GPU Performance vs. Thread-Level Parallelism: Scalability Analysis and a Novel Way to Improve TLP

ZHEN LIN, North Carolina State University
MICHAEL MANTOR, Advanced Micro Devices
HUIYANG ZHOU, North Carolina State University

Graphics Processing Units (GPUs) leverage massive thread-level parallelism (TLP) to achieve high computation throughput and hide long memory latency. However, recent studies have shown that the GPU performance does not scale with the GPU occupancy or the degrees of TLP that a GPU supports, especially for memory-intensive workloads. The current understanding points to L1 D-cache contention or off-chip memory bandwidth. In this article, we perform a novel scalability analysis from the perspective of throughput utilization of various GPU components, including off-chip DRAM, multiple levels of caches, and the interconnect between L1 D-caches and L2 partitions. We show that the interconnect bandwidth is a critical bound for GPU performance scalability.

For the applications that do not have saturated throughput utilization on a particular resource, their performance scales well with increased TLP. To improve TLP for such applications efficiently, we propose a fast context switching approach. When a warp/thread block (TB) is stalled by a long latency operation, the context of the warp/TB is spilled to spare on-chip resource so that a new warp/TB can be launched. The switched-out warp/TB is switched back when another warp/TB is completed or switched out. With this fine-grain fast context switching, higher TLP can be supported without increasing the sizes of critical resources like the register file. Our experiment shows that the performance can be improved by up to 47% and a geometric mean of 22% for a set of applications with unsaturated throughput utilization. Compared to the state-of-the-art TLP improvement scheme, our proposed scheme achieves 12% higher performance on average and 16% for unsaturated benchmarks.

CCS Concepts: • **Computer systems organization** → *Single instruction, multiple data*;

Additional Key Words and Phrases: GPGPU, TLP, context switching, latency hiding

ACM Reference format:
Zhen Lin, Michael Mantor, and Huiyang Zhou. 2018. GPU Performance vs. Thread-Level Parallelism: Scalability Analysis and a Novel Way to Improve TLP. *ACM Trans. Archit. Code Optim.* 15, 1, Article 15 (March 2018), 21 pages.
https://doi.org/10.1145/3177964

We thank the anonymous reviewers for their valuable comments. This work is supported by NSF Grant No. CCF-1618509 and an AMD gift fund.
Authors' addresses: Z. Lin and H. Zhou, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, North Carolina; emails: {zlin4, hzhou}@ncsu.edu; M. Mantor, Advanced Micro Devices, Inc., Orlando, Florida; email: Michael.Mantor@amd.com.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 ACM 1544-3566/2018/03-ART15 $15.00
https://doi.org/10.1145/3177964

1 INTRODUCTION

State-of-the-art throughput-oriented processors, like Graphics Processing Units (GPUs), have become the dominant accelerator for data-parallel workloads. To achieve high computation throughput and memory bandwidth, GPUs exploit high degrees of thread-level parallelism (TLP). Programmers use the CUDA [2] or OpenCL [1] programming models to define the behavior of each thread. The GPU hardware aggregates multiple threads into a warp as the basic execution unit. The warp scheduler seeks for one ready instruction among multiple warps every cycle. Such fine-grain multithreading is the key to hide the memory latency. As a result, GPUs feature high amounts of on-chip resources to accommodate the contexts of large numbers of concurrent warps. For example, in the NVIDIA GP100 (Pascal) architecture [5], each stream multiprocessor (SM) has a 256KB register file and accommodates up to 64 warps.

The inclusion of multiple levels of caches complicates the relationship between TLP and the overall performance. As studied in prior works [14, 18, 27], high degrees of TLP cause the cache to suffer from the contention problem, which may lead to performance degradation. For example, Figure 1 shows the impact of cache thrashing with various degrees of TLP. The experiment setup is presented in Section 3. The figure shows that the L1 D-cache hit rate dramatically decreases when the number of concurrent warps increases from 1 to 24. Cache thrashing causes the performance to drop when the warp number is larger than 16. Rogers et al. [27] propose to limit the number of active/concurrent warps to alleviate the cache thrashing problem. However, as the warp number increases from 32 to 64, more TLP should have hidden more latency for accessing memory, since the cache performance does not significantly drop. But why isn't the performance improved? Some may conjecture that it is because the off-chip DRAM bandwidth is fully utilized, but Figure 1 shows that the DRAM bandwidth is fairly underutilized.

To find this mysterious performance limitation, we conduct a novel and detailed analysis from the perspective of throughput utilization of GPU components. The result shows that 8 out of 22 benchmarks are actually bounded by the interconnect bandwidth in the baseline architecture. The performance starts to drop when the interconnect throughput is saturated so that higher TLP cannot compensate the loss in cache throughput.

In our experiments, we find that some benchmarks do not fully utilize the throughput of any computation or memory bandwidth resource. These benchmarks would benefit from a higher degree of TLP. However, the number of concurrent warps is limited by the context capacity, such as the register file capacity or warp scheduler capacity, of the GPU. We also noticed that, for many benchmarks, resource usage is unbalanced. Often, shared memory is underutilized and/or the L1 D-cache performance is low. For these benchmarks, shared memory or the L1 D-cache can be used to accommodate more warp contexts.

In this article, we propose GPUDuet, a novel approach using context switching as another level of multithreading for GPU architecture. The key idea is to switch out stalled warps/thread blocks (TBs) to realize much higher degrees of TLP without increasing the size of critical physical resources. To achieve fast context switching, we only use on-chip memory to store the switched-out contexts. Through monitoring the throughput resource utilization, GPUDuet dynamically determines the best TLP degrees.

We evaluate our GPUDuet approach with 10 throughput-unsaturated benchmarks and 5 throughput-saturated ones. We find that it can significantly improve the performance for benchmarks without saturated resources. Compared to the baseline, our technique achieves up to 47% and an average (geometric mean) of 22% performance gain. Compared to the state-of-the-art TLP improvement scheme, Virtual Thread (VT) [35], our proposed scheme achieves 12% higher performance on average and 16% for unsaturated benchmarks.

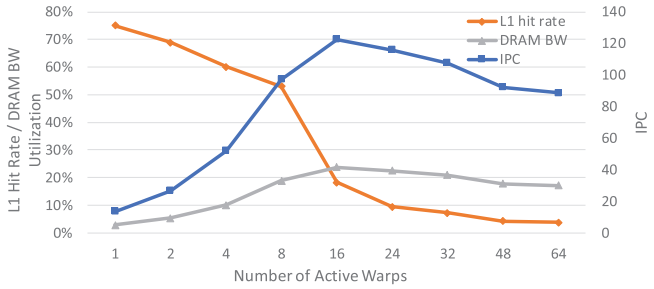


Fig. 1. Instruction per cycle (IPC), L1 D-cache hit rate, and DRAM bandwidth utilization with different numbers of active warps for the SPMV benchmark.

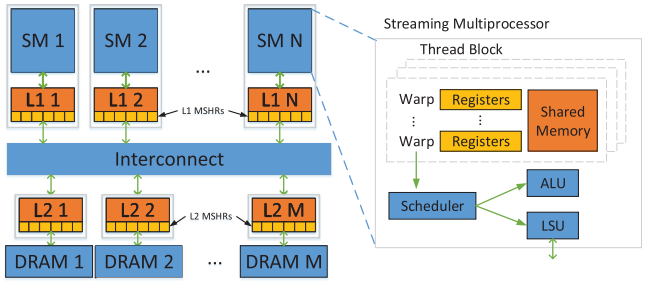


Fig. 2. GPU architecture.

In summary, this article makes the following contributions:

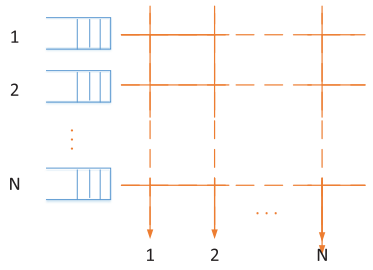
- We present a novel bottleneck analysis on GPU workloads from a resource utilization perspective. It reveals that the interconnect can be a critical bound in GPU performance and provides a detailed categorization of GPU workloads.
- Based on the resource utilization analysis, we reveal insights on GPU performance scalability. We also characterize the features of GPU workloads, whose performance can scale well with TLP.
- For the benchmarks whose performance scales with TLP, we propose lightweight context switching as another level of multithreading support to improve TLP and our experiments show that our proposed TLP improvement technique can significantly enhance the performance.

The rest of the article is organized as follows. Section 2 provides the background. Section 3 describes our experimental methodology. Section 4 presents the throughput utilization study and performance scalability analysis. Section 5 motivates our context switching approach to improve TLP. Section 6 details our GPUDuet approach. Section 7 reports the experiment results. Section 8 addresses the related work. Section 9 concludes the article.

2 BACKGROUND

2.1 GPU Architecture

Contemporary GPU architecture, as illustrated in Figure 2, consists of multiple stream multiprocessors (SMs). Each SM has a private L1 D-cache, which uses an array of miss status handling registers (MSHRs) to support multiple outstanding cache misses. All SMs share a multi-banked L2



(a) Crossbar diagram

N	Sustainable / Peak
1	100.00%
2	75.00%
3	68.25%
4	65.53%
5	63.99%
6	63.02%
7	62.34%
8	61.84%
inf	58.58%

(b) Sustainable throughput ratio

Fig. 3. A crossbar with queueing on inputs, assuming that the input number equals the output number [13].

cache, which also uses MSHRs to send memory requests to the off-chip DRAM. The L1 D-caches and L2 cache banks communicate through a crossbar interconnect network [4, 24].

Each SM hosts a number of warps, each of which is a collection of threads that run in the single-instruction multiple-data (SIMD) manner. Each warp has its private space in the register file and some meta-data, such as the program counter and active thread bit mask, in the warp scheduler to keep track of its execution status. Multiple warps constitute a thread block (TB), within which all warps can communicate and synchronize through shared memory. One SM can accommodate one or more TBs depending on their resource requirement. There are four types of resources that can limit the number of concurrent/active TBs on an SM: register file, shared memory, the warp scheduler slots, and the TB slots. The warp/TB slots include some meta data to keep track of the warp/TB execution status. The warp/TB slot size is much smaller than register file or shared memory [35]. A warp/TB will hold the resources during its whole lifetime. The resources will be released after it finishes execution.

2.2 Crossbar Interconnect Network

State-of-the-art GPUs use crossbars as the interconnect network to connect the SMs and L2 partitions to provide the tremendous data communication demand between them [4, 24]. The crossbar is a non-blocking interconnect network, which means a connection can be established if both input and output are idle. In this article, queueing on inputs (Figure 3(a)) is assumed for simplicity. If more than one packet needs to arrive at the same output at the same time, then only one packet can be accepted by the network and other packets have to be queued on their inputs. Due to such output contention, the sustainable throughput of the crossbar can hardly achieve the peak bandwidth. Based on the assumption that the destinations of packets are independent and uniformly distributed, Karol et al. [13] quantified the ratio of sustainable throughput and the peak bandwidth, as shown in Figure 3(b). As we can see from Figure 3(b), the ratio drops sharply as the number of inputs/outputs increases and saturates around 60% quickly.

3 METHODOLOGY

3.1 Simulation Infrastructure

We use GPGPU-sim v3.2.2 to investigate the GPU throughput utilization and evaluate our context switching approach. Our baseline architecture models the NVIDIA Maxwell architecture and its configuration is shown in Table 1. To get the register and shared memory usage based on Maxwell architecture, we use the nvcc compiler with "- -ptxas-options=-v -arch=sm_50" option. We observe that nvcc typically allocates more registers for the Maxwell architecture than for the older

Table 1. Baseline Architecture Configuration

Overall config.	16 SMs, 32 threads/warp, 16 L2 banks, 16 DRAM chips
SM config.	1,000MHz, 4 schedulers, GTO policy
Context resources/SM	32 TB slots, 64 warp slots, 256KB register file, 98KB shared memory
L1 D-cache/SM	32KB, 32 sets, 8 way, 128B block, 256 MSHRs
Interconnect	16*16 crossbar per direction, 32B wire width, 1200MHz, 614GB/s
L2 cache/partition	128KB, 8 way, 256 MSHRs, 200 cycles latency
DRAM	1,200MHz, 307GB/s, 450 cycles latency

architectures, such as Fermi. This configuration is important as it affects how the TLP of a kernel is limited. If we keep using the code generated for the old architecture, e.g., Fermi, then the register usage of a kernel is smaller than it would be for the Maxwell architecture. In this case, the kernel is more likely to be scheduling limited (due to the limited warp slots) although it should be capacity limited (due to the limited register file size) if we compile the code for the Maxwell architecture.

We implement the crossbar interconnect based on Section 2.2. The crossbar port width can be configured as 16B, 32B, or 64B. We use 16B wire width (307GB/s bandwidth) in Figure 1 and 32B wire width (614GB/s bandwidth) if not specified. When multiple inputs are trying to send to the same output, the crossbar randomly selects one input to connect with the output and the other ones wait for the selected one to finish. Our experiment shows that, for interconnect-intensive benchmarks, the attainable throughput saturates close to 60% of peak bandwidth, confirming the study by Karol et al. [13].

3.2 Benchmarks

We choose 22 GPU kernels from various benchmark suites, including Rodinia [7], Parboil [29], Polybench [11], and CUDA SDK [3], to investigate the throughput utilization on GPU components. Table 2 lists the details of the benchmarks along with their sources and input data sets.

4 DYNAMIC RESOURCE UTILIZATION

4.1 Experiment Description

In this section, we investigate the utilization of six GPU components, including the warp scheduler throughput, L1 D-cache bandwidth, interconnect sustainable throughput along both directions, L2 cache bandwidth, and DRAM bandwidth. In our experiments, the statistic counters are aggregated until the kernel has run for 4 million cycles. The reason is that the variation of statistic counters becomes very small after the kernel is simulated for 2 million cycles. Prior works [6, 31, 34] made similar observations.

The scheduler utilization is the ratio of total instructions issued over the total cycles of all schedulers (i.e., overall cycles X number of schedulers). It also reflects the latency hiding effectiveness for a benchmark.

The L1 D-cache and L2 cache throughput utilization is the ratio of achieved throughput over the peak bandwidth of the cache data port. The data port is utilized when there is a cache hit. The duration depends on the data port width and the requested data size.

The memory requests that miss the L1 D-cache, go through the interconnect along the SM-to-L2 direction. The corresponding replies go through the interconnect along the L2-to-SM direction. Ideally, each input node can transmit a flit (32B) in one cycle so the peak bandwidth is 614GB/s per direction (Table 1). However, due to the output contention as discussed in Section 2.2, at most 60% of the peak bandwidth can be achieved. So, we define the interconnect utilization as the ratio of the

Table 2. Benchmark Specification

Benchmark	Source	Description	Data Set
BFS	bfs [7]	Breath first search	16 million nodes
BP_1	backprop [7]	Machine learning	65,536 nodes
BP_2	backprop [7]	Machine learning	65,536 nodes
BT	b+tree [7]	Graph traversal	1 million nodes
CFD	cfd [7]	Fluid dynamics	0.2 million nodes
CONV	2DCONV [11]	2-D convolution	4,096*4,096 matrix
CP	cutcp [29]	Coulombic potential	large (96,602 nodes)
DWT_1	dwt2d [7]	Image/video compression	1,024*1,024 rgb image
DWT_2	dwt2d [7]	Image/video compression	1,024*1,024 rgb image
FDTD_1	FDTD-2D [11]	2-D stencil operation	2,048*2,048 nodes
FDTD_2	FDTD-2D [11]	2-D stencil operation	2,048*2,048 nodes
GEMM	GEMM [11]	Linear algebra	512*512 matrices
HG	hybridsort [7]	Sorting algorithms	4 million elements
HS	hotspot [7]	Physics simulation	512*512 nodes
KM	kmeans [7]	Data mining	494,020 nodes
LBM	lbm [29]	Fluid dynamics	long
MM	2MM [11]	Linear algebra	2,048*2,048 matrices
MMS	matrixMul [3]	Matrix multiplication	1,024*1,024 matrices
SAD_1	sad [29]	Sum of absolute differences	default
SAD_2	sad [29]	Sum of absolute differences	default
SPMV	spmv [29]	Sparse matrix vector multiply	large
ST	stencil [29]	3-D stencil operation	default

actual throughput over the sustainable throughput (i.e., 60% of the peak bandwidth) rather than the ratio over the peak bandwidth. If we directly use the peak bandwidth, then all benchmarks would show low interconnect utilization overlooking the nature of the sustainable bandwidth of such a crossbar interconnect network.

The DRAM throughput utilization is measured as the ratio of the achieved throughput of the DRAM data transfer over the peak DRAM bandwidth.

The bandwidth utilization of the register file and shared memory is also tested in our experiment. We don't report the details in the article, because they are not the performance bound for any of our benchmarks.

We also tested the instantaneous throughput utilization by collecting the statistic counters every P cycles. We find that the instantaneous utilization is stable if P is large enough (e.g., 10,000).

4.2 Overall Results

Figure 4 presents the overall results of throughput utilization. The Y-axis shows the ratio of achieved utilization of the six components as discussed in Section 4.1. The benchmarks are classified into five categories according to their most utilized component.

The first category is compute-intensive benchmarks, including SAD_1, BP_1, and ST. In this category, the scheduler utilization is high (over 60%), which means they have enough computation instructions to hide the memory latency such that instructions are issued without significant pipeline bubbles.

The second category is L2-to-SM interconnect-intensive benchmarks, including KM, CFD, SPMV, FDTD_2, MMS, and CONV. Those benchmarks have over 60% of the sustainable

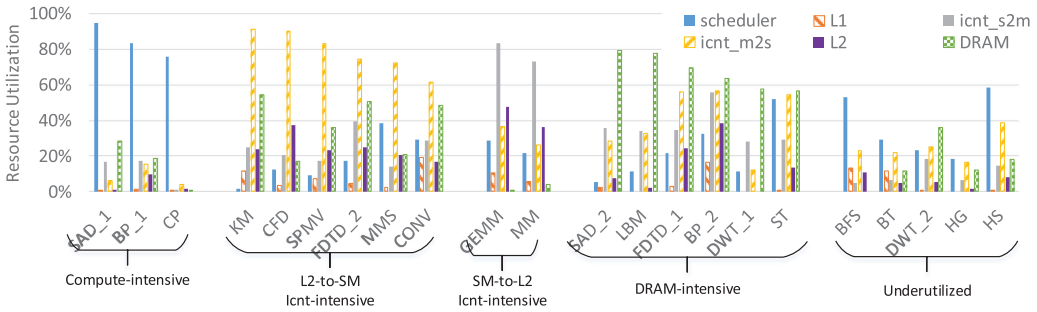


Fig. 4. Benchmark categorization based on throughput utilization of GPU resources.

interconnect throughput along the L2-to-SM direction. The memory replies from either the L2 cache (L2 hits) or DRAM (L2 MSHRs for L2 misses) need to go through this interconnect. Note that neither L2 nor DRAM throughput has been fully utilized for those benchmarks. It means the interconnect fails to provide the data to the SMs at the speed of L2 and DRAM replying the data. The bottleneck in the L2-to-SM interconnect will cause the buffer between L2 and interconnect to become full, which slows down the L2 and DRAM. The cascading effect will stuff all buffers along the data path and eventually cause the pipeline to stall at the SM load store units (LSUs).

The third category is SM-to-L2 interconnect-intensive benchmarks, including GEMM and MM. The SM-to-L2 direction of interconnect is responsible for transferring the memory load and store requests. And because load requests do not contain any payload data, the major consumption is from the store requests. Both GEMM and MM have excessive global memory stores, because they store all intermediate results to global memory rather than registers.

The fourth category is DRAM-intensive benchmarks. Because the DRAM throughput may suffer from row buffer misses, row activation or refreshing, the peak bandwidth is hard to achieve. X. Mei et al. [23] observed that the maximum achievable throughput can be 20%–30% lower than the peak bandwidth. In this article, we define the benchmarks that consumed more than 50% of DRAM peak bandwidth as DRAM-intensive, which includes SAD_2, LBM, FDTD_1, BP_2, DWT_1, and ST. Note that a common approach to categorizing the workloads on GPUs is based on the ratio of the LSU stalls [28, 31]. With this approach, both DRAM-intensive and interconnect-intensive benchmarks are classified as memory-intensive. However, such categorization can be misleading as our results clearly show that the DRAM bandwidth is not highly utilized for many interconnect-intensive benchmarks.

The fifth category, which includes BFS, BT, DWT_2, HG, and HS, doesn't fully utilize the throughput of any of the resources. Such benchmarks essentially lack active warps to fully utilize the throughput of the computation or memory resources. In Section 5, we show that increasing the number of active warps can significantly increase the performance of the benchmarks in this category.

From Figure 4, we can see that the L1 D-cache bandwidth utilization is low for most benchmarks, only 7 benchmarks have higher than 5% utilization. This is because the L1 D-cache size is too small for so many concurrent warps [27]. The L2 cache bandwidth utilization is relatively better than L1. There are 13 benchmarks that achieve higher than 10% L2 cache bandwidth utilization. Because L1 D-cache fails filtering the memory traffic effectively, L2 cache and DRAM become the main resources to answer the memory accesses. When L2 cache filters a large amount of memory traffic, the interconnect between L1 and L2 becomes very busy.

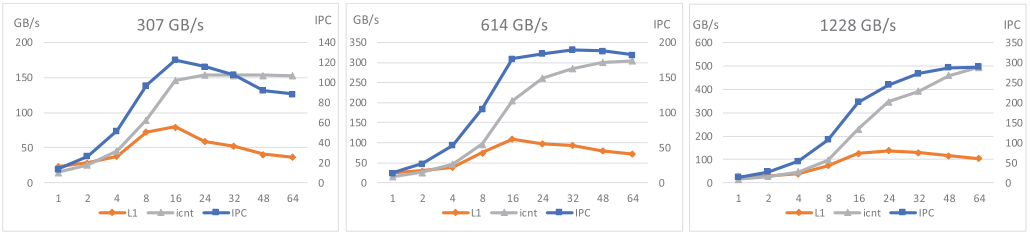


Fig. 5. The impact of varied numbers of active warps for different interconnect bandwidth on SPMV. The left Y-axis is the throughput of L1 D-cache and interconnect (icnt). The right Y-axis is the IPC. The X-axis is the number of active warps.

4.3 Scalability Analysis

In this section, we examine how and why the performance varies with increased TLP. In Figure 1, we have shown that the L1 D-cache performance is not enough to explain the performance degradation for the increased TLP. Here, we tackle the problem from the throughput perspective. Figure 5 presents three diagrams with the interconnect bandwidth varies from 307GB/s to 1,228GB/s through varying the wire width from 16B to 64B. Each diagram shows the IPC (right Y-axis), the throughput of L1 D-cache and interconnect (left Y-axis) with varied numbers of active warps (x-axis). As shown in the first diagram, the L1 D-cache throughput reaches the peak when the warp number is 16. Then, the interconnect bandwidth is saturated. Because the interconnect is saturated, the lost throughput from the L1 D-cache cannot be compensated by the interconnect. So, the performance keeps decreasing as increasing the warp number. In the second diagram, the performance degradation rate becomes smaller, because the interconnect can provide more throughput. In the third diagram, the performance degradation disappears, because the increased interconnect throughput can entirely compensate the decreased throughput of L1 D-cache. From the three diagrams, we can see that the performance reaches the peak when both L1 D-cache and interconnect throughput reach the peak.

Figure 5 also confirms that the interconnect becomes the bottleneck when the bandwidth utilization is close to the sustainable bandwidth. As we double the interconnect bandwidth, the performance increases accordingly.

5 MOTIVATION FOR HIGH TLP

5.1 Which Benchmarks Benefit From Increased TLP

As shown in Section 4.3, when a dynamic resource has been saturated, a benchmark cannot benefit from a higher degree of TLP. Instead, increasing TLP for the benchmarks that have not saturated any resource, should be helpful. To justify our hypothesis, we double the context resources per SM (Table 1) to accommodate the doubled number of warps and test the performance gains.

In Figure 6, the benchmarks from "underutilized" category in Figure 4 achieve the highest speedups. It means lacking active warps is the reason for their resource underutilization. For benchmarks that have saturated a certain resource, increasing the number of active warps can lead to either little improvement (e.g., SAD_1) or degradation (e.g., KM). For the other benchmarks, such as BP_1 and FDTD_1, although they have achieved relatively high utilization of certain resources, some improvement can still be achieved.

5.2 Virtual Thread

Virtual Thread (VT) [35] is a recent work leveraging warp slot virtualization to improve TLP. VT observes that the TLP of some benchmarks is limited by warp or TB slots while the register file and

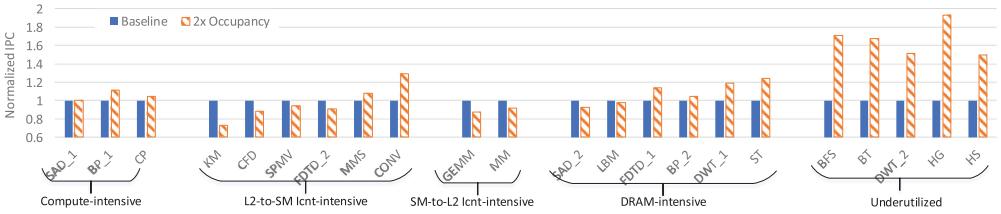


Fig. 6. Speedup for doubling the context resources per SM.

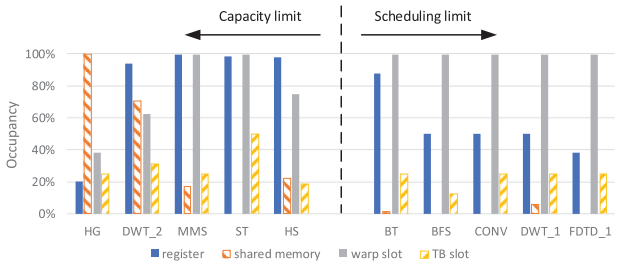


Fig. 7. Occupancy of GPU context resources.

shared memory have some leftover space. VT refers to the benchmarks that are limited by warp or TB slots as scheduling limited while the others are referred as capacity limited. For scheduling limited benchmarks, VT accommodates extra TBs using the spare register file and shared memory. The extra TBs are in the inactive state initially. When an active TB is stalled by memory operations, VT inactivates the stalled TB by moving the warp/TB slot data to shared memory. Then an inactive TB can become active using the vacant warp/TB slots.

There are several limitations of VT to improve TLP.

First, VT only targets at warp/TB slots and it does not work for capacity limited benchmarks. Figure 7 shows the occupancy of the context resources for the top 10 benchmarks that can benefit from increasing of TLP. Although five benchmarks are scheduling limited, the other five benchmarks are capacity limited, either by the register file or shared memory.

Second, VT only leverages TB-level context switching, which requires all warps of a TB to be stalled by the memory operations. Thus, VT loses the latency hiding opportunities for TBs with some but not all stalled warps.

Third, for some scheduling limited benchmarks, their TLP may become capacity limited after removing the scheduling limit with VT. Take BT as an example. Its TLP is initially limited by the warp slot. As its register file occupancy is 88%, the spare register file can only accommodate one more inactive TB using VT. Shared memory, on the other hand, remains highly underutilized.

Fourth, VT does not consider the dynamic resource utilization for improving TLP. As we highlighted in Section 5.1, the performance does not necessarily scale with increased TLP. Especially, for the benchmarks with saturated dynamic resources, increasing the TLP may even lead to performance degradation.

To address these limitations, we propose a novel warp-level context switching scheme. It improves TLP for both scheduling and capacity limited benchmarks, offers latency hiding at both warp and TB levels, and determines the proper degree of TLP based on dynamic resource utilization.

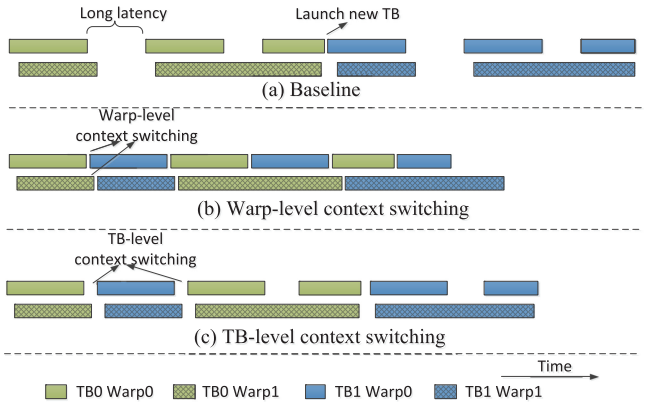


Fig. 8. Context switching to overlap long-latency stalls.

6 IMPROVING TLP WITH FAST CONTEXT SWITCHING

6.1 General Idea

As pointed out in Section 5, increasing TLP can improve the performance for those benchmarks that have not saturated any resources. In this section, we describe our approach to taking advantage of fast context switching for improving TLP. The basic idea is: when a warp/TB is stalled by a long latency operation, the context of the warp/TB is switched out so that the vacant resource can accommodate a new warp/TB. To achieve that, the context switching latency must be much smaller than the latency that we want to hide.

In this article, we consider two reasons that may cause a warp to experience long-latency stalls. The first is global memory loads that miss or bypass the L1 D-cache. On GPUs, the latency to access off-chip memory may take 400–800 cycles [32]. Barrier instructions are the second reason that may cause a warp to stall. When a warp reaches a barrier, it has to wait for other warps in the same TB to arrive. Liu et al. [21] report that the barrier latency can be as high as 3,800 cycles. We discuss additional motivation to enable context switching at barriers in Section 6.3.

To minimize the latency overhead of context switching, we analyze the context resources' occupancy in Figure 7. The number of active warps per SM is limited by four factors: the register file size, shared memory size, scheduler warp slots, and TB slots. From Figure 7, we can see that not all resources are fully occupied. So, we only need to switch out the critical resource that limits the number of active warps. Another key technique that enables fast context switching is that we use spare fast on-chip resources to store the switched-out contexts. Notice that the utilization of register file and shared memory is unbalanced for most benchmarks. So, we can use the spare resource to spill the limited resource. Based on our discussion in Section 4.2, we also use the L1 D-cache as another context buffer if its throughput is low. In addition, we leverage register liveness analysis to reduce the register number that needs to be swapped during context switching. The detail of resource management is described in Section 6.6. We refer to our approach as GPUDuet.

6.2 Latency Hiding with Two-Level Context Switching

We propose to use both warp-level and TB-level context switching for latency hiding. Let us use an example to illustrate how such a scheme works.

As shown in Figure 8, we assume that a kernel has 2 TBs and each TB contains 2 warps. In the baseline architecture, each SM can only accommodate 2 warps or 1 TB. As shown in Figure 8(a),

TB1 can only be launched after TB0 is finished. There are several long-latency events during execution.

In Figure 8(b), we assume that the kernel is either register limited or warp-slot limited (e.g., the warp scheduler only has two entries). In this case, we show that warp-level context switching can be used to overlap the execution latency. When warp0 in TB0 is stalled, we save its architectural states and release its register and warp slot. Then we can launch warp0 from TB1 into the SM. When warp1 in TB0 is stalled and dumped, depending on whether warp0 of TB0 is ready, we can switch warp0 of TB0 back or launch a new warp from TB1. In this way, long-latency gaps can be well hidden as shown in Figure 8(b).

When the kernel is shared memory limited and only one TB can be accommodated in the SM, warp-level context switching does not work, because no more TBs or their warps can be dispatched onto the SM. So, we propose to use TB-level context switching to hide latency in such cases. As shown in Figure 8(c), after warp0 in TB0 is stalled, we cannot launch warps from another TB. But when both warps in TB0 are stalled, we can switch out the architectural states of TB0 such that TB1 can be launched. When both warps in TB1 are stalled, TB0 can be switched back.

Between TB-level and warp-level context switching, TB-level context switching has to wait for all warps in a TB to stall before being switched out, thereby losing latency hiding opportunities compared to warp-level context switching. Therefore, we use TB-level context switching only for shared-memory limited benchmarks.

Note that our approach is not applicable for the benchmarks that are limited by both register and shared memory. In such a scenario, the spare on-chip resource to spill the context is very limited. Besides, the context switching overhead to switch in/out both register and shared memory would be much higher.

6.3 Deadlock Avoidance

If we only enable context switching for memory operations, then the warp-level context switching may suffer from deadlocks in the scenario when all TBs are partially active, meaning that some warps are active and some warps in the same TB are switched out. If all active warps have reached the barriers, then they will be waiting at the barriers forever. This is because the inactive warps may not have enough resource to be switched back, they will never reach the barriers.

The deadlock problem can be solved in two ways.

The first is to always maintain one TB as fully active, which means to disable the context switching of one TB per SM. In this way, when the other TBs are deadlocked, the inactive warps can be switched back when the fully active TB has finished and deallocated all its resources. However, because there is no guarantee of the TB execution time, other TBs may suffer a long period of idle time. Besides, latency hiding effect is reduced as we need to maintain a fully active TB.

The second is to make all warps that wait at the barriers eligible to be switched out. Then, the stalled inactive warps that have not reached the barrier will be able to switch back. Note that a barrier does not require all warps in a TB to reach the barrier at the same time. As long as all warps reach the barrier, even if some of them are switched out after reaching the barrier, the synchronization is achieved and all the warps can continue execution past the barrier. In this article, we use the second way to avoid deadlocks.

6.4 Determining the Number of Extra TBs

To determine how many extra TBs to accommodate in an SM, we first introduce some terms. We use MC to denote the maximum number of TBs that can be natively accommodated per SM. And ME is used to denote the maximum number of TBs that can be launched without all their contexts being active. In other words, $(MC + ME)$ is the maximum number of concurrent TBs per SM using

Table 3. Throughput Utilization Classification

Classification	Throughput Utilization Criteria
Underutilized	Scheduler < 60% and Interconnect < 60% and DRAM < 50% and L1 < 60% and L2 < 60%
Saturated	Scheduler > 80% or Interconnect > 90% or DRAM > 70% or L1 > 80% or L2 > 80%
Moderately Utilized	Others

our proposed GPUDuet approach. However, only ($N \times MC$) warps can be scheduled to run in each cycle, where N denotes the number of warps per TB. We refer to the type of context (e.g., register file) that needs to be switched out to accommodate the ME extra TBs as the limiting context.

As observed in Section 5, increasing the TLP is most helpful for benchmarks that have relatively low throughput utilization. For benchmarks that have high throughput utilization, increasing TLP can lead to little performance improvement or even performance degradation. To determine the most beneficial ME, we first compute its upper bound statically. Then, we initialize ME as 0 when a kernel is launched and adjust it based on dynamic resource utilization. The upper-bound of ME is set as the minimum among the following factors:

- The maximum number of concurrent warps, which are managed by the warp context table (Section 6.6). The number used in this article is 128, compared to the baseline number of warp slots as 64.
- The maximum number of concurrent TBs, which are managed by TB context table (Section 6.6). The number used in this article is 32, which equals to the maximum TB number in the baseline architecture. This is because we observe that the TB number rarely becomes the limitation (Table 7).
- The maximum number of TBs that can be supported considering the unified resource of the register file, shared memory, and L1 D-cache (i.e., unified resource size/TB context size). The reason is that the total register, shared memory and warp slot usage cannot exceed the total capacity of on-chip memories.
- The maximum number of TBs when the register and shared memory are not the limiting context at the same time. When accommodating one extra TB would need both registers and shared memory data to be switched out, the TB-level context switching overhead (solely using the L1 D-cache) becomes too much. In other words, our approach does not work for the benchmarks that are limited by both register file and shared memory.

To collect dynamic resource utilization, GPUDuet implements a throughput monitor to collect the utilization of the scheduler, L1 D-cache, interconnect, L2 cache, and DRAM. The monitor skips the first 10,000 cycles to let the GPU warm up. Then the monitor collects the counters for 10,000 cycles and updates the ME for the first time. As shown in Table 3, the benchmarks are categorized into "underutilized," "moderately utilized," and "saturated." Moderately utilized and underutilized kernels are also referred as "unsaturated" in this article. For underutilized kernels, the ME is set at the upper-bound while the ME is set to zero, i.e., disabling GPUDuet, for saturated kernels. For moderately utilized kernels, the ME is initialized to zero and GPUDuet increases the ME by 1 every 10,000 cycles until it reaches the upper-bound or any resource becomes saturated. For unsaturated benchmarks, if there is no extra space in the register file or shared memory and the L1-D cache bandwidth utilization is low (less than 3%), the L1-D cache is entirely bypassed and used to store the spilled context.

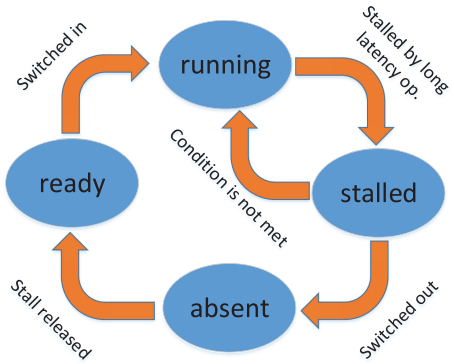


Fig. 9. Warp state transition diagram.

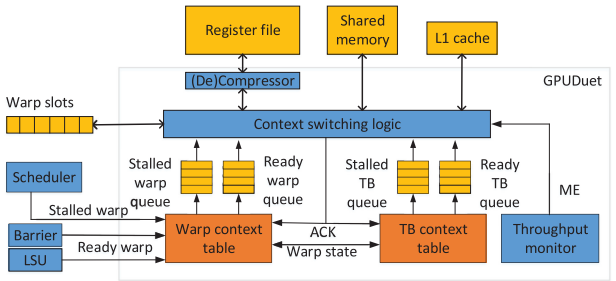


Fig. 10. Block diagram of the GPUDuet architecture.

6.5 Warp States

In GPUDuet, each warp can be in one of the following four states: running, stalled, absent and ready. Figure 9 shows the state transition diagram. A running state means all contexts of a warp are present and the warp is not stalled by a long latency operation. When a running warp is stalled by a long latency operation, its state is changed to the stalled state. A stalled warp will wait to be switched out. Before switching out, GPUDuet checks the switching out condition of the warp. If the condition is not met, then the warp state returns to running. Otherwise, the warp is switched out and its state is changed to be absent. When the long operation is finished, the absent warp becomes ready and wait to be switched in. When there is enough context resource, the warp is switched in and the warp state is changed to running.

If the ME is 0, then all warp states are initialized as running. If the ME is larger than 0, then the state of a newly launched warp is initialized as running if all contexts are present. Otherwise, the state is initialized as ready.

6.6 GPUDuet Architecture

Our proposed GPUDuet architecture is shown in Figure 10. It manages all the on-chip storage and determines when to switch the context of a warp or a TB. The context of a warp includes the registers and meta-data that keeps track of its execution state. The meta-data includes the program counter, the reconvergence stack entries [9] and the score board entries. In Figure 10, the meta-data structures are abstracted as a warp slot array indexed by the (physical) warp id. Besides the contexts of its warps, the context of a TB includes its shared memory. In our context management policy, the contexts can be either present in their original structure or switched out

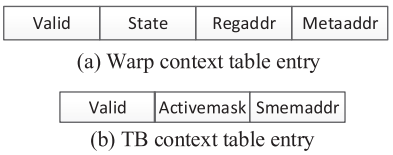


Fig. 11. Warp and TB context table entry.

to the register file/shared memory/L1 D-cache. Only when all contexts of a warp and the context of its corresponding TB are present, the warp can be scheduled by the warp scheduler. As discussed in Section 6.2, GPUDuet performs TB-level context switching only when the TLP of a kernel is limited by shared memory.

The on-chip memory managed by GPUDuet includes the register file, shared memory and L1 D-cache. To manage them, GPUDuet unifies their logical address spaces. The addressable unit is 128B, the same as the vector register width. Note that these memory components remain as separate physical components and there is no change in their hardware organizations. Because the total capacity of the on-chip memory is 384KB (256KB+96KB+32KB), the address space for register file is set as 0x000-0x7ff; the address space for shared memory is set as 0x800-0xaff; and the address space for the L1 D-cache is set as 0xb00-0xbff. A bit map is used to keep track of allocated context resource. Each set bit represents whether a corresponding 128B is allocated. Because the total memory size is 384KB, the bit map has 3,072 bits.

Warp Context Table and TB Context Table: In the GPUDuet architecture shown in Figure 10, the warp context table and TB context table keep the states of the warps and TBs, respectively. The warp context table (WCT) has 128 entries to support up to 128 active warps and is indexed by the warp id. As shown in Figure 11(a), each entry includes four fields: a 1-bit "Valid" field, a 2-bit "State" field indicating the warp state (Figure 9), a 12-bit "Regaddr" field representing the base address of the warp, which can point to anywhere in the unified on-chip memory space and the register context is present only when it falls into the register file address space, and a 12-bit "Metaaddr" field showing where the meta data of the warp is maintained: the index to the warp-slot array if it is present or the spilled address otherwise. The size of the WCT is 3,456 bits (=128 × (1+2+12+12)).

The TB context table (TBCT) is indexed by the TB id. There are 16 entries in the TBCT and each entry contains the following fields as shown in Figure 11(b). "Valid" indicate whether the TB has been launched. "Absmask" is a 32-bit mask indicating which warps in the TB are absent. "Smemaddr" is a 12-bit field pointing to the base address of the shared memory context. The shared memory context is present if the address falls into the shared memory address space. The size of the TBCT is 1,440 bits =(32 × (1+32+12)).

Context Switching Logic: The context switching logic (CSL) is used to spill or restore the limiting context resource of warps/TBs. If the limiting context resource is registers and/or warp slots, then warp-level context switching is used. If the limiting resource is shared memory or both shared memory and warp slots, then TB-level context switching is used. When both registers and shared memory are limiting resources, GPUDuet is disabled.

In warp-level context switching, the CSL switches out a stalled warp when there is enough space in shared memory or the L1 D-cache. It switches in a ready warp when there is enough space in the register file and warp-slot array. When there is both a stalled warp and a ready warp, the CSL will switch the context spaces of them. After switching out/in a warp, the CSL notifies the WCT to update the warp state and changes the address if necessary.

In TB-level context switching, if all warps of a TB have been stalled, the TB is considered as stalled and being pushed to the stalled TB queue. Then the CSL spills shared memory and the warp slots to the register file or L1 D-cache if either has enough space. When all warps, except the ones that are stalled by a barrier, in a TB are ready, the TB is ready to be switched back. Also, if there is a TB in the stalled queue and a TB in the ready queue at the same time, the CSL swaps the context space of them.

Register Context Reduction: We leverage register liveness and compression as proposed by Lin et al. [19] to reduce register context size before context switching. Such liveness and compression can be done either with additional architectural support at run-time or by static analysis using the compiler. In our experiments, we use the compiler approach to simplify the hardware complexity. In our approach, the switch out points can be either the first uses of registers to be loaded from global memory or the barriers. The compiler analyzes the reduced register size at such points with liveness and compression. Then, GPUDuet uses the maximum size among those points as the register allocation size so that we can use the same fixed size to spill live and compressed registers during context switching. In our experiment, the register allocation size is 44% of the original size on average.

6.7 Context Switching Latency

In this article, we model the latency of context switching as follows:

$$Latency = State/BW. \tag{1}$$

In Equation (1), the $State$ represents the total size of the architectural states which are going to be spilled or restored. We assume the bandwidth (BW) of the on-chip memory is 128bytes/cycle for the baseline architecture.

For warp-level context switching, the architectural state size for a warp is calculated as

$$State(warp) = ST + LR \times 128, \tag{2}$$

where ST denotes to the size of a warp slot, which includes the reconvergence stack entries and the score board entries. Fung et al. [9] discussed the details of the size of warp slots. Because GPUDuet only saves live registers, LR is the number of live registers, each of which is 128bytes.

For TB-level context switching, the kernel is limited by shared memory. So the architectural state size for a TB is

$$State(TB) = N \times ST + SS. \tag{3}$$

In Equation (3), N is the number of warps in a TB and SS is the size of shared memory per TB. As discussed in Section 6.4, GPUDuet does not allow registers and shared memory to be spilled at the same time. So, the registers do not need to be saved for TB-level context switching.

6.8 Hardware Overhead Comparing to Virtual Thread

The storage overhead of Virtual Thread (VT) [35] mainly includes the TB status table and the virtual warp IDs in the scoreboard. The TB status table is used to keep track the number of stalled warps in a TB. When all warps in a TB are stalled, the TB is marked as inactive and prepared to be swapped out. In this article, the TB status table size is 32(maximum TB number) × 7(entry size) = 224 bits. The active virtual warp ID storage is used to map the physical warp IDs to virtual warp IDs. We use the following setup: 64(maximum number of physical warps) × 7(size of each virtual warp ID) = 448 bits.

Table 4 summarizes the storage overhead per SM of GPUDuet architecture. Compared to VT, GPUDuet requires higher storage overhead (1,034bytes per SM vs. 84bytes per SM) and additional logic to manage the register file, shared memory, and L1 D-cache as a unified logical memory

Table 4. Storage Overhead per SM

Component	Entry size	# Entries	Total
Allocation bit map	3,072 bits	1	3,072 bits
Warp context table	27 bits	128	3,456 bits
TB context table	45 bits	32	1,440 bits
Stalled/ready warp queues	7 bits	16/16	224 bits
Stalled/ready TB queues	5 bits	8/8	80 bits

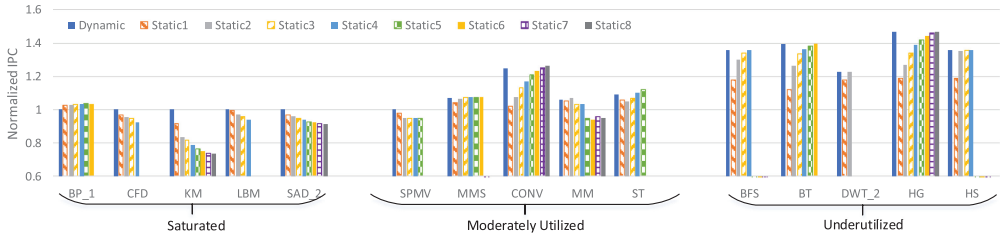


Fig. 12. Normalized speedups of GPUDuet with varied numbers of maximum extra TBs (ME).

region and support both warp-level and TB-level context switching, whereas VT only supports TB-level context switching. To justify such hardware overhead, the performance improvement of GPUDuet compared to VT will be discussed in Section 7.2.

7 EVALUATION

In this section, we evaluate the performance of GPUDuet compared to the baseline architecture and a state-of-the-art approach, VT, to improve GPU TLP. As described in Section 6.4, GPUDuet leverages a training phase to determine the number of extra TBs and the overall performance of GPUDuet includes such training phases. In our evaluation, we run each benchmark for 4 million cycles unless it finishes within 4 million cycles. The same methodology is used for the baseline, VT, and GPUDuet.

7.1 Impact of Maximum Extra TBs

In Figure 12, we select 15 benchmarks (5 from each category in Table 3) to evaluate the performance speedups with a varied number of maximum extra TBs (ME). For each benchmark, the ME can be either dynamically adjusted using the algorithm as described in Section 6.4 or statically selected. The upper-bound of the static ME for each benchmark depends on the context occupancy and the reduction ratio of the registers.

For DWT_2, the L1 D-cache is bypassed and used to store the spilled context. Therefore, we use the baseline architecture with the bypassed L1 D-cache for this benchmark to compute the normalized performance. The performance of DWT_2 decreases by 5% with a bypassed L1 D-cache. However, from Figure 12, we can see that the performance improvement is much higher than 5% with higher TLP.

As shown in Table 3, the benchmarks are classified into three categories. For underutilized benchmarks, which have low utilization for any computation and memory bandwidth resources, GPUDuet has the most significant improvement. These benchmarks show good performance scalability as their performance increases as ME increases and all five underutilized benchmarks achieve the best performance at the upper-bound of ME. For benchmarks in this category, the dynamic ME algorithm can effectively find the ME with the optimal performance. The performance improvement of the underutilized benchmarks ranges from 23% to 47%.

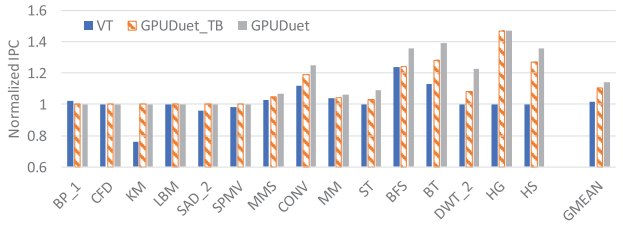


Fig. 13. Performance comparison of GPUDUet with Virtual Thread (VT) and TB-level GPUDuet.

For moderately utilized kernels, because the computation or bandwidth resources are not fully utilized, increasing TLP with GPUDuet can still achieve better performance for most benchmarks. For MMS, CONV, MM and ST, the performance speedups range from 7% to 25%. For SPMV, however, the increasing of TLP aggravates the cache thrashing, which causes the interconnect bandwidth to be saturated. So, increasing ME can only degrade the performance of SPMV. For MM, because the interconnect bandwidth is not saturated in the baseline architecture, increasing the ME from 1 to 3 improves the performance. However, the performance starts to drop when ME is larger than 3 due to the cache thrashing and saturated interconnect bandwidth. In this category, the dynamic ME selection algorithm searches for the optimal ME that can maximize the performance.

For saturated benchmarks, the computation or memory bandwidth cannot provide much room for higher TLP. So, BP_1 have very small performance improvement. For CFD, KM, LBM, and SAD_2, the cache throughput suffers from higher TLP while the interconnect or DRAM cannot compensate the loss. For KM, the performance degradation can be as high as 26%. Therefore, the algorithm in Section 6.4 dynamically sets ME to 0, disabling GPUDuet for such benchmarks with saturated throughput utilization.

7.2 Comparing with Virtual Thread

In Figure 13, we report the normalized IPC of VT [35] and GPUDuet compared to the baseline GPU described in Table 1. To highlight the benefit of warp-level context switching, we also show GPUDuet with only TB-level context switching, which is labeled as "GPUDuet_TB."

Because the TLP of CFD, LBM, ST, MMS, DWT_2, HG, and HS is limited by the register file, no extra TBs can be accommodated by VT. However, by spilling registers to shared memory or L1 D-cache, GPUDuet can achieve higher degrees of TLP for these benchmarks. Take DWT_2 as an example, the baseline architecture can accommodate 10TBs. With register size reduction, the register context size can be reduced to 50%. So the L1-D cache can accommodate the registers of extra 2TBs. The shared memory of the extra 2TBs can reside in the spare shared memory. As HG is limited by shared memory, it is not supported by VT either. With GPUDuet, the shared memory data can be moved to the register file to accommodate an extra 8TBs.

Even for the scheduling-limited benchmarks (i.e., their TLP is limited by the warp slots), GPUDuet can achieve much higher degrees of TLP. Take BT as an example, the spare space of the register file can only accommodate 1 extra TBs. Whereas for GPUDuet, an extra 6TBs can be accommodated by using shared memory to spill registers.

For KM, VT performs worse than the baseline. This is because KM saturates the interconnect bandwidth, increasing the TLP will cause L1 D-cache thrashing and hurt the performance. By leveraging dynamic resource utilization information, GPUDuet disables context switching for KM.

For benchmarks that are limited by registers or warp slots, such as HS and BFS, warp-level context switching performs better than TB-level context switching. This is because warp-level context switching takes advantage of more latency hiding opportunities, as discussed in Section 6.2.

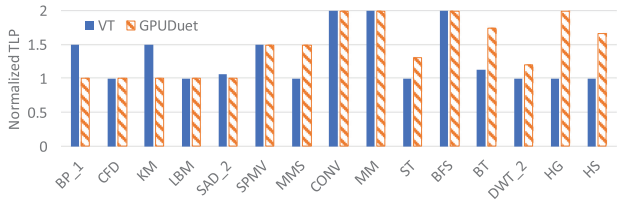


Fig. 14. TLP improvement by Virtual Thread (VT) and GPUDuet over the baseline architecture.

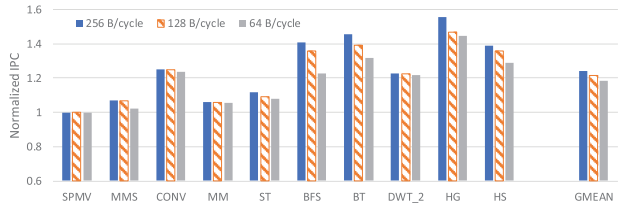


Fig. 15. Impact of context switching latency.

Compared to VT, GPUDuet improves the overall performance by 12% and 16% for unsaturated benchmarks.

7.3 TLP Improvement

In Figure 14, we compare the TLP improvement over the baseline architecture by VT and GPUDuet. The TLP is evaluated as the maximum number of TBs that can be launched to each SM. For the unsaturated benchmarks, such as HG and BT, GPUDuet achieves much higher TLP than VT. This is because VT only works for scheduling-limited benchmarks while GPUDuet also works for capacity-limited ones. In this category, compared to the baseline architecture, the TLP improvement by VT is 18% whereas 70% for GPUDuet. For the saturated benchmarks, such as BP_1 and KM, GPUDuet disables the context switching, therefore VT has higher TLP. However, higher degrees of TLP for these saturated benchmarks lead to little performance improvement or even performance degradation.

7.4 Impact of Context Switching Latency

In Figure 15, we evaluate the impact of context switching bandwidth of GPUDuet. In the baseline architecture, we assume the read/write port width of the register file, shared memory and L1 D-cache is 128bytes. In Figure 15, we evaluate the performance of GPUDuet with 256byte, 128byte, and 64byte port width. In general, benchmarks with high register file or shared memory utilization, such as HG and BT, tend to be more sensitive to the swapping bandwidth. Because such benchmarks have larger architectural states to move during context switching. Comparing to the baseline architecture, the performance improvement of GPUDuet with 256byte, 128byte, and 64byte port width are 24%, 22%, and 18%, respectively.

8 RELATED WORKS

Prior studies on the impact of TLP on throughput-oriented processors, including References [14, 18, 27], are discussed in Section 4.

A few prior works focus on improving the utilization of GPU resources. In Section 5.2 and 7.2, we discussed Virtual Thread [35]. Warped-Slicer [34] proposes an analytical model to determine

the optimal resource partition on one SM for concurrent kernels. Elastic kernel [25] increases register and shared memory occupancy by issuing concurrent kernels on one SM. SMK [31] proposes a dynamic sharing mechanism for concurrent kernels to improve the resource utilization while maintaining the fairness. KernelMerge [12] and Spacial Multiplexing [6] study how to use concurrent kernels to better utilize GPU resources and improve overall throughput. Lee et al. [15] also leverage mixed concurrent kernels to improve GPU utilization. In this work, we target at resource utilization for a single kernel.

Some works observe the GPU underutilization problem for particular scenarios. SAWS [20] and BAWS [21] propose barrier-aware scheduling policies to avoid warps from waiting at barriers for too long. CAWS [17] and CAWA [16] predict and accelerate the warps that lag behind so that the TB can finish faster. Warpman [33] points out the spacial and temporal resource underutilization due to TB-level resource allocation and propose warp-level resource utilization to improve the GPU resource utilization. Although those works can improve the effective TLP at barriers or TB terminations, they are limited by the maximum warp number that allowed being issued to the GPU.

Zorua [30] is a recent work that leverages context virtualization on GPU to provide programming portability and achieve higher levels of TLP. There are two key differences between Zorua and our approach. First, Zorua allocates/deallocates on-chip resources at the phase boundaries. Whereas our approach deallocates the resources when a warp/TB suffers from a long latency operation. Second, Zorua spills the oversubscribed register file and shared memory to global memory while our work leverages the spare on-chip resources to achieve much faster context switching.

Gebhart et al. [10] propose to unify the L1 D-cache, shared memory and register file to improve GPU on-chip resource utilization. We have a similar benefit in terms of increasing occupancy for register- or shared memory-limited applications. But the unified design requires extensive hardware changes and also needs software support. Besides, they have to pay overhead for repartitioning as different kernels have different resource requirements.

Some prior works [19, 26] leverage context switching for preemption on GPUs. Similar to proposed by Lin et al. [19], we leverage liveness analysis and register compression to reduce the context size. However, in this work, we use spare on-chip resources to store the spilled contexts to enable much faster context switching.

Majumdar et al. [22] study the scalability of the GPU kernels with computation units and memory bandwidth. Dublish et al. [8] perform bottleneck analysis on different levels of GPU memory hierarchy, including L1/L2 caches and DRAM. However, neither of them discusses the performance impact of the interconnect between L1 and L2.

9 CONCLUSIONS

In this article, we analyze the relationship between GPU performance and TLP through a novel resource utilization perspective. The GPU performance can be bounded by the scheduler throughput, L1 bandwidth, interconnect bandwidth, L2 bandwidth or DRAM bandwidth. We reveal that many memory-intensive benchmarks are actually bounded by interconnect bandwidth. Then we highlight that for benchmarks not saturating the throughput of any resources, increasing TLP can lead to significant performance improvement. We propose GPUDuet, a fast context switching approach, to increase TLP to better hide the latency of memory load and barrier operations. GPUDuet leverages spare on-chip resources to enable fast context switching. Liveness analysis and register compression are used to reduce the spilled context size. The evaluation shows that GPUDuet achieves up to 47% performance improvement and 22% on average for a set of benchmarks with unsaturated thr oughput utilization.

REFERENCES

[1] 2010. OpenCL—The open standard for parallel programming. KHRONOS Group (2010).

[2] 2011. CCUDA C programming guide. NVIDIA Corporation (2011).

[3] 2011. CUDA C/C++ SDK CODE samples. NVIDIA Corporation (2011).

[4] 2012. AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE White Paper. AMD Corporation (2012).

[5] 2016. NVIDIA tesla P100 whitepaper. NVIDIA Corporation (2016).

[6] J. T. Adriaens, K. Compton, Nam Sung Kim, and M. J. Schulte. 2012. The case for GPGPU spatial multitasking. In *Proceedings of the 2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA'12)*. 1–12. DOI: http://dx.doi.org/10.1109/HPCA.2012.6168946

[7] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J. W. Sheaffer, Sang-Ha Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*.

[8] S. Dublish, V. Nagarajan, and N. Topham. 2016. Characterizing memory bottlenecks in GPGPU workloads. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC'16)*. 1–2. DOI: http://dx.doi.org/10.1109/IISWC.2016.7581287

[9] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2009. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM Trans. Architect. Code Optim.* 6, 2, Article 7 (July 2009), 37 pages. DOI: http://dx.doi.org/10.1145/1543753.1543756

[10] Mark Gebhart, Stephen W. Keckler, Brucek Khailany, Ronny Krashinsky, and William J. Dally. 2012. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE Computer Society, Washington, DC, 96–106. DOI: http://dx.doi.org/10.1109/MICRO.2012.18

[11] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Proceedings of the Conference on Innovative Parallel Computing (InPar'12)*. 1–10.

[12] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. 2012. Fine-grained resource sharing for concurrent GPGPU kernels. In *Presented as Part of the 4th USENIX Workshop on Hot Topics in Parallelism*. USENIX, Berkeley, CA. Retrieved from https://www.usenix.org/conference/hotpar12/fine-grained-resource-sharing-concurrent-gpgpu-kernels.

[13] M. Karol, M. Hluchyj, and S. Morgan. 1987. Input versus output queueing on a space-division packet switch. *IEEE Trans. Commun.* 35, 12 (Dec 1987), 1347–1356. DOI: http://dx.doi.org/10.1109/TCOM.1987.1096719

[14] O. KayÄsŕan, A. Jog, M. T. Kandemir, and C. R. Das. 2013. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 157–166. DOI: http://dx.doi.org/10.1109/PACT.2013.6618813

[15] Minseok Lee, Seokwoo Song, Joosik Moon, J. Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 260–271. DOI: http://dx.doi.org/10.1109/HPCA.2014.6835937

[16] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. 2015. CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 515–527. DOI: http://dx.doi.org/10.1145/2749469.2750418

[17] Shin-Ying Lee and Carole-Jean Wu. 2014. CAWS: Criticality-aware warp scheduling for GPGPU workloads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. ACM, New York, NY, 175–186. DOI: http://dx.doi.org/10.1145/2628071.2628107

[18] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder. 2015. Priority-based cache allocation in throughput processors. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. 89–100. DOI: http://dx.doi.org/10.1109/HPCA.2015.7056024

[19] Zhen Lin, Lars Nyland, and Huiyang Zhou. 2016. Enabling efficient preemption for SIMT architectures with lightweight context switching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*.

[20] Jiwei Liu, Jun Yang, and Rami Melhem. 2015. SAWS: Synchronization aware GPGPU warp scheduling for multiple independent warp schedulers. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*. ACM, New York, NY, 383–394. DOI: http://dx.doi.org/10.1145/2830772.2830822

[21] Yuxi Liu, Zhibin Yu, Lieven Eeckhout, Vijay Janapa Reddi, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Chengzhong Xu. 2016. Barrier-aware warp scheduling for throughput processors. In *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*. ACM, New York, NY, Article 42, 12 pages. DOI: http://dx.doi.org/10.1145/2925426.2926267

[22] A. Majumdar, G. Wu, K. Dev, J. L. Greathouse, I. Paul, W. Huang, A. K. Venugopal, L. Piga, C. Freitag, and S. Puthoor. 2015. A taxonomy of GPGPU performance scaling. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*. 118–119. DOI: http://dx.doi.org/10.1109/IISWC.2015.22

[23] X. Mei and X. Chu. 2016. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans. Parallel Distrib. Syst.* PP, 99 (2016), 1–1. DOI: http://dx.doi.org/10.1109/TPDS.2016.2549523

[24] Online:. 2016. Diving Deeper: The Maxwell 2 Memory Crossbar & ROP Partitions. Retrieved from http://www.anandtech.com/show/8935/geforce-gtx-970-correcting-the-specs-exploring-memory-allocation/2.

[25] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 407–418. DOI: http://dx.doi.org/10.1145/2451116.2451160

[26] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative preemption for multitasking on a shared GPU. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 593–606. DOI: http://dx.doi.org/10.1145/2694344.2694346

[27] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE Computer Society, Washington, DC, 72–83. DOI: http://dx.doi.org/10.1109/MICRO.2012.16

[28] A. Sethia, D. A. Jamshidi, and S. Mahlke. 2015. Mascar: Speeding up GPU warps by reducing memory pitstops. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. 174–185. DOI: http://dx.doi.org/10.1109/HPCA.2015.7056031

[29] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, vLi Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *IMPACT Technical Report* (2012).

[30] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira KhanâĎţ, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B. Gibbons, and Onur Mutlu. 2016. Zorua: A holistic approach to resource virtualization in GPUs. In *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*.

[31] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 358–369. DOI: http://dx.doi.org/10.1109/HPCA.2016.7446078

[32] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*. 235–246. DOI: http://dx.doi.org/10.1109/ISPASS.2010.5452013

[33] P. Xiang, Y. Yang, and H. Zhou. 2014. Warp-level divergence in GPUs: Characterization, impact, and mitigation. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 284–295. DOI: http://dx.doi.org/10.1109/HPCA.2014.6835939

[34] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming. In *Proceedings of the ACM International Symposium on Computer Architecture (ISCA'16)*.

[35] Myung Kuk Yoon, Keunsoo Kim, Sangpil Lee, Won Woo Ro, and Murali Annavaram. 2016. Virtual thread: Maximizing thread-level parallelism beyond GPU scheduling limit. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE Press, Piscataway, NJ, 609–621. DOI: http://dx.doi.org/10.1109/ISCA.2016.59

Received July 2017; revised December 2017; accepted December 2017