TwinKernels: An Execution Model to Improve GPU Hardware Scheduling at Compile Time

Xiang Gong Zhongliang Chen Amir Kavyan Ziabari Rafael Ubal David Kaeli
Northeastern University, USA
xgong,zhonchen,aziabari,ubal,kaeli@ece.neu.edu

Abstract
As throughput-oriented accelerators, GPUs provide tremendous processing power by running a massive number of threads in parallel. However, exploiting high degrees of thread-level parallelism (TLP) does not always translate to the peak performance that GPUs can offer, leaving the GPU’s resources often under-utilized.

Compared to compute resources, memory resources can tolerate considerably lower levels of TLP due to hardware bottlenecks. Unfortunately, this tolerance is not effectively exploited by the Single Instruction Multiple Thread (SIMT) execution model employed by current GPU compute frameworks. Assuming an SIMT execution model, GPU applications tend to send bursts of memory requests that compete for GPU memory resources. Traditionally, hardware units, such as the wavefront scheduler, are used to manage such requests. However, the scheduler struggles when the number of computational operations are too low to effectively hide the long latency of memory operations.

In this paper, we propose a Twin Kernel Multiple Thread (TKMT) execution model, a compiler-centric solution that improves hardware scheduling at compile time. TKMT better distributes the burst of memory requests in some of the wavefronts through static instruction scheduling. Our results show that TKMT can offer a 12% average improvement over the baseline SIMT implementation on a variety of benchmarks on AMD Radeon systems.

1. Introduction
Offloading workload to accelerators, such as Graphics Processing Units (GPUs), has become a popular path to high performance due to the massive parallelism available on these devices. However, peak performance is often challenging to achieve, even for hand-tuned GPU applications optimized by experts. One major cause for this challenge is under-utilized compute resources, which are often caused by memory system bottlenecks.

In comparison to computational stalls, memory requests are responsible for a majority of the stalls on many applications targeting a GPU. When requests arrive at the memory system, they have to compete for limited resources, that include: cache blocks, Miss Status Holding Registers (MSHRs), and memory interconnects. This competition can impact the average response time, and can result in request queuing, cache thrashing, and network congestion. When the memory system is saturated, a new memory request has to wait until older memory requests are serviced, resulting in pipeline serialization and long queuing delays. Furthermore, as memory serialization delays an access to the data cache, the possibility of a data eviction and replacement with an address from an intervening wavefront also increases [19]. As a consequence, it may take several attempts to retrieve an address, generating repeated requests to DRAM, resulting in even more traffic in the memory system.

Despite the impact on the service times of memory requests, memory bottlenecks are further exacerbated due to the design of the GPU programming model. The SIMT execution model employed in modern GPU programming frameworks only take advantage of TLP in compute, and pay little attention to long latency memory operations. GPUs use a form of round-robin wavefront scheduling, so the SIMT model essentially ensures that a large number of memory requests will arrive at the L1 cache within a short time window [15]. This can quickly saturate the bandwidth of the GPU’s memory system.

A number of improvements to the wavefront scheduler have been proposed to alleviate this problem [9, 18, 19]. Wavefront schedulers are custom hardware components that dynamically manage the progress of wavefronts, enabling out-of-order execution on GPUs at the wavefront level. There is no requirement that all wavefronts have to follow the same pattern. When working asynchronously, wavefronts can collaborate and access memory and compute resources in a more efficient manner. Early studies in wavefront scheduling have shown that relaxing the strict order can offer considerable performance improvements for GPU applications.

Wavefront schedulers can evaluate the pressure on hardware resources such that wavefronts can run more efficiently. Our approach is inspired by these early success stories, though we relax strict ordering at a different level. We note that for in-order processors, the order of instructions defines
the characteristics of a program. The type of an instruction determines the hardware resources it requires, and its location in the program roughly reflects the time when the associated hardware is accessed. Therefore, we can guide the execution of a program by manipulating the order of instructions, which is determined by the instruction scheduler in the compiler backend.

We propose the Twin Kernel Multiple Thread (TKMT) execution model, a compiler-centric solution that improves collaboration of wavefronts through instruction scheduling. By reordering instructions for selected wavefronts at compile time, TKMT better distributes memory requests. This can significantly reduce competition for bandwidth, and can help alleviate one cause of memory bottlenecks present on current GPUs. Our execution model can offer out-of-order execution capabilities without changing the hardware-based wavefront scheduler. The TKMT model is also an improvement on the SIMT model that preserves the existing memory consistency model. This paper makes the following contributions:

1. We conduct a range of analyses to assess the impact of instruction scheduling on GPU performance. Our experiments show that by using our instruction scheduler with the SIMT model, we can improve performance by 8% on average.

2. We propose a novel TKMT execution model that improves hardware scheduling at compile time; TKMT does not require any changes in APIs, runtime libraries or the application itself.

3. We develop a complete toolchain to realize and evaluate the TKMT execution model. We show how we can reliably achieve better performance with our tools, using a systematic approach to tune the memory performance of GPU applications.

4. We evaluate a variety of GPU benchmarks and conduct extensive experiments to evaluate our approach using a GPU simulator, Multi2Sim [20]. We carry out our evaluation on four different configurations of an AMD GPU, producing a 12% performance improvement on average over the baseline SIMT model.

2. Background

2.1 GPGPU Execution Model

The execution model in popular GPU frameworks, such as OpenCL, CUDA and HSA, is the Single Instruction, Multiple Thread (SIMT) model. In the SIMT model, an instance of a GPU program, termed a kernel in OpenCL terminology, is described by an NDRange. The whole NDRange index space is divided hierarchically into work-groups, wavefronts and work-items. A number of work-items, typically 32 (on Nvidia hardware) or 64 (on AMD hardware), are grouped into a wavefront. Work-items within a wavefront execute the same instruction in lock-step, while work-items from different wavefronts can execute different instructions. Several wavefronts are grouped together as a work-group and are mapped to a compute unit (CU). Wavefronts within a work-group can coordinate through synchronization primitives, and communicate via global and local memory.

2.2 Scheduling on a GPU

Scheduling on a GPU is a multiple-step process [10] that can be static or dynamic. The first step in scheduling occurs in software before launching the kernel. In this step, the original kernel source code is transformed to instructions of the targeted GPU architecture. The order of these instructions is organized by an instruction scheduler in the compiler backend. After launching the kernel, a second scheduling pass takes place, where work-groups are assigned to CUs by a hardware scheduler. The assignment is based on hardware capability, runtime information and requirements specified in the kernel binary, including registers and Local Data Store (LDS) usage. This step is a static process since remapping is not allowed once execution has started. During kernel execution, the wavefront scheduler, a specialized hardware component in the GPU, dynamically schedules wavefronts to execute on Compute Units (CUs). If a wavefront makes no progress due to long latency memory operations, this wavefront is swapped out in favor of a ready-to-execute wavefront, following an algorithm implemented in the wavefront scheduler.

Our work focuses on the software instruction scheduler, and its interaction with other hardware schedulers. Instruction schedulers can arrange the order of instructions in a distinctive manner. Figure 1 shows the range of instruction sequences that were produced by 10 different instruction schedulers, all from compiling the same kernel source. Instructions executing on different hardware units are denoted with a different color. As we can see, patterns differ from each other significantly. However, as they are all generated from the same Matrix Transpose kernel, the functionality is the same.

3. Motivation

3.1 Instruction Scheduling and Performance

In this section, we study the impact of instruction scheduling and present our execution model that takes advantage of the distinct distribution of instructions in a kernel. Instruction scheduling can significantly impact the performance of applications, without requiring any modifications to the hardware. Figure 2 presents execution timelines for our five workloads across five different system configurations. As shown on the left, the two programs used in these workloads contain the same set of instructions. Kernel 1 starts with 2 successive load instructions, followed by 6 ALU instructions. The second load instruction is moved down to line 5 in Kernel 2. For simplicity, we assume each memory
Figure 1. Instruction sequences produced by different instruction schedulers. Colors denote use of different hardware resources. All sequences are compiled from the same Matrix Transpose kernel source.

Figure 2. Execution timeline for five systems. The top shows Kernel 1 running on a system with infinite resources. Second from the top is Kernel 1 executing on a system with 3 Miss Handling Holding Registers (MSHRs). The middle shows scheduling for Kernel 2 on an ideal system. Second from the bottom shows Kernel 2 running on a system with 3 MSHRs. The bottom shows the program executing on the TKMT model.

instruction takes eight cycles to execute and each arithmetic instruction takes one cycle. Only a single wavefront can issue a memory instruction or arithmetic instruction per cycle in these systems. There are 4 wavefronts executing in parallel and these systems only support in-order execution. All systems employ a round-robin policy (across ready wavefronts) to schedule wavefront. The details of the five timelines are explained as follows.

**Example 1: Kernel 1 with infinite resources**: Instructions in Kernel 1 are scheduled such that the first memory instruction is followed by the second, immediately. In this example, the system has infinite resources for memory requests so that memory instructions can issue at any cycle. Therefore, in wavefront 0, the second load does not need to wait for an available MSHR and can issue at t = 5. When the first load returns at t = 8, the add operation can start at t = 9. With unlimited resources, kernel 1 finishes execution in 32 cycles and a maximum of 8 MSHRs are used. Memory operations take 15 cycles to execute and are overlapped for 7 cycles with compute operations, resulting in an overlap ratio of 21.9%.

**Example 2: Kernel 1 with three outstanding requests**: Compared to the previous configuration, the number of outstanding miss requests is reduced to three. Once the number of memory requests exceeds this limit, no new requests can be issued until outstanding requests return. This limitation causes delays in some memory operations. For wavefront 3, the first load has to wait until t = 9, after the first load of wavefront 0 returns. The second load in wavefront 2 proceeds in a similar fashion. With limited resources, the same
kernel takes 42 cycles to execute, while memory and compute operations overlap for 7 cycles. However, the memory operations take 25 cycles in this scenario, which is 10 cycles longer than the first example. As a consequence, the overlap ratio is decreased to 16.7%.

Example 3: Kernel 2 with infinite resources: In Kernel 2, the two memory instructions are separated by three arithmetic instructions. Constrained by the in-order execution design, the second load in wavefront 0 can not issue until \( t = 20 \), even though there is no resource limitation. Memory operations take 22 cycles in total, which is 7 cycles more than the first example, where both have infinite resources. The kernel finishes execution in 40 cycles, while compute and memory operations overlap for 6 cycles. The overlap ratio drops to 15% as a result of the longer execution time.

Example 4: Kernel 2 with three outstanding requests: Similar to Example 2, some memory operations are delayed due to resource limitations. Consequently, memory operations take 34 cycles to finish and the total execution time increases to 41 cycles. This system exhibits poor utilization of memory resources, as we can find that only 1 MSHR is in use for 17 cycles. The memory and compute operations overlap for 24 cycles, which is significantly better than Example 2. However, even with a much higher overlap ratio (58.5%), we cannot hide the long memory latencies. In the end, this setting is only 1 cycle faster than Example 2, where the overlap ratio was a lowly 16.7%.

Example 5: TKMT with three outstanding requests: Kernel 1 and kernel 2 have the same functionality, but their runtime performances are quite different. Using a round-robin wavefront scheduler, Kernel 1 stresses the memory system, but does not have enough computational intensity to hide the memory latency. Kernel 2 works better at latency hiding, but the memory system is underutilized. We need to compensate for the weaknesses in each approach by working together. However, in the traditional SIMT model, all wavefronts must execute the same kernel. We relax this restriction and allow hybrid execute of kernels for wavefronts. We refer to this execution model as the TKMT execution model. More details of our model are discussed later in Section 4. Using the TKMT model, we assign wavefronts 1 and 3 to execute kernel 1, while wavefronts 0 and 2 execute kernel 2. The MSHR configuration supports three outstanding requests, just as in examples 2 and 4.

As seen in bottom timeline, the program finishes execution in 37 cycles when using the round-robin wavefront scheduler. Load instructions finish in 27 cycles and overlap with 15 arithmetic instructions, resulting in a 40.5% overlap ratio. We also observe that the MSHRs achieve higher utilization during those 27 cycles. Using our TKMT model, this system achieves both improved compute-memory overlap and reduced memory latency.

4. Implementation

SIMT is the prevailing computing model for GPU frameworks. SIMT supports only a single kernel to execute at a time. Multiple Program Multiple Data (MPMD) is an execution model that supports multiple programs to process different data, concurrently. Our proposed model is a hybrid combination of the SIMT and MPMD models: the kernels in our model are identical in terms of functionality, but can differ significantly in terms of instruction schedules. To limit complexity, we only support two concurrent kernels, and we name the individual kernels as Twin Kernels. Consequently, our execution model is referred to as a Twin Kernel, Multiple Thread (TKMT) model. In our discussion here, we will refer to the two kernels as the First and Second Twin Kernels.

Our TKMT execution model schedules workloads executing different Twin Kernels to improve the utilization of GPU hardware. The workload-kernel binding can be carried at a wavefront or work-group granularity. When applied at a wavefront granularity, a pair of Twin Kernels is assigned to the set of wavefronts. Each wavefront can choose one of the Twin Kernels to execute. Similar rules apply when working at a work-group granularity. We are not able to work at a work-item granularity since it would require a major overhaul of both hardware and software. Furthermore, executing different instructions for each work-item can impact memory coalescing opportunities, which often lead to performance benefits.

4.1 Memory Consistency

As an improvement to the SIMT model, the TKMT model preserves the current GPU memory consistency model, which is organized hierarchically as follows [4]:

1. Within a work-item, reads and writes to the same address are not reordered by the hardware.
2. For different work-items belonging to the same work-group, memory consistency is only guaranteed by using barrier operations.
3. Consistency is not guaranteed between different work-groups.

The TKMT model supports, and does not change, these rules. First, there is only one instruction order for a work-item, even though instructions are reordered across work-items, but TKMT does not change the in-order design of GPU hardware, so are no behavioral changes in terms of reads and writes.

Second, TKMT uses the same synchronization semantics as in the SIMT model. Memory barrier instructions may be inserted in different instruction slots. They are not removed, which can guarantee that memory consistency is supported within a work-group.

Third, we did not have to add any mechanism to TKMT in order to support consistency across work-groups. By continuing compatibility with the existing model, TKMT pro-
vides a lightweight solution in terms of software and hardware changes.

4.2 The Twin Kernel Compiler

To realize and evaluate the TKMT execution model, we implemented the Twin Kernel Compiler as an open source tool. The Twin Kernel Compiler is comprised of four major components: a Clang-based frontend, an LLVM-based backend, a Twin Kernel Finalizer written in Python, and a Twin Kernel Assembler implemented with Flex and Bison.

We leverage the native support of OpenCL 1.2 compilation in Clang and use it as our compiler frontend. A few modifications were made in Clang to help with platform detection for our compiler backend. The backend in the Twin Kernel Compiler is based on the AMDGPU backend in LLVM [12], which was originally developed to compile OpenGL shaders for open source MESA drivers. Support for OpenCL in the AMDGPU backend is limited. For instance, the design of some important routines, such as calling conventions, load/store instruction lowering and special registers usage, share no similarity with our target system. Therefore, we modified the design and added a few passes for our target system in the backend. The frontend and backend follow the typical design flow of a compiler.

The Twin Kernel Compiler differs from regular compilers as it can produce multiple binaries from a single source file, which is realized by using different instruction schedulers in the compiler backend. Instruction scheduling can take place both before and after register allocation (RA). Our compiler backend contains 5 pre-RA and 2 post-RA instruction schedulers, adopted from LLVM. The details of the instruction schedulers are presented in Table 1. Therefore, there can be up to ten unique assembly files generated from a single source input. We refer to an individual assembly file as a Candidate Assembly.

Candidate Assembly files need to be converted to binary files and formatted to our target system. The binary executable files generated by our TKMT model, which we call Twin Kernel Binaries, are generated in two steps. The first step is a preprocessing performed by the Twin Kernel Finalizer. The Twin Kernel Finalizer is responsible for filtering, selecting and merging kernels, as well as performing some cleanup. To reduce redundancy, the Twin Kernel Finalizer uses permutations, rather than combinations, of the Candidate Assembly files. The Twin Kernel Combiner is specially designed for the TKMT model. If we skip the preprocessing step, the assembly files are produced for the traditional SIMT model. In the second step, the Twin Kernel Assembler takes the output files from Twin Kernel Finalizer and generates executables. These executables are in ELF format in which instructions and metadata information for our target system are encapsulated.

The workflow of the compiler is shown in Figure 3. The frontend takes a single OpenCL 1.2 kernel file as the input and works with the libclc library to generate the corresponding LLVM IR file. The libclc library is an open source library design of OpenCL that is designed to be used with Clang. The compiler backend generates a number of Candidate Assembly files using built-in instruction schedulers. Then the Candidate Assembly is processed by the Twin Kernel Finalizer. The output files are handed over to the Twin Kernel Assembler to generate a series of executable Twin Kernel Binaries.

4.3 The Twin Kernel Binary

GPU compute applications the use our TKMT model are no longer stuck with the monolithic kernel design of the SIMT model. To accommodate our changes without requiring any changes to the existing software stack, we adopt the structure of a standard kernel binary for our Twin Kernel Binary. The format for a Twin Kernel Binary is organized as follows:
Figure 4. Comparison of a Twin Kernel Binary and two regular kernel binaries. All three are generated from a Matrix Multiplication kernel. The two on the left are generated with different instruction schedulers. The Twin Kernel Compiler concatenates two sets of instructions and packs them into a Twin Kernel Binary, as shown on the right. In the Twin Kernel Binary, an extra branch instruction is added in line 0, which branches to the second kernel that starts at line 242. The vector and scalar register usage are set to the maximum of the two.

1. The Twin Kernel Binary is encapsulated into a single ELF-formatted binary as a regular kernel binary. It has the same structure as the binaries for execution on the SIMT model.

2. An additional unconditional branch instruction is inserted before instructions of the First Twin Kernel. The patched First Twin Kernel is concatenated with the Second Twin Kernel, which is labeled as the destination of the unconditional branch instruction. This extra branch instruction is added to facilitate Twin Kernel Assignment, which is detailed later in Section 4.4.

3. As instruction scheduling may change the usage of registers, the candidate with heavier register usage is used for register information in the metadata section. Local memory usage is not affected.

As shown in Figure 4, the Twin Kernels Binary introduces no structural change to the regular kernel binaries. Therefore, there is no change in kernel loading in the software stack. The additional unconditional branch instructions introduce a small overhead, but they do not lead to dramatic changes in performance, as seen in standard thread divergence.

4.4 Twin Kernel Assignment

In our design, assigning different Twin Kernels to wavefronts or work-groups is done by setting the initial program counter (PC). Traditionally, a kernel begins execution with its PC set to 0, and all wavefronts/work-groups are assigned the same kernel. In our TKMT model and Twin Kernel Binaries, wavefronts/work-groups start with PC = 4 (the size of the extra branch instruction) and executes the First Twin Kernel, while wavefronts/work-groups starting with PC = 0 are redirected to execute the Second Twin Kernel. PC initialization determines which Twin Kernel a wavefront/work-group executes. Switching to another Twin Kernel is not allowed during execution as it can cause program corruption.

We use a simple GreaterThan (GT) PC initialization strategy in our design. When TKMT is applied at wavefront granularity, the IDs of wavefronts are used to determine the initial PCs. For GT, if the ID of a wavefront is greater than a threshold, then its PC is initialized to 0, otherwise, it is set to 4. We also allow the ratio of the two PCs to be adjusted in order to provide more precise control. In addition, three other strategies are implemented for comparison purposes: LessThan (LT), Random (RD) and Round-Robin (RR). As the name suggests, LT is the opposite of GT. RD randomly initializes PCs, while RR initializes PCs to 0 or 4 in a round-robin fashion. Four examples are shown in Figure 5 to illustrate these strategies. We assume each work-group only contains one wavefront, and work-groups are scheduled to two CUs in a round-robin fashion. Eight wavefronts are assigned different PCs using the four strategies. Wavefronts, with the PC set to 4, execute the First Twin Kernel while wavefronts with the PC set to 0 execute the Second Twin Kernel. Similar mecha-
PC initialization is the only addition required to the system to support the TKMT model. For a GPU compute framework that exposes special registers to programmers, such as the %warp_id in CUDA, these strategies can be translated to a few extra instructions. The compiler can insert these instructions into the beginning of a kernel, following a similar approach that we used when adding the extra branch instruction. Instead, we choose a hardware design for the PC initialization due to architectural limitations in the modeled GPUs. As no bookkeeping is involved and the algorithm is not complex, our design adds only a very small amount of hardware real estate.

4.5 The Twin Kernel Execution Engine

Given a number of Twin Kernel Binary and an adjustable mix ratio, we choose to use a performance tuning approach, rather than static analysis, to achieve the best configuration. The major reason is that we found no instruction scheduler can perform consistently better than the others. The decision is even more challenging to make in our TKMT model, as we have more binaries to choose from. To achieve the best performance, we need to exhaustively execute all combinations of Twin Kernel Binaries and mix ratios. To accelerate this process, we analyze the exhaustive execution results and make the following observations:

1. The two kernels in the Twin Kernel Binary should use GPU resources as efficiently as possible in order to run fast in SIMT mode. It is unlikely for two slow kernels to achieve high performance when working together.

2. Memory instructions in two Candidate Assembly files should appear at different lines to avoid memory bottlenecks imposed by the round-robin wavefront scheduler. The difference in line numbers is referred to as the Memory Distance. If a pair of memory instructions has a small memory distance in the assembly, most wavefronts will still issue long latency memory requests roughly at the same time and saturate the memory system.

3. Interleaving the usage of hardware units, such as scalar units, SIMD units, and branch units, can improve performance when we face stalls caused by insufficient capacity in these units.

4. It is common that one kernel in the Twin Kernel Binary dominates the performance, so adjusting the mix ratio does not make any difference. We can recognize this pattern by using Trail Execution: executing on three sequential or random mix ratios. If there is no performance difference, we can safely move to other Twin Kernels, otherwise, execution is performed on the rest of the mix ratios.

5. The two kernels in the Twin Kernel Binary may reduce, rather than enhance, the performance of the other kernel. We can also use Trail Execution to detect this pattern and

---

**Figure 5.** Assign Twin Kernels to wavefronts by setting the initial PC. From left to right, we have 4 strategies: GreaterThan (GT), LessThan (LT), Random (RD), and Round-Robin (RR). The ratio of the PCs for each assignment is adjustable, except for Round-Robin.

**Figure 6.** Performance of 5 instruction schedulers in the SIMT model, relative to the average performance of 10 instruction schedulers.
terminate the execution of these combinations in order to save some time.

Based on these observations, we build an execution tool to automate and accelerate the tuning process. The execution tool can filter the unlikely combinations and prioritize execution on combinations with higher potential. The execution tool works in three steps: i) profiling, ii) scheduling and iii) execution. In the first step, we profile regular kernels executing in SIMT mode. Their collaboration potential is statically evaluated based on memory distance and instruction inter-leaving. In the second step, the profiling and evaluation information are considered together to generate an execution schedule. The Twin Kernel Binaries with a higher probability of producing a faster program are given higher priority in this schedule. Following the schedule generated in the previous step, our execution script performs Trail Execution and updates the record when a better combination is found. With our execution tool, the TKMT model is guaranteed to offer better or equivalent performance than the traditional SIMT model.

5. Evaluation
5.1 Methodology
We select 11 benchmarks from the AMD APP SDK v2.9 to evaluate our approach. The execution parameters of selected benchmarks are presented in Table 2. All kernel binaries in our experiments are generated by the Twin Kernel Compiler. The results of our experiments have been verified against CPU implementations using the self-validation feature in these benchmarks.

We model four GPUs within the AMD Southern Islands architecture family using the Multi2Sim simulator [20], a cycle-level heterogeneous CPU-GPU simulator that supports SIMT execution model by default. Details of the GPU configurations are described in Table 3. We extended the timing module in Multi2Sim and add four PC initializers to support our TKMT execution model. The default settings and GPU configurations provided in Multi2Sim are used in our experiments.

Our key figure of merit is the total number of cycles that it takes to execute a kernel with each GPU model, where a smaller number is better. In our experiments, benchmarks have two execution settings: 1) SIMT mode, where benchmarks use the traditional SIMT model and execute regular kernel binaries, 2) TKMT mode, where we use our execution tool to tune benchmark performance. By default, we choose to work at a wavefront granularity and employ the GT strategy to initialize PCs in TKMT mode.

5.2 Results
5.2.1 Performance Improvements
As mentioned in Section 4.5, the choice of instruction scheduler can affect the performance in the SIMT model. Figure 7 (a) shows the speedup for the best schedule for the SIMT model, relative to the worst performance achieved in the SIMT model. The best instruction scheduler can offer up to a 20% performance improvement, as seen in the Binary Search (BS) benchmark. On average, the best instruction scheduler can outperform the worst scheme by 8% across the four GPU configurations, when using the SIMT model.

To fairly compare the performance benefits that TKMT can afford, we compare its performance to the worst and best performing scheduler in SIMT mode. Figure 7 (b) illustrates the speedup of TKMT over the worst performance achieved in SIMT mode. The results in this comparison represent an upper bound of the speedup that TKMT can achieve when compared to SIMT model. TKMT is able to offer up to 1.45x speedup and we observe a 12.3% average performance improvement across all experiments.

The lower bound of speedup for TKMT is presented in Figure 7 (c), where the best performance of SIMT is used as the baseline. The average performance improvement varies from 3% to 7% on different GPUs. In experiments such as MM on the 7770 and MT on the 7970, TKMT offers more than a 10% speedup. For the BS benchmark on the 7970, it achieves a 1.25x speedup. We analyzed the reasons for this using the Multi2Sim visual profiler and found the average response time for memory requests has been reduced by 45%, which shows congestion in the memory system is significantly reduced. Since memory requests are distributed

<table>
<thead>
<tr>
<th>Table 2. Benchmark Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>BinarySearch</td>
</tr>
<tr>
<td>BitonicSort</td>
</tr>
<tr>
<td>DwtHaar1D</td>
</tr>
<tr>
<td>FastWalshTransform</td>
</tr>
<tr>
<td>FloydWarshall</td>
</tr>
<tr>
<td>MatrixMultiplication</td>
</tr>
<tr>
<td>MatrixTranspose</td>
</tr>
<tr>
<td>QuasiRandomSequence</td>
</tr>
<tr>
<td>Reduction</td>
</tr>
<tr>
<td>ScanLargeArrays</td>
</tr>
<tr>
<td>SimpleConvolution</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3. Model Settings for the 4 GPUs evaluated in this work.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU model</td>
</tr>
<tr>
<td>Num of CUs</td>
</tr>
<tr>
<td>L1 Cache/CU</td>
</tr>
<tr>
<td>L2 Cache</td>
</tr>
<tr>
<td>Registers/CU</td>
</tr>
<tr>
<td>LDS/CU</td>
</tr>
<tr>
<td>MSHR/Cache</td>
</tr>
<tr>
<td>Scheduler</td>
</tr>
</tbody>
</table>
more evenly in time, almost all stalls previously present in vector memory unit due to queuing are eliminated.

The upper bound and lower bound of speedup define a range of improvement that TKMT can offer over SIMT, which we refer to as Potential. The Potential of TKMT is presented in Figure 7 (d). The Potential is calculated as the following:

$$\text{Potential} = 100 \times \frac{\text{Cycle}_{\text{simt worst}} - \text{Cycle}_{\text{simt best}}}{\text{Cycle}_{\text{tkmt}}}$$

As shown in 7 (d), the overall Potential of TKMT is 8% across all platforms. The potential of each benchmark is largely dependent on the scope of the reordering that the instruction schedulers can apply. For the BS, BST, MM and MT workloads, the Twin Kernel Compiler produces a diverse set of unique Candidate Assembly files, diverse in the sense that memory instructions have a variety of memory distances. In contrast, for the SLA workload, the choices to produce unique Candidate Assembly are limited and the memory distance is too small to significantly change the overall performance.

5.2.2 Impact of PC Initializer

Our previous evaluations are based on systems with GT initialization. In this section, we evaluate the three other initializers that were described in Section 4.4. Figure 8 shows the results of running TKMT execution on a 7770 GPU with four PC initializers. The performance is relative to the worst performance in SIMT mode. As seen in the figure, the choice of PC initializer has little impact on the performance in general. The performance difference is within 3% for most benchmarks. There are a few exceptions, such as BS, MT and RD benchmarks. However, no PC initializer always outperforms the others across all benchmarks. Considering the complexity of the mechanism, we prefer less complex ones and recommend GT to be used in future implementations.
5. Related Work

5.2.3 Choice of Granularity

As described in Section 4, TKMT can work at a work-group or wavefront granularity. Intuitively, wavefront granularity is preferred, as it offers more precise control. We evaluate the performance of TKMT configured for a work-group granularity on the 7770. The relative speedup of TKMT with work-group granularity is shown in Figure 9, where the performance of TKMT with wavefront granularity is used as the baseline. As expected, we observe a slowdown for most benchmarks when TKMT works at a work-group granularity.

6. Related Work

Warp Scheduling: Warp scheduling has been studied extensively to improve performance on GPUs. Rogers et al. [18] proposed the cache-conscious warp scheduler (CCWS) to improve cache performance by exploiting intra-warp locality. Narasiman et al. [15] described a two-level warp scheduler to minimize the memory access latency. Jog et al. [9] proposed a coordinated CTA-aware scheduling policy to improve performance by reducing cache contention and improve latency hiding capabilities. Rhu et al. [17] designed a locality-aware memory hierarchy that adapts to fine-grained memory access patterns in irregular applications on a GPU. Sethia et al. [19] proposed an advanced memory-aware scheduler that improves the performance of memory-intensive workloads. These studies have shown the potential impact of scheduling and inspire us to explore better scheduling methods.

Thread Block Scheduler: Previous work has highlighted the problem caused by excessive TLP [2]. Kayiran et al. [10] demonstrated that memory contention can be caused by scheduling the maximum possible number of thread blocks. They proposed a mechanism that uses a dynamic thread block scheduling mechanism to reduce such contention. Lee et al. [13] exploited inter-TB locality by scheduling consecutive thread blocks on the same SMX. We borrow the idea of reducing the maximum TLP to alleviate memory contention. However, it is achieved by producing a better distribution of instructions, versus using thread block throttling, as done in the cited previous work.

Concurrent Kernel Execution: As kernels might not fully utilize GPU resources when running alone, executing them concurrently can effectively improve GPU utilization. Software approaches that combine several kernels into a single kernel function has been proposed in previous work [6][21][5]. Pai et al. [16] proposed elastic kernels to achieve efficient execution of concurrent kernels on GPUs. Adriaens et al. [1] proposed GPU spatial partitioning for concurrent kernel execution. Liang et al. [14] utilized concurrent kernel execution to realize spatial-temporal multitasking on GPUs. Our approach can be seen as concurrent kernel execution applied at a finer granularity.

GPGPU Compiler Optimization: Enhancing the performance of GPU compute applications with compiler techniques have been pursued in previous work. Yang et al. [22] presented a GPU compiler framework to address the challenge of effective utilization of the GPU memory hierarchy and judicious management of parallelism. Based on static branch divergence analysis, Coutinho et al. [3] proposed branch fusion that merges common code from a divergent program. Han et al. [7] employed iteration delay, an optimization focuses on divergent branches within a loop to improve GPU performance. Khorasani et al. [11] introduced a software technique named Collaborative Context Collection to overcome the SIMD inefficiency of divergence caused by intra-warp load imbalance or dissimilar task assignment in GPU kernels. Jablin et al. [8] revisited Trace Scheduling for GPUs, which reorganizes the instruction order to reduce the divergence time. Our approach is a more general optimization that applies at a lower level. Therefore, these particular optimizations and our work can work together.

7. Conclusions and Future Work

In this paper, we studied a compiler-centric approach to improve scheduling of GPU hardware. We analyzed the impact of instruction scheduling in the traditional SIMT model and illustrated the importance of instruction schedulers in compilers. We proposed a novel TKMT execution model and constructed a full toolchain to evaluate our model. Our experiments using 11 benchmarks on 4 modeled GPUs show an average improvement of 12% (25% max) performance improvement over the SIMT model.

In the future, we plan to explore hardware/software co-design to improve this collaborative optimization approach. In the TKMT model, the speedup potential is limited by the inherent reorganization opportunities present in the code. We believe we can explore further opportunities using a wavefront scheduler that works more efficiently with our model. In addition, given the static nature of our implementation, we rely on our execution tool to tune performance, which can be slow and inaccurate. Profile feedback using hardware counters and other mechanisms, can provide us with further guidance to accelerate this process.
References


