Zorua: A Holistic Approach to Resource Virtualization in GPUs

Nandita Vijaykumar[†] Kevin Hsieh[†] Gennady Pekhimenko[‡†] Samira Khan[ℵ]
Ashish Shrestha[†] Saugata Ghose[†] Adwait Jog[┼] Phillip B. Gibbons[†] Onur Mutlu[§†]

[†]*Carnegie Mellon University* [‡]*Microsoft Research* [ℵ]*University of Virginia*
[┼]*College of William and Mary* [§]*ETH Zürich*

Abstract

This paper introduces a new resource virtualization framework, Zorua*, that decouples the programmer-specified resource usage of a GPU application from the actual allocation in the on-chip hardware resources. Zorua enables this decoupling by* virtualizing *each resource transparently to the programmer. The virtualization provided by Zorua builds on two key concepts—*dynamic allocation *of the on-chip resources and their* oversubscription *using a swap space in memory.*

Zorua provides a holistic GPU resource virtualization strategy, designed to (i) *adaptively* control the extent *of oversubscription, and* (ii) coordinate *the dynamic management of multiple on-chip resources (i.e., registers, scratchpad memory, and thread slots), to maximize the effectiveness of virtualization. Zorua employs a hardware-software codesign, comprising the compiler, a runtime system and hardware-based virtualization support. The runtime system leverages information from the compiler regarding resource requirements of each program phase to* (i) *dynamically allocate/deallocate the different resources in the physically available on-chip resources or their swap space, and* (ii) *manage the tradeoff between higher thread-level parallelism due to virtualization versus the latency and capacity overheads of swap space usage.*

We demonstrate that by providing the illusion of more resources than physically available via controlled and coordinated virtualization, Zorua offers several important benefits: (i) ***Programming Ease.*** *Zorua eases the burden on the programmer to provide code that is tuned to efficiently utilize the physically available on-chip resources.* (ii) ***Portability.*** *Zorua alleviates the necessity of re-tuning an application's resource usage when porting the application across GPU generations.* (iii) ***Performance.*** *By dynamically allocating resources and carefully oversubscribing them when necessary, Zorua improves or retains the performance of applications that are already highly tuned to best utilize the hardware resources. The holistic virtualization provided by Zorua can also enable other uses, including fine-grained resource sharing among multiple kernels and low-latency preemption of GPU programs.*

1. Introduction

Modern Graphics Processing Units (GPUs) have evolved into powerful programmable machines over the last decade, offering high performance and energy efficiency for many classes of applications by concurrently executing thousands of threads. In order to execute, each thread requires several major on-chip resources: *(i)* registers, *(ii)* scratchpad memory (if used in the program), and *(iii)* a thread slot in the thread scheduler that keeps all the bookkeeping information required for execution.

Today, these resources are *statically* allocated to threads in the application based on several parameters—the number of threads per thread block, register usage per thread, and scratchpad usage per block. We refer to these static application parameters as the *resource specification* of the application. This static allocation over a fixed set of hardware resources creates a *tight coupling* between the application resource specification and the physical hardware resources. As a result of this tight coupling, for each application, there are only a few optimized resource specifications that maximize resource utilization. Picking a suboptimal specification leads to underutilization of resources and hence, very often, performance degradation. This leads to three key difficulties related to obtaining good performance on modern GPUs.

Programming Ease. First, the burden falls upon the programmer to optimize the resource specification. For a naive programmer, this is a challenging task [14, 47, 59, 60, 61, 65] because, in addition to selecting a specification suited to an algorithm, the programmer needs to be aware of the details of the GPU architecture to fit the specification to the underlying hardware resources. This *tuning* is easy to get wrong because there are *many* highly suboptimal performance points in the specification space, and even a minor deviation from an optimized specification can lead to a drastic drop in performance due to lost parallelism. We refer to such drops as *performance cliffs*. We analyze the effect of suboptimal specifications on real systems for 20 workloads (Section 2.1), and experimentally demonstrate that changing resource specifications can produce as much as a 5× difference in performance due to the change in parallelism. Even a minimal change in one resource can result in a significant performance cliff, degrading performance by as much as 50% (Section 2.1).

Portability. Second, different GPUs have varying quantities of each of the resources. Hence, an optimized specification on one GPU may be highly suboptimal on another. In order to determine the extent of this portability problem, we run 20 applications on three generations of NVIDIA GPUs: Fermi, Kepler, and Maxwell (Section 2.2). An example result demonstrates that highly-tuned code for Maxwell or Kepler loses as much as 69% of its performance on Fermi. This lack of *portability* necessitates that the programmer *re-tune* the resource specification of the application for *every* new GPU generation. This problem is especially significant in virtualized environments, such as clouds or clusters, where the same program may run on a wide range of GPU architectures.

Performance. Third, for the programmer who chooses to employ software optimization tools (e.g., auto-tuners) or manually tailor the program to fit the hardware, performance is still constrained by the *fixed, static* resource specification. It is well known [19, 21, 30, 84] that the on-chip resource requirements of a GPU application vary throughout execution. Since the program (even after auto-tuning) has to *statically* specify its *worst-case* resource requirements, severe *dynamic underutilization* of several GPU resources [19, 21, 30, 37] ensues, leading to suboptimal performance (Section 2.3).

Our Goal. To address these three challenges at the same time, we propose to *decouple* an application's resource specification from the available hardware resources by *virtualizing* all three major resources in a holistic manner. This virtualization provides the illusion of more resources to the GPU programmer and software than physically available, and enables the runtime system and the hardware to *dynamically* manage multiple

978-1-5090-3508-3/16/$31.00 ©2016 IEEE

resources in a manner that is transparent to the programmer, thereby alleviating dynamic underutilization.

Virtualization is a concept that has been applied to the management of hardware resources in many contexts (e.g., [2, 6, 13, 15, 25, 29, 54, 73]), providing various benefits. We believe that applying the general principle of virtualization to the management of *multiple* on-chip resources in GPUs offers the opportunity to alleviate several important challenges in modern GPU programming, which are described above. However, effectively adding a new level of indirection to the management of multiple latency-critical GPU resources introduces several new challenges (see Section 3.1). This necessitates the design of a new mechanism to effectively address the new challenges and enable the benefits of virtualization. In this work, we introduce a new framework, *Zorua*,[1] to decouple the programmer-specified resource specification of an application from its physical on-chip resource allocation by effectively virtualizing multiple on-chip resources in GPUs.

Key Concepts. The virtualization strategy used by Zorua is built upon two key concepts. First, to mitigate performance cliffs when we do not have enough physical resources, we *oversubscribe* resources by a small amount at runtime, by leveraging their dynamic underutilization and maintaining a swap space (in main memory) for the extra resources required. Second, Zorua improves utilization by determining the runtime resource requirements of an application. It then allocates and deallocates resources dynamically, managing them *(i) independently* of each other to maximize their utilization; and *(ii)* in a *coordinated* manner, to enable efficient execution of each thread with all its required resources available.

Challenges in Virtualization. Unfortunately, oversubscription means that latency-critical resources, such as registers and scratchpad, may be swapped to memory at the time of access, resulting in high overheads in performance and energy. This leads to two critical challenges in designing a framework to enable virtualization. The first challenge is to effectively determine the *extent* of virtualization, i.e., by how much each resource appears to be larger than its physical amount, such that we can minimize oversubscription while still reaping its benefits. This is difficult as the resource requirements continually vary during runtime. The second challenge is to minimize accesses to the swap space. This requires *coordination* in the virtualized management of *multiple resources*, so that enough of each resource is available on-chip when needed.

Zorua. In order to address these challenges, Zorua employs a hardware-software codesign that comprises three components: *(i)* ***the compiler*** annotates the program to specify the resource needs of *each phase* of the application; *(ii)* ***a runtime system***, which we refer to as the *coordinator*, uses the compiler annotations to dynamically manage the virtualization of the different on-chip resources; and *(iii)* ***the hardware*** employs mapping tables to locate a virtual resource in the physically available resources or in the swap space in main memory. The coordinator plays the key role of scheduling threads *only when* the expected gain in thread-level parallelism outweighs the cost of transferring oversubscribed resources from the swap space in memory, and coordinates the oversubscription and allocation of multiple on-chip resources.

[1] Named after a Pokémon [51] with the power of illusion, able to take different shapes to adapt to different circumstances (not unlike our proposed framework).

Key Results. We evaluate Zorua with many resource specifications for eight applications across three GPU architectures (Section 6). Our experimental results show that Zorua *(i)* reduces the range in performance for different resource specifications by 50% on average (up to 69%), by alleviating performance cliffs, and hence eases the burden on the programmer to provide optimized resource specifications, *(ii)* improves performance for code with optimized specification by 13% on average (up to 28%), and *(iii)* enhances portability by reducing the maximum porting performance loss by 55% on average (up to 73%) for three different GPU architectures. We conclude that decoupling the resource specification and resource management via virtualization significantly eases programmer burden, by alleviating the need to provide optimized specifications and enhancing portability, while still improving or retaining performance for programs that already have optimized specifications.

Other Uses. We believe that Zorua offers the opportunity to address several other key challenges in GPUs today: (i) By providing an new level of indirection, Zorua provides a natural way to enable dynamic and fine-grained control over resource partitioning among *multiple GPU kernels and applications*. (ii) Zorua can be utilized for *low-latency preemption* of GPU applications, by leveraging the ability to swap in/out resources from/to memory in a transparent manner (Section 7).

The main **contributions** of this work are:

- This is the first work that takes a holistic approach to decoupling a GPU application's resource specification from its physical on-chip resource allocation via the use of virtualization. We develop a comprehensive virtualization framework that provides *controlled* and *coordinated* virtualization of *multiple* on-chip GPU resources to maximize the efficacy of virtualization.
- We show how to enable efficient oversubscription of multiple GPU resources with dynamic fine-grained allocation of resources and swapping mechanisms into/out of main memory. We provide a hardware-software cooperative framework that *(i)* controls the extent of oversubscription to make an effective tradeoff between higher thread-level parallelism due to virtualization versus the latency and capacity overheads of swap space usage, and *(ii)* coordinates the virtualization for multiple on-chip resources, transparently to the programmer.
- We demonstrate that by providing the illusion of having more resources than physically available, Zorua *(i)* reduces programmer burden, providing competitive performance for even suboptimal resource specifications, by reducing performance variation across different specifications and by alleviating performance cliffs; *(ii)* reduces performance loss when the program with its resource specification tuned for one GPU platform is ported to a different platform; and *(iii)* retains or enhances performance for highly-tuned code by improving resource utilization, via dynamic management of resources.

2. Motivation

In this section, we study the performance implications of different choices of resource specifications for GPU applications to demonstrate the key issues we aim to alleviate.

2.1. Performance Variation and Cliffs

To understand the impact of resource specifications and the resulting utilization of physical resources on GPU performance, we conduct an experiment on a Maxwell GPU system (GTX 745) with 20 GPGPU workloads from the CUDA SDK [52], Rodinia [9], GPGPU-Sim benchmarks [5], Lonestar [8], Parboil [66], and US DoE application suites [72]. We use the NVIDIA profiling tool (NVProf) [52] to determine the execution time of each application kernel. We sweep the three parameters of the specification—number of threads in a thread block, register usage per thread, and scratchpad memory usage per thread block—for each workload, and measure their impact on execution time.

Figure 1 shows a summary of variation in performance (higher is better), normalized to the slowest specification for each application, across all evaluated specification points for each application[2] in a Tukey box plot [48]. The boxes in the box plot represent the range between the first quartile (25%) and the third quartile (75%). The whiskers extending from the boxes represent the maximum and minimum points of the distribution, or 1.5× the length of the box, whichever is smaller. Any points that lie more than 1.5× the box length beyond the box are considered to be outliers [48], and are plotted as individual points. The line in the middle of the box represents the median, while the "X" represents the average.

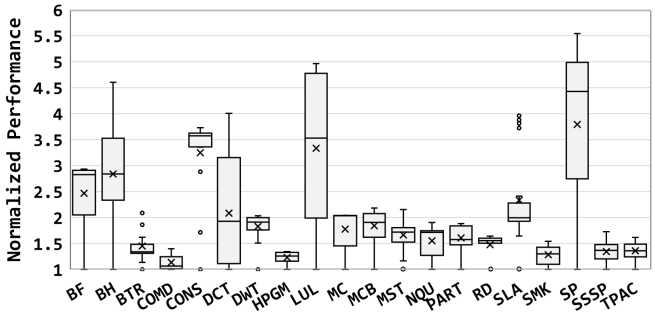
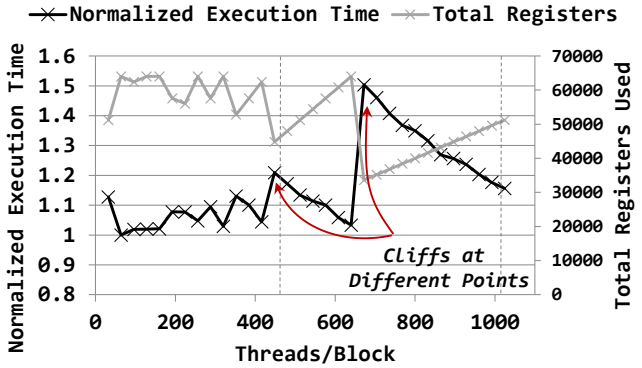


Figure 1: Performance variation across specifications

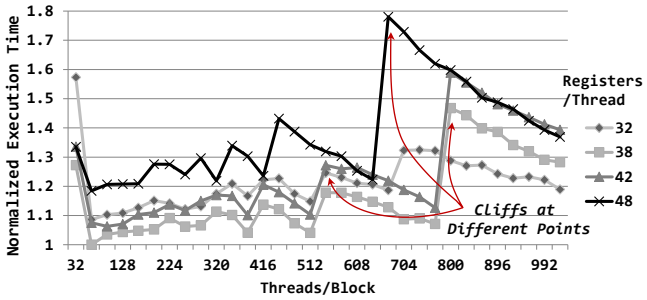
We can see that there is significant variation in performance across different specification points (as much as 5.51× in *SP*), proving the importance of optimized resource specifications. In some applications (e.g., *BTR, SLA*), few points perform well, and these points are significantly better than others, suggesting that it would be challenging for a programmer to locate these high performing specifications and obtain the best performance. Many workloads (e.g., *BH, DCT, MST*) also have higher concentrations of specifications with suboptimal performance in comparison to the best performing point, implying that, without effort, it is likely that the programmer will end up with a resource specification that leads to low performance.

There are several sources for this performance variation. One important source is the loss in thread-level parallelism as a result of a suboptimal resource specification. Suboptimal specifications that are *not* tailored to fit the available physical resources lead to the underutilization of resources. This causes a drop in the number of threads that can be executed concurrently, as there are insufficient resources to support their execution. Hence, better and more balanced utilization of resources enables higher thread-level parallelism. Often, this loss in parallelism from resource underutilization manifests itself in what we refer to as a *performance cliff*, where a small deviation from an optimized specification can lead to significantly worse performance, i.e., there is very high variation in performance between two specification points that are nearby. To demonstrate the existence and analyze the behavior of performance cliffs, we examine two representative workloads more closely.

Figure 2a shows *(i)* how the application execution time changes; and *(ii)* how the corresponding number of registers, statically used, changes when the number of threads per thread block increases from 32 to 1024 threads, for *Minimum Spanning Tree (MST)* [8]. We make two observations.



(a) Threads/block sweep



(b) Threads/block & Registers/thread sweep

Figure 2: Performance cliffs in *Minimum Spanning Tree* (*MST*)

First, let us focus on the execution time between 480 and 1024 threads per block. As we go from 480 to 640 threads per block, execution time gradually decreases. Within this window, the GPU can support two thread blocks running concurrently for *MST*. The execution time falls because the increase in the number of threads per block improves the overall throughput (the number of thread blocks running concurrently remains constant at two, but each thread block does more work in parallel by having more threads per block). However, the corresponding total number of registers used by the blocks also increases. At 640 threads per block, we reach the point where the total number of available registers is not large enough to support two blocks. As a result, the number of blocks executing in parallel drops from two to one, resulting in a significant increase (50%) in execution time, i.e., the *performance cliff*. [3]

[2]Our technical report [71] contains more detail on the evaluated ranges.

[3]Prior work [77] has studied performing resource allocation at the finer warp granularity, as opposed to the coarser granularity of a thread block. As we discuss in Section 8 and demonstrate in Section 6, this does not solve the problem of performance cliffs.

We see many of these cliffs earlier in the graph as well, albeit not as drastic as the one at 640 threads per block.

Second, Figure 2a shows the existence of performance cliffs when we vary *just one* system parameter—the number of threads per block. To make things more difficult for the programmer, other parameters (i.e., registers per thread or scratchpad memory per thread block) also need to be decided at the same time. Figure 2b demonstrates the same results, but when the *number of registers per thread* is also varied from 32 to 48.[4] As this figure shows, performance cliffs now occur at *different points* for *different registers/thread curves*, which makes optimizing resource specification, so as to avoid these cliffs, much harder for the programmer.

We find that performance cliffs are pervasive in the workloads we study. *Barnes-Hut (BH)* in Figure 3 is another example that exhibits very significant performance cliffs depending on the number of threads per block and registers per thread.

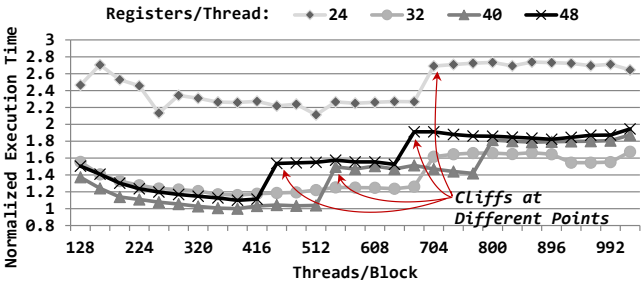


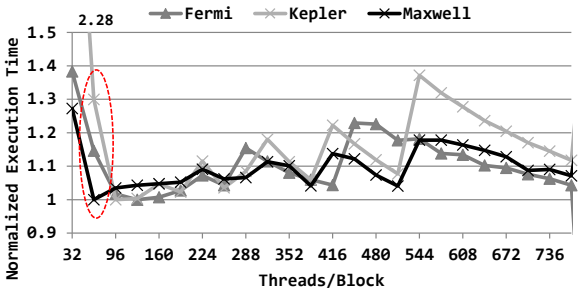
Figure 3: Performance cliffs in *Barnes-Hut* (*BH*)

2.2. Portability

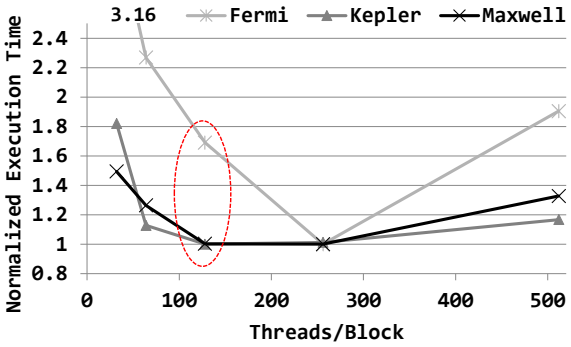
As we show in Section 2.1, tuning GPU applications to achieve good performance on a given GPU is already a challenging task. To make things worse, even after this tuning is done by the programmer for one particular GPU architecture, it has to be *redone* for every new GPU generation (due to changes in the available physical resources across generations) to ensure that good performance is retained. We demonstrate this *portability problem* by running sweeps of the three parameters of the resource specification on various workloads, on three real GPU generations: Fermi (GTX 480), Kepler (GTX 760), and Maxwell (GTX 745).

Figure 4 shows how the optimized performance points change between different GPU generations for two representative applications (*MST* and *DCT*). For every generation, results are normalized to the lowest execution time for that particular generation. As we can see in Figure 4a, the best performing points for different generations occur at different specifications because the application behavior changes with the variation in hardware resources. For *MST*, the *Maxwell* architecture performs best at 64 threads per block. However, the same specification point is not efficient for either of the other generations (*Fermi* and *Kepler*), producing 15% and 30% lower performance, respectively, compared to the best specification for each generation. Similarly, for *DCT* (shown in Figure 4b), both *Kepler* and *Maxwell* perform best at 128 threads per block, but using the same specification for *Fermi* would lead to a 69% performance loss.

[4]We note that the register usage reported by the compiler may vary from the actual runtime register usage [52], hence slightly altering the points at which cliffs occur.



(a) *MST*



(b) *DCT*

Figure 4: Performance variation across different GPU generations (Fermi, Kepler, and Maxwell) for *MST* and *DCT*

2.3. Dynamic Resource Underutilization

Even when a GPU application is *perfectly* tuned for a particular GPU architecture, the on-chip resources are typically not fully utilized [19, 21, 22, 30, 40, 55, 70, 84]. For example, it is well known that while the compiler conservatively allocates registers to hold the *maximum number* of live values throughout the execution, the number of live values at any given time is well below the maximum for large portions of application execution time. To determine the magnitude of this *dynamic underutilization*,[5] we conduct an experiment where we measure the dynamic usage (per epoch) of both scratchpad memory and registers for different applications with *optimized* specifications in our workload pool. We vary the length of epochs from 500 to 5000 cycles (not graphed). We find that even for a reasonably large epoch of 500 cycles, the average utilization of resources is very low (only 12% of allocated scratchpad memory and 37% of registers are utilized). Moreover, even with the largest epoch size we analyze (5000 cycles), the average utilization of allocated scratchpad memory is only 45% and of allocated registers is 68%. This observation clearly suggests that there is an opportunity for better dynamic allocation of these resources that could allow higher effective parallelism.

2.4. Our Goal

As we see above, the tight coupling between the resource specification and hardware resource allocation, and the re-

[5] Underutilization of registers occurs in two major forms—*static*, where registers are unallocated throughout execution [20, 22, 40, 70, 77], and *dynamic*, where utilization of the registers drops during runtime as a result of early completion of warps [77], short register lifetimes [19, 21, 30] and long latency operations [19, 21]. We do not tackle underutilization from long latency operations (such as memory accesses) in this paper, and leave the exploration of alleviating this type of underutilization to future work.

sulting heavy dependence of performance on the resource specification, creates a number of challenges. In this work, our goal is to alleviate these challenges by providing a mechanism that can *(i)* ease the burden on the programmer by ensuring reasonable performance, *regardless of the resource specification*, by successfully avoiding performance cliffs, while retaining performance for code with optimized specification; *(ii)* enhance portability by minimizing the variation in performance for optimized specifications across different GPU generations; and *(iii)* maximize dynamic resource utilization even in highly optimized code to further improve performance. We make two key observations from our studies above to help us achieve this goal:

Observation 1: *Bottleneck Resources.* We find that performance cliffs occur when the amount of any resource required by an application exceeds the physically available amount of that resource. This resource becomes a *bottleneck*, and limits the amount of parallelism that the GPU can support. If it were possible to provide the application with a *small additional amount* of the bottleneck resource, the application can see a significant increase in parallelism and thus avoid the performance cliff.

Observation 2: *Underutilized Resources.* As discussed in Section 2.3, there is significant underutilization of resources at runtime. These underutilized resources could be employed to support more parallelism at runtime, and thereby alleviate the aforementioned challenges.

We use these two observations to drive our resource virtualization solution, which we describe next.

3. Zorua: Our Approach

In this work, we design Zorua, a framework that provides the illusion of more GPU resources than physically available by decoupling the resource specification from its allocation in the hardware resources. We introduce a new level of indirection by virtualizing the on-chip resources to allow the hardware to manage resources transparently to the programmer.

The virtualization provided by Zorua builds upon two *key concepts* to leverage the aforementioned observations. First, when there are insufficient physical resources, we aim to provide the illusion of the required amount by *oversubscribing* the required resource. We perform this oversubscription by leveraging the dynamic underutilization as much as possible, or by spilling to a swap space in memory. This oversubscription essentially enables the illusion of more resources than what is available (physically and statically), and supports the concurrent execution of more threads. Performance cliffs are mitigated by providing enough additional resources to avoid drastic drops in parallelism. Second, to enable efficient oversubscription by leveraging underutilization, we dynamically allocate and deallocate physical resources depending on the requirements of the application during execution. We manage the virtualization of each resource *independently* of other resources to maximize its runtime utilization.

Figure 5 depicts the high-level overview of the virtualization provided by Zorua. The *virtual space* refers to the *illusion* of the quantity of available resources. The *physical space* refers to the *actual* hardware resources (specific to the GPU architecture), and the *swap space* refers to the resources that do not fit in the physical space and hence are *spilled* to other physical locations. For the register file and scratchpad memory, the swap space is mapped to global memory space in the memory hierarchy. For threads, only those that are mapped to the physical space are available for scheduling and execution at any given time. If a thread is mapped to the swap space, its state (i.e., the PC and the SIMT stack) is saved in memory. Resources in the virtual space can be freely re-mapped between the physical and swap spaces to maintain the illusion of the virtual space resources.

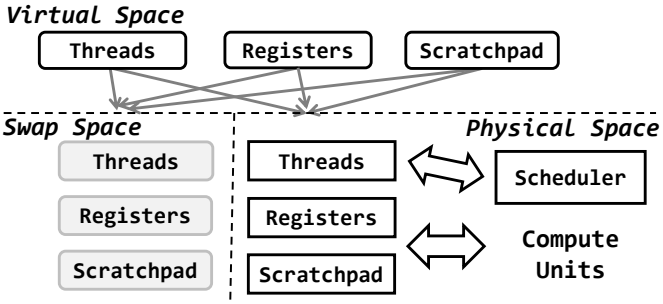


Figure 5: High-level overview of Zorua

In the baseline architecture, the thread-level parallelism that can be supported, and hence the throughput obtained from the GPU, depends on the quantity of *physical resources*. With the virtualization enabled by Zorua, the parallelism that can be supported now depends on the quantity of *virtual resources* (and how their mapping into the physical and swap spaces is managed). Hence, the size of the virtual space for each resource plays the key role of determining the parallelism that can be exploited. Increasing the virtual space size enables higher parallelism, but leads to higher swap space usage. It is critical to minimize accesses to the swap space to avoid the latency overhead and capacity/bandwidth contention associated with accessing the memory hierarchy.

In light of this, there are two key challenges that need to be addressed to effectively virtualize on-chip resources in GPUs. We now discuss these challenges and provide an overview of how we address them.

3.1. Challenges in Virtualization

Challenge 1: *Controlling the Extent of Oversubscription.* A key challenge is to determine the *extent* of oversubscription, or the size of the virtual space for each resource. As discussed above, increasing the size of the virtual space enables more parallelism. Unfortunately, it could also result in more spilling of resources to the swap space. Finding the tradeoff between more parallelism and less overhead is challenging, because the dynamic resource requirements of each thread tend to significantly fluctuate throughout execution. As a result, the size of the virtual space for each resource needs to be *continuously* tuned to allow the virtualization to adapt to the runtime requirements of the program.

Challenge 2: *Control and Coordination of Multiple Resources.* Another critical challenge is to efficiently map the continuously varying virtual resource space to the physical and swap spaces. This is important for two reasons. First, it is critical to minimize accesses to the swap space. Accessing the swap space for the register file or scratchpad involves expensive accesses to global memory, due to the added latency and contention. Also, only those threads that are mapped to the physical space are available to the warp scheduler for selection. Second, each thread requires multiple resources for execution. It is critical

to *coordinate* the allocation and mapping of these different resources to ensure that an executing thread has *all* the required resources allocated to it, while minimizing accesses to the swap space. Thus, an effective virtualization framework must coordinate the allocation of *multiple* on-chip resources.

3.2. Key Ideas of Our Design

To solve these challenges, Zorua employs two key ideas. First, we leverage the software (the compiler) to provide annotations with information regarding the resource requirements of each *phase* of the application. This information enables the framework to make intelligent dynamic decisions, with respect to both the size of the virtual space and the allocation/deallocation of resources (Section 3.2.1).

Second, we use an adaptive runtime system to control the allocation of resources in the virtual space and their mapping to the physical/swap spaces. This allows us to *(i)* dynamically alter the size of the virtual space to change the extent of oversubscription; and *(ii)* continuously coordinate the allocation of multiple on-chip resources and the mapping between their virtual and physical/swap spaces, depending on the varying runtime requirements of each thread (Section 3.2.2).

3.2.1. Leveraging Software Annotations of Phase Characteristics. We observe that the runtime variation in resource requirements (Section 2.3) typically occurs at the granularity of *phases* of a few tens of instructions. This variation occurs because different parts of kernels perform different operations that require different resources. For example, loops that primarily load/store data from/to scratchpad memory tend to be less register heavy. Sections of code that perform specific computations (e.g., matrix transformation, graph manipulation), can either be register heavy or primarily operate out of scratchpad. Often, scratchpad memory is used for only short intervals [84], e.g., when data exchange between threads is required, such as for a reduction operation.

Figure 6 depicts a few example phases from the *NQU* (*N-Queen Solver*) [11] kernel. *NQU* is a scratchpad-heavy application, but it does not use the scratchpad at all during the initial computation phase. During its second phase, it performs its primary computation out of the scratchpad, using as much as 4224B. During its last phase, the scratchpad is used only for reducing results, which requires only 384B. There is also significant variation in the maximum number of live registers in the different phases.

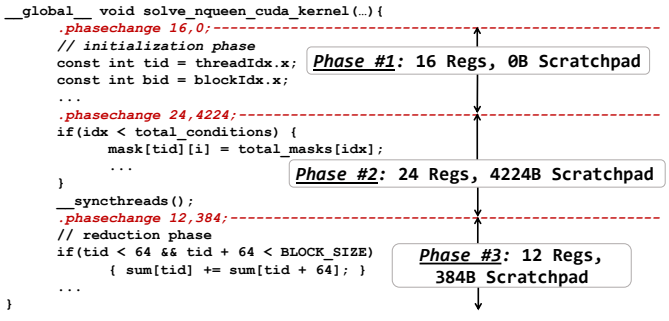


Figure 6: Example phases from *NQU*

In order to capture both the resource requirements as well as their variation over time, we partition the program into a number of *phases*. A phase is a sequence of instructions with sufficiently different resource requirements than adjacent phases. Barrier or fence operations also indicate a change in requirements for a different reason—threads that are waiting at a barrier do not immediately require the thread slot that they are holding. We interpret barriers and fences as phase boundaries since they potentially alter the utilization of their thread slots. The compiler inserts special instructions called *phase specifiers* to mark the start of a new phase. Each phase specifier contains information regarding the resource requirements of the next phase. Section 4.6 provides more detail on the semantics of phases and phase specifiers.

A phase forms the basic unit for resource allocation and deallocation, as well as for making oversubscription decisions. It offers a finer granularity than an *entire thread* to make such decisions. The phase specifiers provide information on the *future resource usage* of the thread at a phase boundary. This enables *(i)* preemptively controlling the extent of oversubscription at runtime, and *(ii)* dynamically allocating and deallocating resources at phase boundaries to maximize utilization of the physical resources.

3.2.2. Control with an Adaptive Runtime System. Phase specifiers provide information to make oversubscription and allocation/deallocation decisions. However, we still need a way to make decisions on the extent of oversubscription and appropriately allocate resources at runtime. To this end, we use an adaptive runtime system, which we refer to as the *coordinator*. Figure 7 presents an overview of the coordinator.

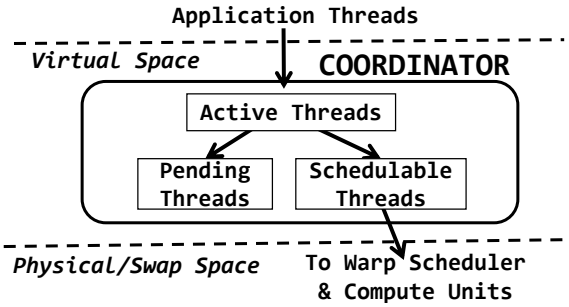


Figure 7: Overview of the coordinator

The virtual space enables the illusion of a larger amount of each of the resources than what is physically available, to adapt to different application requirements. This illusion enables higher thread-level parallelism than what can be achieved with solely the fixed, physically available resources, by allowing more threads to execute concurrently. The size of the virtual space at a given time determines this parallelism, and those threads that are effectively executed in parallel are referred to as *active threads*. All active threads have thread slots allocated to them in the virtual space (and hence can be executed), but some of them may not be mapped to the physical space at a given time. As discussed previously, the resource requirements of each application continuously change during execution. To adapt to these runtime changes, the coordinator leverages information from the phase specifiers to make decisions on oversubscription. The coordinator makes these decisions at every phase boundary and thereby controls the size of the virtual space for each resource (see Section 4.2).

To enforce the determined extent of oversubscription, the coordinator allocates all the required resources (in the virtual space) for only a *subset* of threads from the active threads. Only these dynamically selected threads, referred to as *schedulable*

threads, are available to the warp scheduler and compute units for execution. The coordinator, hence, dynamically partitions the active threads into *schedulable threads* and the *pending threads*. Each thread is swapped between *schedulable* and *pending* states, depending on the availability of resources in the virtual space. Selecting only a subset of threads to execute at any time ensures that the determined size of the virtual space is not exceeded for any resource, and helps coordinate the allocation and mapping of multiple on-chip resources to minimize expensive data transfers between the physical and swap spaces (discussed in Section 4).

3.3. Overview of Zorua

In summary, to effectively address the challenges in virtualization by leveraging the above ideas in design, Zorua employs a software-hardware codesign that comprises three components: *(i)* ***The compiler*** annotates the program by adding special instructions (*phase specifiers*) to partition it into *phases* and to specify the resource needs of each phase of the application. *(ii)* ***The coordinator***, a hardware-based adaptive runtime system, uses the compiler annotations to dynamically allocate/deallocate resources for each thread at phase boundaries. The coordinator plays the key role of continuously controlling the extent of the oversubscription (and hence the size of the virtual space) at each phase boundary. *(iii)* ***Hardware virtualization support*** includes a mapping table for each resource to locate each virtual resource in either the physical space or the swap space in main memory, and the machinery to swap resources between the physical and swap spaces.

4. Zorua: Detailed Mechanism

We now detail the operation and implementation of the various components of the Zorua framework.

4.1. Key Components in Hardware

Zorua has two key hardware components: *(i)* the *coordinator* that contains queues to buffer the *pending threads* and control logic to make oversubscription and resource management decisions, and *(ii) resource mapping tables* to map each of the resources to their corresponding physical or swap spaces.

Figure 8 presents an overview of the hardware components that are added to each SM. The coordinator interfaces with the thread block scheduler (❶) to schedule new blocks onto an SM. It also interfaces with the warp schedulers by providing a list of *schedulable warps* (❼).[6] The resource mapping tables are accessible by the coordinator and the compute units. We present a detailed walkthrough of the operation of Zorua and then discuss its individual components in more detail.

4.2. Detailed Walkthrough

The coordinator is called into action by three events: *(i)* a new thread block is scheduled at the SM for execution, *(ii)* a warp undergoes a phase change, or *(iii)* a warp or a thread block reaches the end of execution. Between these events, the coordinator performs no action and execution proceeds as usual. We now walk through the sequence of actions performed by the coordinator for each type of event.

Thread Block: Execution Start. When a thread block is scheduled onto an SM for execution (❶), the coordinator

[6]We use an additional bit in each warp slots to indicate to the scheduler whether the warp is schedulable.

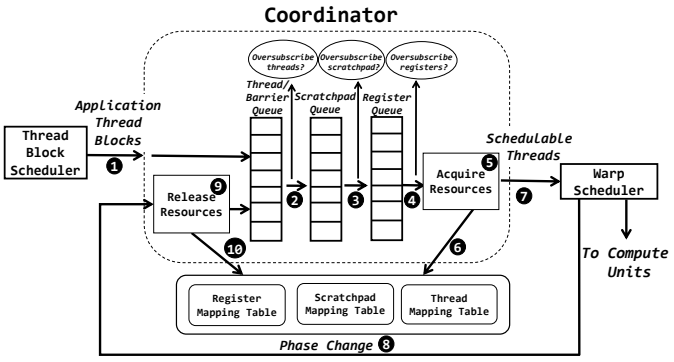


Figure 8: Overview of Zorua in hardware

first buffers it. The primary decision that the coordinator makes is to determine whether or not to make each thread available to the scheduler for execution. The granularity at which the coordinator makes decisions is that of a warp, as threads are scheduled for execution at the granularity of a warp (hence we use *thread slot* and *warp slot* interchangeably). Each warp requires three resources: a thread slot, registers, and potentially scratchpad. The amount of resources required is determined by the phase specifier (Section 4.6) at the start of execution, which is placed by the compiler into the code. The coordinator must supply each warp with *all* its required resources in either the physical or swap space before presenting it to the warp scheduler for execution.

To ensure that each warp is furnished with its resources and to coordinate potential oversubscription for each resource, the coordinator has three queues—*thread/barrier, scratchpad, and register queues*. The three queues together essentially house the *pending threads*. Each warp must traverse each queue (❷ ❸ ❹), as described next, before becoming eligible to be scheduled for execution. The coordinator allows a warp to traverse a queue when *(a)* it has enough of the corresponding resource available in the physical space, or *(b)* it has an insufficient resources in the physical space, but has decided to oversubscribe and allocate the resource in the swap space. The total size of the resource allocated in the physical and swap spaces cannot exceed the determined virtual space size. The coordinator determines the availability of resources in the physical space using the mapping tables (see Section 4.5). If there is an insufficient amount of a resource in the physical space, the coordinator needs to decide whether or not to increase the virtual space size for that particular resource by oversubscribing and using swap space. We describe the decision algorithm in Section 4.4. If the warp cannot traverse *all* queues, it is left waiting in the first (*thread/barrier*) queue until the next coordinator event. Once a warp has traversed *all* the queues, the coordinator acquires all the resources required for the warp's execution (❺). The corresponding mapping tables for each resource is updated (❻) to assign resources to the warp, as described in Section 4.5.

Warp: Phase Change. At each phase change (❽), the warp is removed from the list of schedulable warps and is returned to the coordinator to acquire/release its resources. Based on the information in its phase specifier, the coordinator releases the resources that are no longer live and hence are no longer required (❾). The coordinator updates the mapping tables to free these resources (❿). The warp is then placed into a specific queue, depending on which live resources it retained

from the previous phase and which new resources it requires. The warp then attempts to traverse the remaining queues (❷ ❸ ❹), as described above. A warp that undergoes a phase change as a result of a barrier instruction is queued in the *thread/barrier queue* (❷) until all warps in the same thread block reach the barrier.

Thread Block/Warp: Execution End. When a warp completes execution, it is returned to the coordinator to release any resources it is holding. Scratchpad is released only when the entire thread block completes execution. When the coordinator has free warp slots for a new thread block, it requests the thread block scheduler (❶) for a new block.

Every Coordinator Event. At any event, the coordinator attempts to find resources for warps waiting at the queues, to enable them to execute. Each warp in each queue (starting from the *register queue*) is checked for the availability of the required resources. If the coordinator is able to allocate resources in the physical or swap space without exceeding the determined size of virtual space, the warp is allowed to traverse the queue.

4.3. Benefits of Our Design

Decoupling the Warp Scheduler and Mapping Tables from the Coordinator. Decoupling the warp scheduler from the coordinator enables Zorua to use any scheduling algorithm over the schedulable warps to enhance performance. One case when this is useful is when increasing parallelism degrades performance by increasing cache miss rate or causing memory contention [35, 36, 58]. Our decoupled design allows this challenge to be addressed independently from the coordinator using more intelligent scheduling algorithms [36, 50, 58] and cache management schemes [3, 44, 45, 78]. Furthermore, decoupling the mapping tables from the coordinator allows easy integration of any implementation of the mapping tables that may improve efficiency for each resource.

Coordinating Oversubscription for Multiple Resources. The queues help ensure that a warp is allocated *all* resources in the virtual space before execution. They *(i)* ensure an ordering in resource allocation to avoid deadlocks, and *(ii)* enforce priorities between resources. In our evaluated approach, we use the following order of priorities: threads, scratchpad, and registers. We prioritize scratchpad over registers, as scratchpad is shared by all warps in a block and hence has a higher value by enabling more warps to execute. We prioritize threads over scratchpad, as it is wasteful to allow warps stalled at a barrier to acquire other resources—other warps that are still progressing towards the barrier may be starved of the resource they need. Furthermore, managing each resource independently allows different oversubscription policies for each resource and enables fine-grained control over the size of the virtual space for that resource.[7]

Flexible Oversubscription. Zorua's design can flexibly enable/disable swap space usage, as the dynamic fine-grained management of resources is independent of the swap space. Hence, in cases where the application is well-tuned to utilize the available resources, swap space usage can be disabled or minimized, and Zorua can still improve performance by reducing dynamic underutilization of resources. Furthermore, different oversubscription algorithms can be flexibly employed to manage the size of the virtual space for each resource (independently or cooperatively). These algorithms can be designed for different purposes, e.g., minimizing swap space usage, improving fairness in a multikernel setting, reducing energy, etc. In Section 4.4, we describe an example algorithm to improve performance by making a good tradeoff between improving parallelism and reducing swap space usage.

[7] Our technical report [71] has a more detailed discussion of these benefits.

4.4. Oversubscription Decisions

Leveraging Phase Specifiers. Zorua leverages the information provided by phase specifiers (Section 4.6) to make oversubscription decisions for each phase. For each resource, the coordinator checks whether allocating the requested quantity according to the phase specifier would cause the total swap space to exceed an *oversubscription threshold*, or *o_thresh*. This threshold essentially dynamically sets the size of the virtual space for each resource. The coordinator allows oversubscription for each resource only within its threshold. *o_thresh* is dynamically determined to adapt to the characteristics of the workload, and tp ensure good performance by achieving a good tradeoff between the overhead of oversubscription and the benefits gained from parallelism.

Determining the Oversubscription Threshold. In order to make the above tradeoff, we use two architectural statistics: *(i)* idle time at the cores, *c_idle*, as an indicator for potential performance benefits from parallelism; and *(ii)* memory idle time (the idle cycles when all threads are stalled waiting for data from memory or the memory pipeline), *c_mem*, as an indicator of a saturated memory subsystem that is unlikely to benefit from more parallelism.[8] We use Algorithm 1 to determine *o_thresh* at runtime. Every *epoch*, the change in *c_mem* is compared with the change in *c_idle*. If the increase in *c_mem* is greater, this indicates an increase in pressure on the memory subsystem, suggesting both lesser benefit from parallelism and higher overhead from oversubscription. In this case, we reduce *o_thresh*. On the other hand, if the increase in *c_idle* is higher, this is indicative of more idleness in the pipelines, and higher potential performance from parallelism and oversubscription. We increase *o_thresh* in this case, to allow more oversubscription and enable more parallelism. Table 1 describes the variables used in Algorithm 1.[9]

Algorithm 1 Determining the oversubscription threshold

```
1: o_thresh = o_default
2: for each epoch do
3:     c_idle_delta = (c_idle - c_idle_prev)
4:     c_mem_delta = (c_mem - c_mem_prev)
5:     if (c_idle_delta - c_mem_delta) > c_delta_thresh then
6:         o_thresh += o_thresh_step
7:     end if
8:     if (c_mem_delta - c_idle_delta) > c_delta_thresh then
9:         o_thresh -= o_thresh_step
10:    end if
11: end for
```

4.5. Virtualizing On-chip Resources

A resource can be in either the physical space, in which case it is mapped to the physical on-chip resource, or the swap space, in which case it can be found in the memory hierarchy. Thus, a resource is effectively virtualized, and we need to track the mapping between the virtual and physical/swap

[8] This is similar to the approach taken by prior work [35] to estimate the performance benefits of increasing parallelism.

[9] We include more detail on these variables in our technical report [71].

Variable	Description
o_thresh	oversubscription threshold (dynamically determined)
o_default	initial value for *o_thresh*
c_idle	core cycles when no threads are issued to the core (but the pipeline is not stalled) [35]
c_mem	core cycles when all warps are waiting for data from memory or stalled at the memory pipeline
**_prev*	the above statistics for the previous epoch
c_delta_thresh	threshold to produce change in *o_thresh*
o_thresh_step	increment/decrement to *o_thresh*
epoch	interval in core cycles to change *o_thresh*

Table 1: Variables for oversubscription

spaces. We use a *mapping table* for each resource to determine *(i)* whether the resource is in the physical or swap space, and *(ii)* the location of the resource within the physical on-chip hardware. The compute units access these mapping tables before accessing the real resources. An access to a resource that is mapped to the swap space is converted to a global memory access that is addressed by the logical resource ID and warp/block ID (and a base register for the swap space of the resource). In addition to the mapping tables, we use two registers per resource to track the amount of the resource that is *(i)* free to be used in physical space, and *(ii)* mapped in swap space. These two counters enable the coordinator to make oversubscription decisions (Section 4.4). We now go into more detail on virtualized resources in Zorua.[10]

4.5.1. Virtualizing Registers and Scratchpad Memory. In order to minimize the overhead of large mapping tables, we map registers and scratchpad at the granularity of a *set*. The size of a set is configurable by the architect—we use 4**warp_size*[11] for the register mapping table, and 1KB for scratchpad. The register mapping table is indexed by the warp ID and the logical register set number (*logical_register_number / register_set_size*). The scratchpad mapping table is indexed by the block ID and the logical scratchpad set number (*logical_scratchpad_address / scratchpad_set_size*). Each entry in the mapping table contains the physical address of the register/scratchpad content in the physical register file or scratchpad. The valid bit indicates whether the logical entry is mapped to the physical space or the swap space. With 64 logical warps and 16 logical thread blocks (see Section 5.1), the register mapping table takes 1.125 KB ($64 \times 16 \times 9$ bits, or 0.87% of the register file) and the scratchpad mapping table takes 672 B ($16 \times 48 \times 7$ bits, or 1.3% of the scratchpad).

4.5.2. Virtualizing Thread Slots. Each SM is provisioned with a fixed number of *warp slots*, which determine the number of warps that are considered for execution every cycle by the warp scheduler. In order to oversubscribe warp slots, we need to save the state of each warp in memory before remapping the physical slot to another warp. This state includes the bookkeeping required for execution, i.e., the warp's PC (program counter) and the SIMT stack, which holds divergence information for each executing warp. The thread slot mapping table records whether each warp is mapped to a physical slot or swap space. The table is indexed by the logical warp ID, and stores the address of the physical warp slot that contains the warp. In our baseline design with 64 logical warps, this mapping table takes 56 B (64×7 bits).

[10] Our implementation of a virtualized resource aims to minimize complexity. This implementation is largely orthogonal to the framework itself, and one can envision other implementations (e.g., [30, 84, 85]) for different resources.

[11] We track registers at the granularity of a warp.

4.6. Supporting Phases and Phase Specifiers

Identifying phases. The compiler partitions each application into phases based on the liveness of registers and scratchpad memory. To avoid changing phases too often, the compiler uses thresholds to determine phase boundaries. In our evaluation, we define a new phase boundary when there is *(i)* a 25% change in the number of live registers or live scratchpad content, and *(ii)* a minimum of 10 instructions since the last phase boundary. To simplify hardware design, the compiler draws phase boundaries only where there is no control divergence.[12]

Once the compiler partitions the application into phases, it inserts instructions—*phase specifiers*—to specify the beginning of each new phase and convey information to the framework on the number of registers and scratchpad memory required for each phase. As described in Section 3.2.1, a barrier or a fence instruction also implies a phase change, but the compiler does not insert a phase specifier for it as the resource requirement does not change.

Phase Specifiers. The phase specifier instruction contains fields to specify *(i)* the number of live registers and *(ii)* the amount of scratchpad memory in bytes, both for the next phase. The instruction decoder sends this information to the coordinator along with the phase change event. The coordinator keeps this information in the corresponding warp slot.

4.7. Role of the Compiler and Programmer

The compiler plays an important role, annotating the code with phase specifiers to convey information to the coordinator regarding the resource requirements of each phase. The compiler, however, does *not* alter the size of each thread block or the scratchpad memory usage of the program. The resource specification provided by the programmer (either manually or via auto-tuners) is retained to guarantee correctness. For registers, the compiler follows the default policy or uses directives as specified by the user. One could envision more powerful, efficient resource allocation with a programming model that does *not* require *any* resource specification and/or compiler policies/auto-tuners that are *cognizant* of the virtualized resources. We leave this exploration for future work.

5. Methodology

5.1. System Modeling and Configuration

We model the Zorua framework with GPGPU-Sim 3.2.2 [5]. Table 2 summarizes the major parameters. Except for the portability results, all results are obtained using the Fermi configuration. We use GPUWattch [42] to model the GPU power consumption. We faithfully model the overheads of the Zorua framework, including an additional 2-cycle penalty for accessing each mapping table, and the overhead of memory accesses for swap space accesses (modeled as a part of the memory system). We model the energy overhead of mapping table accesses as SRAM accesses in GPUWattch.

[12] The phase boundaries for the applications in our pool easily fit this restriction, but the framework can be extended to support control divergence if needed.

System Overview	15 SMs, 32 threads/warp, 6 memory channels
Shader Core Config	1.4 GHz, GTO scheduler [58], 2 schedulers per SM
Warps/SM	Fermi: 48; Kepler/Maxwell: 64
Registers	Fermi: 32768; Kepler/Maxwell: 65536
Scratchpad	Fermi/Kepler: 48KB; Maxwell: 64KB
On-chip Cache	L1: 32KB, 4 ways; L2: 768KB, 16 ways
Interconnect	1 crossbar/direction (15 SMs, 6 MCs), 1.4 GHz
Memory Model	177.4 GB/s BW, 6 memory controllers (MCs), FR-FCFS scheduling, 16 banks/MC

Table 2: Major parameters of the simulated systems

5.2. Evaluated Applications and Metrics

We evaluate a number of applications from the Lonestar suite [8], GPGPU-Sim benchmarks [5], and CUDA SDK [52], whose resource specifications (the number of registers, the amount of scratchpad memory, and/or the number of threads per thread block) are parameterizable. Table 3 shows the applications and the evaluated parameter ranges. For each application, we make sure the amount of work done is the same for all specifications. The performance metric we use is the execution time of the GPU kernels in the evaluated applications.

Name (Abbreviation)	(R: Register, S: Scratchpad, T: Thread block) Range
Barnes-Hut (BH) [8]	R:28-44 × T:128-1024
Discrete Cosine Transform (DCT) [52]	R:20-40 × T: 64-512
Minimum Spanning Tree (MST) [8]	R:28-44 × T: 256-1024
Reduction (RD) [52]	R:16-24 × T:64-1024
N-Queens Solver (NQU) [11] [5]	S:10496-47232 (T:64-288)
Scan Large Array (SLA) [52]	R:24-36 × T:128-1024
Scalar Product (SP) [52]	S:2048-8192 × T:128-512
Single-Source Shortest Path (SSSP) [8]	R:16-36 × T:256-1024

Table 3: Summary of applications

6. Evaluation

We evaluate the effectiveness of Zorua by studying three different mechanisms: (i) *Baseline*, the baseline GPU that schedules kernels and manages resources at the thread block level; (ii) *WLM* (Warp Level Management), a state-of-the-art mechanism for GPUs to schedule kernels and manage registers at the warp level [77]; and (iii) *Zorua*. For our evaluations, we run each application on 8–65 (36 on average) different resource specifications[13] (the ranges are in Table 3).

6.1. Effect on Performance Variation and Cliffs

We first examine how Zorua alleviates the high variation in performance by reducing the impact of resource specifications on resource utilization. Figure 9 presents a Tukey box plot [48] (see Section 2 for a description of the presented box plot), illustrating the performance distribution (higher is better) for each application (for all different application resource specifications we evaluated), normalized to the slowest Baseline operating point *for that application*. We make two major observations.

First, we find that Zorua significantly reduces the *performance range* across all evaluated resource specifications. Averaged across all of our applications, the worst resource specification for Baseline achieves 96.6% lower performance than the best performing resource specification. For WLM [77], this performance range reduces only slightly, to 88.3%. With Zorua, the performance range drops significantly, to 48.2%. We see drops in the performance range for *all* applications except *SSSP*. With *SSSP*, the range is already small to begin with (23.8% in Baseline), and Zorua exploits the dynamic underutilization, which improves performance but also adds a small amount of variation.

[13] Our technical report [71] provides the specifications.

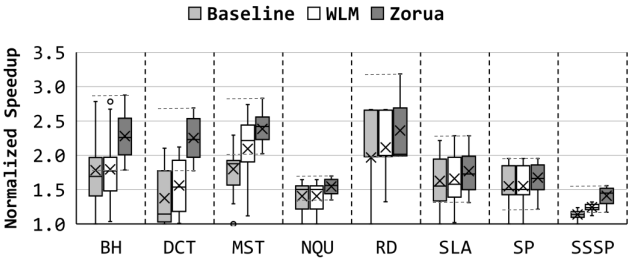
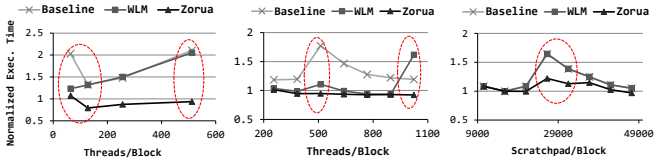


Figure 9: Normalized performance distribution

Second, while Zorua reduces the performance range, it also preserves or improves performance of the best performing points. As we examine in more detail in Section 6.2, the reduction in performance range occurs as a result of improved performance mainly at the lower end of the distribution.

To gain insight into how Zorua reduces the performance range and improves performance for the worst performing points, we analyze how it reduces performance cliffs. With Zorua, we ideally want to *eliminate* the cliffs we observed in Section 2.1. We study the tradeoff between resource specification and execution time for three representative applications: *DCT* (Figure 10a), *MST* (Figure 10b), and *NQU* (Figure 10c). For all three figures, we normalize execution time to the *best* execution time under Baseline. Two observations are in order.



(a) *DCT* (b) *MST* (c) *NQU*

Figure 10: Effect on performance cliffs

First, Zorua successfully mitigates the performance cliffs that occur in Baseline. For example, *DCT* and *MST* are both sensitive to the thread block size, as shown in Figures 10a and 10b, respectively. We have circled the locations at which cliffs exist in Baseline. Unlike Baseline, Zorua maintains more steady execution times across the number of threads per block, employing oversubscription to overcome the loss in parallelism due to insufficient on-chip resources. We see similar results across all of our applications.

Second, we observe that while WLM [77] can reduce some of the cliffs by mitigating the impact of large block sizes, many cliffs still exist under WLM (e.g., *NQU* in Figure 10c). This cliff in *NQU* occurs as a result of insufficient scratchpad memory, which cannot be handled by warp-level management. Similarly, the cliffs for *MST* (Figure 10b) also persist with WLM because *MST* has a lot of barrier operations, and the additional warps scheduled by WLM ultimately stall, waiting for other warps within the same block to acquire resources. We find that, with oversubscription, Zorua is able to smooth out those cliffs that WLM is unable to eliminate.

Overall, we conclude that Zorua *(i)* reduces the performance variation across resource specification points, so that performance depends less on the specification provided by the programmer; and *(ii)* can alleviate the performance cliffs experienced by GPU applications.

6.2. Effect on Performance

As Figure 9 shows, Zorua either retains or improves the best performing point for each application, compared to the Baseline. Zorua improves the best performing point for each application by 12.8% on average, and by as much as 27.8% (for *DCT*). This improvement comes from the improved parallelism obtained by exploiting the dynamic underutilization of resources, which exists *even for optimized specifications*. Applications such as *SP* and *SLA* have little dynamic underutilization, and hence do not show any performance improvement. *NQU does* have significant dynamic underutilization, but Zorua does not improve the best performing point as the overhead of oversubscription outweighs the benefit, and Zorua dynamically chooses not to oversubscribe. We conclude that even for many specifications that are optimized to fit the hardware resources, Zorua is able to further improve performance.

We also note that, in addition to reducing performance variation and improving performance for optimized points, Zorua improves performance by 25.2% on average for all resource specifications across all evaluated applications.

6.3. Effect on Portability

As we describe in Section 2.2, performance cliffs often behave differently across different GPU architectures, and can significantly shift the best performing resource specification point. We study how Zorua can ease the burden of performance tuning if an application has been already tuned for one GPU model, and is later ported to another GPU. To understand this, we define a new metric, *porting performance loss*, that quantifies the performance impact of porting an application without re-tuning it. To calculate this, we first normalize the execution time of each specification point to the execution time of the best performing specification point. We then pick a source GPU architecture (i.e., the architecture that the GPU was tuned for) and a target GPU architecture (i.e., the architecture that the code will run on), and find the point-to-point drop in performance for all points whose performance on the source GPU comes within 5% of the performance at the best performing specification point.[14]

Figure 11 shows the *maximum* porting performance loss for each application, across any two pairings of our three simulated GPU architectures (Fermi, Kepler, and Maxwell). We find that Zorua greatly reduces the maximum porting performance loss that occurs under both Baseline and WLM for all but one of our applications. On average, the maximum porting performance loss is 52.7% for Baseline, 51.0% for WLM, and only 23.9% for Zorua.

Notably, Zorua delivers significant improvements in portability for applications that previously suffered greatly when ported to another GPU, such as *DCT* and *MST*. For both of these applications, the performance variation differs so much

[14] We include any point within 5% of the best performance as there are often multiple points close to the best point, and the programmer may choose any of them.

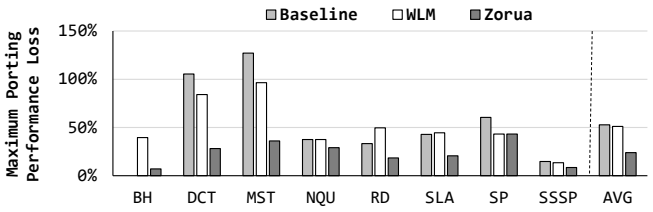


Figure 11: Maximum porting performance loss

between GPU architectures that, despite tuning the application on the source GPU to be within 5% of the best achievable performance, their performance on the target GPU is often more than twice as slow as the best achievable performance on the target platform. Zorua significantly lowers this porting performance loss down to 28.1% for *DCT* and 36.1% for *MST*. We also observe that for *BH*, Zorua actually increases the porting performance loss slightly with respect to the Baseline. This is because for Baseline, there are only two points that perform within the 5% margin for our metric, whereas with Zorua, we have five points that fall in that range. Despite this, the increase in porting performance loss for *BH* is low, deviating only 7.0% from the best performance.

We conclude that Zorua enhances portability of applications by reducing the impact of a change in the hardware resources for a given resource specification. For applications that have already been tuned on one platform, Zorua significantly lowers the penalty of not re-tuning for another platform, allowing programmers to save development time.

6.4. A Deeper Look: Benefits & Overheads

To take a deeper look into how Zorua is able to provide the above benefits, in Figure 12, we show the number of *schedulable warps* (i.e., warps that are available to be scheduled by the warp scheduler at any given time excluding warps waiting at a barrier), averaged across all of specification points. On average, Zorua increases the number of schedulable warps by 32.8%, significantly more than WLM (8.1%), which is constrained by the fixed amount of available resources. We conclude that by oversubscribing and dynamically managing resources, Zorua is able to improve thread-level parallelism, and hence performance.

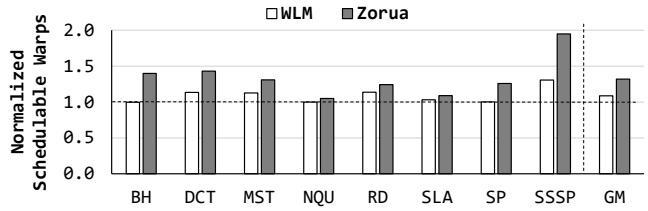


Figure 12: Effect on schedulable warps

We also find that the overheads due to resource swapping and contention do not significantly impact the performance of Zorua. The oversubscription mechanism (directed by the coordinator) is able to keep resource hit rates very high, with an average hit rate of 98.9% for the register file and 99.6% for scratchpad memory (not shown).

Figure 13 shows the average reduction in total system energy consumption of WLM and Zorua over Baseline for each application (averaged across the individual energy consumption over Baseline for each evaluated specification point). We observe that Zorua reduces the total energy consumption across all of

our applications, except for *NQU* (which has a small increase of 3%). Overall, Zorua provides a mean energy reduction of 7.6%, up to 20.5% for *DCT*.[15] We conclude that Zorua is an energy-efficient virtualization framework for GPUs.

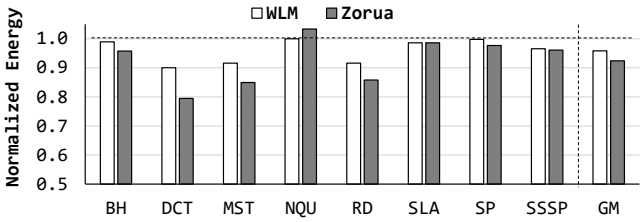


Figure 13: Effect on energy consumption

We estimate the die area overhead of Zorua with CACTI 6.5 [76], using the same 40nm process node as the GTX 480 , which our system closely models. We include all the overheads from the coordinator and the resource mapping tables (Section 4). The total area overhead is 0.735 mm^2 for all 15 SMs, which is only 0.134% of the die area of the GTX 480.

7. Other Applications

By providing the illusion of more resources than physically available, Zorua provides the opportunity to help address other important challenges in GPU computing today. We discuss several such opportunities in this section.

7.1. Multi-Programmed Environments

Executing multiple kernels or applications within the same SM can improve resource utilization and efficiency [24, 33, 55, 75, 87]. Hence, providing support to enable fine-grained sharing and partitioning of resources is critical for future GPU systems. Zorua provides several key benefits for enabling better performance and efficiency in multi-kernel/multi-program environments. First, selecting the optimal resource specification for an application is challenging in virtualized environments (e.g., clouds), as it is unclear which other applications may be running alongside it. Zorua can improve efficiency in resource utilization *irrespective* of the application specifications and of other kernels that may be executing on the same SM. Second, Zorua manages the different resources independently and at a fine granularity, using a dynamic runtime system (the coordinator). This enables the maximization of resource utilization, while providing the ability to control the partitioning of resources at runtime to provide QoS, fairness, etc., by leveraging the coordinator. Third, Zorua enables oversubscription of the different resources. This obviates the need to alter the application specifications [55, 87] in order to ensure there are sufficient resources to co-schedule kernels on the same SM, and hence enables concurrent kernel execution transparently to the programmer.

7.2. Preemptive Multitasking

A key challenge in enabling true multiprogramming in GPUs is enabling rapid preemption of kernels [56, 67, 75]. Context switching on GPUs incurs a very high latency and overhead, as a result of the large amount of register file and scratchpad state that needs to be saved before a new kernel can be executed. Saving state at a very coarse granularity (e.g., the entire SM state) leads to very high preemption latencies. Prior work proposes context minimization [49, 56] or context switching at the granularity of a thread block [75] to improve response time during preemption. Zorua enables fine-grained management and oversubscription of on-chip resources. It can be naturally extended to enable quick preemption of a task via intelligent management of the swap space and the mapping tables (complementary to approaches taken by prior work [49, 56]).

[15] We note that the energy consumption can be reduced further by appropriately optimizing the oversubscription algorithm. We leave this exploration to future work.

7.3. Other Uses

The ability to allocate/deallocate resources on demand during runtime makes Zorua a useful substrate to facilitate support for *(i)* system-level tasks, e.g., interrupts and exceptions (such as page faults); *(ii)* efficient dynamic parallelism [53, 74], which enables nested launching of kernels/thread blocks; and *(iii)* supplying resources for helper threads/background tasks [70]. The indirection offered by Zorua, along with the dynamic management of resources, could also enable better reliability (via simpler remapping of resources) and smaller, less area-intensive, and hence more scalable amounts of on-chip resources [1, 20, 30]. We discuss these use cases in more detail in our technical report [71].

8. Related Work

To our knowledge, this is the first work to propose a holistic framework to decouple a GPU application's resource specification from its physical on-chip resource allocation by virtualizing multiple on-chip resources. This enables the illusion of more resources than what physically exists to the programmer, while the hardware resources are managed at runtime by employing a swap space (in main memory), transparently to the programmer. We design a new hardware/software cooperative framework to effectively virtualize multiple on-chip GPU resources in a controlled and coordinated manner, thus enabling many benefits of virtualization in GPUs.

We briefly discuss prior work related to different aspects of our proposal: *(i)* virtualization of resources, *(ii)* improving programming ease and portability, and *(iii)* more efficient management of on-chip resources.

Virtualization of Resources. *Virtualization* [13, 15, 25, 29] is a concept designed to provide the illusion, to the software and programmer, of more resources than what truly exists in physical hardware. It has been applied to the management of hardware resources in many different contexts [2, 6, 13, 15, 25, 29, 54, 73], with virtual memory [15, 29] being one of the oldest forms of virtualization that is commonly used in high-performance processors today. Abstraction of hardware resources and use of a level of indirection in their management leads to many benefits, including improved utilization, programmability, portability, isolation, protection, sharing, and oversubscription.

In this work, we apply the general principle of virtualization to the management of multiple on-chip resources in modern GPUs. Virtualization of on-chip resources offers the opportunity to alleviate many different challenges in modern GPUs. However, in this context, effectively adding a level of indirection introduces new challenges, necessitating the design of a new virtualization strategy. There are two key challenges. First, we need to dynamically determine the *extent* of the virtualization to reach an effective tradeoff between improved

parallelism due to oversubscription and the latency/capacity overheads of swap space usage. Second, we need to coordinate the virtualization of *multiple* latency-critical on-chip resources. To our knowledge, this is the first work to propose a holistic software-hardware cooperative approach to virtualizing multiple on-chip resources in a controlled and coordinated manner that addresses these challenges, enabling the different benefits provided by virtualization in modern GPUs.

Prior works propose to virtualize a specific on-chip resource for specific benefits, mostly in the CPU context. For example, in CPUs, the concept of virtualized registers was first used in the IBM 360 [2] and DEC PDP-10 [6] architectures to allow logical registers to be mapped to either fast yet expensive physical registers, or slow and cheap memory. More recent works [54, 80, 81], propose to virtualize registers to increase the effective register file size to much larger register counts. This increases the number of thread contexts that can be supported in a multi-threaded processor [54], or reduces register spills and fills [80, 81]. Other works propose to virtualize on-chip resources in CPUs (e.g., [7, 12, 18, 23, 86]). In GPUs, Jeon et al. [30] propose to virtualize the register file by dynamically allocating and deallocating physical registers to enable more parallelism with smaller, more power-efficient physical register files. Concurrent to this work, Yoon et al. [85] propose an approach to virtualize thread slots to increase thread-level parallelism. These works propose specific virtualization mechanisms for a single resource for specific benefits. None of these works provide a cohesive virtualization mechanism for *multiple* on-chip GPU resources in a controlled and coordinated manner, which forms a key contribution of this work.

Enhancing Programming Ease and Portability. There is a large body of work that aims to improve programmability and portability of modern GPU applications using software tools, such as auto-tuners [14, 17, 38, 62, 63, 64], optimizing compilers [10, 28, 34, 46, 82, 83], and high-level programming languages and runtimes [16, 26, 57, 69]. These tools tackle a multitude of optimization challenges, and have been demonstrated to be very effective in generating high-performance portable code. They can also be used to tune the resource specification. However, there are several shortcomings in these approaches. First, these tools often require profiling runs [10, 14, 63, 64, 82, 83] on the GPU to determine the best performing resource specifications. These runs have to be repeated for each new input set and GPU generation. Second, software-based approaches still require significant programmer effort to write code in a manner that can be exploited by software to optimize the resource specifications. Third, selecting the best performing resource specifications statically using software tools is a challenging task in virtualized environments (e.g., clouds), where it is unclear which kernels may be run together on the same SM or where it is not known, apriori, which GPU generation the application may execute on. Finally, software tools assume a fixed amount of available resources. This leads to runtime underutilization due to static allocation of resources, which cannot be addressed by these tools.

In contrast, the programmability and portability benefits provided by Zorua require no programmer effort in optimizing resource specifications. Furthermore, these tools can be used in conjunction with Zorua to further improve performance.

Efficient Resource Management. Prior works aim to improve parallelism by increasing resource utilization using hardware-based [3, 4, 22, 30, 31, 32, 41, 50, 68, 77, 84] and software-based [24, 27, 39, 43, 55, 79, 84] approaches. Among these works, the closest to ours are [30, 85] (discussed earlier), [84] and [77]. These approaches propose efficient techniques to dynamically manage a single resource, and can be used along with Zorua to improve resource efficiency further. Yang et al. [84] aim to maximize utilization of the scratchpad with software techniques, and by dynamically allocating/deallocating scratchpad. Xiang et al. [77] propose to improve resource utilization by scheduling threads at the finer granularity of a warp rather than a thread block. This approach can help alleviate performance cliffs, but not in the presence of synchronization or scratchpad memory, nor does it address the dynamic underutilization within a thread during runtime. We quantitatively compare to this approach in Section 6 and demonstrate Zorua's benefits over it.

Other works leverage resource underutilization to improve energy efficiency [1, 19, 20, 21, 30] or perform other useful work [40, 70]. These works are complementary to Zorua.

9. Conclusion

We propose Zorua, a new framework that decouples the application resource specification from the allocation in the physical hardware resources (i.e., registers, scratchpad memory, and thread slots) in GPUs. Zorua encompasses a holistic virtualization strategy to effectively virtualize multiple latency-critical on-chip resources in a controlled and coordinated manner. We demonstrate that by providing the illusion of more resources than physically available, via dynamic management of resources and the judicious use of a swap space in main memory, Zorua enhances *(i) programming ease* (by reducing the performance penalty of suboptimal resource specification), *(ii) portability* (by reducing the impact of different hardware configurations), and *(iii) performance* for code with an optimized resource specification (by leveraging dynamic underutilization of resources). We conclude that Zorua is an effective, holistic virtualization framework for GPUs. We believe that the indirection provided by Zorua's virtualization mechanism makes it a generic framework that can address other challenges in modern GPUs. For example, Zorua can enable fine-grained resource sharing and partitioning among multiple kernels/applications, as well as low-latency preemption of GPU programs. We hope that future work explores these promising directions, building on the insights and the framework developed in this paper.

Acknowledgments

We thank the reviewers and our shepherd for their valuable suggestions. We thank the members of the SAFARI group for their feedback and the stimulating research environment they provide. Special thanks to Vivek Seshadri, Kathryn McKinley, Steve Keckler, Evgeny Bolotin, and Mike O'Connor for their feedback during various stages of this project. We acknowledge the support of our industrial partners: Facebook, Google, IBM, Intel, Microsoft, Nvidia, Qualcomm, Samsung, and VMware. This research was partially supported by NSF (grant 1409723), the Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation.

References

[1] M. Abdel-Majeed et al. Warped Register File: A Power Efficient Register File for GPGPUs. HPCA, 2013.
[2] G. M. Amdahl et al. Architecture of the IBM System/360. *IBM JRD*, 1964.
[3] R. Ausavarangnirun et al. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. *PACT*, 2015.
[4] R. Ausavarungnirun et al. Staged Memory Scheduling: Achieving High Prformance and Scalability in Heterogeneous Systems. In *ISCA*, 2012.
[5] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.
[6] C. G. Bell et al. The Evolution of the DEC System 10. *CACM*, 1978.
[7] E. Brekelbaum et al. Hierarchical scheduling windows. *MICRO*, 2002.
[8] M. Burtscher et al. A quantitative study of irregular programs on GPUs. In *IISWC*, 2012.
[9] S. Che et al. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.
[10] G. Chen et al. PORPLE: An extensible optimizer for portable data placement on GPU. In *MICRO*, 2014.
[11] P. Chen. N-Queens solver. `http://forums.nvidia.com/index.php?showtopic=76893`, 2008.
[12] H. Cook et al. Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments. *Tech. Rep. UCB/EECS-2009-131*, 2009.
[13] R. J. Creasy. The Origin of the VM/370 Time-sharing System. *IBM JRD*, 1981.
[14] A. Davidson et al. Toward Techniques for Auto-Tuning GPU Algorithms. In *Applied Parallel and Scientific Computing*. Springer, 2010.
[15] P. J. Denning. Virtual memory. *ACM Comput. Surv.*, 1970.
[16] R. Dolbeau et al. HMPP: A hybrid multi-core parallel programming environment. In *GPGPU*, 2007.
[17] Y. Dotsenko et al. Auto-tuning of Fast Fourier Transform on Graphics Processors. *PPoPP*, 2011.
[18] M. Erez et al. Spills, fills, and kills - an architecture for reducing register-memory traffic. Technical report, Stanford University, July 2000.
[19] M. Gebhart et al. A Compile-time Managed Multi-level Register File Hierarchy. In *MICRO*, 2011.
[20] M. Gebhart et al. Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors. *ISCA*, 2011.
[21] M. Gebhart et al. A hierarchical thread scheduler and register file for energy-efficient throughput processors. *TOCS*, 2012.
[22] M. Gebhart et al. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. In *MICRO*, 2012.
[23] A. Gonzalez et al. Virtual-physical registers. In *HPCA*, 1998.
[24] C. Gregg et al. Fine-grained resource sharing for concurrent GPGPU kernels. In *HotPar*, 2012.
[25] P. H. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM JRD*, 1983.
[26] T. D. Han et al. hiCUDA: High-Level GPGPU Programming. *TPDS*, 2011.
[27] A. B. Hayes et al. Unified On-chip Memory Allocation for SIMT Architecture. ICS, 2014.
[28] A. H. Hormati et al. Sponge: Portable Stream Programming on Graphics Engines. *ASPLOS*, 2011.
[29] B. Jacob et al. Virtual memory in contemporary microprocessors. *IEEE Micro*, 1998.
[30] H. Jeon et al. GPU register file virtualization. In *MICRO*, 2015.
[31] A. Jog et al. Orchestrated Scheduling and Prefetching for GPGPUs. In *ISCA*, 2013.
[32] A. Jog et al. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS*, 2013.
[33] A. Jog et al. Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications. In *GPGPU*, 2014.
[34] J. C. Juega et al. Adaptive Mapping and Parameter Selection Scheme to Improve Automatic Code Generation for GPUs. CGO, 2014.
[35] O. Kayiran et al. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT*, 2013.
[36] O. Kayiran et al. Managing GPU Concurrency in Heterogeneous Architectures. MICRO, 2014.
[37] O. Kayiran et al. μC-States: Fine-grained GPU Datapath Power Management. PACT, 2016.
[38] M. Khan et al. A Script-based Autotuning Compiler System to Generate High-performance CUDA Code. *TAC0*, 2013.
[39] R. Komuravelli et al. Stash: Have your scratchpad and cache it too. In *ISCA*, 2015.
[40] N. B. Lakshminarayana et al. Spare register aware prefetching for graph algorithms on GPUs. In *HPCA*, 2014.
[41] M. Lee et al. Improving GPGPU resource utilization through alternative thread block scheduling. In *HPCA*, 2014.
[42] J. Leng et al. GPUWattch: Enabling Energy Optimizations in GPGPUs. *ISCA*, 2013.
[43] C. Li et al. Automatic data placement into GPU on-chip memory resources. In *CGO*, 2015.
[44] C. Li et al. Locality-driven dynamic GPU cache bypassing. In *ICS*, 2015.
[45] D. Li et al. Priority-based cache allocation in throughput processors. In *HPCA*, 2015.
[46] Y. Liu et al. A cross-input adaptive framework for GPU program optimizations. In *IPDPS*, 2009.
[47] J. Matela et al. Low GPU occupancy approach to fast arithmetic coding in JPEG2000. In *Math. and Eng. Methods in Computer Science*. 2011.
[48] R. McGill et al. Variations of box plots. *The American Statistician*, 1978.
[49] J. Menon et al. iGPU: Exception Support and Speculative Execution on GPUs. *ISCA*, 2012.
[50] V. Narasiman et al. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *MICRO*, 2011.
[51] Nintendo/Creatures Inc./GAME FREAK inc. Pokémon. `http://www.pokemon.com/us/`.
[52] NVIDIA. CUDA C/C++ SDK Code Samples, 2011.
[53] NVIDIA. CUDA Dynamic Parallelism Programming Guide. 2014.
[54] D. W. Oehmke et al. How to Fake 1000 Registers. MICRO, 2005.
[55] S. Pai et al. Improving GPGPU concurrency with elastic kernels. In *ASPLOS*, 2013.
[56] J. Park et al. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. *ASPLOS*, 2015.
[57] J. Ragan-Kelley et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *PLDI*, 2013.
[58] T. Rogers et al. Cache-Conscious Wavefront Scheduling. MICRO, 2012.
[59] S. Ryoo et al. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. PPoPP, 2008.
[60] S. Ryoo et al. Program optimization carving for GPU computing. *JPDC*, 2008.
[61] S. Ryoo et al. Program Optimization Space Pruning for a Multithreaded GPU. CGO, 2008.
[62] K. Sato et al. *Automatic Tuning of CUDA Execution Parameters for Stencil Processing*. 2010.
[63] C. A. Schaefer et al. Atune-IL: An instrumentation language for auto-tuning parallel applications. In *Euro-Par*. 2009.
[64] K. Spafford et al. Maestro: data orchestration and tuning for opencl devices. In *Euro-Par*. 2010.
[65] J. A. Stratton et al. Algorithm and data optimization techniques for scaling to massively threaded systems. *IEEE Computer*, 2012.
[66] J. A. Stratton et al. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *UIUC Technical Report IMPACT-12-01*, 2012.
[67] I. Tanasic et al. Enabling Preemptive Multiprogramming on GPUs. In *ISCA*, 2014.
[68] D. Tarjan et al. On demand register allocation and deallocation for a multithreaded processor, 2011. US Patent 20110161616.
[69] Sain-Zee Ueng et al. CUDA-Lite: Reducing GPU Programming Complexity. LCPC, 2008.
[70] N. Vijaykumar et al. A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps. In *ISCA*, 2015.
[71] N. Vijaykumar et al. A New Framework for GPU Resource Virtualization. *CMU SAFARI Technical Report No. 2016-005*, 2016.
[72] O. Villa et al. Scaling the Power Wall: A Path to Exascale. In *SC*, 2014.
[73] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *OSDI*, 2002.
[74] J. Wang et al. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. ISCA, 2015.
[75] Z. Wang et al. Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-Grained Sharing. In *HPCA*, 2016.
[76] S. Wilton et al. CACTI: An enhanced cache access and cycle time model. *JSSC*, 1996.
[77] P. Xiang et al. Warp-level divergence in GPUs: Characterization, impact, and mitigation. In *HPCA*, 2014.
[78] X. Xie et al. Coordinated static and dynamic cache bypassing for GPUs. In *HPCA*, 2015.
[79] X. Xie et al. Enabling coordinated register allocation and thread-level parallelism optimization for GPUs. In *MICRO*, 2015.
[80] J. Yan et al. Virtual Registers: Reducing Register Pressure Without Enlarging the Register File. *HIPEAC*, 2007.
[81] J. Yan et al. Exploiting Virtual Registers to Reduce Pressure on Real Registers. *TACO*, 2008.
[82] Y. Yang et al. A GPGPU Compiler for Memory Optimization and Parallelism Management. *PLDI*, 2010.
[83] Y. Yang et al. A Unified Optimizing Compiler Framework for Different GPGPU Architectures. *TACO*, 2012.
[84] Y. Yang et al. Shared memory multiplexing: a novel way to improve GPGPU throughput. In *PACT*, 2012.
[85] M. Yoon et al. Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit. ISCA, 2016.
[86] J. Zalamea et al. Two-level Hierarchical Register File Organization for VLIW Processors. MICRO, 2000.
[87] J. Zhong et al. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *TPDS*, 2014.