HeteroCore GPU to Exploit TLP-Resource Diversity

Xia Zhao, Zhiying Wang, *Member, IEEE,* and Lieven Eeckhout, *Fellow, IEEE*

Abstract—Graphics processing units (GPUs) are widely adopted as compute accelerators in cloud computing environments and supercomputers. Sharing GPU resources in such environments requires effective multitasking support. Unfortunately, conventional GPUs lack the ability to adapt to diverse thread-level parallelism (TLP) resource demands among co-executing kernels. Previous work such as SM partitioning and simultaneously multitasking (SMK) increase system throughput, however, they degrade per-application performance significantly.

This paper proposes the HeteroCore GPU to significantly improve multitasking performance with a similar area cost as a conventional GPU. After rebalancing TLP-related SM resources, a HeteroCore GPU consists of two types of SMs to support diverse TLP-resource demands. Dynamic scheduling performs low-overhead spatial profiling during runtime across the different SM types and steers scheduling decisions based on the TLP-resource demands of the co-executing kernels. Compared to a conventional GPU, HeteroCore GPU improves system throughput by 20.1% on average (up to 80.9%) and per-application performance by 29.8% on average (up to 50.3%), for workload mixes composed of kernels with different TLP-resource demands.

Index Terms—Heterogeneous, graphics processing units (GPUs), thread level parallelism (TLP), scheduling

1 Introduction

Graphics processing units (GPUs) have become increasingly important components in modern computer systems because of their ability to accelerate highly data-parallel GPU-compute applications [33]. With each technology generation, GPUs have seen a dramatic increase in raw computational power, e.g., the latest Nvidia Pascal GPU delivers performance in the TFlops range [31]. The huge computational power at relatively low energy has spurred the integration of GPUs in supercomputers, cloud computing infrastructures as well as warehouse-scale computers, where GPUs are virtualized and shared by multiple users [11], [12], [37]. A key requirement to support GPU sharing is the ability to support multitasking or concurrent execution of independent kernels.

Time multiplexing is a multitasking technique widely used in CPUs which divides time into slices to time-share the CPU among co-executing applications. An application is preempted by an another application if it runs out of its current time slice. Unfortunately, unlike CPUs, the architecture state of a GPU kernel is large, and hence the overhead of saving and restoring it is high [34], [40], [49]. To make things worse, preemptive multitasking does not make effective use of the available hardware resources, e.g., memory bandwidth may be overutilized and underutilized at different times when executing a memory-intensive versus compute-intensive kernel in a time-sharing GPU environment.

In contrast, spatial multitasking divides GPU resources in space rather than time among co-executing kernels [3], [4], [35], [40]. By concurrently running multiple kernels on different streaming multiprocessors (SMs), spatial multitasking avoids the context switching overhead, and better utilizes the available hardware resources, thereby improving overall system performance. While the number of SMs in GPUs keeps increasing [31], [32] — making spatial multitasking increasingly promising — an inevitable problem in GPU multitasking is that co-executing kernels exhibit different thread-level parallelism (TLP) resource demands that are left unexploited in conventional GPUs.

A GPU kernel features so-called cooperative thread arrays (CTAs) that group threads, out of which warps are formed consisting of 32 threads for SIMD execution. When one warp is stalled, the GPU switches to an another warp to execute to hide memory access latency. The number of CTAs one SM can execute is limited by the available per-SM TLP-related hardware resources, including the register file, shared memory, and the warp slots. These **per-SM TLP resources** are identical for all SMs which leaves significant performance on the table and leads to suboptimal hardware utilization when executing diverse multitasking GPU-compute workloads.

In particular, for **thick-TLP kernels**, the available per-SM TLP resources are insufficient to hide memory access latency — these kernels can achieve higher performance if given more TLP resources. In contrast, for **lean-TLP kernels**, the available TLP resources exceed the kernel's TLP resource demands, and performance does not change or even increases when executed on SMs with less TLP resources. These kernels either saturate memory bandwidth and/or suffer from cache contention. Unfortunately, current GPUs lack the ability to dynamically adapt to and exploit different TLP resource demands among co-executing kernels. In contrary to what conventional GPUs provide, a thick-TLP kernel needs additional TLP resources whereas a lean-TLP kernel does not need as many TLP resources.

Previous solutions including SM partitioning [3] and simultaneous multi-kernel execution (SMK) [49], [51] increase hardware utilization in multitasking GPUs, however, they fall short for workload mixes with different per-SM TLP resource demands, which leads to severe per-application performance degradation. In particular, co-executing a thick-TLP kernel with another kernel on a single SM, as done in SMK, significantly degrades thick-TLP kernel performance. Per-application performance degradation

- *X. Zhao and L. Eeckhout are with the Department of Electronics and Information Systems, Ghent University, Belgium. E-mail: {xia.zhao,lieven.eeckhout}@UGent.be.*
- *Z. Wang is with the State Key Laboratory of High Performance Computing, National University of Defense Technology, China. E-mail: zywang@nudt.edu.cn*

Manuscript received XX, XXXX; revised XX XX, XXXX.

is particularly problematic in the context of fairness, quality-of-service (QoS) and service-level agreements (SLAs) [11], [12], [19].

This paper proposes the **HeteroCore GPU architecture** to improve both system throughput and per-application performance by exploiting different TLP resource demands among co-executing GPU kernels. The core idea of the HeteroCore GPU is to 'rebalance' per-SM TLP resources in an area-normalized way. The proposed HeteroCore GPU supports two types of SMs, the big-SM and small-SM. In particular, we reduce the size of the TLP resources in a **small-SM** and 'migrate' these TLP resources to a **big-SM**. Unlike the widely explored heterogeneous multicore CPU composed of core types with different performance characteristics (e.g., ARM's big.LITTLE), in HeteroCore GPU, the more complex big-SMs and the simpler small-SMs are both used to improve performance. By 'migrating' TLP resources from a small-SM to a big-SM, while keeping the number of functional units and cache size unchanged, we maintain (or even improve) lean-TLP kernel performance while significantly improving thick-TLP kernel performance.

To improve multitasking performance on a HeteroCore GPU, the intuition is to enhance thick-TLP kernel performance by executing on big-SMs while maintaining lean-TLP kernel performance by executing on small-SMs. Although the intuition is simple, designing an effective scheduling algorithm is not. Dynamically discerning thick-TLP kernels from lean-TLP kernels during runtime at low overhead is challenging. To this end, we propose spatial profiling and TLP resource-aware scheduling to optimize total system throughput *and* per-application performance: we profile the co-executing kernels on different SM types at low overhead; after the online profiling phase, our scheduling algorithm decides on the kernel-to-SM mapping based on the kernel's TLP resource characteristics. In addition, we deploy an adaptive preemption policy to minimize the impact of context switching.

Although the HeteroCore GPU architecture is motivated by GPU multitasking, it still maintains single-task performance: lean-TLP kernels perform similarly on the big-SMs and small-SMs — which is why they are lean-TLP kernels — while thick-TLP kernels do worse on the small-SMs but make up for it with better performance on the big-SMs.

In summary, we make the following contributions in this paper:

- We show that kernels exhibit different per-SM TLP-resource demands which previously proposed techniques to improve hardware utilization in multitasking GPUs such as SM partitioning and simultaneous multi-kernel execution, fail to exploit while balancing system throughput and per-application performance.
- We introduce the HeteroCore GPU architecture consisting of big-SMs and small-SMs varying in the degree of TLP resources to significantly improve multitasking performance while keeping hardware cost and single-task performance unchanged.
- We explore HeteroCore GPU scheduling policies to balance system throughput and per-application performance.
- We propose TLP resource-aware scheduling to fully exploit the potential of the HeteroCore GPU architecture by dynamically scheduling kernels to the most suitable SM type, based on a low-overhead spatial profiling phase to dynamically learn a kernel's TLP-resource characteristics.
- We demonstrate the potential of the HeteroCore GPU architecture and comprehensively evaluate its performance. Compared to a conventional GPU with similar hardware cost, the HeteroCore GPU improves system throughput by 20.1% on average (up to 80.9%) and per-application performance by 29.8% on average (up to 50.3%) for multitasking workloads composed of kernels with different TLP-resource characteristics.

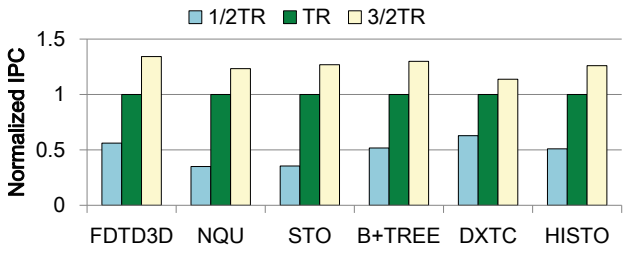


Fig. 1: Thick-TLP kernels: *Performance improves when given more per-SM TLP resources; performance significantly degrades when given less per-SM TLP resources.*

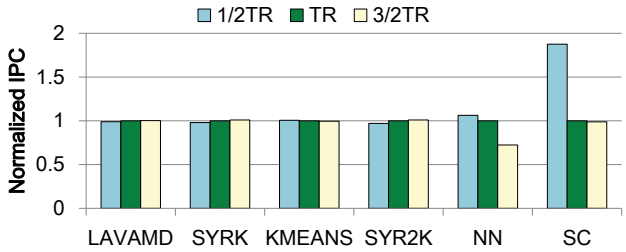


Fig. 2: Lean-TLP kernels: *Performance saturates or degrades with increasing per-SM TLP resources; performance saturates or significantly improves when given less per-SM TLP resources.*

2 Motivation

We first characterize the TLP resource demands in GPU-compute workloads and classify them into thick-TLP versus lean-TLP kernels. We do so based on the observed performance changes as we increase the TLP resources per SM. In particular, we vary the register file size, shared memory and number of warp slots relative to our baseline SM which we denote as 1 unit of TR (*TLP Resource*). We evaluate performance on a GPU with SMs of size $1/2 \cdot$TR, TR, and $3/2 \cdot$TR. ($1/2 \cdot$TR means that the per-SM TLP resources are half the size of our baseline SM.) All other resources, including the number of functional units, cache size and off-chip memory bandwidth, are kept unchanged. (See Section 5 for a detailed description of our experimental setup.)

2.1 Thick-TLP Kernels

Figure 1 shows normalized performance (instructions executed per cycle or IPC) as a function of the per-SM TLP resources for the thick-TLP kernels in our benchmark set. Increasing the TLP resources per SM, e.g., from TR to $3/2 \cdot$TR, enables more CTAs to execute on an SM which in turn leads to more potential for latency hiding and thus better performance, i.e., there is more opportunity to hide latency by scheduling warps when a particular warp is stalled. On the contrary, decreasing the number of TLP resources makes things worse as now there are fewer warps to hide latency.

Conclusion: Thick-TLP kernel performance improves when given more per-SM TLP resources, and significantly degrades when given fewer TLP resources.

2.2 Lean-TLP Kernels

Figure 2 shows similar results for the lean-TLP kernels in our benchmark set. Performance saturates or even degrades with increasing TLP resources per SM. It is interesting to investigate why these benchmarks do not benefit from executing on SMs with more TLP resources.

Performance is unaffected for the four benchmarks on the left. These benchmarks are memory-intensive applications that do not see their performance improve because high TLP induces a large number of memory accesses which saturate the memory system.

We observe a different trend for the two benchmarks on the right-hand side: performance degrades with increasing per-SM TLP resources. Both applications are cache-sensitive: increasing TLP resources enables more CTAs to concurrently execute per SM. This increases contention in the private (per-SM) caches, which as a result leads to a large number of memory accesses being sent to the (shared) L2 cache through the interconnection network, which degrades performance as the memory system stalls under the flood of memory requests.

Conclusion: Lean-TLP kernel performance does not degrade (and in some cases even improves) when given less per-SM TLP resources.

2.3 Opportunity

The observation that kernels exhibit different per-SM TLP resource demands creates an opportunity to improve performance in a GPU multitasking environment. In spatial multitasking on a conventional GPU, thick-TLP kernels and lean-TLP kernels execute on disjoint SMs which all provide the same TLP resources. However, for thick-TLP kernels, these TLP resources are insufficient for optimum performance. Meanwhile, the lean-TLP kernels do not need as many TLP resources, and can maintain (or even improve) performance when given less TLP resources. By 'rebalancing' the TLP resources from small-SMs to big-SMs, the HeteroCore GPU exploits TLP-resource diversity among co-executing kernels while keeping the hardware cost unchanged. By scheduling thick-TLP kernels on big-SMs and lean-TLP kernel on small-SMs, we may improve overall system performance. This key insight motivates the proposal for the HeteroCore GPU architecture.

2.4 Why Existing Solutions Fail

Before describing the HeteroCore GPU architecture in more detail, we first quantify and argue why existing solutions, including SM partitioning and simultaneous multi-kernel execution (SMK), are inadequate to exploit TLP-resource diversity. SM partitioning [3] partitions the available SMs among the co-executing kernels. Extending SM partitioning to be TLP resource-aware can be done by assigning more SMs to the thick-TLP kernel and fewer SMs to the lean-TLP kernel. Simultaneous multi-kernel execution (SMK) is a GPU multitasking approach in which two kernels co-execute on a single SM [49], [51]. Making SMK TLP resource-aware can be done by granting more TLP resources to the thick-TLP kernel while granting fewer to the lean-TLP kernel.

We first qualitatively compare the different ways for exploiting TLP diversity. Figure 3 illustrates even and uneven SM partitioning, SMK and HeteroCore. The thick-TLP kernel and lean-TLP kernel oversubscribe and undersubscribe the per-SM TLP resources, respectively. Even and uneven partitioning, see Figures 3(a) and (b), do not fundamentally address the imbalance problem. SMK makes things even worse: the thick-TLP can use even less TLP resources compared to running in isolation, see Figure 3(c). HeteroCore on the other hand rebalances the per-SM TLP resources so that the available per-SM TLP resources better match the characteristics of the co-executing kernels, see Figure 3(d).

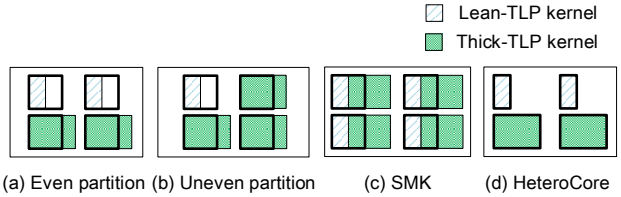


Fig. 3: Possible ways to exploit TLP diversity in a GPU with 4 SMs. *HeteroCore better exploits TLP-resource diversity.*

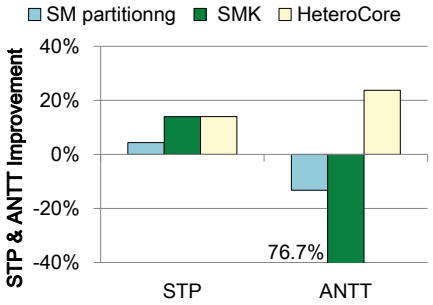


Fig. 4: STP and ANTT improvement across all workloads relative to even SM partitioning: *Uneven SM partitioning improves system throughput (STP) slightly but degrades per-application performance (ANTT). SMK improves STP substantially but severely degrades ANTT. HeteroCore on the other hand significantly improves both STP and ANTT.*

We now compare the four strategies quantitatively. We consider optimum results for SM partitioning and SMK, while reporting effective numbers for HeteroCore. The optimum results are obtained through off-line analysis. In particular, for SM partitioning, we pick the optimum SM partitioning through offline analysis by changing the number of SMs assigned to either kernel in the workload mix in increments of two. For SMK, we identify the optimum partitioning of an SM by exploring all possible combinations of co-executing two kernels on an SM. (To avoid the unfairness caused by GTO under SMK, we use a loose round-robin warp scheduler to guarantee fairness while achieving high STP by first selecting kernels in a round-robin way and then selecting warps within a kernel to issue instructions following the GTO policy [36].)

Figure 4 quantifies overall system throughput (STP) and per-application performance (ANTT) for all four strategies relative to even partitioning. Uneven SM partitioning slightly improves overall system performance. Assigning more SMs to the thick-TLP kernel improves its performance, however, taking away SMs from the lean-TLP kernel degrades its performance. This leads to a net albeit modest improvement in overall system performance compared to even partitioning. Per-application performance degrades because lean-TLP kernel performance suffers.

SMK significantly improves system throughput (by 14% on average), however it severely degrades per-application performance (by 76.7% on average)[1]. The primary reason for the severe degradation in ANTT is that the thick-TLP kernel suffers substantially from not being able to allocate all the TLP resources per SM. Recall that a thick-TLP kernel, by definition, is very sensitive to

1. Optimum SMK here maximizes STP. If optimum SMK were to minimize ANTT, the ANTT degradation is still as high as 62.4% while decreasing the STP improvement to 4.8%.

the assigned TLP resources per SM; hence decreasing the assigned TLP resources leads to a severe per-SM performance drop. In addition, co-executing two kernels on the same SM unavoidably leads to *intra-SM contention* in various resources including the L1 cache and/or the load/store units [13]. Intra-SM contention may slowdown one kernel or in some cases both kernels. The drop in per-SM performance is not compensated for by executing on twice the number of SMs compared to even partitioning. This explains the significant drop in per-application performance under SMK.

Although SMK and uneven SM partitioning both lead to (severe) ANTT degradation compared to even partitioning, this does not necessarily imply that a combination of both also leads to ANTT degradation. One option to combine SMK with uneven SM partitioning may be to apply SMK for some multi-kernel workloads and uneven partitioning for others. This approach could potentially achieve the best of both worlds as some multi-kernel workloads prefer SMK while others prefer uneven SM partitioning. Our evaluation shows that for some workloads however, both SMK and uneven SM partitioning lead to an ANTT performance degradation. Another option may be to assign a subset of the SMs to one kernel exclusively and share the remaining SMs among the various kernels through SMK. This will lead to inferior performance compared to either approach, i.e., if SMK yields better performance on a subset of the SMs, it will be beneficial to apply SMK to all SMs, and if it yields worse performance, then it will be beneficial to apply uneven SM partitioning and not SMK.

In contrast to SMK and uneven SM partitioning, HeteroCore significantly improves both system throughput *and* per-application performance. The fundamental reason is that thick-TLP kernels significantly benefit from being given more per-SM resources when running on big-SMs; the lean-TLP kernels do not see their performance degrade when running on small-SMs.

Conclusion: Existing solutions (SM partitioning and SMK), even when made TLP resource-aware *and* under optimum offline analysis, are ineffective at exploiting TLP-resource diversity. The HeteroCore architecture on the other hand significantly improves thick-TLP kernel performance while not degrading (and in some cases even improving) lean-TLP kernel performance. This leads to a significant improvement in both overall system throughput *and* per-application performance.

3 HeteroCore GPU

In this section, we first discuss the hardware support provided in an SM to exploit TLP. We then propose our HeteroCore GPU architecture to exploit varying TLP resource demands among co-running kernels. We finally describe multitasking support.

3.1 TLP-Related Hardware Structures

As shown in Figure 5, GPUs typically consist of a number of SMs that are connected to the last-level cache and memory controllers through the interconnection network. When launching a kernel to the GPU, the CTAs within the kernel are assigned to the SMs by the CTA scheduler in a (typically) round-robin fashion. The number of CTAs that an SM can execute concurrently is determined by various hardware resource constraints such as the register file, shared memory, warp and CTA slots [23]. No CTAs will be dispatched to an SM if one of these resources is insufficient to support a new CTA. We now describe these TLP-related structures in more detail, as previously detailed in the literature [5], [29], [44], [52].

Register file: The maximum number of concurrent threads per SM is a function of register file capacity on the one hand, and the number of registers allocated per thread on the other hand. To reduce hardware cost, instead of employing a multiported register file, GPUs typically feature a multibanked register file in which a crossbar network and operand collectors are used to transport operands from the banks to the execution units [5], [29]. Within an SM, 32 threads from a given CTA are grouped into a warp which is the basic unit to schedule and issue. The execution context of all warps executing on the SM is stored in the register file.

Shared memory: The shared memory is on-chip scratchpad memory that is allocated per CTA and is visible to all threads within the same CTA. The amount of shared memory required per CTA is specified by the programmer. The shared memory not only provides a mechanism for inter-thread communication within a CTA, but also serves as a software-managed cache due to its small access latency and high bandwidth. Similar to the register file, shared memory is typically multibanked and connected through a crossbar network [5], [29].

Warp slots: The number of warp slots is the third resource that may limit the maximum number of CTAs per SM [44], [52]. As mentioned before, a warp is the basic unit to schedule and issue instructions on a GPU. As shown in Figure 5, the number of warp slots is related to various components in different pipeline stages, such as the program counter array (PC array), instruction buffer (I-Buffer), SIMT stack and scoreboard.

CTA slots: The CTA slots administer the CTAs currently running on an SM. Hence, the number of CTA slots puts a cap on the maximum number of CTAs one SM can execute, e.g., Fermi GPUs have 8 CTA slots per SM [2].

TABLE 1: SM configurations for the baseline conventional GPU versus the big-SM and small-SM configurations in the HeteroCore GPU architecture.

TLP resource	Baseline	Big-SM	Small-SM
Register File	32768 (registers)	49152	16384
Shared Memory	48 KB	64 KB	32 KB
Threads Slots	1536 (threads)	2304	768
Warp Slots	48 (warps)	72	24
CTA Slots	8 (CTAs)	10	6

3.2 HeteroCore SM Architecture

As mentioned before, the basic idea of the HeteroCore GPU architecture is to 'rebalance' the per-SM resources in an area-normalized way to exploit different TLP resource demands in concurrently executing kernels. The HeteroCore GPU consists of two types of SMs: the big-SM exploits thick-TLP kernels whereas the small-SM is optimized for lean-TLP kernels. The number of threads or warps an SM can support depends on the available TLP resources such as the register file, shared memory, warp slots and CTA slots. In our baseline GPU, there are 16 SMs, all configured the same way. To achieve an area-normalized rebalancing of per-SM TLP resources, the HeteroCore GPU 'migrates' TLP resources from half the SMs to the other half, to effectively construct a HeteroCore GPU with 8 big-SMs and 8 small-SMs. Compared to an SM in the baseline GPU, the big-SM consumes more chip area as it features larger TLP resources. On the other hand, the reduced small-SM consumes less area. As we will later show in the evaluation section, the total chip area of the HeteroCore GPU is indeed nearly the same as the baseline conventional GPU. It is worth noting that by reducing the number of TLP resources in the small-SM while keeping cache size constant, each thread benefits from a larger effective cache space.

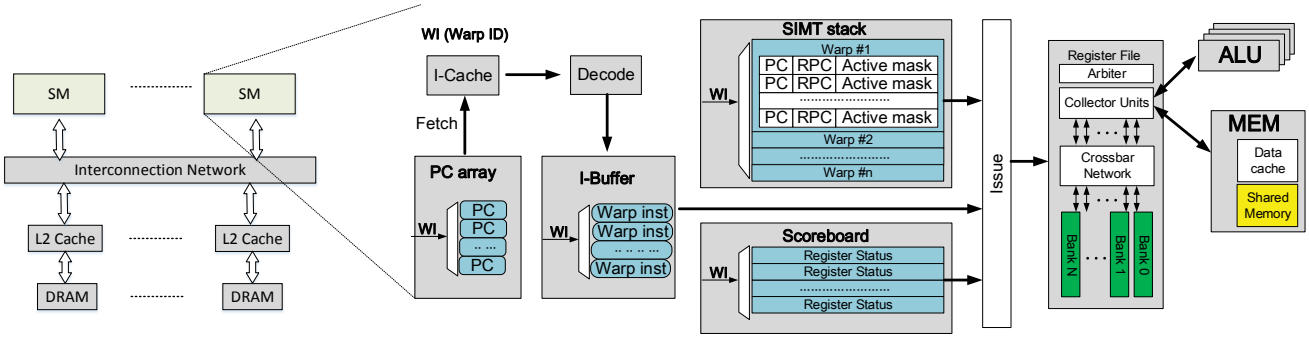


Fig. 5: Overview of a conventional GPU architecture. *The per-SM TLP resources are highlighted.*

The detailed SM configurations for the baseline GPU and HeteroCore GPU are listed in Table 1. The HeteroCore GPU SMs are configured by modifying the size of the TLP-related structures only; we do not change the structures themselves. For example, to increase the size of the register file, we only increase the size of each bank instead of increasing the number of banks. By doing so, the other components of the register file such as the crossbar network remains unchanged.

Cycle time. One concern may be that because the big-SM has larger sized structures compared to the baseline SM, cycle time and therefore clock frequency may be affected, and as a result, the HeteroCore GPU may not be clocked as fast as a conventional GPU. Previous work has shown that the warp scheduler of the issue unit is on the critical path as it needs to access the scoreboard to identify the ready warps among all active warps and then choose the ready warp with the highest priority to issue [6], [52]. We hence focus the discussion here on the warp scheduler. Although GPUs typically employ multiple warp schedulers per subset of warps, increasing the number of warp slots in the big-SM leads each warp scheduler to consider more warps, which may affect cycle time. To solve this problem, we divide the warp slots per scheduler into two groups: the first group contains as many warps slots as the baseline SM; the other group consists of the remaining warp slots. Initially, the warp scheduler only considers the first group. If the warp scheduler cannot find a ready warp, it will consider the other group in the next cycle. The warp scheduler continues considering one group until it cannot find ready warps. Although switching between groups incurs a lost cycle, this does not impact big-SM performance much because (i) switching typically happens when the system suffers from severe resource contention, e.g., memory contention, in which case a one-cycle bubble is small compared to the long memory access latency [16]; and (ii) this problem only occurs when both groups are used, i.e., because of limitations in the other TLP-related resources, many thick-TLP applications may only use the warp slots in the first group. We take this one-cycle switching overhead into account in our evaluation.

3.3 HeteroCore GPU Multitasking Support

To efficiently utilize the available hardware resources, architectural extensions such as spatial multitasking have been proposed for sharing the GPU among kernels from different processes [3], [40]. Spatial multitasking divides the SMs in a GPU into several disjoint subsets and allows concurrently executing kernels to run on different subsets of SMs. When a new kernel is launched, it can preempt some SMs to execute and this avoids the starvation of the newly arrived kernel.

To fully support multitasking in a HeteroCore GPU, a good preemption policy is critical. Previously proposed GPU preemption policies include SM draining and context switching [35], [40]. The SM draining policy exploits the GPU execution model that different CTAs are independent from each other. After finishing a CTA, no information of the finished CTA needs to be stored. In the SM draining policy, if an SM is preempted, no more CTAs are allowed to be issued on the SM. After finishing all the executing CTAs, the SM becomes idle after which it can execute CTAs from another kernel. Unlike the SM draining policy, the context switching policy saves the contexts for all threads currently running on the SM. GPUs support up to a few thousand threads per SM, which incurs significant overhead for saving and restoring architecture state which can be as large as 256 KB for the register file and 48 KB for shared memory per SM. The SM is halted and can no longer execute instructions during preemption.

Simply employing the draining policy or the context switching policy in the HeteroCore GPU without considering a kernel's execution characteristics is not the best choice. In particular, if the CTAs of the currently executing kernel are likely to finish soon, the draining policy seems a better fit as it avoids the overhead of saving and restoring architecture state which may incur significant memory contention and increase the preemption latency. On the other hand, if the CTAs currently running on the SM need a very long time to finish their execution, the draining policy would increase the preemption latency. The high preemption latency may also significantly decrease overall system performance as the SMs cannot utilize the available hardware while draining the SM. In this case, context switching is the better option. As these two preemption policies are suitable to different kernels and execution contexts, the proposed HeteroCore GPU exploits an adaptive preemption policy that chooses either the draining policy or the context switching policy based on the kernel's execution characteristics, as we describe in the following section.

4 HeteroCore GPU Scheduling

Scheduling kernels onto the different SM types is critical to HeteroCore GPU performance. In this section we describe different scheduling algorithms which we then evaluate in Section 6. We consider two kernels when describing these algorithms, however, this does not affect the generality — the algorithms are easily extended to more than two co-executing kernels. We further consider a baseline conventional GPU with 16 SMs versus a HeteroCore GPU with 8 big-SMs and 8 small-SMs.

4.1 TLP Resource-Agnostic Scheduling

A naive TLP resource-agnostic scheduling algorithm simply divides the 8 big-SMs and 8 small-SMs into two groups and assigns each of the co-executing kernels 4 big-SMs and 4 small-SMs. By not

Algorithm 1
Static TLP-aware STP-optimized scheduling

1: $IPC^{base}_{K_i} \leftarrow$ IPC for kernel K_i on 16 baseline SMs
2: $IPC^{B}_{K_i} \leftarrow$ IPC for kernel K_i on 8 big-SMs
3: $IPC^{S}_{K_i} \leftarrow$ IPC for kernel K_i on 8 small-SMs
4: $STP_{K_0(B)\_K_1(S)} = \frac{IPC^{B}_{K_0}}{IPC^{base}_{K_0}} + \frac{IPC^{S}_{K_1}}{IPC^{base}_{K_1}} \leftarrow$ estimate STP for K_0 on big-SMs and K_1 on small-SMs
5: $STP_{K_1(B)\_K_0(S)} = \frac{IPC^{B}_{K_1}}{IPC^{base}_{K_1}} + \frac{IPC^{S}_{K_0}}{IPC^{base}_{K_0}} \leftarrow$ estimate STP for K_1 on big-SMs and K_0 on small-SMs
6: **if** $STP_{K_0(H)\_K_1(L)} > STP_{K_1(H)\_K_0(L)}$ **then**
7: map K_0 on 8 big-SMs and K_1 on 8 small-SMs
8: **else**
9: map K_1 on 8 big-SMs and K_0 on 8 small-SMs
10: **end if**

considering the TLP resource characteristics of the co-executing kernels, TLP resource-agnostic scheduling is the simplest yet naive approach for scheduling kernels on the HeteroCore GPU. Although TLP resource-agnostic scheduling avoids the overheads incurred by the offline and online algorithms described in the following sections, it is unable to fully exploit the performance potential of the HeteroCore GPU. In particular, when a lean-TLP kernel and a thick-TLP kernel co-run, TLP resource-agnostic scheduling assigns 4 big-SMs and 4 small-SMs to the lean-TLP kernel, which unfortunately does not improve its performance; what is even worse, the thick-TLP kernel suffers from a performance degradation by executing (in part) on the small-SMs.

4.2 Static TLP Resource-Aware Scheduling

TLP resource-aware scheduling tackles this shortcoming. Although our final goal is a dynamic scheduling algorithm, we first describe static scheduling as it will serve as a point of comparison for our dynamic scheduler. We consider three static TLP-aware scheduling policies: (i) classification, (ii) STP-optimized, and (iii) STP/ANTT-balanced scheduling.

4.2.1 Classification scheduling

Classification scheduling first classifies kernels in either the thick-TLP versus lean-TLP category. We first run each kernel in isolation on a GPU with 8 baseline SMs, 8 big-SMs and 8 small-SMs. We then compare the performance results and classify kernels accordingly. A kernel for which performance does not degrade relative to the baseline, when running on the small-SMs, is classified as a lean-TLP kernel. All other kernels are classified as a thick-TLP kernel.

Classification scheduling maps the thick-TLP kernel to big-SMs and the lean-TLP kernel to small-SMs. This way, the thick-TLP kernel benefits a significant performance improvement by exploiting more TLP on the big-SMs. Meanwhile, the lean-TLP kernel does not suffer and in some cases, performance even improves by executing on the small-SMs. However, when two applications with the same TLP resource characteristics co-execute, classification scheduling reverts to TLP resource-agnostic scheduling and assigns 4 big-SMs and 4 small-SMs to each kernel.

4.2.2 STP-optimized scheduling

STP-optimized scheduling aims at optimizing system throughput. This policy is motivated by the observation that classification scheduling as just described does not fully exploit the potential of the HeteroCore GPU in case kernels from the same category need to be co-scheduled. For example, two co-running thick-TLP kernels only get half the big-SMs assigned which leads to suboptimal performance. Higher overall system throughput can be achieved by scheduling the kernel that benefits the most from running on the big-SM, on the big-SMs.

STP-optimized scheduling, during the offline profiling phase, runs each of the two kernels on the conventional GPU with 16 baseline SMs and on the HeteroCore GPU with just the 8 big-SMs and just the 8 small-SMs. The respective performance numbers are then used to determine the schedule that optimizes overall system throughput, see Algorithm 1. Performance scheduling computes overall system throughput for both scheduling options, i.e., kernel K_0 on the big-SMs and kernel K_1 on the small-SMs, and vice versa, and then picks the schedule that maximizes performance. Note that system performance is computed following the notion of the system throughput (STP) metric [15] (which is equivalent to weighted speedup), as we detail in Section 5.

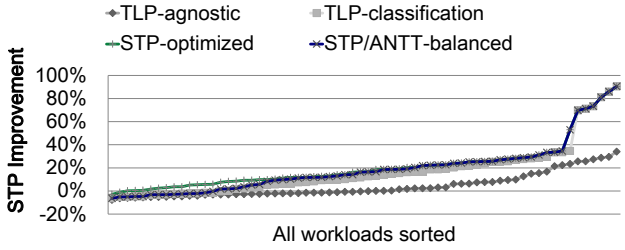
4.2.3 STP/ANTT-balanced scheduling

STP/ANTT-balanced scheduling aims at balancing system throughput and per-application performance. STP-optimized scheduling severely degrades per-application performance for some workload mixes. In particular, if two thick-TLP kernels co-execute concurrently, STP-optimized scheduling assigns all big-SMs to the thick-TLP kernel that benefits the most. This, however, penalizes the other thick-TLP kernel, which gets executed on small-SMs and hence experiences a severe performance degradation. To achieve better per-application performance, it is better to assign both kernels half the big-SMs and half the small-SMs, as done in classification scheduling.

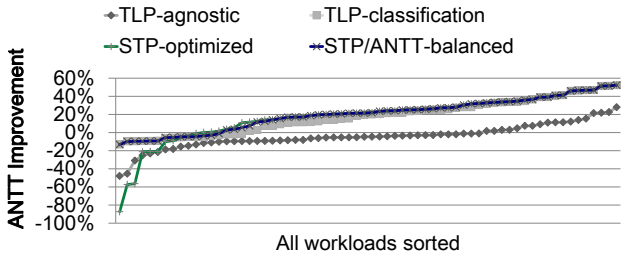
STP/ANTT-balanced scheduling achieves the best of both classification and STP-optimized scheduling. If both kernels benefit from executing on the big-SMs, STP-ANTT balanced scheduling assigns 4 big-SMs and 4 small-SMs to each kernel — just like classification scheduling. Otherwise, it assigns all 8 big-SMs to the kernel that benefits the most — just like STP-optimized scheduling. It is worth noting that for workload mixes consisting of two lean-TLP kernels, STP/ANTT-balanced scheduling works well as most lean-TLP kernels do not see their performance degrade when executing on big-SMs; a lean-TLP kernel that gets executed on a small-SM might see a performance benefit. This improves both STP and ANTT.

4.2.4 Performance evaluation

We now evaluate the different static TLP-aware scheduling policies. We report STP and ANTT relative to a conventional GPU for TLP-agnostic scheduling, versus TLP-aware classification, STP-optimized and STP/ANTT-balanced scheduling. We sort the workloads along the horizontal axis, see Figure 6. (See Section 5 for details regarding the experimental setup.) There are two key take-away messages: (i) the HeteroCore GPU clearly outperforms the conventional GPU, and (ii) the scheduling policy plays a critical role in improving STP and ANTT. STP-optimized scheduling clearly outperforms the other three scheduling policies in terms of STP. However, its impact on ANTT is also obvious: for some workload mixes that consist of two thick-TLP kernels, ANTT may degrade by up to 87.4%. Classification scheduling does not lead to a significant drop in ANTT, however, STP is not nearly as good as for STP-optimized scheduling. STP/ANTT-balanced scheduling hits the middleground by balancing STP and ANTT, i.e., STP is comparable to STP-optimized scheduling, yet ANTT does not degrade as much for thick-TLP kernel workload mixes. Overall, across all multikernel workloads, STP/ANTT-balanced scheduling



(a) STP for all multikernel workloads



(b) ANTT for all multikernel workloads

Fig. 6: Offline analysis: *STP and ANTT improvement for various HeteroCore scheduling policies over a conventional GPU.*

Algorithm 2
Dynamic TLP-aware scheduling

if new kernel gets launched **then**
  initiate profiling phase
end if
if execution phase **then**
  $IPC^{B}_{K_i} \leftarrow$ measured IPC on big-SMs
  $IPC^{S}_{K_i} \leftarrow$ measured IPC on small-SMs
  $IPC^{base}_{K_i} \leftarrow$ estimated IPC on baseline SMs
  if $IPC^{B}_{K_0} > IPC^{S}_{K_0}$ and $IPC^{B}_{K_1} > IPC^{S}_{K_1}$ **then**
    map K_0 and K_1 to 4 big-SMs and 4 small-SMs
  else
    compute $STP_{K_0(B)\_K_1(S)} = \frac{IPC^{B}_{K_0}}{IPC^{base}_{K_0}} + \frac{IPC^{S}_{K_1}}{IPC^{base}_{K_1}}$
    compute $STP_{K_1(B)\_K_0(S)} = \frac{IPC^{B}_{K_1}}{IPC^{base}_{K_1}} + \frac{IPC^{S}_{K_0}}{IPC^{base}_{K_0}}$
    if $STP_{K_0(B)\_K_1(S)} > STP_{K_1(B)\_K_0(S)}$ **then**
      map K_0 on 8 big-SMs and K_1 on 8 small-SMs
    else
      map K_1 on 8 small-SMs and K_0 on 8 big-SMs
    end if
  end if
end if

improves STP by 13.9% on average (up to 90.4%) and improves ANTT by 23.8% on average (up to 52.3%).

4.3 Dynamic TLP Resource-Aware Scheduling

Static TLP-aware scheduling relies on offline profiling which is impractical. To this end, we propose dynamic TLP resource-aware scheduling which is inspired by static TLP-aware STP/ANTT-balanced scheduling, yet performs profiling during runtime at low overhead.

When launching two kernels to co-execute on the HeteroCore GPU (i.e., when two kernels start their execution at the same time, or when a new kernel comes in while another kernel was already running), we first initiate a *spatial profiling* phase in which we partition the HeteroCore GPU into two groups with 4 big-SMs and 4 small-SMs each. Each kernel gets to run on a partition with 4 big-SMs and 4 small-SMs, during which we measure big-SM and small-SM performance. After this profiling phase, we determine the types of the two co-executing kernels. If both kernels favor big-SMs, i.e., they are both thick-TLP kernels, we assign 4 big-SMs and 4 small-SMs to each kernel. Otherwise, we determine which kernel benefits most from running on the big-SM. We then assign all the big-SMs to the application that benefits the most from big-SM execution towards overall system throughput; the other application gets to run on the small-SMs. We re-initiate spatial profiling whenever a kernel finishes its execution and a new kernel is launched.

Algorithm 2 describes the dynamic scheduling algorithm in more detail. Spatial profiling takes 40K cycles in our setup, of which we consider the first 20K cycles for warmup, and we then measure performance during the next 20K cycles. To estimate system throughput for the different scheduling alternatives, we need an arbitrarily chosen baseline to normalize to. Static TLP-aware scheduling considers a conventional GPU as its baseline. Unfortunately, we cannot measure baseline SM performance during online profiling. Hence we have to estimate it. This is done by re-scaling the big-SM and small-SM performance numbers relative by the number of CTAs running on either SM type, as shown in the below formulas, with $\#CTA^{Baseline}_{K_i}$, $\#CTA^{big}_{K_i}$ and $\#CTA^{small}_{K_i}$ the number of CTAs per SM in the conventional GPU, big-SM and small-SM, respectively. These numbers are easy to compute as the resource requirements per CTA are known as well as the amount of resources per SM.

$$\text{IPC}^{base}_{K_i} = IPC_{scaled} \times (IPC^{big}_{K_i} - IPC^{small}_{K_i}) + IPC^{small}_{K_i}$$

$$\text{IPC}_{scaled} = \frac{\#CTA^{base}_{K_i} - \#CTA^{small}_{K_i}}{\#CTA^{big}_{K_i} - \#CTA^{small}_{K_i}}$$

As mentioned before, to reduce preemption latency, the HeteroCore GPU exploits an adaptive preemption policy that chooses between the draining versus context switching policy based on the kernel's execution characteristics. In particular, the adaptive preemption policy considers the kernel's execution behavior during warmup. If a kernel can finish a CTA's execution during the warmup phase, it is likely to assume that other CTAs will also finish soon, hence the adaptive preemption policy employs the draining policy to preempt the SMs occupied by this kernel. If not, the adaptive policy employs context switching.

Note that dynamic TLP resource-aware scheduling relies on an initial profiling phase whenever a new kernel comes in. This works because GPU kernels are made up of many CTAs that exhibit relatively consistent behavior, so we can use some initial CTAs to help us guide scheduling for future CTAs [27]. Although fine-grained phase behavior may be observed at the warp level [21], [44], this gets leveled out at the SM level as several CTAs execute concurrently on an SM.

Finally, new CUDA features such as dynamic parallelism in which parent kernels can launch child kernels to run alongside the parent kernels, may affect kernel launch. Although our current evaluation infrastructure does not support dynamic parallelism, we believe that dynamic parallelism can be supported by regarding a child kernel as a new incoming kernel. In that case, when a child kernel is launched, we re-initiate the spatial profiling phase. One potential limitation may occur for workloads with many small child kernels, which may lead to considerable profiling overhead. In such a case, we may need to employ an adaptive approach in which we dynamically determine whether or not to initiate spatial profiling — such a decision can be made based on the number of CTAs per kernel, i.e., a kernel with a large number of CTAs is likely to run longer than a kernel with a small number of CTAs, hence it may be worth paying the profiling overhead. The evaluation of such a mechanism falls beyond the scope of this paper and is left for future work.

TABLE 2: Baseline GPU architecture.

Parameter	Value
Streaming Multiprocessors (SM)	16 SMs, 700 MHz
Warp Size	32
Schedulers/Core	2 (GTO)
Number of Threads/Core	1536
Registers/Core	32768
Shared Memory/Core	48 KB
Constant/Core	8 KB
L1 Data Cache/Core	16 KB, 4-way, LRU, 128 B line
Memory Controllers	6
L2 Cache/MC	128 KB, 8-way, LRU, 128 B line
Interconnection Network	Crossbar, 32 B channel width
DRAM Model and Bandwidth	FR-FCFS, 16 banks/MC, 177.4 GB/s
GDDR5 Timing	t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=6, t_{CCD}=2, t_{WR}=12

TABLE 3: Benchmarks considered in this paper.

Benchmark	Abbr.	CTAs / SM	CTAs / Big-SM	CTAs / Small-SM	TLP Demand
LavaMD [8]	LAVAMD	6	9	3	Lean-TLP
Symmetric Rank-k Operations [17]	SYRK	6	9	3	Lean-TLP
K-means [8]	KMEANS	6	9	1	Lean-TLP
Symmetric Rank-2k Operations [17]	SYR2K	6	9	1	Lean-TLP
Neural Network [5]	NN	8	10	4	Lean-TLP
Streamcluster [8]	SC	3	4	1	Lean-TLP
3D Finite-Difference Time-Domain [1]	FDTD3D	2	3	1	Thick-TLP
N-Queens Solver [5]	NQU	3	4	2	Thick-TLP
StoreGPU [5]	STO	3	4	2	Thick-TLP
B+TREE Search [8]	B+TREE	5	9	3	Thick-TLP
DirectX Texture Compressor [1]	DXTC	8	10	6	Thick-TLP
Histogram [39]	HISTO	8	10	6	Thick-TLP

5 Experimental Setup

Simulated System: We use a modified version of GPGPU-sim v3.2.2 [5] to evaluate the proposed HeteroCore GPU architecture. The modifications allow GPGPU-sim to run multiple applications concurrently through spatial multitasking. Table 2 lists the configuration for our baseline GPU architecture. The HeteroCore GPU architecture consists of 8 big-SMs and 8 small-SMs. Apart from the SM-type specific configuration parameters listed in Table 1, the HeteroCore GPU parameters are the same as for the baseline configuration. To estimate power consumption and chip area, we use GPUWattch [29].

Context Switching: Upon a context switch, the context of the currently executing kernel is switched out and stored in off-chip memory. To incorporate the overhead of context switching in our measurements, not only because of stalling the preempted SM but also because of increased network and memory contention, we implement context switching in our simulator by simulating the packets transferring the context through the interconnection network as well as writing to main memory. When an SM is preempted, it is stalled until it finishes sending all packets to main memory. The number of packets is calculated based on the size of the context.

Workloads: We consider a wide range of CUDA GPU-compute benchmarks from a range of application domains including data mining, search, deep learning, engineering, compression, etc., see Table 3 These benchmarks are selected from Rodinia [8], Parboil [39], CUDA SDK [1], PolyBench [17] and GPGPU-sim [5]. KMEANS, B+TREE and NN constitute of two kernels; the other benchmarks consist of a single kernel. We classify these benchmarks into two classes following the procedure described in Section 2.[2] For generating multikernel workloads, we pair all the thick-TLP and lean-TLP applications (see Table 3 for the classification) to obtain 66 multikernel workloads. Based on the TLP resource demands of the constituting benchmarks, the 66 workloads can be classified into two categories, named heterogeneous workloads and homogeneous workloads. The multi-kernel workloads in the heterogeneous workload category consist of two kernels with different TLP resource demands, i.e., one thick-TLP kernel and one lean-TLP kernel. The workloads in the homogeneous workload category consist of two kernels with similar TLP resource demands.

Performance and Power Metrics: We simulate for 10 million cycles. If a benchmark finishes before others, it is re-launched and re-executed from the beginning.[3] The reported performance results pertain the whole execution. System throughput (STP) and average normalized turnaround time (ANTT) [15] are used to evaluate multikernel performance. STP takes a system's perspective and quantifies total system throughput — STP is also referred to as weighted speedup. ANTT takes a user's perspective and quantifies average per-application performance. Energy per instruction (EPI) is used to measure energy efficiency.

6 Evaluation

We now evaluate the HeteroCore GPU architecture. We first quantify the improvement in STP and ANTT. We then evaluate the impact on hardware cost and energy efficiency, and we evaluate single-kernel and four-kernel performance. Finally, we perform sensitivity analyses.

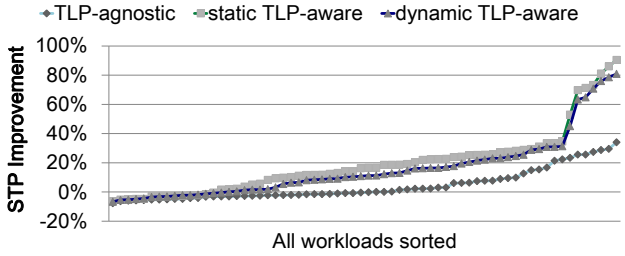
6.1 STP and ANTT

We first evaluate how HeteroCore affects system throughput (STP) and per-application performance (ANTT) compared to our baseline GPU. These results assume that we profile and schedule kernels during runtime. In other words, we account for the overhead of spatial profiling and context switching. Obviously, TLP resource-agnostic scheduling does not incur any overhead as it does not require a profiling phase, in contrast to dynamic TLP resource-aware scheduling. We also compare against static TLP resource-aware performance scheduling to quantify the impact of profiling and context switching overhead.

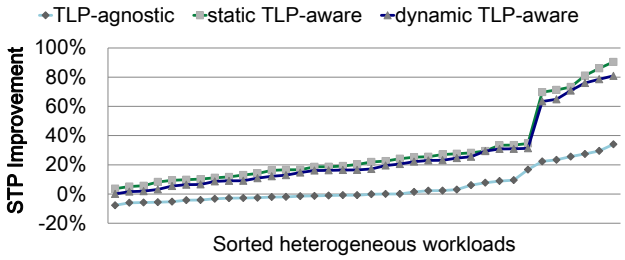
Figure 7 quantifies STP improvement for the HeteroCore GPU over the conventional GPU under TLP resource-agnostic, static and dynamic TLP resource-aware scheduling. Clearly, the HeteroCore GPU with dynamic TLP resource-aware scheduling outperforms the conventional GPU. The runtime overhead of spatial profiling and context switching is minor, i.e., dynamic scheduling is nearly as effective as static scheduling. Overall, the HeteroCore GPU

2. Note that when a benchmark is classified as a lean-TLP application, that does not mean that the benchmark performs poorly on a GPU. In contrast, the lean-TLP benchmarks achieve very high performance (average IPC of 107 and up to 467). These benchmarks are classified as lean-TLP because performance does not improve when given more TLP-related resources per SM over the baseline, see Section 2.

3. We verified that 10 million cycles is representative for all workloads, i.e., performance characteristics do not change afterwards, which is in line with current practice [49], [50], [51]. For some workloads, we need to re-launch (and thus re-profile) up to 3 times.

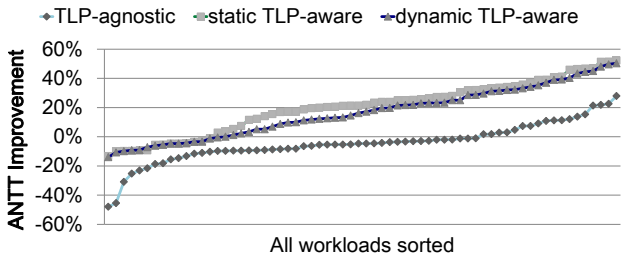


(a) All multikernel workloads

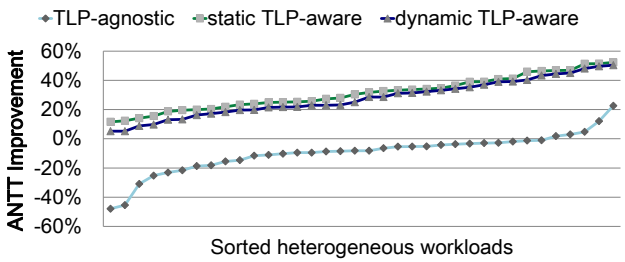


(b) Heterogeneous multikernel workloads

Fig. 7: STP improvement over a conventional GPU under dynamic scheduling. *Dynamic TLP-aware scheduling improves STP by 20.1% on average and up to 80.9% for the heterogeneous workload mixes.*



(a) All multikernel workloads



(b) Heterogeneous multikernel workloads

Fig. 8: ANTT improvement over a conventional GPU under dynamic scheduling. *Dynamic TLP-aware scheduling improves ANTT by 29.8% on average and up to 50.9% for the heterogeneous workload mixes.*

improves system performance by 20.1% on average, and up to 80.9%, for workloads composed out of kernels with different TLP-resource characteristics. Across all workloads considered in this work, HeteroCore GPU improves performance by 11% on average. For only a few workloads does the HeteroCore GPU lead to a performance degradation of at most 6.7%. This happens when two memory-intensive kernels experience similar performance benefits from executing on the big-SMs. In such a case, assigning all 8 big-SMs to one kernel is not the best choice as this kernel may clog memory resources slowing down the other kernel, which in the end degrades overall system performance; assigning 4 big-SMs and 4 small-SMs to each of the co-executing kernels leads to higher performance.

Figure 8 provides similar curves for ANTT. HeteroCore with dynamic TLP resource-aware scheduling leads to significant improvements in ANTT for all heterogeneous workloads, by 29.8% on average and up to 50.3%, see Figure 8(b). The fundamental reason is that thick-TLP kernel performance improves substantially by running on big-SMs; at the same time, lean-TLP kernel performance does not degrade and in some cases even improves. Across all workloads, see Figure 8(a), we observe substantial improvements in per-application performance (by 20.2% on average). For some workload mixes that consist of two thick-TLP kernels, ANTT decreases by at most 14%. For these thick-TLP kernels which both favor big-SMs, the performance benefits on big-SMs is less than the performance degradation on small-SMs.

We also find dynamic scheduling to be within 3% and 3.6% of static scheduling for STP and ANTT, respectively. For some workloads, the performance gap between static and dynamic scheduling is somewhat higher, and the worst is 17.3%. We find this to be the case for a multikernel workload that consists of two lean-TLP kernels that both favor small-SMs. The initial profiling is not that accurate and happens to make the wrong scheduling decision. However, even for this particular workload, HeteroCore still outperforms a conventional GPU by 8.3%.

6.2 Hardware Cost and Energy

We now evaluate area cost and energy consumption using GPUWattch [29]. The per-SM TLP resources (shared memory, register file, warp slots and CTA slots) change across SM types; the ALUs and other components are not affected. Compared to a baseline SM, the area of a big-SM increases by 6.9% whereas the area of a small-SM decreases by 3.1%. Overall, the total area of the HeteroCore GPU increases by 3.8% over the conventional GPU. Taking into account the fraction taken up by the SMs relative to the entire chip, based on the Nvidia Fermi die photo [14], this translates into the HeteroCore GPU occupying 1.8% more chip area. Note that a 3.8% increase in chip area is (much) smaller than the area cost of one SM. Moreover, even when assuming that GPU performance increases linearly with SM count in the ideal scenario, increasing the number of SMs from 16 to 17 would improve performance by at most 6.2%.

The HeteroCore GPU improves energy consumption per instruction by 11.2% for the heterogeneous workloads and is energy-neutral for the homogeneous workloads. Overall, the HeteroCore GPU improves energy consumption per instruction by 7.2% on average. The reduction in energy consumption comes from improved performance which compensates for the slight power consumption increase of 4.1% on average for the HeteroCore GPU.

6.3 Single-Kernel Performance

So far we considered multikernel workloads. Obviously, in a real execution context, there might be periods of execution during which only a single kernel is running. The question then is what performance to observe on the HeteroCore GPU. Figure 9 reports single-kernel performance, normalized to the conventional GPU, along with big-SM and small-SM performance. Overall, performance is largely unaffected for the majority of the benchmarks. A couple benchmarks, i.e., NQU, STO, B+TREE, DXTC and HISTO, experience a performance degradation of at most 6.3% (B+TREE). A couple benchmarks, i.e., NN, SC and FDTD3D, experience a significant performance improvement reaching up to 46.6% (SC).

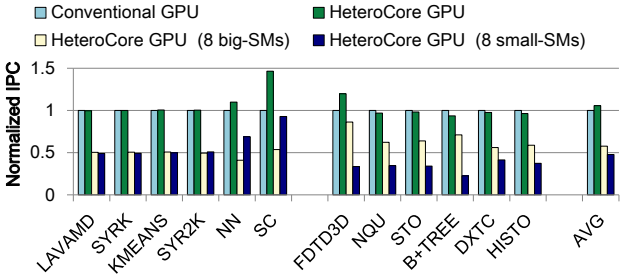


Fig. 9: Single-kernel performance: *HeteroCore does not degrade single-kernel performance.*

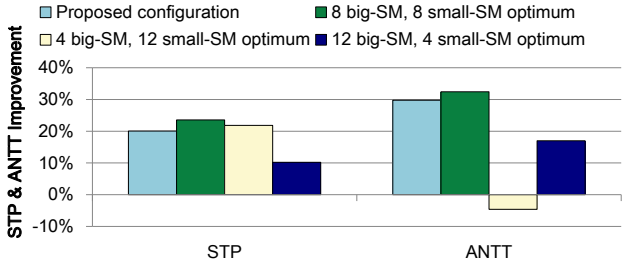


Fig. 10: Sensitivity analyses: *Proposed HeteroCore configuration against optimum configuration with 8 big-SMs and 8 small-SMs, versus 4 big-SMs and 12 small-SMs, versus 12 big-SMs and 4 small-SMs.*

Most of the lean-TLP kernels, see Figure 9 on the left, are unaffected because big-SM and small-SM performance is similar. We observe a significant performance improvement for the cache-sensitive applications NN (9.9%) and SC (46.6%). The reason is high performance on the small-SM because of less cache contention when co-executing fewer CTAs. For the thick-TLP kernels, see Figure 9 on the right, high performance is achieved on the big-SMs, as expected. The improvement is not as big as the performance drop on the small-SMs, hence a net but small performance degradation for most thick-TLP kernels. The exception is FDTD3D with a performance improvement of 19.9%: FDTD3D is a memory-intensive benchmark that greatly benefits from increased memory access latency hiding when running on the big-SMs.

6.4 Four-Kernel Performance

We next evaluate the performance of the HeteroCore GPU architecture with more than two co-executing kernels. In particular, we generate 310 4-kernel workloads containing kernels with different TLP characteristics, such as 3T1L, 2T2L and 1T3L (3T1L means 3 thick-TLP kernels co-running with 1 lean-TLP kernel), out of which we randomly choose 50 workloads. The dynamic TLP resource-aware performance scheduling algorithm employed here for 4 co-running kernels is a straightforward extension upon the one described in Section 4: we first profile performance for each kernel on 2 big-SMs plus 2 small-SMs, and then determine the kernel-to-SM mapping that maximizes STP assuming that each kernel occupies either 4 big-SMs or 4 small-SMs. The results are in line with the ones for two kernels: the HeteroCore GPU improves STP by 19.0% on average (and up to 59.8%), while at the same time improving ANTT by 33.2% on average.

6.5 Sensitivity Analyses

As a final step in the evaluation, we perform two sensitivity analyses to better justify the design choices made in this work. First, we compare our dynamic scheduling policy with optimal scheduling. Second, we explore different forms of heterogeneity with 4 big-SMs and 12 small-SMs versus 12 big-SMs and 4 small-SMs.

Optimal Scheduling: Recall from Section 4 that we assign 8 SMs of the same type to both co-running kernels in a heterogeneous workload mix. The question may be asked whether this is optimal. Why do we evenly split the SMs and why do we give all big-SMs to one kernel? Why not give more SMs to one kernel? And why do all SMs assigned to a given kernel need to be from the same type? To answer this question, we consider uneven distributions (i.e., give 4 SMs to one kernel and 12 to the other), and for each of those 4 SMs assigned, we consider all possible combinations of big-SMs (B) versus small-SMs (S), i.e., 4B, 1B3S, 2B2S, 1B3S and 4S, and pick the optimum. The results are shown in Figure 10 for the heterogeneous workloads: the first bar shows our proposed scheduling algorithm that dynamically assigns 8 SMs to both kernels; the second bar shows the optimum determined statically across all possible combinations of 4, 8 or 12 SMs for either kernel. The key conclusion is that our proposed solution is within 3.5.% on average compared to the optimum; we conclude that assigning 8 SMs to both kernels is optimal and no additional performance benefit is obtained through an uneven split of SMs to kernels.

Exploring Heterogeneity: Another question that may be raised is why we consider 8 big-SMs and 8 small-SMs. Why not 4 big-SMs and 12 small-SMs, or vice versa, 12 big-SMs and 4 small-SMs? To answer this question we construct HeteroCore GPUs with the same hardware area cost using the following scenario: we keep the small-SM configuration unchanged but change the big-SM configuration accordingly in an area-normalized way. Hence, when there are only 4 big-SMs, they are sized bigger with 81,920 registers, 96 KB shared memory and 120 warp slots; when there are 12 big-SMs, they are sized with 38,229 registers, 53 KB shared memory and 56 warp slots. In this experiment we also exhaustively enumerate all scheduling possibilities and pick the optimum. The results are also shown in Figure 10. For the HeteroCore GPU with 4 big-SMs and 12 small-SMs, we observe that the optimum scheduling policy assigns 12 small-SMs to the lean-TLP kernel and 4 big-SMs to the thick-TLP kernel. This keeps STP nearly unchanged, however ANTT suffer severely (by 34.4%) — although per-SM performance improves for the thick-TLP kernel, its overall performance degrades because it has fewer SMs assigned. For the HeteroCore GPU with 12 big-SMs and 4 small-SMs, the available TLP resources are too little in the big-SMs, so that the thick-TLP kernel does not benefit much; the lean-TLP kernel suffers from limited SMs. This leads to both STP and ANTT degradation.

7 Related Work

TLP in GPUs: TLP is fundamental to GPU performance. To increase TLP without incurring extra hardware, Yoon et al. [52] propose the Virtual Thread (VT) architecture: by storing the context of inactive CTAs in shared memory, VT enables assigning more CTAs to an SM. Vijaykumar et al. [44] introduce Zorua, a resource virtualization framework for on-chip TLP-related resources. On the other hand, Kayıran et al. [23] and Lee et al. [28] make the observation that performance decreases with an increasing number of CTAs per SM for some kernels, and hence modulate TLP by allocating the optimal number of CTAs per SM to improve performance. Recent work by Wang et al. [45] analyze the resource contention problem in GPGPU multitasking and propose pattern-based bandwidth management policies to find the proper TLP configuration for each application. Compared to these

previous works, our work is different in scope and contribution by rebalancing the architecture to benefit both thick-TLP and lean-TLP kernels.

GPU Heterogeneity: Kayıran et al. [24] propose μC-states to power-gate or clock-gate datapath components upon under-utilization and/or when memory contention happens. A heterogeneous GPU consisting of big SMs and small SMs is proposed with a different number of streaming processors (SP), special function units (SFU) and load/store (LDST) units. To our knowledge, this is the only work introducing heterogeneity in the GPU. However, the nature of heterogeneity is completely different in the HeteroCore GPU: we focus on heterogeneity to exploit TLP-resource diversity and do not change the number of SPs, SFUs and LDSTs per SM.

GPU multitasking and dynamic parallelism: GPU multitasking has been explored through software-only solutions [18], [34], [48]. On the architecture side, spatial multitasking has been proposed and optimized to co-execute kernels on the GPU [3], [4], [22], [35], [40]. In particular, these prior works focus on SM allocation [3], [4], memory scheduling [22] and preemption policies [35], [40]. Instead of executing tasks on different SMs, simultaneous multi-kernel (SMK) execution [49] and Warp-Slicer [51] co-execute CTAs from different kernels on the same SM to exploit kernel diversity. Maestro [36] dynamically selects SMK versus spatial multitasking. A number of papers target dynamic parallelism (DP), in which a kernel launches child kernels to increase resource utilization, and reduce the launch overhead, exploit data locality and improve load balancing [9], [20], [41], [46], [47]. All of these prior works focus on resource partitioning and optimization within a conventional GPU; none of these prior works explore the opportunity for exploiting TLP-resource diversity.

Heterogeneity in CPUs: A sizable body of work has looked into exploiting heterogeneity in the CPU world. Kumar et al. [25], [26] were the first to exploit the potential of introducing heterogeneity on a CPU chip. Follow-on work [7], [10], [30], [38], [42], [43] looked into further improving heterogeneous multicore performance through scheduling. The motivation for HeteroCore GPU is completely different from heterogeneous CPUs. Whereas heterogeneous CPUs are motivated by power efficiency, the key idea for the HeteroCore GPU is to 'rebalance' TLP resources from the small-SM to the big-SM, while keeping the number of functional units and cache size unchanged to improve GPU performance.

8 Conclusion

Current GPUs lack the ability to adapt to TLP-resource diversity among co-executing kernels in multitasking GPU-compute workloads. As a result, thick-TLP kernels lose the opportunity of achieving high performance due to insufficient TLP resources within an SM, whereas lean-TLP kernels waste the available TLP resources without getting any performance benefit. In this paper, we propose the HeteroCore GPU architecture consisting of different SM types to improve multitasking performance by exploiting TLP-resource diversity. A big-SM features a bigger register file, bigger shared memory and more warp slots than a small-SM, while keeping the number of ALUs, load/store units and L1 cache size the same. HeteroCore GPU employs spatial profiling to learn big-SM versus small-SM performance during runtime at low overhead, and dynamically schedules kernels to the big-SMs or small-SMs based on the kernels' TLP resource characteristics. Experimental results show that HeteroCore GPU delivers significantly higher performance. For the multitasking workloads with different TLP-resource characteristics, HeteroCore GPU improves overall system throughput by 20.1% on average (up to 80.9%) and per-application performance by 29.8% on average (up to 50.3%) compared to a conventional GPU with similar hardware cost. Moreover, single-task performance is unaffected on average.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We thank HPC-UGent for the computing infrastructure. This work is supported by the European Research Council (ERC) Advanced Grant agreement No. 741097, FWO projects G.0434.16N and G.0144.17N, NSFC under Grant No. 61572508 and 61672526, NUDT Research Project No. ZK17-03-06. Xia Zhao is supported through a CSC scholarship and UGent-BOF co-funding.

References

[1] NVIDIA CUDA SDK Code Samples. https://developer.nvidia.com/cuda-downloads.

[2] NVIDIA's Next Generation CUDA Compute Architecture:Fermi. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[3] Jacob T. Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. The Case for GPGPU Spatial Multitasking. In *Proceedings of the 18th International Symposium on High-Performance Computer Architecture, HPCA*, Feb. 2012.

[4] P. Aguilera, K. Morrow, and N. S. Kim. Fair Share: Allocation of GPU Resources for Both Performance and Fairness . In *The 32nd IEEE International Conference on Computer Design, ICCD*, Oct. 2014.

[5] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceeding of the International Symposium on Performance Analysis of Systems and Software, ISPASS*, April 2009.

[6] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, and Karthikeyan Sankaralingam. Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU. *ACM Trans. Archit. Code Optim.*, 12(2), June 2015.

[7] Michela Becchi and Patrick Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proceedings of the 3rd Conference on Computing Frontiers, CF*, May 2006.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the International Symposium on Workload Characterization, IISWC*, Oct. 2009.

[9] G. Chen and X. Shen. Free Launch: Optimizing GPU Dynamic Kernel Launches through Thread Reuse. In *Proceedings of the 48th Annual International Symposium on Microarchitecture, MICRO*, Dec. 2015.

[10] Jian Chen and Lizy K. John. Efficient Program Scheduling for Heterogeneous Multi-core Processors. In *Proceedings of the 46th Design Automation Conference, DAC*, July 2009.

[11] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the 22Nd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, April 2017.

[12] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, April 2016.

[13] Hongwen Dai, Zhen Lin, Chao Li, Chen Zhao, Fei Wang, Nanning Zheng, and Huiyang Zhou. Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls. In *Proceedings of the International Symposium on High Performance Computer Architecture, HPCA*, Mar. 2018.

[14] Bill Dally. Next Gen CUDA GPU Architecture, Code-Named "Fermi". http://storageapplicationsinc.com/html/nvidia_fermi_external.pdf, 2014.

[15] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.

[16] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. A Hierarchical Thread Scheduler and Register File for Energy-Efficient Throughput Processors. *ACM Trans. Comput. Syst.*, 30(2):8:1–8:38, April 2012.

[17] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a High-Level Language Targeted to GPU Codes. In *2012 Innovative Parallel Computing (InPar)*, May 2012.

[18] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained Resource Sharing for Concurrent GPGPU Kernels. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism, HotPar*, June 2012.

[19] H. Guan, J. Yao, Z. Qi, and R. Wang. Energy-Efficient SLA Guarantees for Virtualized GPU in Cloud Gaming. *IEEE Transactions on Parallel and Distributed Systems*, 26(9):2434–2443, Sep. 2015.

[20] I. E. Hajj, J. Gomez-Luna, C. Li, L. W. Chang, D. Milojicic, and W. m. Hwu. KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism. In *Proceedings of the 49th Annual International Symposium on Microarchitecture, MICRO*, Oct. 2016.

[21] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. GPU Register File Virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO*, Dec. 2015.

[22] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. Anatomy of GPU Memory System for Multi-Application Execution. In *Proceedings of the 2015 International Symposium on Memory Systems, MEMSYS*, Oct. 2015.

[23] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques, PACT*, Sep. 2013.

[24] Onur Kayıran, Adwait Jog, Ashutosh Pattnaik, Rachata Ausavarungnirun, Xulong Tang, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. μC-States: Fine-grained GPU Datapath Power Management. In *Proceedings of the 25th International Conference on Parallel Architectures and Compilation, PACT*, Sep. 2016.

[25] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th International Symposium on Microarchitecture, MICRO*, Dec. 2003.

[26] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st International Symposium on Computer Architecture, ISCA*, June 2004.

[27] J. Lee and H. Kim. TAP: A TLP-Aware Cache Management Policy For a CPU-GPU Heterogeneous Architecture. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture, HPCA*, Feb 2012.

[28] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture, HPCA*, Feb. 2014.

[29] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th International Symposium on Computer Architecture, ISCA*, June 2013.

[30] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. Composite Cores: Pushing Heterogeneity Into a Core. In *Proceedings of the 45th International Symposium on Microarchitecture, MICRO*, Dec. 2012.

[31] Nvidia. NVIDIA GP100 Pascal Architecture. White paper. http://www.nvidia.com/object/pascal-architecture-whitepaper.html, 2016.

[32] Nvidia. NVIDIA Tesla V100 GPU Architecture The Worlds Most Advanced Data Center GPU. White paper. http://www.nvidia.com/object/volta-architecture-whitepaper.html, 2017.

[33] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[34] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, Mar. 2013.

[35] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 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*, Mar. 2015.

[36] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. In *Proceedings of the 22th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, Apr. 2017.

[37] Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. Supporting GPU Sharing in Cloud Environments with a Transparent Runtime Consolidation Framework. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing, HPDC*, June 2011.

[38] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. HASS: A Scheduler for Heterogeneous Multicore Systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, April 2009.

[39] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, Mar. 2012.

[40] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling Preemptive Multiprogramming on GPUs. In *Proceeding of the 41st International Symposium on Computer Architecuture, ISCA*, June 2014.

[41] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *Proceedings of the 23rd Annual International Symposium on High Performance Computer Architecture, HPCA*, Feb. 2017.

[42] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. Fairness-Aware Scheduling on Single-ISA Heterogeneous Multi-Cores. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT*, Sep. 2013.

[43] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *Proceedings of the 39th International Symposium on Computer Architecture, ISCA*, June 2012.

[44] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B Gibbons, and Onur Mutlu. Zorua: A Holistic Approach to Resource Virtualization in GPUs. In *Proceedings of the 49th International Symposium on Microarchitecture, MICRO*, Oct. 2016.

[45] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog. Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management. In *Proceedings of the International Symposium on High Performance Computer Architecture, HPCA*, pages 247–258, Feb. 2018.

[46] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA*, June 2015.

[47] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture, ISCA*, June 2016.

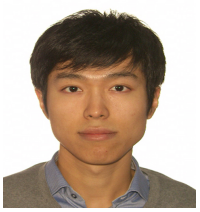
[48] L. Wang, M. Huang, and T. El-Ghazawi. Exploiting Concurrent Kernel Execution on Graphic Processing Units. In *Proceedings of the 2011 International Conference on High Performance Computing & Simulation, HPCS*, July 2011.

[49] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-Grained Sharing. In *Proceedings of the 22nd IEEE Symposium on High Performance Computer Architecture, HPCA*, Mar. 2016.

[50] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Quality of Service Support for Fine-Grained Sharing on GPUs. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA*, June 2017.

[51] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *Proceedings of the 43th International Symposium on Computer Architecture, ISCA*, June 2016.

[52] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram. Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit. In *Proceedings of the 43th International Symposium on Computer Architecture, ISCA*, June 2016.



Xia Zhao is a third year PhD student at Ghent University, Belgium. He received his M.S. in Computer Science from the National University of Defense Technology (NUDT), Changsha, China, in 2015. His research interests include GPGPU architecture in general, and multi-program execution and Network-on-Chip (NoC) design more in particular.

Zhiying Wang is a Professor with the School of Computer, NUDT. He received his PhD in electrical engineering from the National University of Defense Technology (NUDT), Changsha, China, in 1988. He has contributed over 10 invited chapters to book volumes, published 240 papers in archival journals and refereed conference proceedings, and delivered over 30 keynotes. His main research fields include computer architecture, computer security, VLSI design, reliable architecture, multicore memory system, and asynchronous circuit. He is an IEEE member.

Lieven Eeckhout is Professor at Ghent University, Belgium. He received his PhD in Computer Science and Engineering from Ghent University in 2002. His research interests are in the area of computer architecture, with a specific interest in performance analysis, evaluation and modeling. He is the current editor-in-chief of IEEE Micro (2015-2018). He is the recipient of the 2017 Maurice Wilkes Award. His research is funded by the European Research Council under the European Communitys Horizon 2020 Programme / ERC Advanced Grant agreement no. 741097, as well as Research Foundation – Flanders (FWO) grants no. G.0434.16N and G.0144.17N. He is an IEEE Fellow.