Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit

Myung Kuk Yoon*, Keunsoo Kim*, Sangpil Lee*, Won Woo Ro*, and Murali Annavaram†
* *School of Electrical and Electronic Engineering, Yonsei University*
{*myungkuk.yoon, keunsoo.kim, madfish, wro*}*@yonsei.ac.kr*
† *Ming Hsieh Department of Electrical Engineering, University of Southern California*
annavara@usc.edu

***Abstract*—Modern GPUs require tens of thousands of concurrent threads to fully utilize the massive amount of processing resources. However, thread concurrency in GPUs can be diminished either due to shortage of thread scheduling structures (scheduling limit), such as available program counters and single instruction multiple thread stacks, or due to shortage of on-chip memory (capacity limit), such as register file and shared memory. Our evaluations show that in practice concurrency in many general purpose applications running on GPUs is curtailed by the scheduling limit rather than the capacity limit. Maximizing the utilization of on-chip memory resources without unduly increasing the scheduling complexity is a key goal of this paper.**

This paper proposes a Virtual Thread (VT) architecture which assigns Cooperative Thread Arrays (CTAs) up to the capacity limit, while ignoring the scheduling limit. However, to reduce the logic complexity of managing more threads concurrently, we propose to place CTAs into active and inactive states, such that the number of active CTAs still respects the scheduling limit. When all the warps in an active CTA hit a long latency stall, the active CTA is context switched out and the next ready CTA takes its place. We exploit the fact that both active and inactive CTAs still fit within the capacity limit which obviates the need to save and restore large amounts of CTA state. Thus VT significantly reduces performance penalties of CTA swapping. By swapping between active and inactive states, VT can exploit higher degree of thread level parallelism without increasing logic complexity. Our simulation results show that VT improves performance by 23.9% on average.

***Keywords*-GPU; GPGPU; Warp Scheduling; Virtual Thread (VT); Capacity Limit; Scheduling Limit;**

I. Introduction

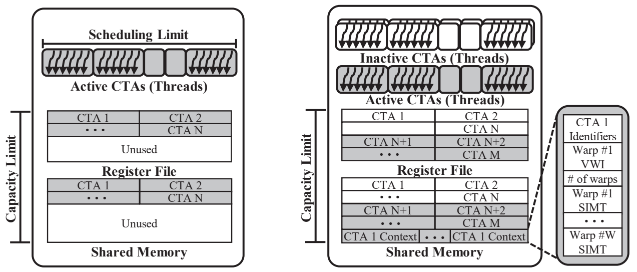
Modern Graphics Processing Units (GPUs) are now widely used for executing general purpose applications. These general purpose applications are ported to be GPU kernels to create massive Thread Level Parallelism (TLP), which is then exploited by GPUs to concurrently execute thousands of threads. Each GPU has dozens of Streaming Multiprocessor cores (SM), where each SM can execute dozens of threads concurrently using Single Instruction Multiple Thread (SIMT) execution model. GPU kernels are divided into large Cooperative Thread Arrays (CTAs) or thread blocks. CTAs, in turn, are divided into smaller groups of threads called *warps* or *wavefronts*. GPUs support fast context switching between warps to allow execution of other warps when one warp stalls, so as to hide instruction latencies [1], [2]. To support fast context switching, GPUs have a large register file to store the architectural context of multiple concurrent warps without the need to save and restore the context.

The size of each CTA is usually decided by the programmers [3], [4], however the number of CTAs that can be concurrently executed on an SM is determined by various hardware resource constraints: the number of issuable CTAs, the number of issuable threads, the size of the register file, and the size of shared memory [5], [6]. In this paper, we divide the constraints into two groups: *scheduling limit* and *capacity limit*. Scheduling limit constrains the maximum number of issuable CTAs and threads due to the limited thread concurrency that the hardware can support. On the other hand, capacity limit constrains the maximum number of CTAs due to limited register file and limited amount of shared memory for storing concurrent thread contexts. If the cumulative resource demand of all the assigned CTAs reaches any one of these limits, no further CTAs can be dispatched to SMs even though the other resources may still be available.

To understand the severity of both these limits, scheduling and capacity limits, we measured the number of CTAs assigned to an SM across a wide range of GPU applications (details in Section III). We observed that 18 applications out of 35 applications that we analyzed have at least one kernel that reaches the scheduling limit first when the applications are launched on the Fermi architecture [7], [8]. Among these 18 applications, 10 applications reach thread count limit, and the other 8 applications reach CTA count limit. In these benchmarks, on average, 49.8% of the register file, and 84.6% of the shared memory are unused on Fermi. The resource underutilization problem becomes even more acute in newer architecture such as Kepler [9]. Kepler increased the maximum thread count to 2,048 (compared to 1,536 in Fermi) and the maximum CTA count was increased to 16 (compared to 8 in Fermi). Kepler also increased the total number of registers to 64K $\times$ 32-bit registers, compared to 32K $\times$ 32-bit registers in Fermi [7], [8], [9]. Even though the scheduling limit was increased the register file grew larger. As a result, in the case of the Kepler architecture, 27 applications out of 35 applications reach the scheduling limit. Among these 27 applications, 22 applications cannot dispatch more CTAs onto SMs due to the thread count limit, and 5 applications cannot dispatch more CTAs due to the CTA count limit. In summary, the scheduling limit, especially the thread count limit, is a major TLP limiting factor on GPUs.

Figure 1(a) shows a conceptual view of resource underutilization due to the scheduling limit on GPUs. The thread counts and CTA counts that are allowed in each GPU were fully occupied. But the register file and shared memory structures were underutilized. In fact, the underutilization of register file has also been observed in many prior studies [10], [11], [12], [13], [14], [15]. Register file and shared memory are valuable resources that can be used to further improve parallelism but unfortunately scheduling limits prevent these resources from being fully utilized.

Prior research studies [11], [16], [17], [18], [19] have



(a) Baseline GPU (b) VT Architecture

Figure 1. Conceptual View of Resource Management on Baseline GPU and VT Architecture

shown increasing TLP is beneficial for many GPU applications. Our evaluations (shown in Section V) also confirm that additional threads can improve performance. Dispatching more CTAs can improve performance and resource utilization, however the scheduling limit cannot be scaled easily due to several reasons. First, it requires additional scheduling support such as larger instruction buffer, scoreboard, and SIMT stack. These structures cause area and power overheads on GPUs [20], [21], [22], [23], [24]. Second, implementing the warp scheduler for a large number of warps is a challenging design problem [25], [26]. Every cycle, the scheduler must access the scoreboard to verify ready warps and a subset of ready warps must be selected for issue. Therefore, scaling scheduling limit increases the scheduling complexity and may increase scheduling latencies.

As a cost-effective alternative, we propose a Virtual Thread (VT) architecture which relaxes the scheduling limit and forces the capacity limit to be the only concurrency limiter. To restrict the corresponding increase in scheduler complexity, we propose to keep the physical scheduling limit unchanged. The scheduler still handles the same number of CTAs it is provisioned to handle in the baseline architecture without VT. Figure 1(b) shows the conceptual view of resource management on VT. In VT, all the CTAs are placed either in active or inactive state. The scheduler only deals with active CTAs which are still confined by the baseline scheduler limit. When all the warps in an active CTA hit a long latency operation, such as waiting for a response from the global memory, the active CTA is switched to inactive state. After the state change, another ready CTA is swapped into the active state. As we will describe in detail shortly, an inactive CTA will not be part of any scheduling decisions and hence the scheduling complexity is not increased.

Given that the total CTAs (active and inactive) are still bounded by the capacity limit, there is no need to save and restore the entire architectural state while switching CTAs between active and inactive states. In particular, the large register file and shared memory state of the swapped CTAs [27] stay unperturbed, and only a small amount of per-CTA state data is saved when a CTA is moved from active to inactive state, and the small amount state is restored when an inactive CTA is switched back to active state.

While increasing TLP generally helps performance, increasing TLP unchecked, particularly in the presence of global memory stalls may increase memory contention [5], [28]. To overcome this problem, VT uses a memory request prioritization technique which is inspired by prior works [29], [30]. By exploiting the increased TLP and managing the memory contention, VT increases average performance by 23.9%. VT's performance is only 3% lower than a hardware-intensive traditional approach, called MAX in our evaluations, that increases the scheduling limit through more scheduling hardware.

The remainder of this paper is organized as follows. Section II presents the GPU programming model and the baseline GPU architecture. In Section III, we present motivational data for this work. In Section IV, we propose the VT architecture. The experimental results of VT are presented in Section V. In Section VI, we present related work. Finally, we conclude in Section VII.

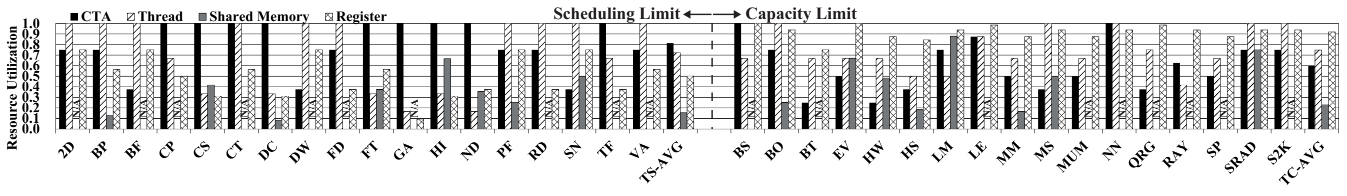
II. Background

For consistency, we use NVIDIA terminology to refer to various hardware and software components throughout this paper [7], [8], [9], [31]. Many general purpose GPU applications typically have multiple compute kernels which are the primary computational blocks that can be executed on GPUs. These kernels are comprised of tens of thousands of threads which are organized as CTAs [31]. Each CTA is further sub-divided into a collection of threads called a warp, which is the minimum scheduled unit of work in GPUs [31]. The threads in each CTA can synchronize via hardware barriers. GPUs are provisioned with a range of specialized memories. The largest memory is global memory which is accessible to all threads in a kernel and accessing the global memory space takes hundreds of clock cycles [26], [32]. Then there is an on-chip shared memory which is available for inter-thread communication within a CTA, but different CTAs cannot access each other's shared memory resources [33]. Lastly, each thread can access private local memory.

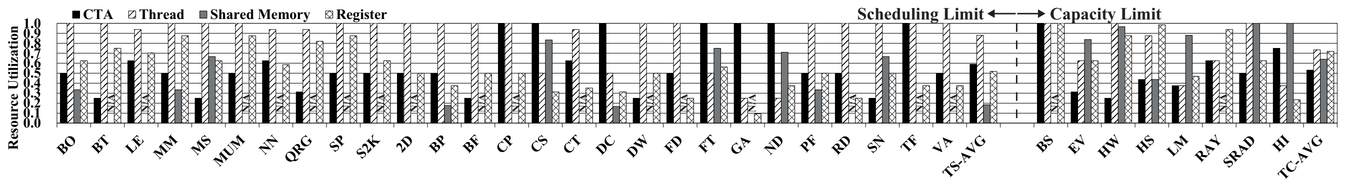
GPUs consist of thousands of processing units to execute a massive number of threads concurrently. For example, NVIDIA Kepler GPU includes 2,048 processing units in each Streaming Multiprocessor (SM or SMX) to execute up to 30,720 (15 SMXs $\times$ 2,048) threads concurrently [9]. The CTA scheduler dispatches CTAs to SMs, and the number of CTAs assigned to an SM is constrained by various factors: the number of issuable CTAs, the number of issuable threads, the size of the register file, and the size of the shared memory [5], [6]. We classify TLP limiting constraints into two categories: *scheduling limit* and *capacity limit*.

The scheduling limit is due to the logic complexity of managing a massive number of threads and CTAs. For instance, one scheduling limit that curtails the number of issuable warps is the number of program counters since GPUs have to provide a separate program counter per each concurrent warp. The capacity limit is due to the cumulative storage demand from all the assigned CTAs. For instance, the number of architected registers used in a warp may determine the total number of warps that can be assigned per SM since the cumulative register usage cannot exceed the available register file size. The amount of shared memory used by each warp may also limit the total number of warps assigned as the total memory usage across all warps must fit the available shared memory. As can be seen from these examples, TLP can be curtailed due to a number of competing design constraints.

Due to varying application demands, some resource limits may be reached first while there are plenty of other available resources. If the total resource usage of current in-flight CTAs reaches the limit of any one of various resource constraints, no more CTAs can be scheduled even though



(a) Resource Utilization on Fermi Architecture



(b) Resource Utilization on Kepler Architecture

Figure 2. Resource Utilization on Various GPU Architectures

all other resources are still available. For example, NVIDIA Kepler GPU has 64K × 32-bit register file and 48KB shared memory per SMX, categorized as capacity limit in this work. In addition, each SMX allows up to 2,048 threads or up to 16 CTAs (whichever limit reaches first) to be issued concurrently on each SMX [9], categorized as scheduling limit in this work. Let us assume that an example application which consists of hundreds of CTAs is launched onto the Kepler GPU. If each CTA contains 256 threads and each thread only needs 10 registers then the number of issuable CTAs is limited to 8 due to the the maximum thread count (scheduling limit), even though there are plenty of available registers. A well designed system should be able to maximize all the available resources to achieve the highest throughput.

While resource underutilization may appear to be a design imbalance that may be fixed by either reducing the resources or increasing the scheduling structure sizes, we should point out that GPU designs are tuned for highly parallel applications, such as graphics computing [34], [35], [36]. The use of GPUs for general purpose computing creates the imbalances [37], [38]. In spite of these concerns, GPUs are rapidly being deployed for general purpose computing. For general purpose computing with diverse application demands, it is much more challenging to tune the size of the structures at design time. Rather we believe that existing GPU designs can be enhanced with features such as VT to improve their execution efficiency even for general purpose computing.

III. Understanding GPU TLP Limits

To quantify resource utilization in GPUs, we characterize the resource demands of various applications in two different GPU configurations, which are similar to Fermi and Kepler architectures. Detailed simulation environment is presented on Section V. Figure 2 shows the fraction of available SM resources used by each application. We consider four different limitations: the maximum number of CTAs, the maximum number of threads, the total register file usage, and the total shared memory usage [5], [6]. The first two limits are scheduling limits and the last two are capacity limits. In the figure, we divide applications into two types. Type-C applications which are shown to the right of the dotted line in Figure 2(a) are constrained by the capacity limit. Most Type-C applications utilize almost the entire 32K × 32-bit register file in Fermi, except that *LM* is limited by the shared memory capacity. On average, 91.9% of register file and 22.8% of shared memory are occupied for the Type-C applications. Therefore, the Type-C applications require more on-chip memory capacity for storing thread contexts to improve TLP regardless of the other limiting factors. In contrast to the Type-C applications, the scheduling limit is the bottleneck in scaling TLP for the Type-S applications, which are shown to the left of the dotted line in the figure. For these applications the number of allowable threads or CTAs is the TLP limiting factor. As these applications do not fully utilize the on-chip memory, a large fraction of register file and shared memory are wasted. As shown in Figure 2(a), only 50.2% of register file and 15.4% of shared memory space are used in these applications.

The resource underutilization problem becomes even more magnified when the available resources are scaled in future GPUs. From Fermi to Kepler generation [8], [9], [31], per-SM CTA limit is increased from 8 to 16, thread limit is increased from 1,536 to 2,048, and the register file size is doubled. Figure 2(b) shows the resource utilization with the Kepler architecture for the same set of benchmarks. Compared to Fermi, 27 applications are now classified as Type-S; 22 of these applications reach the thread count limit and 5 applications reach the CTA count limit. Consequently, 48.2% of register file and 81.6% of shared memory space are underutilized in Kepler. From these results, we surmise that as resources are scaled in future for many general purpose applications running on GPUs the TLP limiting factor is the scheduling limit (especially thread count limit) rather than the capacity limit. Therefore, relaxing the scheduling limit constraint can extract more TLP and improve the utilization of the on-chip memory structures.

To investigate the extent of additional scheduling resources needed to reach the capacity limit, either register file capacity or shared memory capacity, we calculate the maximum number of threads that can be assigned in the two generations of GPU architecture: Fermi, and Kepler. Figure 3 shows the fraction of additional threads required to reach the capacity limit (either the register file or shared memory whichever saturates first) for the applications classified as Type-S in Figure 2. In case of Fermi, on average 38.2%

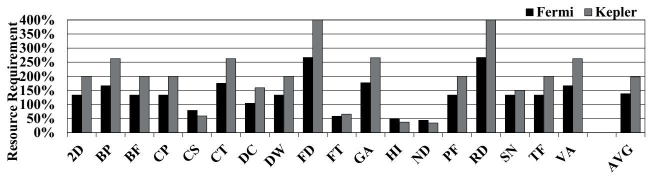


Figure 3. Thread Counts to Saturate Register File/Shared Memory for Type-S Applications

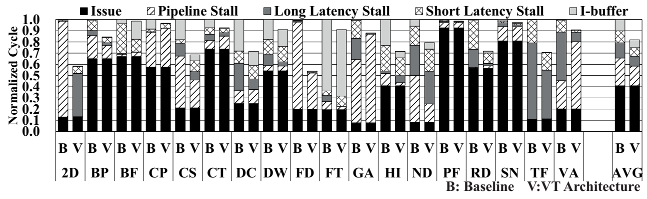


Figure 4. Cycle Distribution for Type-S Applications

more threads are required to reach the capacity limit. In Kepler, on average 97.7% more threads are needed to fully utilize on-chip storage, and 300.0% more threads are needed in the worst case.

However, implementing GPUs which can support 3× more threads is a non trivial challenge due to the need for providing additional hardware resources such as instruction buffers, scoreboards, and SIMT stack. Based on the previous studies, these hardware logic blocks increase power and area overheads [21], [22], [23], [24], [20]. In addition to the hardware overheads, implementing a warp scheduler for the large number of threads is also a difficult problem. In modern GPUs, the scheduler accesses the scoreboard to find the ready warps and selects the highest priority warp among the ready warps every cycle [25], [26]. Therefore, if the number of concurrently running threads is increased significantly, the scheduler complexity is increased and the searching time for an issuable warp can be increased.

A. Advantages of Higher TLP

GPUs exploit high TLP to hide processing latency of individual warps [1], [28]. If a warp is stalled by a long latency memory access or data dependency, then warp schedulers issue another ready warp from the warp pool so that execution of warps are interleaved. The effectiveness of stall hiding depends on the number of available warps in the warp pool, which is the key reason why GPUs require a large number of concurrent threads [5], [11], [19], [39].

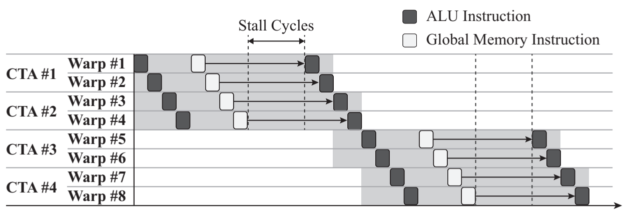
To quantify whether the existing level of TLP is already sufficient to hide long latency memory stalls, we classified the scheduling cycles into several categories. The first bar in each group in Figure 4 (B for the baseline architecture) shows the breakdown of scheduling cycles for the Type-S applications on the Fermi architecture (other bar in each group will be described in Section V). We divide the cycles into two different types: issue and stall. The bottom category shows the fraction of cycles where the scheduler is able to find a warp which is ready for issue. All the other categories are stall categories where the scheduler cannot issue an instruction. The warp schedulers cannot issue instructions during 59.1% of total execution cycles. Hence, more than half of the available issue bandwidth is unused.

To better understand the reasons for these stalls, we further divide the stalls into four different types: pipeline, long latency, short latency, and I-buffer stalls. Pipeline stall (the second component from the bottom), which takes 24.8% of total cycles, indicates that the schedulers have at least one ready warp, however these ready warps could not be issued since the functional units, such as load/store units, do not accept more instructions. In particular, we observed that the majority of pipeline stalls are caused by the memory subsystem saturation, which occurs when load/store units are not able to handle more memory requests [29]. Long latency stall accounts for 13.6% of missed scheduling opportunities, which occurs when the warps in the scheduler pool are all waiting for memory responses. Thus 38.4% of the total scheduling cycles are wasted due to memory related stalls (pipeline and long latency stalls). Recall that GPUs fundamentally rely on TLP to hide memory latency related stalls. Hence, 38.4% stalls could have been potentially avoided with increasing TLP. Finally, short latency stall takes 10.7% of total cycles, which happens when no more instructions are issued due to preceding non-memory dependent instructions. The other reasons including instruction fetch buffer stall (I-buffer), which indicates next instructions of all warps are not yet fetched, are only 10.0%.

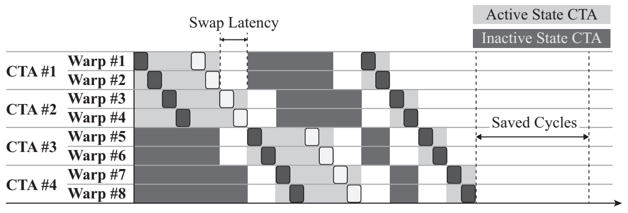
IV. Virtual Thread Architecture

As seen from Figure 4 improving TLP still has a significant potential for higher performance with better latency hiding. The data shown in Figure 2(b) shows that 48.2% of register file and 81.6% of shared memory space are underutilized in Kepler for a majority of benchmark applications. Thus the option we explore to increase TLP is to allow more threads into the GPU to fill up the available register file or shared memory. At the same time we do not increase the scheduling limit to deal with the increased thread count. Scaling the scheduling limit using more program counters, larger scheduling windows, and deeper fetch and scheduling queues is non-trivial due to power and area limitations as shown in prior works [22], [25], [26]. As a cost-effective alternative, we propose the Virtual Thread (VT) architecture. The proposed architecture takes advantage of the fact that on-chip memory capacity is not a limiting factor for most applications. With VT, CTAs are allocated on each SM up to the capacity limit regardless of the scheduling limit. Thus the number of CTAs allocated to each SM (and consequently the number of threads) will either cumulatively occupy the entire register file or the entire shared memory, whichever limit reaches first. Based on the data in Figure 2, the register file limit is reached earlier than the shared memory for the majority of the applications.

To keep the scheduling limit unchanged, the VT architecture maintains only a fraction of CTAs in active state which will be managed within the current scheduling limit, and the other CTAs are left in inactive state. When all the warps from an active CTA stall due to a long latency memory operation, VT swaps out the stalled CTA into the inactive state and brings in another ready CTA. We exploit the fact that even though the swapped out CTA's architected state is in the register file and shared memory, the incoming CTA does not need to use any of those registers or shared memory. In fact all the allocated CTAs' architected state fit within the capacity limit, irrespective of whether a CTA is active or inactive. Thus rather than naively saving and restoring CTA's register or memory state on a context switch, each CTA needs to save a much smaller per-CTA state information. As we will describe shortly, the per-CTA state information



(a) Execution Order on Baseline GPU



(b) Execution Order using VT Architecture

Figure 5. Execution Order with and without VT

is composed of warp identifiers, CTA identifiers, and SIMT stack including the program counter.

Given the limited amount of per-CTA state information that needs to be saved and restored, it is feasible to save the per-CTA state information in the shared memory itself, which is often even more underutilized than the register file, as can be seen from Figure 2. Shared memory access latency is significantly shorter than global memory [16], [40], [41], and therefore CTA swapping latency is negligible compared to the global memory access latency stalls that are suffered by a stalled CTA. This dynamic context switching of CTAs increases TLP allowing VT to improve performance. If there is not enough shared memory to store the context information of all the CTAs that can be assigned with VT, we can scale back VT to support fewer CTAs.

Figure 5 illustrates an example execution with and without VT. In this example, we assume for illustrative purposes that each SM can issue two CTAs concurrently due to the scheduling limit, however the SM has unused registers and shared memory which can support two additional CTAs. Figure 5(a) shows an execution order of conventional GPU without VT. After issuing a series of instructions from each warp in round-robin order, each warp hits a global memory access stall. Eventually both CTAs are stalled, but no new CTAs can be brought in. Only after a CTA completes execution, the SM releases CTA resources and then assigns new CTA into the SM.

With VT, we reduce the long latency stalls using more CTAs as illustrated in Figure 5(b). At the beginning of execution, all four CTAs are assigned to the SM ignoring the fact that only two CTAs can be actively scheduled. Each of four CTAs gets its own register file and shared memory allocation to improve resource utilization. At the start of execution to honor the existing scheduling limit, only two CTAs are placed in the active state and the scheduler only deals with active CTAs. After detecting Warp #1 and #2 from CTA #1 are all waiting for returning responses from the global memory, CTA #1 is swapped out and CTA #3 is swapped in. For illustration purposes, the swapping latency is entirely hidden by the execution cycles of CTA #2. When CTA #2 is stalled, it is swapped with CTA #4. The warp scheduler issues instructions from the newly activated CTAs (#3 and #4). Eventually warps from the newly activated CTA #3 and #4 issue global memory access instructions and then the VT architecture swaps out CTA #3 and #4 with CTA #1 and #2 since memory requests from CTA #1 and #2 are processed. To summarize, the long latency operations are more effectively hidden and Memory Level Parallelism (MLP) is increased due to additional CTAs.

A. Architectural Support for VT

In this section, we describe the microarchitectural support for VT. The number of microarchitectural changes are quite minimal and are highlighted in Figure 6. In order to manage CTA context swap, we add Virtual Thread Controller (VTC) logic which is shown in Figure 7. To provide the complete operational detail of VT architecture, we describe how each pipeline stage works in baseline and highlight if there are any changes that are needed for the VT architecture.

Fetch and Decode Stages: In our baseline, a Fermi-like architecture, the fetch stage of the pipeline selects one of the warp program counters and fetches an instruction from that warp into an instruction fetch buffer. The instruction fetch buffer is indexed using the warp ID to locate the buffer entries. The VT architecture enables many more CTAs (and hence more warps) to be assigned to the SM up to the capacity limit. Thus indexing the instruction fetch buffer with a larger warp ID is not feasible without unduly increasing the complexity of the fetch buffer design. Hence, whenever a CTA is swapped out (referred to as CTA_{old} for simplicity) and a new CTA (CTA_{new}) is brought in, we re-assign warp ids of CTA_{old} to CTA_{new}. We will use the term Physical Warp ID (PWI) to refer to the fact that PWI is used to physically index the instruction fetch buffers. Thus warps from CTA_{new} essentially use the same PWIs as CTA_{old}.

When a CTA swaps out, the PC of each warp in the the CTA_{old}, which is usually stored within the SIMT stack, is stored in a dedicated context storage area within the shared memory. The size and content of each CTA context will be described in detail shortly. After saving the CTA_{old} context, PCs of all the warps in the CTA_{new} are now restored into the PC array by reading the new context from the shared memory. Then the fetch buffers for CTA_{old} are flushed. After flushing the buffers, warps from CTA_{new} fetch instructions from their own PCs. Since these warps reuse the PWIs, they simply bring instructions into the same entries as warps from the CTA_{old}.

Issue Stage: Our baseline architecture uses a scoreboard to keep track of instruction dependencies. The scoreboard can be implemented in multiple ways. Without loss of generality in our baseline, the scoreboard is implemented as an array of bit-vectors, where each bit-vector in the array tracks the dependencies of one warp. Each bit-vector in the scoreboard array is thus accessed by the warp id. As soon as a warp instruction is issued, the destination register number of that warp is marked as busy in the corresponding bit-vector of that warp. Any younger instruction that needs a register marked as busy in the scoreboard is stalled from issuing. When the instruction completes its execution and writes back its results to the register file, it then resets the busy bit in the scoreboard. The scoreboard is accessed every cycle to decide which warp is ready for issue [25].

Rather than increasing the scoreboard size to accommodate more warps, VT reuses the scoreboard entries from

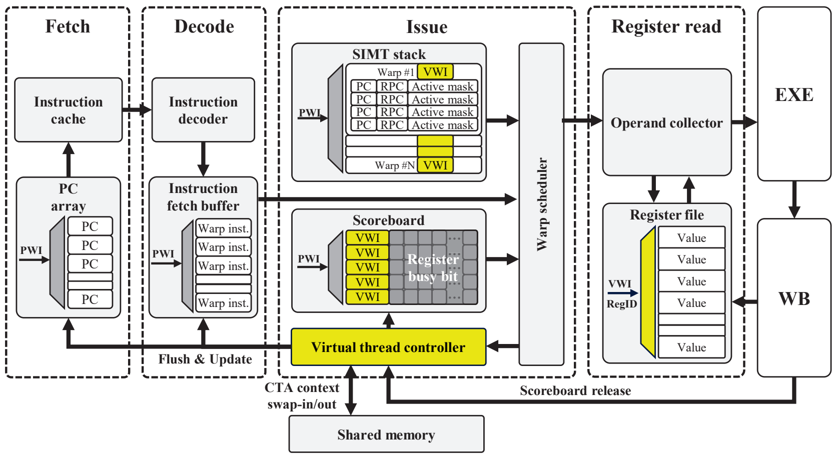


Figure 6. GPU Architecture with VT

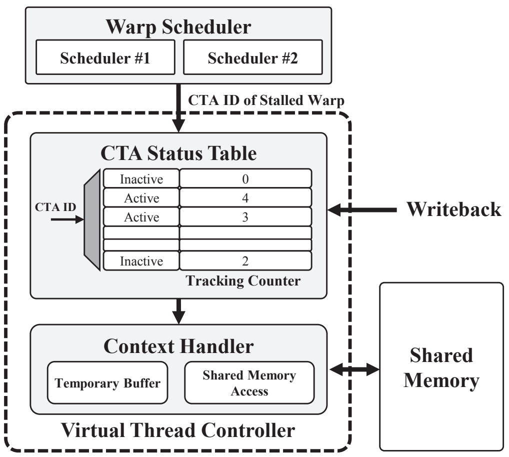


Figure 7. Virtual Thread Controller

CTA_{old} for CTA_{new}. After CTA_{old} is switched out, each bit-vector associated with the warps in CTA_{old} are all reset to zero. At that time, VT can reuse the scoreboard bit-vectors for warps from CTA_{new}. VT uses the PWI to index the scoreboard bit-vector. Since VT reuses the PWIs for CTA_{new}, the newly initiated warps reuse the the same bit-vectors.

Note that some operations from CTA_{old} that are still in the pipeline may be completed after CTA_{old} is swapped out. Generally when an instruction completes it would update the scoreboard. However, in VT any warps from the swapped out CTA_{old} do not update the scoreboard. Instead they simply write to the register file associated with CTA_{old}. Hence, in VT any swapped out CTA_{old} operations are not allowed to access or clear the scoreboard bits since the scoreboard vectors are reassigned to CTA_{new}. This process is described in more detail in the writeback stage discussion.

Register Read Stage: Once an instruction is issued the input operands are read from the register file and stored in the operand collector buffer. In the baseline architecture, each warp's register file base location is computed using the PWI. With VT, the PWIs are reused and hence warps from CTA_{new} cannot use them to access their own local registers. We introduce Virtual Warp ID (VWI) to each warp that is assigned to an SM. VWI is a unique warp identifier across all the assigned warps to the SM, including both active and inactive CTAs. Only when a CTA finishes execution its VWIs are released and reused for another CTA. Hence, context switching a CTA does not release the VWI. Initially when VT assigns the maximum number of CTAs that can fit within the capacity limit, VWIs are assigned sequentially starting from the first CTA. For instance, if there are five CTAs with eight warps per CTA, then VWIs 0-7 are assigned to the eight warps in CTA#0, VWIs 8-15 are assigned to the eight warps in CTA#1 and so on.

The VT architecture uses VWIs to access the register file, instead of PWIs. Since the baseline architecture allows only 48 physical warps they can be encoded in 6 bits. However, the VT architecture requires more bits to encode the VWI. As shown in our motivation results earlier, most benchmarks reach the capacity limit well below 256 warps (5× the baseline) in total. Thus we limit the total number of virtual warps to 256 which can be encoded using 8 bits. Note that the limitation of 256 warps is an empirically-driven choice and the primary impact of using a larger number of VWIs is that the VWI encoding requires additional bits.

The VT architecture simply fills up the underutilized registers to pack more warps. Hence using the VWIs, instead of the PWIs, does not perturb the register allocation process or the number of register accesses of the baseline architecture. In fact the total number of register accesses remains the same over the entire kernel execution with or without VT.

Writeback Stage: After an instruction is completed in the functional unit, VWI is also used to determine the location in register file where the new value of destination register should be written. Typically the destination register is also released from the scoreboard at that time. As noted earlier, VT resets the scoreboard bit-vector from CTA_{old} and reassigns it to CTA_{new} on initiating a context switch. Hence, when any pending operations from CTA_{old} complete, they should not be allowed to clear the scoreboard bits since they are reassigned to CTA_{new}. To handle this scenario correctly, we store VWI for each warp in the scoreboard. We use PWIs to index the scoreboard array. But during writeback we compare the VWI field of the scoreboard entry with VWI of the completing instruction to make sure that the entry belongs to the same warp. If two VWIs mismatch, this means the current warp belongs to an inactive CTA, therefore the scoreboard is not updated by CTA_{old}. While CTA_{old} may not update the scoreboard it is still necessary to keep track of the progress of CTA_{old}'s long latency operations so as to recognize when a stalled CTA is ready for execution again. For this purpose, we use a CTA status table incorporated within the virtual thread controller. The controller structure and operation will be described in detail shortly.

Branch Divergence Handling: The baseline architecture uses SIMT stack to store next PC and reconvergence PC to handle branch divergence [21], [42], [43]. As discussed in prior work [21], a portion of the SIMT stack per each warp may be cached on-chip for fast branch handling and any entries that do not fit the SIMT stack are stored in global memory. The cached SIMT stack usually holds a fixed number of entries per each warp. In our implementation, we assume that each warp has a 4-entry SIMT stack that is cached and any warp that requires more than four entries will spill to the global memory. In practice across a broad range of applications we studied, the 4-entry SIMT stack never overflowed. With VT, only the cached portion of the SIMT stack is saved/restored in shared memory as part of the CTA context. The global memory portion of the SIMT stack, if any, is left untouched. We simply need to use VWI, instead of PWIs, to access the global memory portion of the SIMT stack to uniquely identify each CTA's complete SIMT stack.

CTA Swap Operation: To manage CTA context swap, the virtual thread controller (VTC), which is a schedule monitoring logic is inserted in the VT architecture. VTC keeps track of the state of all CTAs in order to determine which CTA can be brought back to active from inactive state or vice versa. VTC employs a table called CTA status table, as shown in Figure 7. The table is indexed by CTA ID, and each entry of the table has two fields. One-bit status field indicates the corresponding CTA is whether active or inactive. The second field is called tracking counter, and the counter is used for two different purposes depending on the CTA state. If the CTA is active, it is used for indicating the number of stalled warps that belong to that CTA. The counter is incremented when the warp scheduler detects a warp is stalled by a long latency memory operation and decremented when the warp is released from the stall. The warp schedulers send associated CTA ID of the warp to VTC for this operation. If the counter value reaches the number of warps in the CTA, the CTA is marked for swap out. Note that the number of warps per CTA is precomputed at kernel launch and is already available in the SM for tracking CTA progress.

When a CTA is marked for swap out, the necessary context for restarting its execution is saved in the shared memory through the context handler. The per-CTA state information includes VWI, CTA ID, and SIMT stack (including PC). All these elements are fixed size and hence the per-CTA context size is also fixed. Furthermore, the context information is saved in consecutive memory location, the shared memory stores generated due to context saving are all coalesced memory accesses. The SIMT stacks entries are first moved into a temporary buffer and are then scheduled for storing into the shared memory. The temporary buffer allows the context handler to save the SIMT stack outside of the CTA switching critical path. Finally the SIMT stack, instruction buffer, program counter array, and scoreboard entries of all warps belonging to the swapped out CTA are invalidated so the new CTA can reuse these structures. More details of the context saving overhead will be discussed in Section IV-C.

Before invalidating the scoreboard, the tracking counter associated with the swapped-out CTA is initialized with the number of write-pending registers that belong to the CTA in order to handle the pending operations from the swapped-out CTA. The number of pending register writes can be obtained by counting the busy bits in the scoreboard bit-vector entries of all the warps that belong to the swapped-out CTA. Using the counter, VTC is able to collectively track whether all current in-flight instructions of the swapped-out CTA have completed. When an instruction belonging to a swapped-out CTA is completed at the writeback stage, the tracking counter is decremented. An inactive CTA becomes a candidate for scheduling once the counter value is zero. All such ready but pending CTAs are candidates for execution once a currently active CTA is marked for swap out.

In our current implementation, we use a first-ready-first-serve policy to select the CTA for swapping. Once the CTA is selected then VTC reads its context from the context space in the shared memory. VTC does not require additional access ports for the shared memory. VTC shares access port with load/store units and requests are scheduled so that demand requests have always a higher priority than the CTA swap operation. We observed that such additional accesses have only negligible performance impact on the original demand requests.

B. Shared Memory Management

Figure 1(b) shows how the shared memory space is managed with VT. We partition the shared memory into two regions: data and context. The data region is used for data which is allocated for each CTA, and swapped-out CTA states are stored in the context region. Note that the memory management is simplified by allocating data region from low to high shared memory address and context region from high to low address.

In VT, the context size of each CTA is fixed which simplifies shared memory management, at the expense of some wasted space. At the beginning of a kernel launch, the GPU runtime calculates the number of CTAs needed to reach the capacity limit, as well as the number of warps in each CTA. Once the total number of virtual warps are calculated then the runtime reserves shared memory which is equal to the number of virtual warps $\times$ the context size of each warp. As stated earlier, the context of each warp is primarily the 4-entry SIMT stack, VWI, and CTA identifiers.

C. Cost of Context Swapping

In this section, we discuss the size and timing overheads for the CTA swap operation. The details of CTA-specific data structures in existing GPUs are not disclosed in any publicly accessible literature, and we believe they are implementation dependent. Therefore, the following discussion is based on our baseline design assumptions. However, we have evaluated performance impact with various cost scenarios in Section V, and conclude that the performance impact of context saving/restoring of SIMT stack is minimal.

We first calculate the total size of state data for a CTA. The following is the list of CTA state data that we store/restore on a CTA switch in our implementation.

Virtual Warp Identifiers: For each swapped out CTA, we need to store W VWIs, where W is number of warps in each CTA. In VT, instead of saving all VWIs, we save the first VWI and the number of warps in each CTA since VWIs are sequentially assigned. Assuming the first VWI requires N bits, the total size for warp identifiers is $N + log_2 W$ per CTA. In our particular baseline, the VT architecture has a maximum of 256 virtual warps and each CTA can have up to 32 warps $(1,024/32)$ [9]. Hence, an 8-bit VWI of the first warp in the CTA, and a 5-bit for the number of warps in CTA must be stored; a total of 13 bits.

CTA Identifiers: Each CTA has a unique identifier based on three different dimensional indices, blockIdx.x, blockIdx.y, and blockIdx.z [31], [44]. Based on PTX specification [33], we assume that each index requires 32 bits. Therefore, the total size of CTA identifiers is 96 bits per CTA.

SIMT Stack: SIMT stack maintains thread divergence information of a warp [42], [43]. We assume a stack entry consists of thread active mask and two program counter fields [43]. Each stack entry requires 32 bits for a 32-lane active mask, 64 bits to save the Program Counter (PC), and another 64 bits to save the Reconvergence PC (RPC). In VT, the SIMT stack size is fixed at D entries (D=4 in our base machine). Then, total size of stack is $160 \times D$ bits. Given W warps in a CTA, the total size of stack for the CTA is about $160 \times D \times W$ bits.

In summary, the state data size for a CTA is as follows:

$$State(bits) = (N + log_2 W) + 96 + (160 \times D \times W) \quad (1)$$

From Equation 1, we estimate CTA swapping latency. Based on the previous studies [5], [31], we assume the shared memory has total 32 banks and each bank can read 32-bit data per one or two clock cycles. Therefore, the total bandwidth of shared memory (B) is 1,024 bits or 512 bit per clock cycle. Therefore, the total CTA swapping latency can be computed using a parametrized model as follows:

$$Cycles = \frac{(N + log_2 W) + 96 + (160 \times D \times W)}{B} \quad (2)$$

D. Resolving Memory Contention

One side effect of increasing TLP is the increase in MLP. Typically the number of memory instructions issued per unit time also increase as more warps are concurrently executed. Note that the total number of memory requests over the kernel execution time does not change. It is the rate of memory requests that changes. The increase in MLP can be beneficial or detrimental, depending on the memory system capability, such as available bandwidth. Previous studies [5], [28] have shown that increasing MLP may hurt performance if they increase memory system contention. To alleviate any potential contention, we propose memory request reordering technique built on previous studies [29], [30].

Mascar [29] technique places a reordering queue between the load/store unit and L1 cache. Once a load instruction is issued, it checks the cache tags to find the hit/miss status. If the load is a hit, data is provided. If the load is a miss, then it may need additional resources to handle the miss such as Miss Status Handling Registers (MSHRs). If the miss cannot be properly handled due to structural hazards, or failure to reserve a cache entry for replacement (reservation failures), then that load instruction is moved to a re-execution queue for later replay, instead of stalling the entire memory system.

Inspired by this prior work [29], we create two load queues: one queue to manage loads from active CTAs and another queue to manage loads from inactive CTAs that are swapped out after the loads were issued. Every load enters the active queue first and if corresponding CTA is swapped out then that load is moved to inactive queue. The loads from the active queue are always given higher priority than the loads from the inactive queue. By giving higher priority to the active queue, any load requests from a swapped-out CTA are handled only after servicing active CTA requests. If no new memory requests are generated from the active CTAs, the load requests from the swapped-out CTAs are handled. Therefore, the VT architecture is able to detect load requests from swapped out CTAs and gives them lower priority to resolve memory contention.

Based on the previous study [29], each entry for the load queue has 301 bits to store information. In VT, each entry is 309 bits because of extra VWI (8bits). In our evaluation, total 60 (30 for active queue and 30 for inactive queue) entries are used, therefore the total size of queues is approximately 2,317 bytes. The space overhead of queues is about 3.5% compared to the L1 cache/shared memory which is about 64 KB [7], [9].

Table I
GPU MICROARCHITECTURAL PARAMETERS

Parameters	Value
Number of SMs	15
Core Clock	1.4 GHz
SIMD Pipeline Width	16
Warp Size	32-threads
Max Number of CTAs / Core	8
Max Number of Threads / Core	1,536
Max Number of Warps / Core	48
Number of Warp Schedulers / Core	2
Number of Scoreboard Counters / Core	256
Number of Registers	32,768
Size of Active & Inactive Queue	30 & 30
L1 Cache Size / Core	16 KB
Shared Memory / Core	48 KB
Shared Memory Bandwidth / Cycle	512 bits
MSHRs/Core	64
Warp Scheduler Policy	LRR & GTO & TW
CTA Scheduler Policy	Round Robin

Table II
LIST OF BENCHMARK APPLICATIONS

Type-S Applications		Type-C Applications	
Application Name	**Abbr.**	**Application Name**	**Abbr.**
2D Convolution [45]	2D	BlackScholes [46]	BS
Back Propagation [47]	BP	B+ Tree [47]	BT
Breadth First Search [48]	BF	Heart wall [47]	HW
Coulombic Potential [5]	CP	EigenValues [46]	EV
Convolution Separable [46]	CS	Binomial Options [46]	BO
Convolution Texture [46]	CT	HotSpot [47]	HS
Discrete Cosine Transform [46]	DC	LavaMD [47]	LM
Discrete Haar Wavelet Decomposition [46]	DW	Single Asian OptionP [46]	SP
2D finite different time domain [45]	FD	Matrix Multiplication [46]	MM
Fast Fourier Transform [48]	FT	Speckle Reducing Anisotropic Diffusion [47]	SRAD
Gaussian Elimination [47]	GA	Neural Network Digit Recognition [5]	NN
Histogram [46]	HI	MUMmerGPU [47]	MUM
Needleman Wunsch [47]	ND	QuasiRandom Generator [46]	QRG
Path Finder [47]	PF	Ray Tracing [5]	RAY
Reduction [46]	RD	Leukocyte [47]	LE
Sorting Networks [46]	SN	Merge Sort [46]	MS
Thread Fence Reduction [46]	TF	Symmetric Rank-2K Operations [45]	S2K
Vector Addition [46]	VA		

V. EVALUATION

A. Methodology

Our baseline configuration is similar to an NVIDIA Fermi-like GPU architecture. The baseline architecture is able to execute a maximum of 8 CTAs and 1,536 threads concurrently, and each SM has 32K $\times$ 32-bit registers with 64 KB reconfigurable on-chip memory [31]. We configure the size of shared memory to be 48 KB. The detailed microarchitecture configuration that we simulated is presented in Table I. We have chosen 18 applications from NVIDIA CUDA SDK [46], Rodinia benchmark suite [47], GPGPU-Sim [5], scalable heterogeneous computing benchmark suite [48], and polybench [45] that are Type-S applications which are bottlenecked by the scheduling limit. Clearly, Type-C applications show no better or worse performance compared to the baseline. They have reached the capacity limit and no

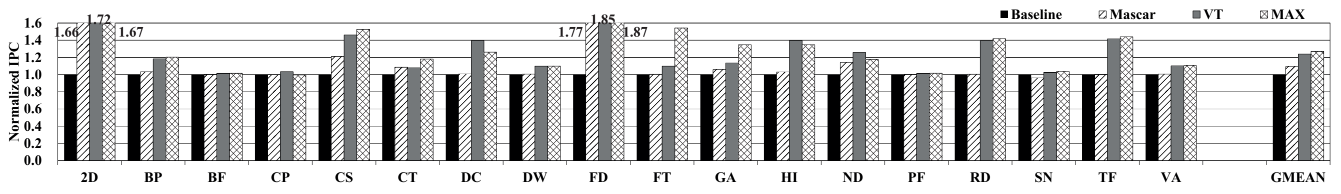


Figure 8. Performance Improvement of VT Architecture

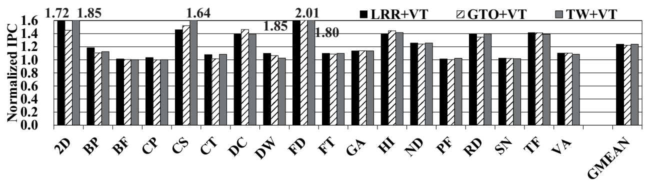


Figure 9. Impact of Various Warp Schedulers

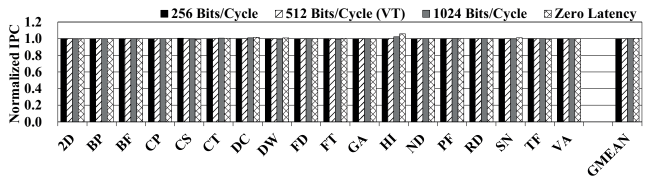


Figure 10. Impact of Various Swapping Delays

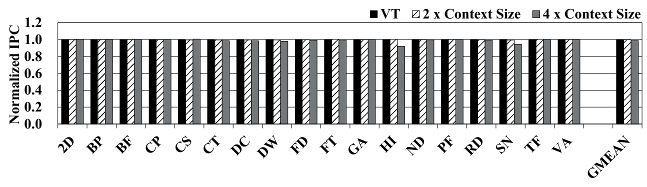


Figure 11. Impact of Various Context Sizes

more virtual CTAs are created, hence SMs will schedule the same number of threads to the baseline. In this case, the VT architecture does not improve the performance for the Type-C applications since CTA swapping will not happen. Therefore, there is no need to apply VT when applications are not bottlenecked by the scheduling limit.

The applications from Type-S and Type-C are listed in Table II. All of our applications run from beginning to 1 billion instructions using GPGPU-sim [5]. We also use the largest available input set for each benchmark since some benchmarks dispatch only a small number of CTAs with the small input set which results in an artificially high resource underutilization. For the evaluation, we use Loose Round Robin (LRR) warp scheduler since the LRR warp scheduler is commonly implemented on GPUs [32], [49], [50]. However, to verify the impact of various warp schedulers with VT, we also evaluate VT using Greedy-Then-Oldest (GTO) [51] and TWo-level (TW) [32] warp schedulers and compare the results to GTO and TW warp schedulers, respectively.

B. *Performance of VT Architecture*

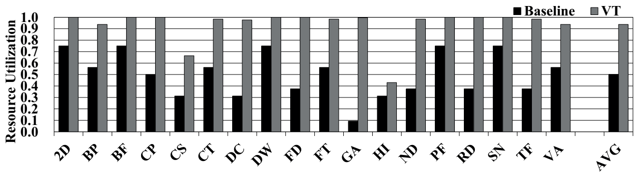
Figure 8 shows the normalized IPC of four different GPU configurations: baseline (Baseline), Mascar [29], VT architecture, and unlimited scheduling resources (as many program counters as needed to fill the capacity, as many SIMT stacks as needed etc.) with Mascar (MAX). Our results show that an average performance improvement for VT is 23.9% compared to baseline. The performance improves universally across almost all the benchmarks. However, *BF*, *PF*, and *SN* will see no benefits from VT. *SN* and *PF* do not suffer from any stalls to begin with and even in the baseline instructions are issued for more than 80% of total execution cycles as shown in Figure 4. In addition, *BF* has low ratio of memory related stalls. Therefore, VT can only hide a small portion of stall cycles in these applications resulting in small performance improvement.

We compare VT with Mascar which is a recently proposed memory reordering technique [29] since VT has the memory reorder scheme which is inspired from previous studies [29], [30]. Based on our evaluation, Mascar improves an average 9.3% performance while VT improves an average 23.9% compared to the baseline. With Mascar, applications *2D*, *CS*, *FD*, and *ND* show high performance improvement compared to the baseline. These applications suffer from relatively high pipeline stalls and Mascar is well suited for reducing the pipeline stalls. With VT, the performance improves universally across almost all the benchmarks because VT is designed to increase TLP to hide memory related stalls such as long latency stalls and pipeline stalls. Overall, VT outperforms Mascar by 13.4%.

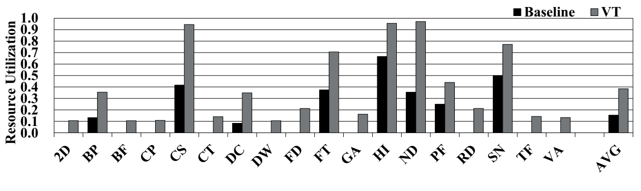
We also compare VT with a hardware-intensive traditional approach, shown as MAX in our evaluations. MAX increases the scheduling limit to fill up all the available registers or shared memory (whichever comes first) assuming scheduling hardware can be scaled at zero cost. Hence, MAX assumes no latency penalties for supporting larger scoreboards, bigger instruction buffers, and larger comparison logic to pick the ready warps. MAX also includes Mascar on top of unlimited scheduling resources. Based on the results, the performance of VT is nearly equal to the performance of MAX. On average, VT improves the performance by about 23.9% and MAX improves the performance about 26.9%. In *2D*, *CP*, *DC*, *HI*, and *ND*, VT yields higher performance than MAX. The reason behind this is due to the small changes in warp scheduling in VT due to the memory request prioritization.

We also measured the detailed cycle breakdown with VT. The second bar in each group in Figure 4 shows the cycle distributions for VT. The stall cycle distribution shows that VT reduces 18.1% of stall cycles compared to baseline. Especially, the pipeline stalls and long latency stalls are significantly reduced. VT exploits higher TLP to hide the long latency stalls more effectively.

1) Impact of Warp Scheduler: In this section, we evaluate the performance impact of VT using various warp schedulers: LRR [32], [49], [50], GTO [51], and TW warp schedulers [32]. Figure 9 shows the performance improvement of VT using various warp schedulers. As mentioned in the previous section, using the LRR warp scheduler, VT improves the performance by about 23.9% compared to the baseline LRR configuration. The performance of GTO+VT

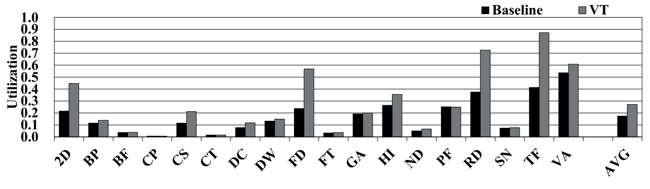


(a) Register File Utilization

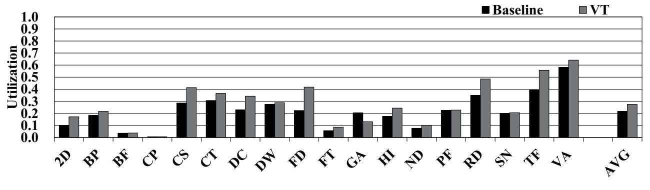


(b) Shared Memory Utilization

Figure 12. Resource Utilization with VT



(a) L1 MSHR Utilization



(b) DRAM Bandwidth Utilization

Figure 13. Impact on Memory System

is about 22.2% compared to the baseline GTO configuration. In addition, TW+VT improves the performance by about 23.9% compared to the baseline TW configuration. As can be seen from the results, VT can be combined with any warp scheduling policies and it provides fairly consistent performance improvements.

2) Impact of Swapping Delay and Context Size: We evaluate the performance impact of VT using various swapping delays. In the prior section, we assumed that the context can be swapped to shared memory using 512-bits/cycle bandwidth. In this section, we evaluate VT using 256 bits, 512 bits, and 1,024 bits of shared memory bandwidth. In addition, we have evaluated VT using no swapping overhead. Figure 10 shows the evaluation results. Based on the results, the performance gaps are very small. VT without the swapping overhead is 0.4% faster than the baseline that uses 512-bits/cycle bandwidth. Also, VT with 256 bits of shared memory bandwidth shows no performance degradation. The only time swapping overhead has an impact on performance is when there are no CTAs in active state or an occasional conflict in using the shared memory to save/restore CTA context and a demand request.

We also evaluate the performance impact of the size of the CTA context data that is needed to be saved and restored with VT. We increase the context size to be twice as large and four times as large. Large context is typically due to increasing the number of entries in the SIMT stack. Figure 11 shows the performance results. As the context size doubles, VT performance is only degraded by about 0.2% on average. VT performance improvements are reduced by 1.1% on average when context size is quadrupled. The performance degradation of increasing the context size is not due to increased latency to save/restore context state. Rather it is due to the increased demand for more shared memory where the context is saved. As the size of CTA contexts is increased, the number of CTAs that can be dispatched to SMs is decreased for some applications such as *HI* and *SN* as they hit the capacity limit quicker.

C. Resource Utilization

In this section, we present the resource utilization for the register file and shared memory with VT. Figure 12 shows the resource utilization graph for the register file and the shared memory. Figure 12(a) shows the register file utilization. In Baseline, register file resource utilization is around 50.2%, however VT increases the register utilization to 93.8%. The register file utilization may not reach 100% since the unused register file is too small to assign another CTA. Also, for applications such as *CS* and *HI* more CTAs cannot be assigned as the shared memory capacity limit is reached.

Figure 12(b) shows the result of shared memory utilization on GPUs. On average, shared memory space utilization is about 15.4% in the baseline. However, with VT, an average utilization is increased to 38.5%. Note that most applications reach the register file capacity limit well before reaching the shared memory limit.

D. Impact on Memory System

In this section, we show the impact on memory system when VT is activated. We measured the L1 cache miss rate and L1 MSHR utilization. Our results on L1 cache miss rates showed that VT increases cache miss rates by less than 1.7%, which has negligible performance impact. Most applications only rarely reuse cache lines [52], [53] temporally. When a CTA is swapped out only a small fraction of its data in L1 cache is reused when the CTA is swapped back in. As a result, VT does not degrade cache miss rate. L1 MSHR utilization is increased with VT. Note that even though the overall cache miss rate is not decreased, MLP is increased because more memory requests are initiated within a shorter time interval with VT. However, VT does not generate any useless data requests since most of the data brought into the cache is used before the CTA is swapped out. As a result, Figure 13(a) shows that MSHR utilization is increased from 17.5% to 27.1% on average. However, the performance impact of the increased utilization is negligible.

In addition, we measure DRAM bandwidth utilization. Figure 13(b) shows the bandwidth utilization on Baseline and VT. On average, the DRAM utilization is increased from 21.7% to 27.4% when VT is activated. In *GA*, the utilization is decreased because L2 cache miss rate is slightly decreased. The reason behind the lower miss rate is due to the improved inter-CTA data locality. Overall, the DRAM utilization increase is also for the same reason as MSHR utilization increase. Namely, MLP is improved with VT as additional CTAs generate more requests.

VI. Related Work

To the best of our knowledge, this is the first paper that proposes a simple context switching solution to improve TLP using underutilized register file and shared memory on GPUs. In this section, we describe the closely related work.

Latency Tolerance Techniques on CPUs: Previously, many studies proposed various latency tolerance techniques on CPUs [54], [55], [56]. In these techniques, instructions which depend on a long latency memory operation are removed from the instruction window to allow executing near future instructions. By doing this, instruction level parallelism is successfully increased and the memory latency can be well tolerated. In this paper, we propose a light weight context switching technique which increases TLP to hide the long latency memory stalls more effectively on GPUs.

Application Level Context Switching on GPUs: Application level context switching in current GPUs is analogous to process scheduling in CPUs involving full context swap [8]. According to Fermi whitepaper this switch takes up to 25us, which corresponds to tens of thousands of cycles. However, VT targets to improve scheduling efficiency for applications, and VT does not perform full context swap and switches only a limited CTA context information which takes only tens of cycles.

Resource Underutilization Problem on GPUs: Several power management policies have been proposed for reducing leakage power consumption for unused on-chip resources [10], [57], [58], [59]. For instance, Abdel-Majeed and Annavaram [10] proposed the warped register file. In the proposed technique, the tri-modal register file is inserted into GPUs. The tri-modal register file can change the register state into ON, OFF, and drowsy states. The registers are power-gated if the registers are not needed to execute the kernels (OFF state). In addition, the state of registers is changed to ON state when the registers are being used by the processors, otherwise the state of registers is in drowsy state. In contrast to the technique, VT targets to increase performance by minimizing underutilized resources and improving resource utilization.

In addition, flexible resource management schemes have been proposed to overcome the capacity limit [37], [60], [61]. In this approach, all on-chip storages in an SM, especially register file and shared memory are unified into a single malleable memory. This approach alleviates capacity limit by more flexibly managing storage demands of various applications. However, as the capacity limit is relaxed, more applications will be constrained by the scheduling limit. Therefore, VT complements potential TLP bottleneck with this approach by relaxing the scheduling limit.

Programming and compile time approaches have also been proposed for improving resource utilization [62], [63]. For instance, in shared memory multiplexing [63], allocated shared memory is released as soon as the space is no longer used by CTAs, even though the CTAs are not yet completed. These approaches better utilize on-chip storage space by activating more threads, however in common such techniques still capped to both thread and CTA limits. VT allows to issue more threads even beyond the scheduling limit, therefore further improves TLP and enforces capacity limit to be the only TLP limiting factor.

Thread Scheduling on GPUs: Within the scheduling limit, a solution for squeezing more TLP is relaxing the restriction of CTA-granularity resource allocation. Xiang *et al.* proposed the warp level resource management technique [11] called WarpMan. In WarpMan, the CTA scheduler allocates the threads on SMs at a warp granularity instead of a CTA granularity. By doing that, the proposed technique can exploit higher degree of TLP using additionally issued threads. The number of thread that can be dispatched on SMs cannot exceed the conventional hardware limit, however VT can dispatch more threads on SMs.

GPUs exploit TLP for hiding long processing delay, which is the primary objective of warp schedulers [26], [32], [64]. Narasiman *et al.* proposed the two-level warp scheduler for maximizing latency hiding effect [32]. The scheduler divides the warps into multiple fetch groups and the long latency operation stalls of each group can be hidden by executing warps from other fetch groups. All these techniques improve performance by rearranging the order of execution within the scheduling limit. Our approach provides more warps and CTAs to be scheduled therefore improves latency hiding effect as can be seen in the previous section.

Some previous studies also have noted that issuing more CTAs is not always beneficial for GPGPU applications [5], [28], [65]. Issuing additional threads can increase memory contention on GPUs, therefore the performance may degrade for the applications. Researchers have explored minimizing memory contention by improving CTA or warp scheduling policy [28], [50], [51]. Kayiran *et al.*, or proposed a dynamic CTA scheduling mechanism [28], which throttles the number of active CTAs based on the degree of memory contention. The OWL scheduler [50] further improves performance by reordering CTAs for maximizing bank level parallelism and row-buffer locality of DRAM. Rogers *et al.* proposed Cache-Conscious Wavefront Scheduling [51], which issues warp instructions based on the intra-wavefront locality, therefore the scheduler reduces the operation stalls by increasing cache efficiency. In this work we show that increasing TLP is still beneficial for many GPGPU applications in terms of the performance, along with memory requests reordering.

Memory Request Scheduling on GPUs: To maximize the benefits of additional TLP and minimize its side effects, we have proposed the memory enhancement technique on VT which is inspired by recently proposed memory reordering approaches [29], [30]. As we discussed previously, the design of VT adopts the concept of the re-execution queue from Mascar [29]. The queue resolves blocking problem of cache requests, and VT applies prioritization techniques for further improving its effect in the context of VT. MRPB [30] prioritizes requests using a similar scheduling buffer, and more focuses on the reordering policy of cache access requests to be cache friendly. Our VT architecture incorporates memory reordering approaches after providing extra CTAs to be scheduled, thereby achieves even better performance compared to applying only a single technique.

VII. Conclusion

In this paper, we observe that the scheduling limit is a major TLP bottleneck on GPUs. To relax the scheduling limit constraint, we propose the VT architecture which can dispatch CTAs to SMs until the on-chip storage capacity limit is reached regardless of the scheduler limit. We use the notion of active and inactive CTAs where the number of active CTAs is still limited by the scheduling logic. VT swaps out a CTA when warps in the active CTA are all stalled by the long latency memory operations. By switching between active and inactive states, VT can exploit higher degree of TLP without increasing logic complexity. Based on our evaluation, VT improves performance by 23.9% on average.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP)(No. NRF-2015R1A2A2A01008281), by Memory Division, Samsung Electronics Co., Ltd., and by the following grants: DARPA-PERFECT-HR0011-12-2-0020 and NSF-CAREER-0954211, NSF-0834798. W. W. Ro is the corresponding author.

References

[1] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "APOGEE: Adaptive Prefetching on GPUs for Energy Efficiency," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, 2013.

[2] Y. Zhang and J. Owens, "A Quantitative Performance Analysis Model for GPU Architectures," in *Proceedings of IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[3] C. Wojek, G. Dorkó, A. Schulz, and B. Schiele, "Sliding-windows for Rapid Object Class Localization: A Parallel Technique," in *Pattern Recognition*, Springer, 2008.

[4] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU Concurrency with Elastic Kernels," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[5] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads using a Detailed GPU Simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS '09, 2009.

[6] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A Simulation Framework for CPU-GPU Computing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, 2012.

[7] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, vol. 31, no. 2, 2011.

[8] "Fermi: NVIDIA's Next Generation CUDA Compute Architecture." http://goo.gl/zAtZMY", 2009. Accessed: 24 April 2014.

[9] "NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110." http://goo.gl/DLi9nu. Accessed: 23 Dec 2014.

[10] M. Abdel-Majeed and M. Annavaram, "Warped Register File: A Power Efficient Register File for GPGPUs," in *Proceedings of the 19th International Symposium on High Performance Computer Architecture*, HPCA '13, 2013.

[11] P. Xiang, Y. Yang, and H. Zhou, "Warp-level Divergence in GPUs: Characterization, Impact, and Mitigation," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[12] H. Jeon and M. Annavaram, "GPGPU Register File Management by Hardware Co-operated Register Reallocation," tech. rep., 2014.

[13] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "GPU Register File Virtualization ," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '15, 2015.

[14] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A Case for Core-assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, 2015.

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

[16] C. Li, Y. Yang, H. Dai, S. Yan, F. Mueller, and H. Zhou, "Understanding the Tradeoffs between Software-managed vs. Hardware-managed Caches in GPUs," in *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '14, 2014.

[17] M. Rhu and M. Erez, "The Dual-path Execution Model for Efficient GPU Control Flow," in *Proceedings of IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, 2013.

[18] W. W. L. Fung, "GPU Computing Architecture for Irregular Parallelism," 2015.

[19] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-Preexecution: A GPU Pre-execution Approach for Improving Latency Hiding," in *Proceedings of the 22th International Symposium on High Performance Computer Architecture*, HPCA '16, 2016.

[20] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam, "Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU," *ACM Trans. Archit. Code Optim.*, vol. 12, June 2015.

[21] S. Collange, "Stack-less SIMT Reconvergence at Low Cost," tech. rep., Technical Report HAL-00622654, INRIA, 2011.

[22] NVIDIA, "Whitepaper: NVIDIA GeForce GTX 680." http://goo.gl/f58MQ, 2012.

[23] N. Brunie, S. Collange, and G. Diamos, "Simultaneous Branch and Warp Interweaving for Sustained GPU Performance," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, 2012.

[24] B. Coon, P. Mills, S. Oberman, and M. Siu, "Tracking Register Usage during Multithreaded Processing using a Scoreboard having separate memory regions and storing sequential register size indicators," 2008. US Patent 7,434,032.

[25] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *Micro, IEEE*, vol. 28, no. 2, 2008.

[26] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "A Hierarchical Thread Scheduler and Register File for Energy-Efficient Throughput Processors," *ACM Trans. Comput. Syst.*, vol. 30, Apr. 2012.

[27] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling Preemptive Multiprogramming on GPUs," in *Proceedings of the 41st International Symposium on Computer Architecture*, ISCA '14, 2014.

[28] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, 2013.

[29] A. Sethia, D. Jamshidi, and S. Mahlke, "Mascar: Speeding up GPU Warps by Reducing Memory Pitstops," in *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, HPCA '15, 2015.

[30] W. Jia, K. Shaw, and M. Martonosi, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[31] C. Nvidia, "CUDA C Programming Guide," *NVIDIA Corporation*, 2014.

[32] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '11, 2011.

[33] N. Compute, "PTX: Parallel Thread Execution ISA Ver-

sion 2.3," *Dostopno na: http://developer. download. nvidia. com/compute/cuda/3*, vol. 1, 2010.

[34] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann Publishers Inc., 1st ed., 2010.

[35] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, May 2008.

[36] S. Stone, J. Haldar, S. Tsao, W. m.W. Hwu, B. Sutton, and Z.-P. Liang, "Accelerating Advanced MRI Reconstructions on GPUs," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, 2008.

[37] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor," in *Proceedings of the 45th Annual International Symposium on Microarchitecture*, MICRO '12, 2012.

[38] C.-C. Wu, J.-Y. Ke, H. Lin, and W. chun Feng, "Optimizing Dynamic Programming on Graphics Processing Units via Adaptive Thread-Level Parallelism," in *Proceedings of IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, 2011.

[39] P. Xiang, Y. Yang, M. Mantor, N. Rubin, and H. Zhou, "Many-thread Aware Instruction-level Parallelism: Architecting Shader Cores for GPU Computing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, 2012.

[40] F. Ji and X. Ma, "Using Shared Memory to Accelerate MapReduce on Graphics Processing Units," in *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium*, IPDPS '11, 2011.

[41] P. Micikevicius, "3D Finite Difference Computation on GPUs Using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU '09, 2009.

[42] W. Fung and T. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," in *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[43] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware," *ACM Trans. Archit. Code Optim.*, vol. 6, July 2009.

[44] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, 2008.

[45] "PolyBench: The Polyhedral Benchmark Suite." http://web.cse.ohio-state.edu/~pouchet/software/polybench/. Accessed: 30 April 2015.

[46] "NVIDIA CUDA SDK Code Samples." http://developer.nvidia.com/cuda-downloads. Accessed: 24 April 2014.

[47] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '09, 2009.

[48] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, 2010.

[49] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[50] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[51] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, 2012.

[52] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal, "A Detailed GPU Cache Model based on Reuse Distance Theory," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[53] Y. Oh, K. Kim, M. K. Yoon, J. H. Park, Y. Park, W. W. Ro, and M. Annavaram, "APRES: Improving Cache Efficiency by Exploiting Load Characteristics on GPUs," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[54] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A Large, Fast Instruction Window for Tolerating Cache Misses," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, 2002.

[55] M. Pericas, A. Cristal, R. Gonzalez, D. Jimenez, and M. Valero, "A Decoupled KILO-instruction Processor," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, HPCA '06, 2006.

[56] Y. Kora, K. Yamaguchi, and H. Ando, "MLP-aware Dynamic Instruction Window Resizing for Adaptively Exploiting Both ILP and MLP," in *Proceedings of the 46th International Symposium on Microarchitecture*, MICRO '13, 2013.

[57] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[58] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs," in *Proceedings of the 46th International Symposium on Microarchitecture*, MICRO '13, 2013.

[59] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-Compression: Enabling power efficient GPUs through register compression," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, 2015.

[60] A. B. Hayes and E. Z. Zhang, "Unified On-chip Memory Allocation for SIMT Architecture," in *Proceedings of the 28th International Conference on Supercomputing*, ICS '14, 2014.

[61] O. Villa, D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally, "Scaling the Power Wall: A Path to Exascale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, 2014.

[62] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, 2008.

[63] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou, "Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, 2012.

[64] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, 2011.

[65] X. Chen, L.-W. Chang, C. Rodrigues, J. Lv, Z. Wang, and W. mei Hwu, "Adaptive Cache Management for Energy-Efficient GPU Computing," in *Proceedings of the 47th International Symposium on Microarchitecture*, MICRO '14, 2014.