Automatically Exploiting Implicit Pipeline Parallelism from Multiple Dependent Kernels for GPUs

Gwansun Kim, Jiyun Jeong, John Kim
KAIST
{gskim, cgy9037, jjk12}@kaist.ac.kr

Mark Stephenson
NVIDIA
mstephenson@nvidia.com

ABSTRACT

Execution of GPGPU workloads consists of different stages including data I/O on the CPU, memory copy between the CPU and GPU, and kernel execution. While GPU can remain idle during I/O and memory copy, prior work has shown that overlapping data movement (I/O and memory copies) with kernel execution can improve performance. However, when there are multiple dependent kernels, the execution of the kernels is serialized and the benefit of overlapping data movement can be limited. In order to improve the performance of workloads that have multiple dependent kernels, we propose to automatically overlap the execution of kernels by exploiting implicit pipeline parallelism. We first propose Coarse-grained Reference Counting-based Scoreboarding (CRCS) to guarantee correctness during overlapped execution of multiple kernels. However, CRCS alone does not necessarily improve overall performance if the thread blocks (or CTAs) are scheduled sequentially. Thus, we propose an alternative CTA scheduler – Pipeline Parallelism-aware CTA Scheduler (PPCS) that takes available pipeline parallelism into account in CTA scheduling to maximize pipeline parallelism and improve overall performance. Our evaluation results show that the proposed mechanisms can improve performance by up to 67% (33% on average). To the best of our knowledge, this is one of the first work that enables overlapped execution of multiple dependent kernels without any kernel modification or explicitly expressing dependency by the programmer.

Keywords

GPGPU; Pipeline parallelism; Overlapping Kernels; Thread Block Scheduling

1. INTRODUCTION

GPUs have been widely used in various domains to accelerate data-parallel workloads. While significant research effort has been made to improve the performance and efficiency of GPUs, relatively little research has been done to reduce the overhead of non-kernel portions of GPGPU workloads. The overhead mainly consists of two different components – accessing I/O devices (e.g., HDD, SSD, or network card), and memory copy between host and device memory. The time spent on accessing I/O devices to initialize input data or storing results can significantly outweigh the memory copy overhead since I/O devices provide much lower bandwidth than the PCIe bus (for discrete GPUs) or on-chip bus (for integrated GPUs). Even the state-of-the-art SSD has ~3.2× lower bandwidth than PCIe 3.0 x16 (and 16-40× lower than NVLink). Thus, in a typical usage scenario, where input data are read from HDD or SSD to host memory and then copied to device memory, the time for I/O device access will surpass the time for memory copy. Furthermore, integrated GPUs with unified main memory for CPU and GPU [23] can eliminate the memory copy overhead and make the I/O overhead even more significant. The I/O overhead can be mitigated by overlapping the kernel execution with I/O device access. GPGPUfs [30] and GPUnet [21] enable file and network access within a kernel, respectively, and HSA (Heterogeneous System Architecture) [23] can also provide similar functionality.

However, prior work [30, 21] results in little improvement for many workloads that have multiple kernels with inter-kernel dependencies – 20 out of 30 workloads from the Rodinia [10] and Parboil [34] benchmark suites combined have such characteristics. Under the current bulk-synchronous programming model, implicit synchronization barriers exist between dependent kernels to prevent data hazards and guarantee correct execution. As a result, while execution of the first kernel can be overlapped with input file access (or I/O), the subsequent kernels cannot start execution until the first kernel is finished due to possible dependency. If there are many kernels, only the first kernel can be overlapped with the I/O device access, resulting in minimal overall performance improvement.

Overlapping multiple kernels with I/O device access and appropriately leveraging pipeline parallelism that exists between the dependent kernels can result in a significant improvement in performance. However, currently available manual optimization methods to leverage pipeline parallelism among kernels pose a significant challenge. While optimizing a workload with a single kernel to achieve such an overlap can be relatively straightforward by using CUDA...
streams [32, 28], for workloads with multiple dependent kernels, splitting each kernel into separate kernels can drastically increase the complexity of the dynamic dependency tracking that needs to be done by the programmer. With multiple streams, there can be intra-stream and inter-stream dependencies for complex workloads, but the CUDA runtime only checks intra-stream dependencies. Thus, the programmer has to manually manage the inter-stream dependencies, which require a significant programmer effort. Task-parallel languages [33] have been proposed as an alternative to better exploit the high parallelism in GPUs, but they still require additional programmer effort. Although they provide high-level APIs to aid synchronization and communication between tasks, dependency information still needs to be explicitly expressed by the programmer. On the other hand, enabling implicit pipeline parallelism for CUDA and OpenCL, which are currently the de facto standard for GPU computing, can achieve high performance without degrading programmability.

In this work, we propose a novel, hardware-software-based approach to automatically exploit implicit pipeline parallelism from multiple dependent kernels of GPGPU workloads without any kernel modification by the programmer. Since our approach moves dynamic dependency tracking into hardware, the programmer does not have to manage dependencies nor modify existing CUDA or OpenCL kernels. By combining our mechanisms with existing approaches that overlap a kernel with I/O access (e.g., GPUfs [30] or GPUnet [21]), the multiple dependent kernels can be overlapped with I/O device access, resulting in a significant overall speedup.

In order to avoid data hazards while overlapping dependent kernels, we propose a lightweight, profiler-assisted dependency tracking mechanism on the GPU. The profiler is automatically generated at compile time as a kernel from the original kernel code by a source-to-source translator and executed on the GPU. The profiler uses a sampling method [20] to determine the bounds of the memory addresses accessed by each thread block or CTA (Cooperative Thread Array) within each kernel with minimal overhead. The profiler kernel is executed before the main kernel begins and stores the access range information in the GPU’s reference count table, which is used during runtime to properly issue instructions while guaranteeing correct execution.

Due to a large number of threads (and CTAs) in modern GPUs, it is infeasible to keep all of the profiled dependency information in the GPU. Thus, we propose Coarse-grained Reference Counting-based Scoreboarding (CRCS) that only requires keeping reference counters for each kernel in the reference count table. It is coarse-grained as dependency tracking is done in CTA granularity (compared to thread granularity) and per page (instead of per word). CRCS is also based on reference counting, rather than keeping exhaustive dependency information.

However, even with CRCS, the performance improvement can be limited if pipeline parallelism is not properly considered in scheduling CTAs across multiple kernels. Thus, we propose a Pipeline Parallelism-aware CTA Scheduler (PPCS) that takes into account the amount of available data for each kernel and determines how many SMs (Streaming Multiprocessors) will be assigned for each kernel. We show how our mechanisms significantly reduce overall runtime by overlapping multiple kernels with I/O access. To summarize, the contributions of this work include the following:

- We propose novel mechanisms to automatically exploit implicit pipeline parallelism that exists between multiple dependent kernels for GPGPU workloads without any programmer effort to modify the kernel.
- In order to automatically track dependencies between a large number of CTAs from different kernels at low cost, we propose Coarse-grained Reference Counting-based Scoreboarding (CRCS) that guarantees correct execution while overlapping multiple dependent kernels with other stages.
- Since the baseline FIFO scheduler does not take data availability into account when scheduling CTAs from multiple kernels, we propose a Pipeline Parallelism-aware CTA Scheduler (PPCS) that considers data availability to further improve performance.
- We show that while prior work to overlap I/O access with kernel execution alone results in limited improvement for workloads with multiple kernels, by combining our proposed CRCS and PPCS with I/O access overlap, significant speedup can be achieved up to 67% (33% on average).

2. BACKGROUND / MOTIVATION

2.1 GPGPU Pipeline Parallelism

GPGPU workloads typically consist of several different stages executed serially. Figure 1(a) illustrates an example

![Figure 1: Timeline diagram of (a) baseline execution with no optimization, (b) manually optimized workload execution with dependency tracking by the programmer, (c) execution with the state-of-the-art approaches, and (d) our proposed approach (CRCS and PPCS) combined with the existing approaches.](image-url)
timeline of a GPGPU workload that consists of five kernels (Kernel_0 to Kernel_4) that have dependencies - the result of the first kernel is input to the second kernel, which in turn generates input to the third kernel, and so on. Before the kernels can start execution, input data need to be initialized by the CPU (e.g., read input data from disk) and we refer to this stage as a *prelude*. The input data also needs to be copied from CPU memory to GPU memory (H2D_copy). After the kernels finish execution, the results are copied back to the CPU memory (D2H_copy). The CPU thread then post-processes or writes the final result to an I/O device which we refer to as a *postlude* stage.

For workloads with a non-negligible prelude and postlude, there is an opportunity to overlap them with other stages. GPUs have much higher throughput compared to the PCIe and I/O devices, and a significant amount of computation can be done on the GPU while the prelude is being executed. For example, NVIDIA Tesla K40 GPU has a peak throughput of 4.79 Tflops (for single-precision) and with 4 bytes for each single-precision floating number, the peak throughput amounts to 17.2 TB/s, which is multiple orders of magnitude higher than the peak throughput of the state-of-the-art SSD [1]. Even with memory-bound workloads, Tesla K40 has a peak memory bandwidth of 288 GB/s which is two orders of magnitude higher than the peak SSD throughput.

Thus, if input data needs to be processed through multiple kernels, overlapping them with prelude can potentially provide significantly higher performance than overlapping a single kernel only. Figure 2 shows the execution time breakdown of two workloads measured on an NVIDIA Tesla M2050. For each workload, we varied the input data size to evaluate its impact on the prelude and postlude stages. While the overall execution time increases with a larger input data size, the execution time is normalized to illustrate the ratio of the different stages. While the portion of the kernel can either vary or remain constant as the input problem is varied depending on the complexity of the kernel computation, the ratio of time spent on prelude and postlude can be as high as approximately 86%. Prior work on the Rodinia benchmark [10] has also shown that a significant portion of overall execution time can be spent on the I/O overhead.² Properly overlapping kernel execution with I/O device access can potentially reduce the overall runtime considerably, but given the large number of kernels (denoted by \( N_k \) in Figure 2), only overlapping the first and the last kernel with I/O device access can have a very small impact overall.

### 2.2 Exploiting Pipeline Parallelism in GPU Computing

In a baseline system (Figure 1(a)), all stages are serialized and result in poor performance and system resource underutilization – the GPU stays idle during prelude, memory copy, and postlude stages, and conversely, the CPU remains idle during data transfer and kernel execution.

However, by overlapping different stages, the overall system resource utilization can be increased, resulting in runtime reduction. Overlapped execution can be realized by manual optimization (Figure 1(b)) through CUDA stream and asynchronous memory copy API [32, 28] to exploit pipeline parallelism explicitly. For very simple workloads, this can be easily done since there would be only intra-stream dependencies. However, in general, such optimizations become very complicated due to inter-stream dependencies. For example, there can be data shared by multiple split-kernels or the output data may need to be merged across outputs from multiple split-kernels. With manual optimization, the dependencies have to be dynamically managed during runtime by the programmer or the workload algorithm needs to be modified such that the kernel can be split into independent, smaller kernels. Figure 3 shows an example of an iterative 1D stencil computation implemented as multiple dependent kernels. It highlights the data dependencies in calculating the \( j \)th and \( k \)th elements at the \( i \)th iteration. Manual optimization with \( N_s \) streams results in \( N_s N_k \) kernels, where \( N_s \) is the number of total iterations. Across each iteration, there can be \( N_s^2 \) possible dependencies, resulting in a total of \( N_s N_k^2 \) possible dependencies. The dependency tracking will be further complicated as the input and output array are swapped across each iteration, especially with 2D or 3D data. Prior work [30, 21, 23] overlaps the execution of a kernel with prelude or postlude by enabling I/O device access within a kernel as shown in Figure 1(c). However, dependent kernels are still serialized and limit the overall performance.

In this work, we propose to overlap execution of the kernels by automatically tracking dependency between CTAs across kernels. By combining our approach with existing approaches, as shown in Figure 1(d), all of the different stages in a GPGPU workload can be overlapped and the overall utilization of CPU, GPU, and I/O devices can be maximized. In addition, our approach exploits pipeline parallelism implicitly as we do not require a programmer to express dependency or parallelism available in a workload.

---

²This overhead is not to be confused with memory copy overhead between host and device. Running a workload on a GPU instead of a CPU can remove memory copy overhead but not the I/O overhead.
3. OVERLAPPING NON-KERNEL STAGES WITH KERNEL EXECUTION

While our main contribution is the mechanism for overlapping execution of multiple kernels, it can be effective when non-kernel stages (i.e., prelude, membrane copy, and postlude) are overlapped with the kernel. Prior work, such as HSA [23], GPUs [30], and GPUnet [21], can be used for the overlapping. In this work, without any loss of generality, we assume a simple host model that is presented in this section. This model can be considered as an example implementation of overlapping non-kernel stages with the kernel and is orthogonal to our main contribution described in Section 4.

3.1 Host Thread Code Modification

In order to overlap a kernel the non-kernel stages, kernel execution needs to be initiated before the prelude and memory copy begins, and a separate host thread for postlude needs to be created. The reordering and thread creation is transparently done by the host model’s library, but we assume the programmer expresses what needs to be done at each stage.

In the original host code, the prelude comes first and then H2D copy, kernel launch, D2H copy, and postlude follow as shown in Figure 4(a).4 A modified example code that uses the OverlappedExec class provided by the library is shown in Figure 4(b). The code uses lambda functions supported by standard C++11 and recent CUDA to express each stage’s task in a simple way. An OverlappedExec object is instantiated by providing five lambda functions to the constructor in the order of prelude, H2D copy, kernel, D2H copy, and postlude. For memory copy API calls, MemcpyH2DOverlap or MemcpyD2HOverlap was given as the type of operation. Then the different tasks start execution when the run method is called for the object (line 20). The library first initiates the D2H and H2D memory copy functions to set up DAT entries and the CPU and GPU page tables (Section 3.2). Then, the kernel is launched and another host thread is created to execute the postlude function.5 Finally, the prelude is started and the execution of different stages will be overlapped. The run method is a blocking function and will return after all tasks are finished, but the library can also provide a non-blocking version as well. This approach is not necessarily optimal and is only one of many possible approaches. The goal of this section is to present a reasonable, programmer-friendly approach to express different stages of GPGPU workloads.

3.2 Overlapping Data Transfer with CPU/GPU Execution

In the host model that we assume, the CPU needs to automatically transfer input data to device memory as they become available. The Data Availability Tracker (DAT) shown in Figure 5 is placed near the memory controller and is responsible for detecting data availability and initiating data transfer. When the overlapped H2D copy API is called (line 8-9 in Figure 4(b)), an entry is created at DAT for each page in host memory through OS. For correct operation, the corresponding host pages need to be page locked or pinned to avoid changing the mapping between pages and frames. Each entry consists of a frame number, a write counter, and a F/E bit vector. The write counter signifies the number of words initialized and the F/E bit vector indicates which words are initialized. The write counter is initialized to zero, and as words are written to the frame, the F/E bit vector is checked for the corresponding offset in the frame, and if it toggles to 1, the counter is incremented. If there are multiple writes to the same memory address, the F/E bit will stay at 1 and the write counter will be only incremented for the first write. The write counter will reach the size of a page when its corresponding F/E bit vector is all 1, which

---

4The kernel code is shown in Figure 7.

5If there are shared data between the prelude and postlude, the programmer is responsible for ensuring mutual exclusion between them or choosing not to overlap either of them.
indicates that all words in the page are ready. Then, H2D transfer is issued through DMA and the entry is destroyed. Similar to page tables, the DAT entries are stored in DRAM, but recently used entries are cached on-chip. If the on-chip entries are insufficient, an LRU entry is evicted to DRAM and fetched back as needed. The GPU page table has a ready bit that indicates whether the page has been transferred from the CPU and this bit will be set after DMA transfer is done. Until the ready bit is set, memory requests from GPU threads will not be serviced as the data are not available.

This approach can support different types of data sources during the prelude such as a file, network, or host thread computation. However, one limitation is that it assumes all words in the input array will be initialized by the CPU at least once. This is not the case if the input data from an I/O source is partially pre-processed before being copied to the GPU. In that case, since the DAT is not aware of when each page is initialized, the memory copy cannot be automatically done. In such a case, the original data can be first written to a temporary memory region, pre-processed, and written to the final host memory region from which data will be transferred to the GPU. Alternatively, the programmer can still avoid such a modification by choosing not to overlap some part of the pre-processing with kernel execution.

For some workloads, the input data initialization sequence during the prelude can affect the portion of prelude that can be actually overlapped with kernel execution. If the GPU threads access multiple input arrays of similar sizes, it is desirable to interleave the initialization of the arrays during the prelude. For example, if there are two arrays A and B, a chunk of array A is initialized, a chunk of array B is initialized, and then other chunks of array A and B are initialized, and so on. However, even with an arbitrary initialization order, most workloads can still effectively overlap prelude with other stages as shown in Section 5. Another common pattern in GPU workloads is having a small data structure shared by all threads to perform computation on a large data structure (e.g., image filtering). In that case, it is desirable to initialize the small input data first, and then start initializing the large data such that the small shared data quickly become available to all threads.

In the baseline, non-overlapped execution, the postlude on the GPU can only begin after the final kernel’s output data are transferred to host memory. In order to overlap postlude with other stages, our library creates a separate host thread for postlude before the prelude begins. However, the postlude thread should be allowed to access data only after the page is transferred to host memory. This can be done by clearing the valid bits in the CPU page table for D2H copy range at first, and setting the valid bits after the pages are transferred from the GPU. When the host accesses a page that is not ready, a page fault occurs and the thread is blocked by the OS handler until the page is transferred and becomes ready.

4. OVERLAPPING MULTIPLE DEPENDENT KERNELS

This section describes our proposed approach to exploit implicit pipeline parallelism from multiple kernels with software and hardware support. To provide correct overlapped execution, dynamic dependency tracking is performed by CRCS (Section 4.1) on the GPU with support from GPU MMU. It requires a source-to-source translator at compile time to generate profiler code that analyzes the memory access range (Section 4.2). To further improve performance, we propose PPCS (Section 4.4) that takes into account available parallelism while scheduling CTAs from multiple kernels.

4.1 Coarse-grained Reference Counting-based Scoreboarding

Warp schedulers in GPUs use scoreboarding hardware to check data availability and determine which warp to schedule [2]. In order to guarantee correct execution while overlapping different stages in this work, the availability of a page needs to be checked by the scheduler. In addition, for overlapped execution of multiple dependent kernels, the scheduler needs to check if dependencies between threads from the different kernels can cause data hazards in determining the eligibility of each warp instruction. In particular, global memory accesses should occur in program order except for orderings between reads, and the warp scheduler should not allow memory accesses to the same location that can cause RAW, WAR, or WAW hazards. However, identifying and keeping data dependency information for each CTA at a word granularity is not feasible due to the large number of CTAs – e.g., ≈ 9.22 × 10^{18} CTAs per kernel – for recent GPUs[2]. Even at a page granularity, the number of pages that a CTA can access is only bound by the size of the virtual address space and keeping such information is also not feasible. Thus, we employ a per-page, CTA-wise reference counting to minimize the overhead of tracking dependencies between CTAs across kernels. Reference counting is commonly used for garbage collector based resource management [12]. Based on the technique and CTA-granularity dependency tracking, we propose Coarse-grained Reference Counting-based Scoreboarding (CRCS) that augments existing scoreboarding hardware in warp schedulers.

CRCS requires a memory access range pre-profiler (Section 4.2) that analyzes how many CTAs in each kernel access each page and records the information in a reference count table shown in Figure 6, which resides in the GPU DRAM. Each table entry consists of a kernel ID, and write and read CTA reference counts that indicate how many CTAs in the kernel might read from or write to the page. The number of accesses to the page by a CTA does not impact the reference count as it only signifies if a CTA will ever access the page. The reference count is conservative and some of the CTAs that have incremented the counter may not actually access the page. However, if all the reference counts of pages incremented for a CTA are properly decremented when the CTA finishes, the dependencies between kernels can be correctly tracked. It is important to note that false dependencies only impact performance, but do not affect the correctness of our approach.

Each GPU page table entry keeps a pointer to its corresponding reference count table and holds owner kernel ID, and remaining read and write reference counts by current owner kernel. We define owner kernel of a page as the oldest kernel that has unfinished CTAs that have one or more dependencies to the particular page. All threads in the owner kernel can safely access the owned page without causing any

\footnote{For each CTA, the profiler increments the reference count for all pages within the analyzed low and high bounds for each array.}
CTAs) are updated with that of the next kernel. A CTA is ready to be accessed by the current owner kernel any longer. Thus, read CTA counters and write CTA counters are decremented. If both read and write referencing to the type of accesses (read-only, write-only, or read-and-write) are decremented. However, the challenge is that the accessed page range for this CTA obtained by the pre-profiler (Figure 7(b)) is not kept in the GPU memory to minimize overhead. Thus, we append a post-profiler code to the kernel (line 5-22 in Figure 7(c)) to execute it at the end of the kernel for this CTA. This can be done with low overhead since the profiler code only has few instructions. For all the pages in the range, the reference counters corresponding to the type of accesses (read-only, write-only, or read-and-write) are decremented. If both read and write reference counters reach zero, it indicates that the page will not be accessed by the current owner kernel any longer. Thus, the owner kernel ID and the counters (number of remaining W_CTA/R_CTA) are updated with that of the next kernel by looking up the reference count table of the page. When the kernel owner is updated, the MMU notifies other SMs of the change such that the next kernel’s warps can be executed on other SMs. If the current kernel is the last kernel that accesses the page, depending on the mechanism used for overlapping memory copy and the CPU thread, device-to-host memory copy and postlude execution will be done. For the example approach described in Section 3, if the D2H bit in the page table is true, the MMU notifies the DMA engine to transfer the page to the host for the postlude thread.

CRCs is used only for global memory and is not needed for private local memory or other read-only memory such as texture and constant memory. For an irregular kernel, the profiler detects irregularity (Section 4.2) and CRCs only overlaps the current oldest, unfinished kernel with the prelude by looking up the page’s ready bit. Other kernels fall back to the baseline, non-overlapped execution.

4.2 Profiler Code Generation

4.2.1 Sampling Method

While it is possible to statically classify the memory access pattern of a kernel into different categories (e.g., regular/irregular or contiguous/discontiguous) detailed dependency information between CTAs and pages can be dependent on runtime arguments and only available at runtime. Thus, we profile the data dependencies of each kernel at runtime before kernel execution. Our profiler collects dependency information at the granularity of a CTA. In order to identify the pages accessed by each CTA, we leverage the sampling-based memory access range analysis proposed by Kim et al. [20]. Figure 7(a,b) shows examples of an original kernel and a corresponding pre-profiler that will be executed before the main kernel, respectively. This method only requires sampling the memory addresses accessed by threads at the corners of the CTA (i.e., thread index for each dimension of the CTA is either 0 or \( N_d - 1 \), where \( N_d \) is the size of the CTA in dimension \( d \)) to obtain the lower and upper bounds of access range for an array. Specifically, among the memory addresses calculated for sampling, the lowest and highest addresses give the lower and upper bounds, respectively, for the particular array. If an array is accessed multiple times with different index functions (e.g., to perform a calculation with multiple neighboring data elements

---

**Figure 6:** GPU page table with reference counters for Coarse-grained Reference Counting-based Scoreboarding (CRCs). hazard. In the beginning, the first kernel that appears in the reference count table is designated as the owner of the page and the remaining read and write CTA counters are initialized with the reference counts stored in the reference counter table for the kernel. When a warp accesses global memory, scoreboard logic checks if the kernel owns the page to determine the warp’s eligibility. After the instruction is decoded and all accessed memory addresses are known, the scoreboard accesses the SM’s TLB [36] and compares the owner of each accessed page with current kernel ID (stored in a per-SM register). The instruction can be issued only if all accessed pages are owned by the kernel; otherwise, the warp stalls.

During execution, when a CTA is finished, the remaining reference counters of the pages accessed by the CTA are decremented. However, the challenge is that the accessed page range for this CTA obtained by the pre-profiler (Figure 7(b)) is not kept in the GPU memory to minimize overhead. Thus, we append a post-profiler code to the kernel (line 5-22 in Figure 7(c)) to execute it at the end of the kernel for this CTA. This can be done with low overhead since the profiler code only has few instructions. For all the pages in the range, the reference counters corresponding to the type of accesses (read-only, write-only, or read-and-write) are decremented. If both read and write reference counters reach zero, it indicates that the page will not be accessed by the current owner kernel any longer. Thus, the owner kernel ID and the counters (number of remaining W_CTA/R_CTA) are updated with that of the next kernel by looking up the reference count table of the page. When the kernel owner is updated, the MMU notifies other SMs of the change such that the next kernel’s warps can be executed on other SMs. If the current kernel is the last kernel that accesses the page, depending on the mechanism used for overlapping memory copy and the CPU thread, device-to-host memory copy and postlude execution will be done. For the example approach described in Section 3, if the D2H bit in the page table is true, the MMU notifies the DMA engine to transfer the page to the host for the postlude thread.

CRCs is used only for global memory and is not needed for private local memory or other read-only memory such as texture and constant memory. For an irregular kernel, the profiler detects irregularity (Section 4.2) and CRCs only overlaps the current oldest, unfinished kernel with the prelude by looking up the page’s ready bit. Other kernels fall back to the baseline, non-overlapped execution.

**Figure 7:** VectorAdd workload code example with (a) original kernel, (b) pre-profiler code, and (c) kernel with post-profiler code generated by a source-to-source translator appended.
as shown in Figure 3), all of the accesses are taken into account in obtaining the bounds.

The prior work [20] specified the conditions to use the sampling method. As one condition is that the array index is an affine function of CTA ID and thread ID, it cannot be used for irregular memory access patterns (e.g., an array referenced by an index read from another array). In that case, since the exact memory access range cannot be determined before kernel execution, we assume that the kernel can access the entire memory address space. If the kernel includes an indirect memory reference, the profiler does not perform sampling and simply marks the kernel as irregular and passes the information to the GPU for scoreboardng (Section 4.1).

Another condition is that the array reference is not contained in a conditional or branch statement. However, since such a statement frequently appears in many workloads, we release the condition by performing sampling for both cases of a conditional statement to make it usable for such workloads. Since this can only increase the detected memory access range, it still gives the correct range bound while it is more conservative and can result in false dependencies in some cases.

4.2.2 Pre-profiler

We execute the profiler on the GPU since we use the memory access range on the GPU. At compile time, our source-to-source translator takes the original kernel code and generates a pre-profiler kernel. The pre-profiler is launched with the same grid dimensions as the original kernel, but each CTA requires only one thread to calculate the CTA’s memory access range bounds. Thus, the profiler kernel takes the original kernel’s CTA block dimension as an argument in addition to all arguments to the original kernel (line 0-1 in Figure 7(b)). For each array access in the original kernel, access range bounds are calculated and the SetRange function is called once for each array. It takes the starting address of the array, low and high bounds of access to the particular array, and access type (either read-only, write-only, or read-and-write). The function increments the count in the reference count table (Figure 6) for the given access type for all pages within the memory range. If the calculated bounds exceed the region allocated for the array by the CUDA memory allocation API call, the function ignores the out-of-bound region. The dependency information collected by the profiler will be used by our scoreboard mechanism (Section 4.1) to correctly execute the main kernel.

4.2.3 Post-profiler

After each CTA in the main kernel is finished, the remaining reference counters need to be decremented based on the pre-profileredependency. However, the dependency information is not kept in memory to avoid the high overhead. Thus, we re-compute the dependency information for each CTA after it is finished to minimize overhead. This update is done by a post-profiler that is appended to the original kernel by the source-to-source translator. Figure 7(c) shows a modified kernel with appended code to release accessed memory region by decrementing the remaining reference counters (Section 4.1). The same bound calculation as in the pre-profiler is performed but instead of SetRange, the ReleaseRange function is called, which decrements the remaining reference counters in the page table entry for all pages within the bounds. However, since the bound calculation is conservative, the bound can include pages that are never accessed by the CTA and currently owned by an earlier kernel. In that case, the counter in the reference count table is decremented instead. Since the main kernel can have many threads in a CTA unlike the pre-profiler that spawns only one thread per CTA, a conditional statement (line 9 in Figure 7(c)) is added to only execute it with a single thread for each CTA. A synchronization statement (line 6 in Figure 7(c)) is also added to wait for all other threads in the CTA to finish execution. Since the pre- and post-profilers perform the same computation, they both obtain the exactly same memory access range and will not result in incorrect execution (e.g., a kernel indefinitely waiting due to false dependencies). In addition, as SetRange and ReleaseRange are called within a GPU kernel, we assume they can access the GPU page table and update the counters (e.g., through a special instruction).

4.2.4 Implementation and Overhead

Our library will setup a modified version of CUPTI [5] library to automatically launch the pre-profiler kernels through GPU driver support before main kernels are executed. The pre- and post-profilers have minimal overhead as they only involve simple calculations and do not require any memory access except for updating the reference count table. In our evaluation, the pre-profilers represented only 0.85% of the runtime on average (up to 2.7%) for CRCS with FIFO scheduling. To reduce the overhead of the pre-profiler, the profiler kernel can be concurrently executed with any other kernels except for its corresponding main kernel, which can start execution after its pre-profiler.

4.3 Cache Coherence

Current GPUs provide L1 and L2 caches, but they do not employ cache coherence. In particular, a write to the global memory by an SM does not invalidate other SMs’ L1 cache lines and it is allowed for an L1 cache to hold a stale copy. The L1 cache is not flushed when a new CTA from the same kernel is issued to an SM and the programmer is responsible for appropriately handling possible stale L1 accesses across CTAs if it affects the program’s correctness. However, the stale data access should not occur across kernels when multiple dependent kernels are overlapped. Thus, an SM has to flush its L1 cache when all CTAs running in an SM are finished such that the next kernel’s CTAs that will be scheduled on the SM can fetch the most recent data from L2 (or DRAM) instead of accessing possibly stale data in the local L1 fetched by an earlier kernel.

To guarantee correct execution, CTA scheduling also has to be carefully done such that CTAs from different kernels are not executed in an SM at the same time. Thus, scheduling CTAs from the next kernel has to be delayed until all CTAs executing the current kernel on the current SM are finished. Due to this requirement, the maximum number of kernels that can be simultaneously executed needs to be less than or equal to the total number of SMs.
4.4 Pipeline Parallelism-aware CTA Scheduler

Even with CRCS, if a baseline FIFO CTA scheduler is used where older kernel’s CTAs are always prioritized regardless of data availability, GPU utilization cannot be maximized. Thus, our proposed scheduler takes data availability into account to improve pipeline parallelism and performance. With the baseline (non-overlapped execution of kernels), kernels are serially executed one at a time as shown in Figure 8(a) which we refer to as a FIFO (First-In-First-Out) CTA scheduler. In this example, we assume that the execution order is prelude, kernel 0-3, and then postlude, and the CTA \#N is dependent on page \#N from its prior stage (e.g., CTA 0 in kernel 0 is dependent on page 0 initialized in the prelude, CTA 0 in kernel 1 is dependent on page 0 processed by CTA 0 in kernel 0, and so on). With the baseline, a later kernel cannot start execution until all CTAs from its preceding kernel finished execution and the synchronization barrier at the end of the kernel is reached, even if there are idle SMs.

Figure 8(b) shows how FIFO scheduler works for overlapped execution. The CTAs from the first kernel are scheduled on SMs in the beginning (before \(t_0\)), but they cannot start execution until the required pages become ready in the GPU. In addition, there is an overlap of kernels as some CTAs from the second kernel (Kernel 1) are scheduled while some CTAs from the first kernel are still executing (e.g., during \(t_6 - t_8\)). However, the SMs remain idle for a long time since the SMs are dependent on unavailable data (e.g., CTA 4 during \(t_1 - t_4\) that is dependent on page 4). Furthermore, for most of the prelude (\(t_0 - t_6\)), all kernels except for the first are not executed even though there are available data for later kernels. For example, at \(t_1\), the CTA 0 from kernel 0 finishes execution and the data for CTA 0 in kernel 1 become available, but the CTA is only scheduled at \(t_6\).

In order to better exploit parallelism with overlapped kernel execution, we propose Pipeline Parallelism-aware CTA Scheduler (PPCS). Since accurately tracking dependencies for each individual CTA is not feasible due to a large number of CTAs possible, we propose a best-effort scheduler that can be effective for workloads with regular access patterns while the optimal scheduling decision is not guaranteed to minimize complexity. We first define an \(SM\) share of a kernel as the portion of SMs that the kernel is scheduled on. Similarly, \(page\) share of a kernel is defined as the portion of the pages owned by the kernel out of all available pages (i.e., pages that are initialized by the prelude and transferred).

The intuition behind PPCS is that parallelism can be maximized when CTAs across multiple kernels are scheduled such that the SM share of each kernel matches the page share of each kernel. At first, CTAs from the first kernel are scheduled at all SMs in a round-robin manner (i.e., first \(n\) CTAs on SM 0, the next \(n\) CTAs on SM 1, etc., where \(n\) is the maximum number of CTAs that can be scheduled on SM). Next scheduling does not occur until an SM becomes completely idle (i.e., all CTAs executing in that particular SM are finished). This is because our cache coherence policy (Section 4.3) prevents CTAs from different kernels being executed on the same SM simultaneously. Although it is still possible to schedule CTAs from the same kernel on the same SM, such scheduling can prevent fully exploiting pipeline parallelism since other kernels cannot be scheduled. For example, the SM can stay idle for a long time if the currently scheduled CTAs are waiting for the data that will be initialized at a later time by the prelude. Thus, scheduling is done when an SM becomes completely idle. The scheduler computes the SM share and page share of each kernel and chooses the kernel that has the maximum value of \((page\_share - SM\_share)\). Then, CTAs from the chosen kernel are scheduled on the idle SM. PPCS can be implemented with \(O(n)\) complexity, where \(n\) is the total number of kernels, by maintaining a sorted list of kernels with respect to the \((page\_share - SM\_share)\) difference and performing incremental sorting whenever a page’s owner is changed, an SM becomes idle, or another kernel is scheduled.

An example scheduling by PPCS is shown in Figure 8(c). At \(t_1\), SM 0 becomes partially idle, but since there is a CTA still executing on the SM, no other CTA is scheduled on the SM. At \(t_2\), SM 0 becomes completely idle and as kernel 0 decrements reference counters for page 0 and 1, kernel 1 becomes the owner of the pages. At this point, since only page 0 and 1 are available, the page share is 0.0 for kernel 0, and 1.0 for kernel 1. In addition, as no CTA is scheduled on SM 0 and kernel 0’s CTAs are scheduled on SM 1, the SM share is 0.5 for kernel 0 and 0.0 for kernel 1. The difference \((page\_share - SM\_share)\) is 0.0 = 0.5 = –0.5 for kernel 0 and 1.0 – 0.0 = 1.0 for kernel 1. Thus, kernel 1 has a greater difference and CKTs from kernel 1 are scheduled on SM 0. As the SMs become idle, CTAs from subsequent kernels are scheduled and pipelined execution of kernels as well as postlude (for pages processed by the last kernel) is achieved. As a result, the total runtime can be significantly reduced.

However, as PPCS is conservative (i.e., it waits until an SM becomes idle before scheduling another set of CTAs) it...
can perform worse than the FIFO scheduler when all data are available to the GPU. Thus, we only use PPCS while prelude is still in progress and switch to the baseline FIFO CTA scheduler when prelude is finished\(^8\) and all data becomes available to the GPU. For our evaluated workloads, if PPCS was used past the end of prelude, the performance was almost the same as switching to FIFO, but if imbalance between CTAs exists, continuing to use PPCS can degrade performance. The imbalance between CTAs can occur when the CTAs have a different amount of computation to perform. With PPCS, if the CTAs running on the same SM finish at different time, the SM’s resources freed up by a short CTA will remain idle until the other CTAs on the SM are finished. When there is only a single kernel in the workload, we use the baseline FIFO CTA scheduler.

5. EVALUATION

5.1 Methodology

We used GPGPU-sim version 3.0.1 [7] to model the proposed approach with parameters given in Table 1 that results in similar peak performance as NVIDIA Kepler GK110 [2]. For prelude and postlude, we employ a simple model and assume that the data is initialized by reading files from a solid-state drive (SSD) at 500 MB/s [1] to host memory and the postlude writes the output to a file at another SSD with the same bandwidth. We assumed the PCIe bandwidth of 15.75 GB/s in each direction (Gen 3.0 x16) and the PCIe bus is fully utilized during memory copy. We assume 4KB page for both the CPU and GPU. We used clang, a front-end for LLVM compiler [25], to implement our source-to-source translator that receives kernel code as input and generates the profiler code. The runtime is measured from the beginning of the prelude to the end of the postlude. We used workloads with multiple kernels from NVIDIA CUDA SDK examples [4], Parboil [34], Rodinia [10], PolyBench [15], and CUDA example code [19] listed in Table 2. Some workloads were not included because of the limitation of our simulation infrastructure. The evaluated workloads have diverse characteristics in runtime ratios of different stages and we also present a sensitivity study result with varying ratio of kernel execution time. We modified the host code of the workloads to read input from file(s), instead of generating random input in the CPU. For workloads that require multiple input data arrays, we assumed that the initialization of the arrays is interleaved (i.e., one page at a time for each array in turn) but we also provide a result with an adversarial, randomized initialization.

5.2 Performance Improvement

Figure 9 shows the speedup of different execution models compared to that of baseline, non-overlapped execution. In addition to the proposed mechanisms, we present the speedup from completely eliminating the memory copy while other stages are not overlapped. This represents the performance bound of integrated GPUs and full/empty bit approach [27], which only addresses the memory copy overhead. We also show the performance bound of the state-of-the-art that only overlaps a single kernel with other stages while kernels cannot be overlapped. This model assumes that the first and last kernel are completely overlapped with prelude and postlude, and zero memory copy overhead, which may not be possible to realize.

Our proposed mechanisms provided more significant speedups than the bound of prior approaches. While the state-of-the-art was bounded by 11% speedup, combining CRCS with PPCS resulted in an overall speedup of 33% (up to 67%). PPCS resulted in up to 40% (14% overall) higher performance compared to the FIFO scheduler by more efficiently scheduling CTAs from multiple kernels based on data availability. Figure 10 compares the progress of each kernel in HSP during runtime for FIFO scheduler and PPCS. Each curve represents how many CTAs in each kernel have finished execution by the time given in x-axis. The vertical dashed line at the center indicates the time the prelude is finished. With FIFO scheduler, all kernels except for the first one were not well overlapped with the prelude and resulted in a relatively small speedup of 10%. On the contrary, PPCS executed most of all kernels even before the prelude is finished, and resulted in a significant speedup of 50%. While kernels cannot be overlapped. This model assumes that the first and last kernel are completely overlapped with prelude and postlude, and zero memory copy overhead, which may not be possible to realize.

The detection of the end of prelude can be done in different ways depending on how prelude is overlapped with a kernel. For our host model assumed in Section 3, it can be detected when the prelude lambda function (e.g., line 3-5 in Figure 4(b)) is finished.

---

\(^8\)The detection of the end of prelude can be done in different ways depending on how prelude is overlapped with a kernel. For our host model assumed in Section 3, it can be detected when the prelude lambda function (e.g., line 3-5 in Figure 4(b)) is finished.
Figure 9: Speedup with different overlapped execution mechanisms compared to the baseline with no overlap.

Figure 10: Curves showing progress of kernels in HSP workload. Each curve corresponds to a kernel.

(a) Overlapped execution with CRCS + FIFO scheduler.

(b) Overlapped execution with CRCS + PPCS.

Figure 11: Runtime breakdown of the workloads.

Figure 12: Speedup of HSP as the number of iterations (kernels) is varied.

5.3 Sensitivity Study

The fraction of the total runtime for the different stages can impact the potential improvement from overlapped execution. The maximum performance benefit can be obtained if the different stages take a similar portion of total runtime, but if there is a significant imbalance in the portion of different stages during runtime, the performance improvement decreases. Thus, as a sensitivity study, we varied the number of iterations in HSP from 5 to 100 while keeping other parameters constant and compared how performance improvement is varied in Figure 12. All numbers are normalized to that of the baseline, non-overlapped execution.

With 5 and 100 kernels, the kernels took 9.0% and 66% of total runtime, respectively, with the baseline. With 5 kernels, the kernels and postlude were well overlapped with the prelude by PPCS, resulting in a 51% speedup, whereas the performance of single-kernel overlap was bound by a 7% speedup. With 100 kernels, the performance improvement was reduced as the fraction of the prelude is reduced and thus, the opportunity to overlap it with kernels is also decreased. However, PPCS still resulted in a 26% speedup, while the performance of single-kernel overlap was bound by only 2% speedup. PPCS provided better performance than FIFO scheduler by properly scheduling CTAs from the kernels that had available data.

To study the sensitivity of performance to the order of input data initialization (Section 3.2), we randomly initialized the input data during the prelude while initializing the arrays one by one. Performance improvement for CONV, LPC, and HSP was reduced but still resulted in 8%, 20%, and 38%
speedups over the baseline, respectively. For other workloads, the impact of initialization order was very small and the performance was very similar (within 2%).

5.4 Hardware Cost

The CRCS needs a single ready bit for each instruction buffer entry. To support the modified GPU page table structure, assuming that 128-entry TLB [36] keeps a ready bit, D2H copy bit, owner kernel ID (10 bits to support overlapping 1024 kernels), two 10-bit remaining reference counters, and a 32-bit pointer to the reference count table in each entry, the overhead is 1KB per SM. Additional hardware required to support PPCS includes 26 bits of storage per kernel (10-bit kernel ID, 5 and 11 bits to keep the number of SMs scheduled and pages owned, respectively), resulting in 3.25KB overhead for the entire GPU, assuming 1024 kernels. Assuming the baseline GPU has 32KB L1 data cache, 32KB shared memory, 48 KB read-only cache for each of 64 SMs, and 1.5MB L2 cache, the CRCS, and PPCS only results in 0.77% storage overhead for the entire GPU.

6. RELATED WORK

While several prior works addressed the overhead of non-kernel stages, they were limited to only overlapping them with a single kernel. They can be combined with our mechanisms to overlap multiple kernels to further improve performance.

6.1 Memory Copy Overhead

Gregg and Hazelwood [16] showed that memory copy overhead can be large in GPGPU workloads and the speedup by GPU reduces if the overhead is properly considered. Prior work focused on reducing memory copy overhead, but did not enable overlapped execution of multiple dependent kernels. ADSM [14] and DyManD [18] provide automatic memory management between CPU and GPU and overlap memory copy with prelude and postlude. GPUDmm [22] uses the device memory as a cache of host memory and their hardware-based dynamic GPU memory management can overlap data transfer with the kernel execution. COMP [31] proposed several compiler optimizations for Intel Xeon Phi coprocessor that can overlap data transfer with the kernel but is not directly applicable to GPUs. Dymaxion [11] proposes memory remapping and overlapping it with data transfer to hide its overhead. NVIDIA Unified Memory (UM) [17] removes GPU memory allocation and memory copy code by automatically performing them to improve programmability, but can result in performance degradation for many workloads [24]. Zheng et al. [36] proposed several optimization techniques to improve the performance of UM. Integrated GPUs can share the same physical memory with the CPU, and thus, remove the need for memory copy between them [3]. However, discrete GPUs provide much higher performance and still require memory copy.

6.2 Prelude and Postlude Overhead

GPUs [30] and GPUnet [21] enable file and network access within a kernel, respectively, but the kernel needs to be rewritten to use their API. GPUdrive [29] proposed optimized SSD array and software stack to improve I/O performance. HSA [23]-based integrated GPUs can handle page faults from the GPU core, and thus, I/O device access and memory copy can occur within a kernel. Current HSA only supports integrated GPUs and it is unclear how HSA will support discrete GPUs in the future. GPGPU spatial multitasking [6] is proposed to address the underutilization of GPU resources by simultaneously executing multiple workloads. However, it does not necessarily address the resource underutilization due to I/O overhead since the I/O access from different workloads can occur simultaneously. In addition, it does not improve the runtime of each workload.

6.3 Exploiting Pipeline Parallelism from Kernels

There are task-parallel languages designed to exploit pipeline parallelism [35, 8, 33], but the programmer has to explicitly express the dependencies and programmability is degraded. On the contrary, our approach implicitly leverages pipeline parallelism and is based on CUDA and OpenCL, which are currently the de facto standard. While NVIDIA GPUs [2] can simultaneously execute multiple kernels through Concurrent Kernel Execution (CKE), it is limited to independent kernels; dependent kernels still have to be executed one at a time with an implicit synchronization barrier at the end of each kernel. Lustig and Martonosi [27] addressed memory copy overhead with fine-grained full/empty-bit based synchronization to overlap data transfer with kernel execution. For some workloads where multiple kernels do not write to the same memory location, their approach can overlap the kernels. However, most iterative algorithms such as stencil shown in Figure 3 require multiple kernels to repeatedly update memory data in the same region and cannot be parallelized by their approach. In general, overlapping kernels requires properly preventing read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) hazards. To the best of our knowledge, no prior work supports data hazard prevention between dependent kernels while overlapping them. Alternative CTA scheduling [26] has been proposed to improve overall performance but prior CTA scheduling does not consider the impact of overlapping the data movement with the kernel execution.

7. CONCLUSION

In this work, we proposed a hardware-software approach to overlap multiple dependent kernels without the programmer modifying the kernel. The Coarse-grained Reference Counting-based Scoreboarding (CRCS) was proposed to guarantee correct execution while overlapping multiple dependent kernels. In addition, Pipeline Parallelism-aware CTA Scheduler (PPCS) was proposed to improve performance from dependent kernel overlapping by taking into account the amount of data available for the different kernels. Our evaluation results show that combining our approach with existing mechanisms to overlap kernels with non-kernel stages can provide a speedup of up to 67%.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments. This research was supported in part by the Mid-career Researcher Program (NRF-2013R1A2A2A01069132), in part by the MSIP under the ITRC support program (HITP-2016-I8501-16-1005), and in part by Next-Generation Information Computing Development Program through NRF funded by the Ministry of Science, ICT & Future Planning (2015M3C4A7065647).
8. REFERENCES