parallelism [6]. A stream program consists of a sequence of tasks designated as kernels [5]. Input data are usually organized in batches, which are often called as streams. Tasks in a stream program are usually rich of data parallelism. Figure 1 (a) illustrates the concept of stream processing. Multiple kernels constitute a pipeline that conducts different functions on continuously arriving data, i.e., stream. The data transferring between two neighboring kernels follows a producer-consumer fashion. Stream processing is a fundamental pattern that is pervasive in virtually all scientific and engineering applications [7]. With the rapid growth of the data-intensive applications such as sensor network, data mining and signal processing [8]–[10], stream processing will play an even more important role.

A large body of research (e.g., [11]–[18]) has been proposed to support GPU based stream execution. In spite of the encouraging results, a series of problems have to be resolved before the task-pipelined processing pattern can be fully streamlined on GPUs. First, most GPUs can only launch one kernel at a time. When running kernels with relatively small data sets, a GPU's computing resources have to be under-utilized. Second, current GPUs cannot directly support the producer-consumer pattern of data transfer between two adjacent stages in a task pipeline. A preceding kernel in a task pipeline has to write its output to the off-



(a) A task pipeline (b) Serialized execution of tasks on GPUs

Fig. 1 A task-pipeline flow and its GPU execution.

Manuscript received February 25, 2013.
Manuscript revised May 13, 2013.
[†]The authors are with Institute of Microelectronics, Tsinghua University, Beijing, China.
a) E-mail: mus04ster@gmail.com
DOI: 10.1587/transinf.E96.D.2194

chip memory, whereas the succeeding kernel then fetches the data from off-chip memory to its local memory [19]. The problem can be mitigated by buffering data in on-chip cache. However, GPUs usually only have a relatively small capacity cache. Thus, the cached intermediate results tend to be replaced by other data before it can be consumed by the succeeding kernel. Figure 1 (b) depicts the serialized execution process of task-pipelined applications on GPUs. The above two problems seriously drag down the efficiency of running stream applications on GPUs.

The first problem mentioned above has been addressed by a few recent works through software [20]–[23] and/or hardware techniques [1], [24]. Among software techniques, uber-kernel [22] is a novel solution that packs multiple kernels corresponding to various stages of a pipeline into a single one. This approach can be efficient for task-pipelined applications on GPUs as it helps eliminate the explicit barriers and lower the overhead of launching kernels. Although this technique can improve the programming flexibility and execution efficiency, it is ultimately constrained by the hardware scheduling mechanism.

Meanwhile, hardware techniques were first introduced on NVIDIA's Fermi GPU to concurrently execute multiple kernels as long as there are available computing resources [4]. This feature, together with an asynchronous data transferring mechanism [4], constitutes Fermi's stream processing capability and proves to be effective for certain applications [1]. Fermi GPU uses a fixed preemptive strategy to map kernels to computing resources. If the first kernel cannot occupy all cores (i.e. streaming multiprocessors in NVIDIA's terminology), then the second kernel can be launched on the remaining cores. Otherwise, if a kernel consumes all the resources, no other kernels can be started until there are idle cores available. Therefore, the resource allocation among concurrent kernels cannot be adjusted as soon as a kernel is initialized. NVIDIA's latest GPU architecture called Kepler [24] enhances the flexibility of dynamic kernel launching. Kepler supports dynamic parallelism that allows on-the-fly GPU kernels to launch new kernels. Such an innovation is convenient for developers to program and optimize recursive and data-dependent execution patterns without the need to returning control to a host CPU. However, Kepler's hardware scheduling mechanism is still based on the preemptive strategy, as children kernels can only either occupy the idle computing resources or share the busy computing resources with their parent kernels. Such a solution still lacks the capability of allocating computing cores with a globally optimized scheme according to the different kernels' workload. Such a dynamic parallelism mechanism is mainly designed to ease the programming process, instead of offering a microarchitecture-level automatic performance tuning technique.

A static scheduling mechanism cannot adapt to the dynamic execution characteristics of kernels. In addition, the performance of a kernel generally does not scale linearly with the number of processors. Previous work [25] already indicates that the computing efficiency of a kernel declines when the number of processors reaches a certain threshold. As we will elaborate later in Sect. 2.3, an extensive characterization of task-pipelined applications confirms that single-kernel performance will saturate beyond a given number of processors. On the other hand, our experimental results prove that simultaneously running two kernels can often be more efficient than dedicating the whole GPU's computing resources to a single kernel even when this kernel is able to occupy the resources fully. Therefore, a static allocation of processors is generally not the optimal strategy. Our results demonstrate that there exists an optimal allocation for overall performance, but it can only be determined dynamically.

Based on the above observations, this work focuses on developing microarchitectural techniques to enable efficient task-pipelined execution on GPUs. The major contributions are as follows.

- We develop a quantitative analysis to identify the pros and cons of running task-pipelined applications on current GPU microarchitectures. The results provide key insights on devising microarchitectural features for proficient task-pipelined execution on GPUs.
- An efficient adaptive task scheduler for GPUs is proposed to orchestrate the execution of kernels in a pipelined fashion. By monitoring the usage of computing resources and measuring performance, the proposed scheduler dynamically adjusts the allocation of processors among multiple kernels.
- We propose a moderately modified L2 cache structure for GPUs. By revising the replacement policy, we can significantly reduce the number of off-chip memory accesses by 13%.
- By integrating the techniques proposed in this work, we construct an enhanced GPU microarchitecture to support task-pipelined applications. Experiments on a set of typical benchmark applications show that our techniques enable an 18% higher throughput than the serialized implementation does and a 7% improvement over Fermi GPU's preemptive scheduling method.

The rest of this paper is organized as follows. Section 2 gives an overview of current GPU architecture and a quantitative analysis of running task-pipelined applications on current GPUs. Section 3 introduces our microarchitectural enhancements for supporting task-pipelined execution. In Sect. 4, we evaluate the proposed techniques with extensive experimental results. Section 5 reviews related work. The paper is concluded in Sect. 6.

2. Characteristics of Task-Pipelined Processing on GPUs

2.1 Overview of Modern GPU Microarchitecture

We use NVIDIA's GPU, Fermi [1], as an example to review modern GPU microarchitectures that support general purpose computing. Computing resources of the Fermi GPU

are organized into 16 multiprocessors. As the basic unit for program execution, a multiprocessor is equipped with 32 scalar processing cores as well as local memory in terms of shared memory and L1 cache.

NVIDIA proposed a general-purpose programming model, CUDA (i.e. Compute Unified Device Architecture). A CUDA program deploys up to tens of thousands of threads, which are distributed onto multiprocessors by an on-chip global thread scheduler. The threads are organized into thread blocks, while multiple thread blocks constitute a grid. A GPU computation corresponding to such a grid organization is designated as a kernel, which is equivalent to a task in the terminology of stream processing. Threads in different blocks can only share data through off-chip global memory. Data sharing and synchronization between different multiprocessors is expensive. NVIDIA's Fermi GPU is equipped with an on-chip unified L2 cache used to hide the long latency to off-chip memory. However, the relatively low cache capacity (768KB on Fermi) makes it less effective for applications with intensive or unpredictable memory accesses [26].

2.2 Characteristics of Task-Pipelined Benchmarks

We picked up a series of task-pipelined applications from three typical GPGPU benchmark suites, HPEC [27], NVIDIA CUDA SDK [28] and Rodinia [29], as benchmarks for this research. The benchmarks are widely used for evaluating GPU microarchitecture. As shown in Table 1, these benchmarks cover a broad range of application domains including image processing, high performance embedded computing, computational finance and computational biology. Such benchmarks as MT, PM and TDFIR are compute-bound, while benchmarks like MUM, HG and LOS are memory-bound.

We simulate the benchmarks with a cycle-accurate many-core GPU microarchitecture simulator, GPGPU-sim [30], which provides a detailed simulation model of contemporary GPUs. The detailed execution characteristics of the benchmarks are shown in Table 2. Each benchmark has at least two kernels and these kernels are executed in a task-pipelined manner. The 2nd column describes how many streams, i.e. input data sets, are fed to the task pipeline. Note that a stream has to be processed by every kernel serially. As each data set can be quite large, increasing the number of streams will not affect the overall performance, but incurs a longer simulation time. We select the number of streams based on the principle that the performance can be stable and simulation time can be kept at a reasonable level. The numbers of kernels are listed in the 3rd column. Columns 4 and 5 enumerate the thread organizations in terms of grid and block dimensions for each kernel, which are consistent with the original released benchmarks and/or manually fine-tuned for better performance. The last column describes the active blocks per multiprocessor. This metric indicates the maximum number of blocks distributed onto a multiprocessor in one shot.

Table 1 Description of benchmarks.

Benchmark	*Description*
NVIDIA's SDK[28]	
Mersenne Twister (MT)	A random number generator used in Monte-Carlo simulation
Histogram (HG)	Histogram computing, commonly used in image processing and data mining
Dct8x8 (DCT)	Discrete cosine transformation
Line-of-Sight (LOS)	Graphics processing
HPEC[27]	
Time Domain Finite Impulse Response (TDFIR)	A basic operation in signal processing
Pattern Matching (PM)	Feature-aided tracking module in an integrated radar-tracker
Constant False-Alarmed Rate (CRAR)	A basic radar processing step for removing varying background noise
Rodinia[29]	
MUMerGPU (MUM)	A high-throughput parallel pairwise local sequence alignment program
Neural Network (NN)	A convolution neural network for handwritten digit recognition

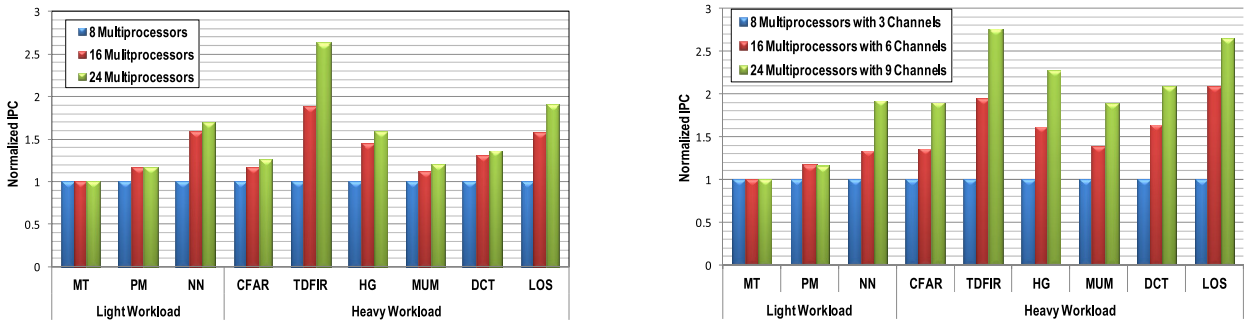
Table 2 Characteristics of task-pipeline applications.

Name	*Streams*	*Kernels*	*Grid Dimension*	*Block Dimension*	*Active Blocks/Multiprocessor*
Light-load benchmarks					
MT	8	2	(32,1,1);	(128,1,1);	8
			(32,1,1)	(128,1,1)	8
PM	8	2	(64,1,1);	(128,1,1)	8
			(64,1,1)	(128,1,1)	8
NN	28	4	(6,1,1);	(13,13,1);	5
			(50,1,1);	(5,5,1);	8
			(100,1,1);	(1,1,1);	8
			(10,1,1)	(1,1,1)	8
Heavy-load benchmarks					
CFAR	8	2	(256,1,1);	(128,1,1);	8
			(256,1,1)	(128,1,1)	8
TDFIR	8	2	(256,1,1);	(256,1,1);	8
			(256,1,1)	(128,1,1)	8
HG	8	2	(1093,1,1);	(64,1,1);	3
			(128,1,1)	(256,1,1)	4
MUM	24	2	(1024, 1, 1)	(192, 1, 1)	2
			(1024, 1, 1)	(256, 1, 1)	3
DCT	16	3	(128,128,1);	(8,4,2);	7
			(256,128,1);	(8,8,1);	8
			(64,64,1)	(8,4,2)	7
LOS	32	5	(256,1,1);	(256,1,1);	4
			(256,1,1);	(256,1,1);	4
			(1,1,1);	(128,1,1);	8
			(256,1,1);	(256,1,1);	4
			(256,1,1)	(256,1,1)	4

These benchmarks can be classified into two categories light-load and heavy-load, according to the computation intensity measured by the number of required multiprocessors as the following equation:

$$Num_{required} = \frac{Product\ of\ Grid\ Dimension}{Active\ Blocks/mutiprocessors} \quad (1)$$

If the required number of multiprocessors derived by (1) does not exceed the number of multiprocessors in a GPU,



(a) Performance with fixed memory bandwidth (6 channels) (b) Performance with proportional memory bandwidth

Fig. 2 Performance scalability of serialized kernel execution.

Table 3 Configuration of Simulated GPU Microarchitecture.

Configuration	*Parameter*
Streaming Multiprocessors	8/**16**/24
Warp Size	32
SIMD Pipeline Width	32
Number of Threads/Core	1024
Shared Memory/Core	48KB (16 banks)
Constant Cache Size/Core	8KB 2-way set assoc. 64B lines LRU
Number of Memory Channels	3/**6**/9
L1 Cache/Core	24KB 4-way set assoc. 64B lines LRU
L2 Unified Cache	384KB/**768KB**/1152KB 8-way set assoc. 64B lines LRU
Interconnect Configuration	Mesh
GDDR3 Memory Timing	TCL=9, TRP=13, TRC=34, TRAS=21,TRCD=12, TRRD=8

all the blocks can be assigned onto the multiprocessors in one shot. Such type of applications is designated as the light-load benchmark. Otherwise, some blocks can only be launched after certain assigned blocks finish execution. We call such applications as heavy-load benchmarks. Note that a kernel in the light-load benchmarks deploys a relatively small number of thread blocks and does not occupy all the multiprocessors fully, whereas a kernel in the heavy-load benchmarks will consume all computing resources. In the following evaluation and experiments, we will consider three hardware configurations with the number of multiprocessors chosen as 8, 16 and 24, which are listed in Table 3. For all three configurations, Eq. (1) leads to the same classification as shown in Table 2. MT, PM and NN are light-load benchmarks, while the others belong to the category of heavy-load benchmarks.

2.3 Analysis of Task-Pipelined Execution on GPU

For task-pipelined applications, both light-load and heavy-load benchmarks have great potential for performance improvement by optimizing the allocation of computing resources. For light-load applications, it is intuitive that two or more continuous kernels along the pipeline can run simultaneously to occupy the GPU's processors fully as long as they handle different data stream. For heavy-load applications, we will show that it is also possible for performance improvement by appropriately reallocating the computing resources of GPU between two kernels.

To understand the performance implications of running two or more kernels concurrently, we first analyze the efficiency of the kernel execution on current GPU by running the benchmarks with increasing number of multiprocessors. The kernels in each benchmark are executed sequentially. The analysis is performed on a cycle-accurate GPU microarchitecture simulator [30]. Table 3 gives the detailed configurations with the numbers in a bold font showing the configuration of Fermi GPU. We will consider two scenarios: 1) Memory bandwidth remains fixed at six channels[†] with an increasing number of multiprocessors; 2) The memory bandwidth increases proportional to the number of multiprocessors (i.e. 3, 6 and 9 channels, respectively).

Figure 2 shows the simulation results for all the benchmarks with the number of multiprocessors set as 8, 16, and 24, respectively. The performance of each benchmark is evaluated with instruction throughput (IPC[††]) normalized to that of an 8-multiprocessors configuration. The results of the first scenario are shown in Fig. 2 (a). The key observation is that, except TDFIR, none of the benchmarks exhibit good performance scalability when more multiprocessors are installed. It suggests that generally the total throughput is a sub-linear function of the number of multiprocessors. Figure 2 (b) illustrates the performance trend of the second scenario. It follows that the performance scalability of memory-bound programs (i.e. MUM and LOS) is better than that of the first configuration, but it is still far from perfect linearity. On the other hand, it must be noticed that off-

[†]Each channel has a 64-bits bus. The GDDR3 memory clock frequency is 1107MHz. Thus, the bandwidth of each channel is $1107 \times (64/8) \times 2/1024 = 17$GB/s.

[††]IPC is short for Instruction per Cycle, a widely adopted metric for performance in microarchitecture design. Given an application and a compilation flow, the overall number of instructions is fixed no matter how the microarchitecture is altered. Therefore, it is a more appropriate metric than the computing throughput. The performance evaluation results of this paper are based on IPC.

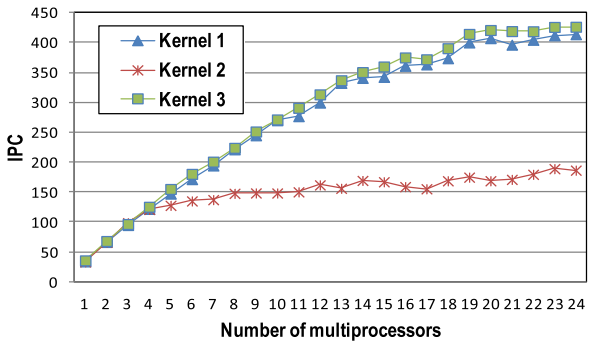


Fig. 3 Performance scalability of single kernel in DCT.

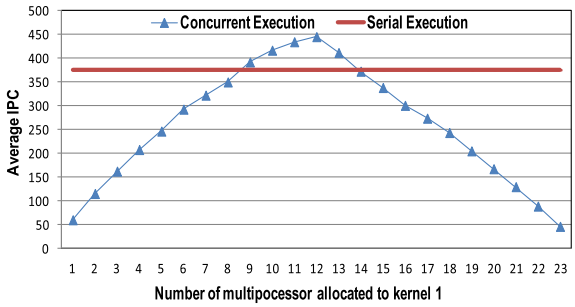


Fig. 4 Performance comparison of varying allocations of multiprocessors for kernel 1 and kernel 2 of DCT application.

chip memory bandwidth might not be able to always keep pace with the increasing of multiprocessors. With the technology scaling, the number of multiprocessor increases with the area of a GPU chip, while the number of memory bus pins (or equivalently the off-chip memory bandwidth) only grows with the perimeter of the GPU chip. So ultimately, the second scenario cannot hold. We expect that the future scalability of memory bandwidth should be somewhere in between scenarios 1 and 2. Although the above sub-linear trend is derived from a GPU simulator, it has also been verified on real GPU hardware running large volume of experimental data [25], [31], [32].

For both scenarios in Fig. 2, a further analysis reveals that light-load benchmarks exemplified by MT and PM hardly benefit from a larger number of multiprocessors. The reason is that there are no more workloads available for the idle multiprocessors. A better way to utilize the computing resources is to run two or more kernels simultaneously. Launching concurrent kernels has to meet the constraint that these kernels are independent from each other. In task-pipelined application, the independence widely exists along the pipeline. In fact, two pipeline stages are independent as long as they are handling different data streams. We will elaborate this issue later in Sect. 3.

The performance of heavy-load benchmarks tends to saturate as the number of processors exceeds a threshold. Taking the DCT application with three kernels as an example, we can derive the performance curves of its three kernels executed individually as shown in Fig. 3 (Here the memory bandwidth is fixed when the number of multiprocessors is increasing). The rate of performance growth drops down when the number of multiprocessors is beyond 16. The situation is even worse for the second kernel. It suggests that, beyond a given threshold, we can only receive diminishing gain in performance when dedicating more computing resources to a single kernel. Considering the task-pipelined parallelism, it is appealing to investigate the feasibility of expediting such heavy-load benchmarks through concurrent kernel execution.

Accordingly, we perform a series of experiments. First, kernels 1 and 2 of DCT are executed sequentially with both fully occupying 24 multiprocessors. The IPC of the whole application is shown as the red line in Fig. 4. Next, we modify the simulator to run two kernels concurrently and allocate the multiprocessors between two kernels. The blue curve in Fig. 4 depicts the performance trend of DCT by changing the number of multiprocessor allocated to kernel 1 from 1 to 23 (i.e., 23 to 1 multiprocessors to kernel 2).

Figure 4 delivers a few important messages. First, unbalanced allocations (i.e., those at the two ends of the blue curve) of computing resources lead to poor performance. Second, running two kernels in parallel does offer opportunities to outperform the sequential execution as long as a proper allocation of multiprocessors can be identified. In Fig. 4, when we allocate 9-13 multiprocessors to kernel 1 and remaining to kernel 2, the overall performance is higher that the serialized solution by up to 17%. Similar performance curves have been observed on other benchmarks.

2.4 Implication for Cache Design

The performance saturation of the single kernel execution is largely due to the severe jam of memory traffic when a large enough number of multiprocessors are working. With the number of multiprocessors increasing linearly, more memory requests will be delivered. Typically, the memory overhead for these requests cannot be linear due to the limit in both memory bandwidth and latency of off-chip GPU memory. Such a problem is hard to solve for an arbitrary application, as generally the memory requests from different kernels are independent from each other. However, for task-pipelined applications, the abovementioned problem can be greatly mitigated because two neighboring kernels in a task-pipeline have a producer-consumer relation. When running sequentially, a producer kernel generates intermediate results and stores them in the off-chip memory, while a consumer kernel loads the data back. In the case of parallel execution of two or more kernels, writing and reading to off-chip memory will both go through L2 cache. This fact suggests that L2 cache can actually serve as a shortcut between concurrently running producer and consumer kernels. The reduced number of accesses to off-chip memory leads to the overall performance improvement, as illustrated in Fig. 4. The following section explains the details of our microarchitecture modification to support task-pipelined application efficiently by taking advantage of the above observations.

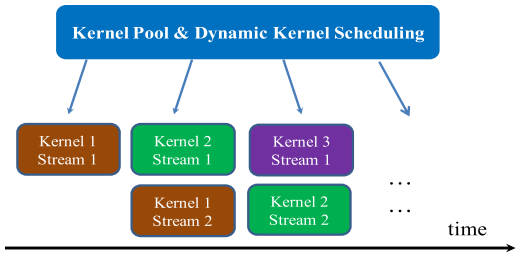


Fig. 5 Parallel execution of task-pipelined kernels.

3. GPU Microarchitectural Enhancements

In this section, we elaborate our work on enabling efficient task-pipelined execution on GPUs. The proposed microarchitectural enhancements include two key components, an adaptive dynamic scheduling mechanism to orchestrate pipelined kernels and a moderately modified L2 cache structure to exploit the producer-consumer pattern between neighboring kernels.

Our task-pipeline oriented microarchitecture allows parallel execution of multiple kernels in a manner as illustrated in Fig. 5, where kernels can run concurrently by sharing the GPU computing resources. The characterization results presented in Sect. 2.3 suggest that such pipeline-parallel execution has potential to deliver higher performance. If an application has more than two kernels, e.g. 3, we can still issue two kernels each time by following an execution sequence as K1-K2, K2-K3 and K3-K1. A critical question, therefore, has to be raised: how to allocate and schedule GPU processors among multiple kernels for an optimal overall throughput. In Sect. 3.1, we propose an efficient scheduling mechanism to answer the above question. In Sect. 3.2, we introduce our modified cache architecture and corresponding policy tailored for producer-consumer pattern. In the last sub-section, the hardware implementation and its cost overhead will be estimated.

3.1 Dynamic Kernel Scheduling

The general scheduling problem under the context of multi-/many-core processors involves both allocating computing resources and determining the start time for each task. In this work, the temporal scheduling is implied by the pipelined dependency. Hence, the objective of our scheduling problem is to allocate multiprocessors on a GPU chip between K_1 and K_2 such that the overall throughput is optimized. In other words, a multiprocessor is the basic unit of resource allocation.

Here we propose a two-kernel scheduling mechanism that dynamically adjusts the resource allocation by analyzing the execution history of kernels and predicting the performance implications. The prediction is made possible by the fact that a kernel needs to be executed for multiple times and statistical behaviors of the workload generally change slightly across multiple executions. The scheduling mechanism is implemented in GPU's global scheduler.

Although we can try all possible allocations of multiprocessors to identify the solution for the best performance, such a process will be costly. In particular, the performance loss during the search process is unaffordable. As a result, we take a heuristic approach to solve the problem. The proposed scheduling mechanism works as follows. We have two kernels designated as K_1 and K_2, which potentially share N multiprocessors. The execution time of these two kernels can be designated as $T_1(n)$ and $T_2(n)$, where n is the number of multiprocessors assigned to them. Since the total workloads of two kernels are fixed, the overall computation time is constrained by the slower one of the two kernels, as the fast kernel has to wait for the slower one in each step of pipelined execution. Then the total execution time of concurrently running two kernels is:

$$T_{execution}(n) = MAX(T_1(n), T_2(N-n)) \tag{2}$$

The objective of the scheduling problem is to find n such that the total execution time in (2) can be minimized. $T_1(n)$ and $T_2(n)$ can be arbitrarily complex and generally cannot be derived analytically. According to our experimental results, $T_1(n)$ and $T_2(n)$ can be approximated with monotonically decreasing functions. Under such an assumption, the sufficient condition for (2) to reach its minimum is:

$$T_1(n) = T_2(N-n) \tag{3}$$

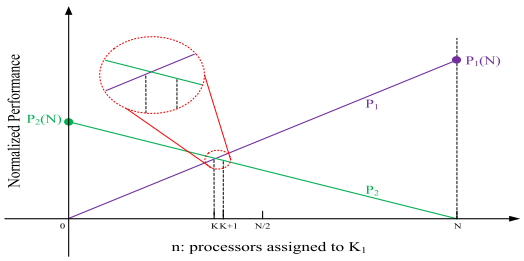
However, it must be realized that $T_i(n)$ is relatively hard to handle because $T_i(n) \to \infty$, $i = 1, 2$ when $n = 0$. A commonly used numerical technique is to use the reciprocal of $T_i(n)$:

$$P_1(n) = \frac{1}{T_1(n)} \; and \; P_2(n) = \frac{1}{T_2(n)} \tag{4}$$

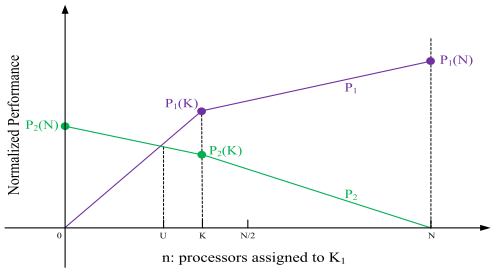
Again we use piecewise-linear functions to approximate $P_1(n)$ and $P_2(n)$. Then the problem reduces to use measured data to derive $P_1(n)$ and $P_2(n)$, and then identify an n such that $P_1(n) = P_2(N-n)$.

Upon the arrival of the 1st data stream, we run K1 and K2 sequentially and measure P1(N) and P2(N). The measurement is done by an on-chip performance counter as those already deployed on GPU chips [33]. Assuming the performance of both kernels is linearly scalable, we can derive two lines that predicting the performance of both kernels under varying allocation of multiprocessors. In Fig. 6 (a), the purple line is the linear approximation of P1 and the green line is that of P2. The two lines have a cross point, which suggest a performance optimal under the above assumptions[†]. However, this point generally does not correspond to an integer number of processors. Suppose the cross

[†]It should be noticed that the performance curve of each kernel is different from that calculated individually in a static manner. The points on the performance curve are calculated dynamically under real cache and memory constraints. Hence, the performance inherently includes the effect of competition for cache and memory bandwidth.



(a) The 1st performance interpolation



(b) The 2nd performance interpolation

Fig. 6 Using a piecewise-linear approximation to get the optimal processor allocation. The convergence will be achieved after 4∼5 iterations on average. The figure only shows the first and second iteration.

point is located between integers K and K + 1 as magnified in Fig. 6 (a). Then we have to choose the optimal allocation of either K or K + 1. At either point, there will be a faster kernel and a slower one. We make a selection such that the performance of slower kernel is larger. The situation is illustrated in Fig. 6 (a), where choosing K leads to a higher overall throughput.

Then the multiprocessors are allocated between kernels K_1 and K_2 accordingly, i.e. K multiprocessors for K_1 and N-K for K_2. Of course, such an initial prediction may be inaccurate. Therefore, we use new execution results to continuously fine tune the allocation as shown in Fig. 6 (b). We then re-calculate the performance line and find the next optimized allocation under the same criteria. The scheduler will repeat this process until convergence. Table 4 gives the number of iterations for reaching the optimal allocation in the benchmarks. On average, the convergence to the optimal allocation can be achieved in 4-5 iterations. Although some applications, such as TDFIR and HG, achieve the convergence point until almost all the stream data are processed, the performance of the last sub-optimal allocation is already very close to the final optimal configuration. Therefore, even such a late convergence does not lead to obvious performance degradation.

3.2 Modified L2 Cache Structure

As analyzed in Sect. 2.3, L2 cache can serve as a shortcut for data transferring between two neighboring kernels in a task-pipeline. To fully exploit such an opportunity, data produced by the preceding kernel should be kept in L2 cache as much as possible until the succeeding kernel finishes consuming the data. Thus, we can improve the chance of cache hit when the succeeding kernel begins executing. Meanwhile, the contents written by the preceding kernel should be flushed† immediately after the succeeding kernel finishing execution, as these data have little chance to be reused in the next round of stream execution.

Table 4 Number of iterations for optimal kernel allocation.

Application	*Kernel Pair*	*Number of Iterations*
MT	K1-K2	3
PM	K1-K2	4
NN	K1-K2	4
	K2-K3	5
	K3-K4	6
	K4-K1	6
CFAR	K1-K2	6
TDFIR	K1-K2	8
HG	K1-K2	7
MUM	K1-K2	6
DCT	K1-K2	3
	K2-K3	4
	K3-K1	3
LOS	K1-K2	3
	K2-K3	5
	K3-K4	7
	K4-K5	6
	K5-K1	3

Today's GPUs are equipped with L1 and L2 cache. However, due to the limited cache size, some cache lines tend to be replaced by later requested data during the execution. Typical cache replacement policies like LRU cannot specify exactly which cache lines have the potential to be reused in the succeeding execution. In a task-pipelined application, a preceding kernel processes the input stream and generates new data, which will be consumed by a succeeding kernel. The intermediate data will be written to global memory through cache. If the corresponding cache lines are replaced, the data have to be fetched again from global memory to the cache when the succeeding kernel needs them. Obviously, such memory traffic can lead to a performance overhead, especially when cache capacity is limited. The observation suggests that it can be beneficial to keep these cache lines until the succeeding kernel accessed them. After that, the intermediate data will hardly be reused and we can flush these cache lines to save space for other work.

Based on these observations, we modified the on-chip L2 cache of GPU by adding two labels in each cache line as shown in Fig. 7 (a). One label, kernel_ID, is used to mark which kernel writes to the cache line, while another label, stream_ID, records to which stream the data in this cache line belong. Both kernel_ID and stream_ID of all cache lines all initialized as −1. When a memory request is sent by a multiprocessor, the kernel_ID and stream_ID are added to the memory request. Given a memory-writing request,

†The cache flushing refers to writing the cache data back to global memory normally and labeling the cache line as invalid. It will take only several extra cycles.

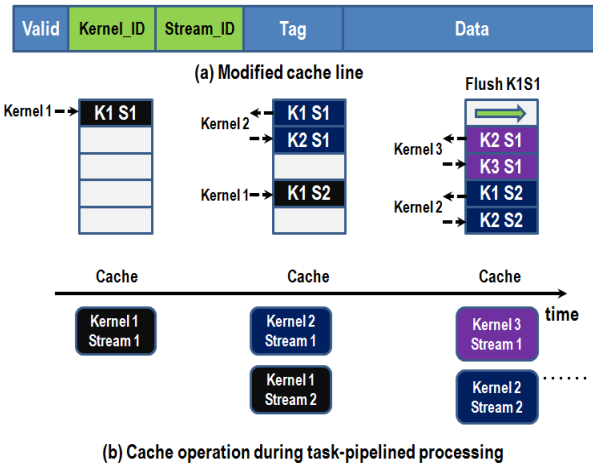


Fig. 7 Stream-based cache structure and replacement policy.

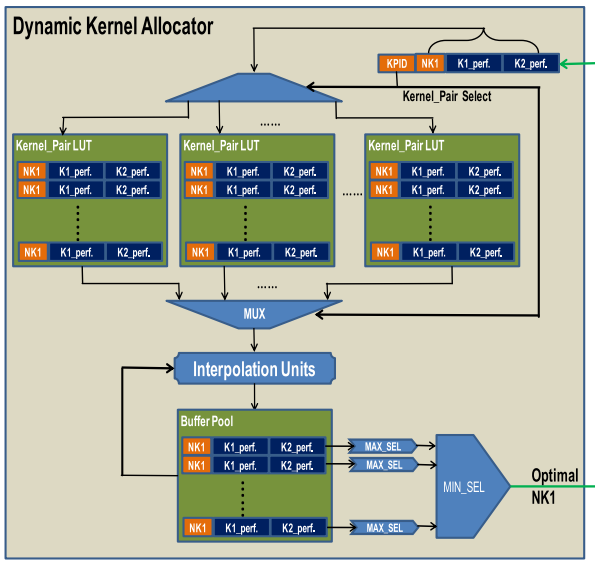


Fig. 8 Implementation of dynamic kernel allocator.

the kernel_ID and stream_ID information in the memory request will be copied to the cache line after the regular cache operations. Memory read requests are handled exactly the same as in a conventional cache. Therefore, the major difference from the previous cache design is that write request will record or update the label information of the cache line. As a result, we are able to specify to which data stream and kernel a cache line belong.

Our design ensures the data stored in each cache line to have unique kernel_ID and stream_ID label. As read request will not alter the label information, we only need to consider write request to maintain the uniqueness. When a write miss happens, we fetch the data of the whole cache line directly from the off-chip global memory. The label information in cache line is then updated by that of the write request. Upon a write hit, it is possible that the label information in the write request is different from that of the corresponding cache line. One option is to store each data's label information separately, but it will pose a relatively high storage overhead. Actually, it is rare for write requests to hit the same cache line with different labels, as different streams have varying writing address space in the global memory. Therefore, to alleviate the storage pressure, we record the labels at the granularity of the whole cache line. In other words, we use the label information of the memory request to replace that of the corresponding cache line. Figure 7 (b) elaborates the detailed working procedure. At each step, concurrent kernels finish the processing for a data stream. Then the thread scheduler will broadcast a message containing both kernel_ID and stream_ID corresponding to the kernel and data stream that have just finished execution to all cache lines. Each cache line will compare its label information with the counterpart of the message. When matched, the corresponding cache line will be flushed.

An exemplar execution process is illustrated in Fig. 7 (b). At step 1, kernel 1 processes data stream 1. Therefore, the cache line written by kernel 1 is labeled with K1S1. At step 2, kernel 2 will read the data produced by kernel 1 at step 1. As the data are already cached, kernel 2 read them directly from cache as long as the cache lines have matched thread_ID and stream_ID. Meanwhile kernel 2 writes its own results in cache lines with label as K2S1. After kernel 2 finishes executing, the global thread scheduler broadcasts a message to L2 cache indicating that the cache lines labeled with K1S1 can be flushed safely. This implies that the flushed cache line can be reused. Without the cache modification, at step 3 when all the cache lines are occupied, new data requests will replace cache lines in an unpredictable manner, while many cache lines actually store data needed by the next step execution. In such a case, the replaced data have to be fetched back and the extra read accesses lead to overhead of time and energy.

3.3 Hardware Implementation and Cost Estimation

Our microarchitectural enhancements only incur moderate hardware overhead to GPU's global scheduler and L2 cache. Figure 8 shows the major hardware organization of our dynamic kernel distribution mechanism, which is designated as dynamic kernel allocator (DKA) and implemented in GPU's global scheduler. DKA is equipped with a series of lookup tables (LUTs) to record the performance metrics. Each kernel pair[†] has its own LUT, which has multiple entries. Each entry corresponds to a possible allocation of multiprocessors. An entry consists of three fields, the number of multiprocessors allocated to the 1st kernel in the kernel pair (NK1[††]), the performance metric of the 1st kernel (K1_perf), and the performance metric of the 2nd kernel (K2_perf), respectively. The performance metrics fields are empty

[†]A kernel pair refers to two kernels that are two adjacent stages are in the pipeline and execution concurrently in our work.

[††]The NK1 label only records the number of multiprocessors allocated to kernel 1. Then the remaining multiprocessors are assigned to kernel 2.

before a program starts[†]. The allocation is performed in an iterative fashion. At the beginning of one round of the iteration process, GPU global scheduler sends a message to DKA indicating which LUT to choose. Meanwhile, the performance metric of each kernel-pair's previous execution is updated in corresponding entries. Then, the performance metrics from the entries of LUTs will be sent to the linear interpolation units to derive the performance metrics of the remaining allocation solutions of multiprocessors with a linear approximation method described in Fig. 6. Meanwhile, the predicted performance metric by the interpolation will be stored in a temporal buffer pool. Finally, a series of simple comparisons are conducted to identify the optimal processors allocation, which is sent back to global scheduler.

The storage units in a DKA are the main contributor of hardware cost since the logic units are relatively simple. Although DKA needs a 16-bit floating-point processing unit (FPU) for performance prediction (e.g. reciprocal computation), the overhead of a FPU is negligible considering the fact that modern GPU already have hundreds of FPUs deployed. We choose the execution time as the performance metric. The measuring of execution time can be achieved by reusing the program cycle counters that are already installed on GPU chips. Assuming there are 24 multiprocessors in the GPU, for a given pair of kernels, a LUT of each kernel pair has 25 entries. Each entry contains 5 bits, 16 bits and 16 bits for three regions in a total of 37 bits. Currently we allow a task-pipelined consisting of up to 8 kernels[††]. Thus, the total LUT of neighboring kernel pairs is 8 plus 1 in the buffer pool. In summary, the scheduler needs a total of 1KB (i.e., $(8 + 1) \times 25 \times 37$bits) SRAM for storage. The modified L2 cache needs extra storage for the kernel and stream labels as well as corresponding comparison logic. In the modified L2 cache, each cache line needs 3 extra bits for kernel ID (up to 8 kernels) and 5 bits for stream ID (the stream ID could be reused when exceeding the 32 streams). For a typical L2 cache with 12K cache lines (with the 768KB cache size), we only need an extra 12KB of storage, which occupies only 1.5% of total L2 cache size. The hardware overhead of the comparison logic is even more negligible.

4. Methodology and Experiments Results

We implemented the proposed microarchitecture features in a cycle-accurate simulator, GPGPU-sim (version 3.0.1), for many-core GPU microarchitectures [30]. The benchmarks have been summarized in Table 1. Table 3 lists the detailed hardware configurations. We compare three GPU microarchitectures, baseline GPU running kernels serially, the preemptive scheduling mechanism used in Fermi and Kepler GPUs and our proposed techniques. The preemptive scheduling mechanism has been incorporated in current version of GPGPU-sim. We also implemented our proposed techniques in GPGPU-Sim.

The performance is evaluated in the following aspects. Section 4.1 compares the performance improvement for three microarchitecture configurations. Section 4.2 analyzes the breakdown of performance improvement by the dynamic scheduling and the modified L2 cache. Section 4.3 compares the reduction of off-chip memory accesses. Section 4.4 discusses the concurrent execution of three or more kernels.

4.1 Performance Evaluation

As introduced in Sect. 1, the preemptive scheduling mechanism is a straightforward method to support multiple kernels. The idea is that a succeeding kernel can be launched on the remaining idle multiprocessors if a preceding kernel does not occupy all multiprocessors. Otherwise, when a preceding kernel consumes all the resources, no other kernels can start until some multiprocessors finish their workloads. Such a scheduling strategy considers neither the performance scalability of each kernel nor the optimal number of multi-processors allocated to each kernel.

Figure 9 shows the performance comparison normalized to the baseline GPU. To evaluate the scalability of our proposed techniques, we consider three configurations with the number of multiprocessors set as 8, 16 and 24, respectively. The L2 cache size and memory bandwidth increase proportionally as Table 3 shows. The results in Fig. 9 lead to a few important insights.

First, compared to the baseline, the performance improvement of light-workload is much better than the heavy-workload benchmarks for both the preemptive scheduling and our methods. For example, on the 8-multiprocessors configuration, the performance improvement of PM for two scheduling methods can be about 40% than the baseline. However, the performance improvement of light-load benchmarks does not scale prominently with the number of multiprocessors increasing. The reason is intuitive: the workload of a single kernel in these benchmarks does not fully occupy the computing resources.

Therefore, both the preemptive scheduling and our task-pipelined method can better exploit the computing resources for multiple small kernels execution. The results reveal that our dynamic scheduling can perform as well as the preemptive scheduling on light workload benchmarks.

Second, Fig. 9 demonstrates that our method has a better scalability than preemptive scheduling for the heavy-workload benchmarks. On the 8-multiprocessors configuration, our method does not pose significant performance advantage. However, with the number of multiprocessors increasing to 16, our method delivers slightly better results. Further, when the number of multiprocessors reaches 24, our method is superior to the preemptive method. The per-

[†]As described in Sect. 3.1 of this section, the initial data streams of each kernel pair are processed serially and the performance metrics will be initialed accordingly. The scheduling of allocator actually starts working after the serial execution.

[††]The decision of supporting at most 8 kernels is based on our observations that almost all the benchmarks have less than 8 kernels. Actually, this value can be set to a larger number with a trivial hardware overhead.

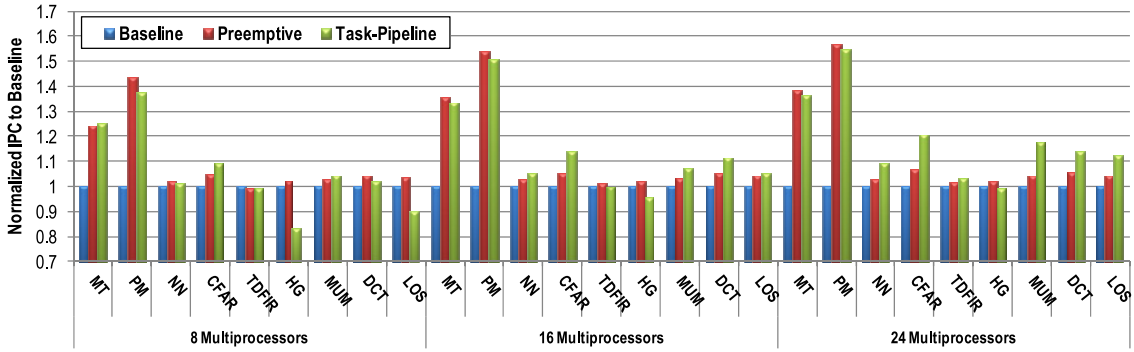


Fig. 9 Performance comparison on three different microarchitecture configurations.

formance improvement of our task-pipeline method can be 12%, while that of the preemptive scheduling method is only 4%. The performance scalability can be explained as follows. With a small number of multiprocessors, our method does not have much freedom to make an optimized allocation of multiprocessors. Taking HG as an example, it has two kernels and the execution time of the 2nd kernel is about 30X longer than that of the 1st kernel. If we ignore the influence of memory bandwidth, the best allocation of computing resources should be 3% to the 1st kernel and 97% to the 2nd kernel, which means the 2nd kernel has to occupy most of the computing resources. However, on 8-processors configuration, the maximum computing resources allocated to the 2nd kernel is 87.5% (7 of the total 8 multiprocessors) as we have to allocate at least one multiprocessor to the 1st kernel. In this case, not enough computing resources allocated to the 2nd kernel will be the major performance bottleneck. This problem can be alleviated when the number of multiprocessors is larger. On the 24-multiprocessors configuration, the computing resources allocated to the 2nd can be 95%, which is near to the optimal allocation. Therefore, we can see that larger number of multiprocessors brings more benefits for our scheduling method, which makes it suitable for future microarchitectures with even more processors.

Overall, on the 24-multiprocessors configuration, 7 out of 9 benchmarks perform much better than baseline while only TDFIR and HG are less efficient. For TDFIR, as pointed out earlier in Fig. 2, it has almost a linear scalability. Therefore, the choice of scheduling method does not significantly influence the performance. The reason of the performance degradation for HG is that the workloads of its kernels are extremely unbalanced and a feasible allocation is far from optimal. On average, our task-pipeline method delivers an 18% improvement compared to the baseline and a 7% improvement over Fermi's preemptive scheduling method.

4.2 Performance Breakdown

We have proposed two mechanisms to support task-pipelined execution, i.e. the dynamic kernel scheduling mechanism and the modified L2 cache. Figure 10 shows the breakdown of performance improvement due to these two mechanisms on 24-multiprocessors configuration. We

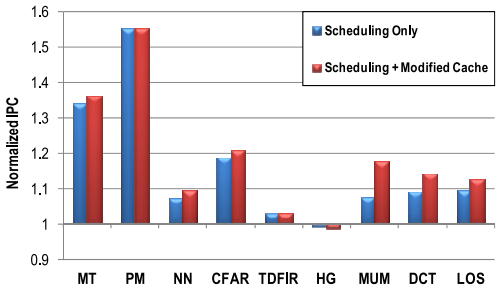


Fig. 10 Performance breakdown of task-pipelined execution.

consider two scenarios of our techniques. The first one only uses the proposed dynamic scheduling technique, while the second one integrates both techniques. For light-workload application, almost all contributions to the enhanced performance come from the dynamic scheduling. As the performance bottleneck of these benchmarks is the insufficient usage of computing resources, instead of the contention to memory resources. Therefore, the modified cache policy is less influential in performance improvement.

For heavy-workload benchmarks, the modified cache plays a more important role in performance improvement. The MUM benchmark gets most benefits from the modified cache, as it is memory-intensive and cache friendly. Improving cache behavior can contribute to the performance directly.

4.3 Memory Access

As we have explained in Sect. 3.2, the important objective of our modified cache design is to reduce the number of off-chip memory accesses, which improves performance and lowers power consumption. Figure 11 illustrates the number of off-chip memory accesses normalized to the results of baseline. We can see that the preemptive scheduling only slightly changes the number of memory accesses and leads to diverse behaviors on different benchmarks. As the preemptive scheduling does not change the cache's architecture, the memory behaviors are determined by the execution sequence of threads and the streaming characteristics have not been exploited. On the contrary, our stream-oriented cache architecture effectively reduces the number of mem-

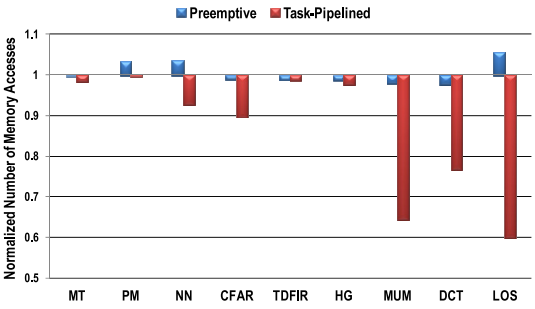


Fig. 11 Normalized number of off-chip memory accesses.

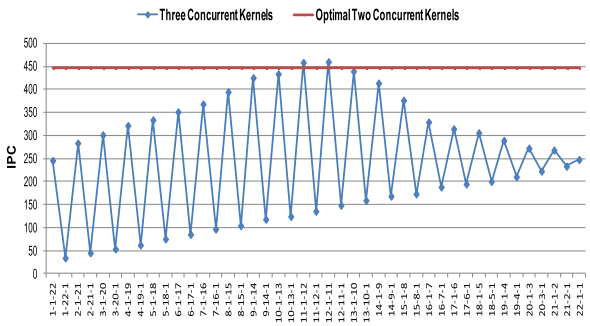


Fig. 12 Performance comparison between optimal two kernels execution and three kernels execution under different configurations.

ory accesses on most benchmarks. Among these, MUM and LOS require large-volume of data transfer between neighboring kernels. Thus, the proposed technique brings up to 40% reduction in off-chip memory requests for these two benchmarks. In the case of HG, which incurs a large number of memory references, the reduction is only 2% as most of the requests are read operations and few data need to be transferred between kernels. For MT and PM, the number of accesses changes slightly. The reason is that they only need a small volume of data transfer between neighboring kernels and most data can be contained in cache. Overall, the total number of off-chip memory accesses on the proposed architecture is reduced by 13%.

4.4 Concurrent Execution of Three or More Kernels

In the previous design and implementation, we focus on the concurrent execution of two kernels. It is proved that running two kernels tends to be more efficient than dedicating the whole GPU's computing resources to a single kernel, even when this kernel is able to fully occupy the resources. Scheduling two kernels simultaneously has a direction application in computer graphics rendering, in which vertex processing and fragment processing are two major steps of a typical graphics pipeline.

A generic task-pipelined application may have over two kernels. Such applications can be solved in two different approaches. The first approach is implemented in our scheduling mechanism, which decomposes a task pipeline into stages with each stage consisting of two kernels. Then each stage of two kernels can be executed concurrently. The second approach is to concurrently schedule three or more kernels by extending our techniques. For the second approach, we need to consider two scenarios: 1) All the kernels' workloads are small and not all the SMs are required; and 2) The kernels' workloads are large enough to occupy all the SMs. In the first scenario, the preemptive scheduling adopted by current GPUs can distribute the computing resources efficiently as different kernels do not compete for resources. It must be noted that such a case rarely happens in current GPGPU applications because the purpose of GPGPU is to accelerate the processing of large data sets. In the second scenario, running three or more kernels is likely to cause increasing traffic jam and imbalance computing resources allocation that hurts the overall performance. We can use benchmark DCT as an example. DCT is composed of three kernels as a task pipeline. Figure 12 illustrates the performance of running three kernels under all possible configurations of task mapping on a 24-multiprocessors GPU. The X axis of Fig. 12 lists the allocations of multiprocessors to the three kernels. The zigzagged curve shows the IPC value of each configuration[†]. The configuration space is quite large. Therefore, we only show the results on the "boundary" of performance data[††]. The performance attained by our proposed two-kernels scheduling is also shown as the straight red line. It shall be noted that the performance data of optimal two concurrent kernels is fixed as it has no relation to the X axis. Figure 12 illustrates that only two out of the over 40 configurations deliver a slightly higher level of performance than our two kernels execution and the performance advantage is less than 2%. In addition, most configurations actually perform worse. On the other hand, finding the optimal configuration for three or more kernels leads to significantly increased computing time and storage space. Overall, our dynamic two-kernels scheduling is suitable for current stream applications. From the above analysis, we believe that it is not beneficial to directly schedule task pipeline with three or more kernels for concurrent execution.

5. Related Work

Mapping stream programs to multiprocessors has received significant attention in the past [11]–[18]. Recently there has been a wave of research work on mapping multi-task applications to GPUs. The existing work can be classified into three categories. The first line of research focuses on constructing automatic flow of mapping task-pipelined streaming programs onto GPU [12], [13] and auto-tuning work for optimizing program parameters [34]–[36]. The second category aims at software programming techniques to exploit

[†]The three figures on the horizontal axis represent the numbers of multiprocessors allocated to the three kernels.

[††]There are at most $C_{23}^{2} = 253$ kinds of configuration for 24 multiprocessors. We only show the boundary performance. The performance of other configurations all falls in the range of the boundary performance.

task management on GPUs [20]–[23]. The third category makes efforts to optimize microarchitectures for scheduling multi-tasks among many processing cores [1]. Our work is similar to the 3rd category.

The first category work aims at developing efficient automatic tools to map stream programs on GPU platforms and auto-tuning program parameters for different GPU platforms. Udupa *et al.* [12] developed efficient techniques to map stream programs to CUDA by following a software pipeline approach, in which GPU global memory is used as the data buffer for concurrently executing kernels. However, this work can only handle relatively small streaming data because their techniques require all thread blocks to be simultaneously active. There has been a number of work [34]–[36] about tuning and optimizing GPU's parameters for a higher efficiency on different GPU hardware. They adopt empirical optimization techniques to generate a large number of parameterized code variants for a given algorithm and run these variants on a given platform to discover the one that gives the best performance. These auto-tuning works only focus on the efficient software implementations and cannot be directly applied to the hardware resources adjusting.

The work of second category put an emphasis on supporting multi-task execution in the framework of the CUDA programming model. Various software techniques were proposed to enable efficient and flexible execution of multi-task workload. Gupa *et al.* [23] study using persistent thread style programming to solve irregular GPU workloads. The persistent thread approach is to keep threads alive during the whole process of kernel execution. Upon finishing processing a task, a thread fetches a new task from a work queue. This approach effectively reduces the overhead of launching new threads. Tatarinov and Kharlamov [22] proposed a novel method called uber-kernel to merge multi-kernels to one single kernel for any workload benchmarks. This approach can be efficient for exploiting task-pipelined applications on GPUs as it can eliminate the explicit barriers and lower the overhead of launching kernels. Based on the uber-kernel method, Tzeng [20] developed software mechanism to decompose the tasks into the granularity of warps and record the warps in distributed queues allocated on the device memory as well as task donation mechanism to balance the workload of each task and ensure all processors can get work quickly. The abovementioned works are valuable in improving the performance and programming flexibility of GPGPU applications, but they still rely on the existing scheduling method of current GPUs.

The third category cares about hardware optimization for a better allocation of processing cores among multi-tasks. Early generations of GPUs supporting general purpose computing do not allow directly running multiple kernels. The concurrent kernel execution was first introduced on NVIDIA's Fermi GPU [1]. NVIDIA's Kepler GPU offered the support for dynamic parallelism [24]. Although both Fermi and Kepler GPUs support concurrently running multiple kernels, they still follow a preemptive scheduling strategy, where an idle multiprocessor is immediately assigned to execute a kernel as long as it still has unfinished blocks. Such a preemptive scheduling mechanism does not consider the dynamic behaviors of kernel execution and thus leaves space for optimization. For graphics applications, current GPUs can allocate a varying distribution of multiprocessors to various tasks in a graphics pipeline [37]. For example, the GPUs might use 90 percent of its processors as pixel shaders and 10 percent as vertex shaders when drawing a certain scene, and then reverse that ratio when drawing another. However, this scheduling mechanism is only suitable for graphics applications, which have relatively fixed computing patterns so that the performance is relatively easy to predict. However, GPGPU applications involve more complicated computation patterns and their performance is much more difficult to estimate.

Similar to the third category, our work focuses on microarchitecture optimization to support concurrent kernels. We provide a new solution for GPU-alike many-core microarchitectures to exploit the potential of simultaneously exploiting task-pipeline and data parallelism. Completely following the CUDA programming style, our hardware-based techniques do not require programmers to change their code explicitly.

6. Conclusion

The history of graphics processing units has consistently witnessed the enhancements of flexibility. Task-pipelined parallelism is a fundamental computing pattern that is commonly found in scientific and engineering applications, especially. As GPUs are playing an increasingly important role in computing, this work focuses on developing microarchitectural techniques to support efficient task-pipelined execution on GPUs. Based on an extensive quantitative analysis, we identify opportunities and challenges to utilize the task-pipelined parallelism. We propose a kernel scheduling mechanism that dynamically tunes the resource allocation between two concurrent kernels for an optimized overall throughput. We also introduce techniques to moderately modified GPU's L2 cache to ease the data transfer between kernels and reduce the off-chip memory access. Simulation results on real-world applications demonstrate the effectiveness and scalability of our techniques.

References

[1] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2009.
[2] AMD, "http://www.amd.com/us/products/desktop/graphics/7000/7970ghz/Pages/radeon-7970GHz.aspx," 2012.
[3] W. Mark, "Future graphics architectures," ACM Queue, March 2008.
[4] NVIDIA, "CUDA programming Guide 4.0," 2011.
[5] R. Stephens, "A survey of stream processing," Acta Informatica, vol.34, pp.491–54, 1995.
[6] M. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data and pipeline parallelism in stream programs," ASPLOS, 2006.

[7] V. Strumpen, H. Hoffmann, and A. Agarwal, "Stream algorithms and architecture," Journal of Instruction-Level Parallelism, no.6, pp.1–31, 2004.
[8] J. Gama and M. Mohamed, Learning from Data Streams – Processing Techniques in Sensor Networks, Springer, 2007.
[9] S. Mu, C. Wang, M. Liu, D. Li, M. Zhu, and X. Chen, "Evaluating the potential of graphics processors for high performance embedded computing," Design, Automation & Test in Europe Conference & Exhibition, pp.1–6, March 2011.
[10] A. Kerr, D. Campbell, and M. Richards, "GPU VSIPL: high-performance VSIPL implementation for GPUs," Proc. High Performance Embedded Computing Workshop, 2008.
[11] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," NEPLS, Aug. 2002.
[12] A. Udupa, R. Govindarajan, and M. Thazhuthaveetil, "Software pipelined execution of stream programs on GPUs," CGO, pp.200–209, 2009.
[13] A. Hagiescu, H. Huynh, W. Wong, and R. Goh, "Automated architecture-aware mapping of streaming applications onto GPUs," IEEE International Parallel and Distributed Processing Symposium, 2011.
[14] G.A. Elliott and J.H. Anderson, "Globally scheduled real-time multiprocessor systems with GPUs," Real-Time Systems, pp.34–74, Jan. 2012.
[15] J. Gummaraju and M. Rosenblum, "Stream programming on general purpose processors," Annual International Symposium on Microarchitecutre, pp.343–354, 2005.
[16] D. Sanchez, D. Lo, R.M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic fine-grain scheduling of pipeline parallelism," PACT, pp.22–32, Oct. 2011.
[17] S. Liao, Z. Du, G. Wu, and G. Lueh, "Data and computation transformations for brook streaming applications on multiprocessors," International Symposium on Code Generation and Optimization, pp.196–207, 2006.
[18] I. Buck, et al. "Brook for GPUs: Stream computing on graphics hardware," ACM Trans. Graphics, pp.777–786, Aug. 2004.
[19] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA tesla: A unified graphics computing architecture," IEEE Micro, vol.28, pp.39–55, March 2008.
[20] S. Tzeng, A. Patney, and J. Owens, "Task management for irregular-parallel workloads on the GPU," High Performance Graphics, pp.29–37, June 2010.
[21] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron, "Enabling task parallelism in the CUDA scheduler," Proc. Workshop on Programming Models for Emerging Architectures, pp.69–76, Sept. 2009.
[22] A. Tatarinov and A. Kharlamov, "Alternative rendering pipelines using NVIDIA CUDA," SIGGRAPH 2009, http://developer.nvidia.com/object/siggraph-2009, Aug. 2009.
[23] K. Gupta, J.A. Stuart, and J.D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," High Performance Graphics (HPG), 2011.
[24] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.
[25] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," Proc. International Symposium on Computer Architecture (ISCA), 2009.
[26] N. Lakshminarayan and H. Kim, "Effect of Instruction Fetch and Memory Scheduling on GPU Performance," Workshop on Language, Compiler, and Architecture Support for GPGPU, in conjunction with HPCA/PPoPP, 2010.
[27] High Performance Embedded Computing Benchmark Suites, MIT Lincoln Laboratory, http://www.ll.mit.edu/HPECchallenge.
[28] NVIDIA, "NVIDIA CUDA SDK code samples," http://developer.download.NVIDIA.com/compute/cuda/sdk/website/samples.html.
[29] S. Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary CMP workloads," Proc. IEEE International Symposium on Workload Characterization, Dec. 2010.
[30] A. Bakhoda, G. Yuan, L. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," ISPASS, pp.163–174, April 2009.
[31] S. Hong and H. Kim, "An integrated GPU power and performance model," ISCA, pp.280–289, June 2010.
[32] S. Huang, S. Xiao, and W. Feng, "On the energy efficiency of graphics processing units for scientific computing," IPDPS, pp.1–8, 2009.
[33] NVIDIA, Compute Visual Profiler: User Guide, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Compute_Visual_Profiler_User_Guide.pdf
[34] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," Proc. Conference on High Performance Computing Networking, Storage and Analysis, 2009.
[35] Y. Li, J. Dongarra, and S. Tomov, "A note on auto-tuning GEMM for GPUs," Proc. 9th International Conference on Computational Science (ICCS), pp.884–892, 2009.
[36] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3D stencil codes on GPU clusters," Proc. Tenth International Symposium on Code Generation and Optimization (CGO), pp.155–164, 2012.
[37] D. Luebke and G. Humphreys, "How GPUs work," Computer, vol.40, no.2, pp.96–100, 2007.

Shuai Mu received his B.S. degree in Institute of Microelectronics from the University of Tsinghua, China in 2008. He is currently pursuing the Ph.D. degree in Institute of Microelectronics in the University of Tsinghua. His research interests mainly focus on massively parallel computing and general purpose computing on graphics processing hardware (GPGPU).

Dongdong Li received his B.S. degree in Computer Science from Beihang University, Beijing, in 2012. He is currently a Ph.D Candidate in Computer Engineering of British Columbia University, Vancouver, Canada. His research interests include computer hardware, GPU architecture design.

Yubei Chen received his B.S. degree in Electrical and Electronics Department from Tsinghua University, Beijing, in 2012. He is currently a Ph. D. Candidate in Computer Science of University of California, Berkeley. His research interests mainly focus on the parallel system design for different application areas and human intelligence.

Yangdong Deng received his Ph.D. degree in Electrical and Computer Engineering from Carnegie Mellon University, Pittsburgh, PA, in 2006. He received his ME and BE degrees in Electrical and Electronics Department from Tsinghua University, Beijing, in 1998 and 1995, respectively. His research interests include parallel electronic design automation (EDA) algorithms, parallel program optimization and general purpose computing on graphics processing hardware.

Zhihua Wang received the B.S., M.S. and Ph.D. degree from Tsinghua University, Beijing, China in 1983, 1985, and 1990, respectively. His research interests include design methodology of integrated circuits and systems, lower power analog and RF ICs for medical and communication, high-speed real-time signal processing. He has published over 180 papers and 4 books, and he holds 25 patents, with over 10 pending, and accomplished over 15 research projects.