EffiSha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU

Guoyang Chen, Yue Zhao, Xipeng Shen
Computer Science, North Carolina State University
{gchen11, yzhao30, xshen5}@ncsu.edu

Huiyang Zhou
Electrical and Computer Engineering, North Carolina State University
hzhou@ncsu.edu

Abstract

Modern GPUs are broadly adopted in many multitasking environments, including data centers and smartphones. However, the current support for the scheduling of multiple GPU kernels (from different applications) is limited, forming a major barrier for GPU to meet many practical needs. This work for the first time demonstrates that on existing GPUs, efficient preemptive scheduling of GPU kernels is possible even without special hardware support. Specifically, it presents *EffiSha*, a pure software framework that enables preemptive scheduling of GPU kernels with very low overhead. The enabled preemptive scheduler offers flexible support of kernels of different priorities, and demonstrates significant potential for reducing the average turnaround time and improving the system overall throughput of programs that time share a modern GPU.

1. Introduction

As a kind of massively parallel architectures, GPUs have attained broad adoptions in modern computing systems. Most of these systems are multitasking with more than one application running and requesting the usage of GPU simultaneously. In data centers, for instance, many customers may concurrently submit their requests, multiple of which often need to be serviced simultaneously by a single node in the data center. How to manage GPU usage effectively in such environments is important for the responsiveness of the applications, the utilization of the GPU, and the quality of service of the computing system.

The default management of GPU is through the undisclosed GPU drivers and follows a first-come-first-serve policy. Under this policy, the system-level shared resource, GPU, may get unfairly used: Consider two requests from applications A and B; even though A may have already used the GPU for many of its requests recently and B has only issued its first request, the default GPU management—giving no consideration of the history usage of applications—may still assign the GPU to A and keep B waiting if A's request comes just slightly earlier than B's request. Moreover, the default scheduling is oblivious to the priorities of kernels. Numerous studies [1–4] have shown that the problematic way to manage GPU causes serious unfairness, response delays, and low GPU utilizations.

Latest GPUs (e.g., Pascal from NVIDIA) are equipped with the capability to evict a GPU kernel at an arbitrary instruction. However, the kernel eviction incurs substantial overhead in state saving and restoring. Some recent work [2, 4, 5] proposes some hardware extensions to help alleviate the issue. But they add hardware complexities.

Software solutions may benefit existing systems immediately. Prior efforts towards software solutions fall into two classes. The first is for trackability. They propose some APIs and OS intercepting techniques [1, 3] to allow the OS or hypervisors to track the usage of GPU by each application. The improved trackability may help select GPU kernels to launch based on their past usage and priorities. The second class of work is about granularity. They use kernel slicing [6, 7] to break one GPU kernel into many smaller ones. The reduced granularity increases the flexibility in kernel scheduling, and may help shorten the time that a kernel has to wait before it can get launched.

Although these software solutions may enhance GPU management, they are all subject to one important shortcoming: None of them allow the eviction of a running GPU kernel before its finish—that is, none of them allows preemptive GPU schedules. The request for GPU from an application, despite its priority, cannot get served before the finish of the GPU kernel that another application has already launched. The length of the delay depends on the length of the running kernel. To reduce the delay, some prior proposals (e.g., kernel slicing [6, 7]) attempt to split a kernel into many smaller kernels such that the wait for a kernel to finish

PPoPP '17, February 04-08, 2017, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3018743.3018748

gets shortened. They however face a dilemma: The resulting increased number of kernel launches and the reduced parallelism in the smaller kernels often cause some substantial performance loss (as much as 58% shown in our experiments in Section 8).

In this work, we propose a simple yet effective way to solve the dilemma. The key is a novel software approach that, for the first time, enables efficient preemptions of kernels on existing GPUs. Kernels need not be sliced anymore; they voluntarily suspend and exit when it is time to switch kernels on GPU.

How to enable efficient kernel preemption is challenging for the large overhead of context savings and restoration for the massive concurrent GPU threads. Before this work, all previously proposed solutions have relied on special hardware extensions [2, 4].

Our solution is pure software-based, consisting of some novel program transformations and runtime machinery. We call our compiler-runtime synergistic framework EffiSha (for efficient sharing). As a pure software framework, it is immediately deployable in today's systems. Instead of slicing a kernel into many smaller kernels, EffiSha transforms the kernel into a form that are amenable for efficient voluntary preemptions. Little if any data need to be saved or restored upon an eviction. Compared to prior kernel slicing-based solutions, EffiSha reduces the runtime eviction overhead from 58% to 4% on average, and removes the need for selecting an appropriate kernel size in kernel slicing.

EffiSha opens the opportunities for preemptive kernel scheduling on GPU. We implement two priority-based preemptive schedulers upon EffiSha. They offer flexible support for kernels of different priorities, and demonstrate significant potential for reducing the average turnaround time (18-65% on average) and improving the overall system throughput of program executions (1.35X-1.8X on average) that time-share a GPU.

This work makes the following major contributions:

1) It presents EffiSha, a compiler-runtime framework that, for the first time, makes beneficial preemptive GPU scheduling possible without hardware extensions.

2) It proposes the first software approach to enabling efficient preemptions of GPU kernels. The approach consists of multi-fold innovations, including the *preemption enabling program transformation*, the creation of GPU proxies, and a three-way synergy among applications, a CPU daemon, and GPU proxies.

3) It demonstrates the potential of EffiSha-based schedulers for supporting kernel priorities and improving kernel responsiveness and system throughput.

2. Terminology and Granularity

We use CUDA terminology for our discussion, but note that the problem and solution discussed in this paper also apply to other GPU programming models (e.g., OpenCL).

```
Kernel of matrix addition:
%threadIdx.x: the index of a thread in its block;
%blockIdx.x: the global index of the thread block;
%blockDim.x: the number of threads per thread block

 idx = threadIdx.x + blockIdx.x * blockDim.x;
 C[idx] = A[idx] + B[idx];
```

Figure 1. A kernel for matrix addition.

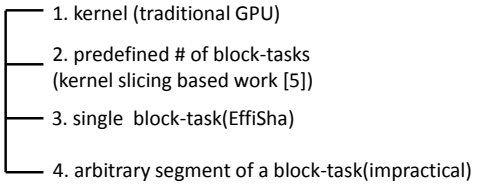

Figure 2. Four levels of GPU scheduling granularity.

A GPU kernel launch usually creates a large number of GPU threads. They all run the same kernel program, which usually includes some references to thread IDs to differentiate the behaviors of the threads and the data they work on. These threads are organized into groups called *thread blocks*.

Note that in a typical GPU program[1], no synchronizations across different thread blocks are supported. Different thread blocks can communicate by operating on the same data locations on the global memory (e.g., reduction through atomic operations), but the communications must not cause dependence hazards (i.e., execution order constraints) among the thread blocks. Otherwise, the kernel could suffer deadlocks due to the hardware-based thread scheduling.

We call the set of work done by a thread block a **block-task**. For the aforementioned GPU property, the block-tasks of a kernel can run in an arbitrary order (even if they operate on some common locations in the memory). Different block-tasks do not communicate through registers or shared memory. The execution of each block-task must first set up the states of their registers and shared memory. Typically, the ID of a thread block is taken as the ID of its block-task. Figure 1 shows the kernel for matrix addition.

Existing GPUs do not allow the eviction of running computing kernels. A newly arrived request for GPU by a different application must wait for the currently running kernel to finish before it can use the GPU [2].

Scheduling Granularity. Scheduling granularity determines the time when a kernel switch can happen on GPU. This section explains the choice we make.

[1] Exceptions are GPU kernels written with persistent threads [8], which are discussed later in this paper.

[2] On some recent GPUs (e.g., the K40m used in our experiments), the hardware sometimes evicts a kernel, but the undisclosed control is entirely oblivious to the priorities of the kernels, OS control, and QoS requirements.

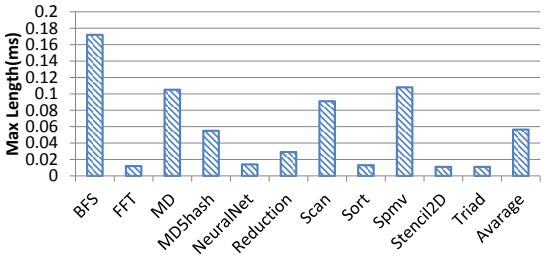

Figure 3. The maximum length of a task in SHOC benchmarks (the largest input in the benchmark suite are used).

Figure 2 lists four granularities for GPU scheduling. At the top is an entire kernel execution; kernel switches can happen only at the end of the entire kernel. This is what traditional GPUs support. All previously proposed software solutions have tried to support a level-2 granularity. The *preemptive kernel model (PKM)* [6], for instance, breaks the original kernel into many smaller kernels, with each processing a pre-defined number (K) of the block-tasks in the original kernel. Even though with this approach the GPU still switches kernels at the boundary of a kernel, the slicing reduces the size of the kernel and hence the granularity. A challenge at this level is to determine the appropriate value of K. Previous work has relied on profiling to do so, feasible to restricted real-time environment, but not for data center-like general sharing environments. It further suffers a dilemma between responsiveness and overhead as Section 8 will show.

The level most flexible for scheduling is the lowest level in Figure 2, where, GPU may switch kernels at an arbitrary point in the execution. The latest generation of GPU, Pascal, supports this level of preemption through hardware. However, the feature is rarely usable in practice due to the tremendous overhead in saving and restoring the state of massive GPU threads [2, 4]. Moreover, the decisions for the preemptions are not software-controllable.

In this work, we design EffiSha to offer the level-3 scheduling granularity. GPU kernel switch can happen at the end of an arbitrary block-task. This choice has several appealing properties. It offers more scheduling flexibility than level 1 or level 2 does. It requires no kernel slicing, and hence, circumvents the difficulty of level 2 for selecting the appropriate K value and the overhead associated with kernel slicing. And finally, unlike level-4, no thread states are needed to save or restore at this level, since preemptions do not happen in the middle of a thread execution.

The size of a block-task determines the worst kernel eviction delay (i.e., the time between the eviction flag is set and the time when the kernel gets actually evicted). Figure 3 shows the maximum length of a block-task in the level-1 SHOC benchmark suite [9]. All block-tasks are shorter than 0.8 milliseconds. The actual eviction delay is even shorter, only 0.08ms on average (detailed in Section 8), much shorter than a typical context switch period in Linux. For the rare

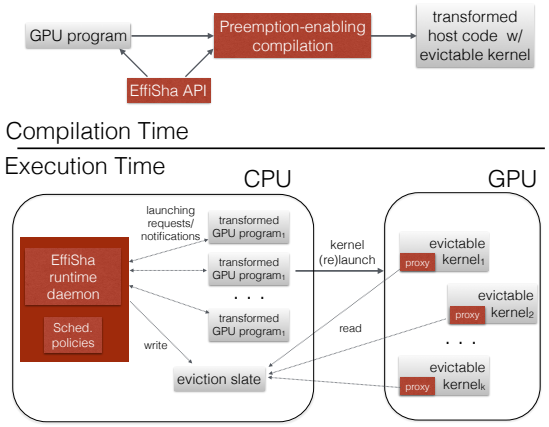

Figure 4. Overview of EffiSha.

cases where a block-task is too large, it is possible to develop some compiler techniques to reduce the granularity by reforming a block-task; the study is left to future.

Terminology clarification: In some (real-time systems) literatures, "preemptive scheduling" exclusively refers to Level-4 scheduling (eviction at an arbitrary instruction). In this article, we adopt a broader meaning of the term, calling a scheduling scheme preemptive if it makes a running kernel evict before its finish.

3. Overview of EffiSha

EffiSha is the first software framework that offers the level 3 scheduling granularity efficiently. As Figure 4 shows, EffiSha consists of four major components, working at the times of both compilation and execution. The compilation step is through a source-to-source compiler that transforms a given GPU program into a form amenable for runtime management and scheduling. It replaces some GPU-related function calls in the host code with some APIs we have introduced, such that at the execution time, those API calls will pass the GPU requests and related information to the EffiSha runtime. It reforms the GPU kernels such that they can voluntarily stop and evict during their executions. The eviction points in the kernels are identified by the compiler such that little if any data would need to be saved and restored upon an eviction.

The EffiSha APIs are mostly intended to be used by the compiler, but could be also used by programmers in a GPU kernel for offering optional hints to the compiler and runtime. Some high-level APIs are designed to allow users to specify the intended priority of kernels.

The EffiSha runtime consists of a daemon on the host side, and a "proxy" of the daemon on the GPU side. The latter is in form of a special thread block of the executing GPU kernel; its creation is facilitated by the *preemption-*

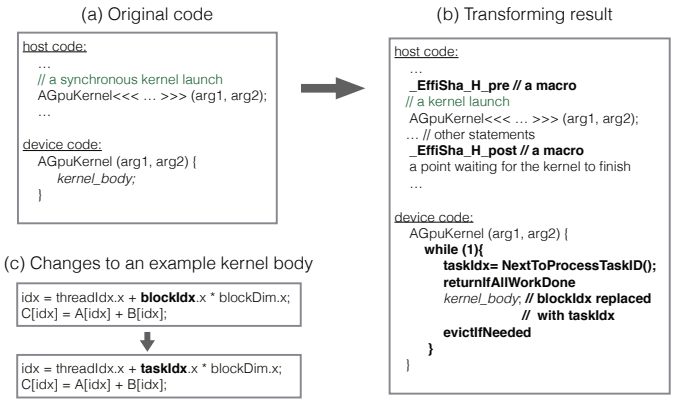

Figure 5. Illustration of the effects of *preemption-enabling* code transformation. Graphs (a) and (b) show the code of a GPU program before and after the transformation; graph (c) uses matrix addition as an example to show the changes to the body of the kernel: from the default thread-block ID indexed form to a block-task ID indexed form.

enabling code transformation done by the compiler. This novel design is key to the low runtime overhead of EffiSha.

The EffiSha daemon receives all the GPU requests from the applications. These requests could be of different priorities. The EffiSha daemon organizes the requests in queues. Based on the scheduling policy that we have designed, it decides when to evict the current running kernel and which kernel to launch next. The interplay between the daemon and the "proxy" of the daemon efficiently notifies the executing GPU kernel when it is time for it to get evicted. Those kernels, thanks to the reformation of their code by the compiler, then voluntarily exit from the GPU. They are recorded in the scheduling queues of EffiSha. When the EffiSha daemon decides that it is time for a kernels to start or resume their executions on GPU, it notifies the hosting process of the kernel, which launches or relaunches the kernels accordingly.

The design of EffiSha ensures a very low overhead of a GPU kernel preemption. We next use an example to first explain how the essential machinery of EffiSha works, and then describe the implementations in the compiler module and the runtime.

4. Preemption-Enabling Code Transformation

Figure 5 illustrates the effects of the *preemption-enabling* transformation to a GPU program. It converts the GPU kernel into a form that uses persistent threads [8] (explained later), and inserts some assistant code into the host side to help with scheduling.

The statements in bold font in Figure 5 are inserted by the EffiSha compiler. A transformation critical for enabling the low-overhead preemption is the changes applied to the device code, shown by Figure 5 (c), which replaces the reference to thread block ID with a block-task ID. An important observation underlying the transformation is that even though in most GPU kernels the ith block-task is processed by the ith thread block, it does not have to be like that. The block-task keeps its integrity as long as it is indexed with i (its task ID), regardless of which thread block processes it.

Based on the observation, the preemption-enabling transformation replaces the variable of thread-block ID in a kernel with a variable of block-task ID as illustrated by the replacement of "blockIdx" with "taskIdx" in Figure 5 (c). By eliminating the tie between a block-task and a thread block ID, it offers the freedom for an arbitrary thread block to process an arbitrary block-task.

With that freedom, in an execution of our transformed GPU kernel, each thread block typically processes not one but a number of block-tasks. That is materialized by the second part of the transformation. As illustrated in the device code in Figure 5 (b), the transformation adds a *while* loop around the body of the kernel such that when a thread block finishes processing a block-task, it tries to process more yet-to-process block-tasks, until all block-tasks are finished or it is time for the kernel to get evicted. Because the threads stay alive across block-tasks, they are called *persistent threads* [8].

An important benefit from using persistent threads is that the transformed program can set the number of thread blocks of a GPU kernel to be just enough to keep the GPU fully utilized. All thread blocks can then be active throughout the kernel execution; global synchronizations among them (e.g., through global memory) hence become possible. It offers some important support to EffiSha as shown later.

Our description has been for the typical GPU kernels that are data-parallel kernels. In some rare cases where the original kernel is already in the form of persistent threads, the compiler can recognize that based on the outmost loop structure, skip this step of code transformation, and move on to the next step.

Low-Overhead Kernel Preemption With the kernel in the form of persistent threads, *preemption-enabling transformation* puts the eviction point at the end of the block-task loop (i.e., the *while* loop in Figure 5 (b).) It gives an appealing property: Upon preemptions, no extra work is needed for saving or restoring threads states. The reason is that since the current block-tasks have finished, there is no need for restoring the threads states. For the block-tasks yet to process, later relaunches of the kernel can process them based on the memory state left by the previous launch of the kernel. (The memory data of the evicted kernel are not copied out or removed from the GPU memory.) The memory state contains the effects left by the earlier block-tasks, including the counter indicating the next block-task to process. If later block-tasks depend on some values produced by earlier block-tasks (e.g., for a reduction), they can find the values on the memory. No extra data saving or restoring is needed upon evictions.

It is worth noting that EffiSha does not cause extra overhead in setting up the states of GPU shared memory. On GPU, by default, the data in shared memory loaded by a thread block cannot be accessed by another thread block. Therefore, in a typical GPU program, each block-task must first load their needed data into shared memory in order to use such memory. It is the same in our transformed code; such load instructions are enclosed inside the "while" loop in Figure 5 (b) as part of the kernel body. Because evictions happen only at the end of a block-task, the shared memory for that block-task needs to be set up only once just as in the default case, regardless of kernel evictions.

The placement of eviction points offer the level-3 scheduling granularity as Section 2 has discussed. This level of scheduling granularity offers more flexibilities than previous software methods do.

As typical as in time-sharing systems, the evicted unfinished kernels and the next-to-run kernel may contend for the space of device memory. Section 6.4 gives further discussions.

In addition to the described code changes, the *Preemption-Enabling* transformation adds two predefined macros respectively at the points before and after the kernel invocation in the host code. It also injects code to create a proxy of the EffiSha runtime daemon on GPU. For their close connections with the EffiSha runtime, we postpone their explanations to the next section where the runtime is explained.

5. EffiSha API and Runtime: Basic Design

The main functionality of the EffiSha runtime is to manage the usage of GPU, making a kernel get launched, evicted, or relaunched at appropriate times. With GPU drivers remaining undisclosed, it is hard for the EffiSha daemon to directly manipulate a GPU kernel inside the context of another process. The runtime management hence involves the cooperations among the EffiSha daemon, the GPU threads, and the CPU processes that host the GPU kernels. A careful design is necessary to ensure that they work together smoothly and efficiently.

To ease the understanding, our description starts with a basic design. It conveys the basic idea, but lacks the support to multiple GPU streams, and suffers large overhead. Section 6 describes the enhanced design and optimizations that address those limitations. For now, the description assumes that each application has only one GPU stream (i.e., one sequence of GPU kernels to run on the GPU).

5.1 GPU-Stubs and State Transitions

We first introduce a runtime data structure named *GPU-stubs* and the states that a GPU kernel could have and their transitions. They are essential for understanding the runtime cooperations.

The EffiSha daemon hosts a number of *GPU-stubs*. Each stub holds a record for a kernel that is yet to finish. As

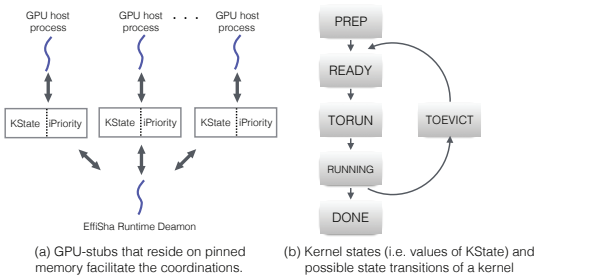

Figure 6. GPU-stubs and the possible state transitions of a GPU kernel.

Figure 6 (a) shows, a GPU-stub contains two fields: One is *KState*, which indicates the current state of the kernel, the other is *iPriority*, which indicates the initial priority of the kernel.

Each GPU-stub is a small piece of (CPU-side) shared memory created by the EffiSha daemon. When an application requests a GPU-stub for a GPU kernel call (through an EffiSha API call), the daemon allocates a stub for it, maps the stub to the address space of the host thread, and returns the address to the host thread. The usage of shared memory allows the stub to be both read and written by the daemon and the host thread directly. We have also considered an alternative design which leaves the stub in the daemon space; the host thread then must access the stub through system calls. It incurs much more overhead because the stubs are frequently read by the host thread as we shall see in the next section.

The field *KState* in a GPU-stub can have one of six values, corresponding to all the six states that a GPU kernel can possibly have. Figure 6 (b) shows the set of states and the possible transitions among them. A GPU-stub is created before the host thread reaches kernel invocations; it is in a preparation state (PREP). Before a kernel invocation, it transits to the READY state, indicating that a kernel is ready to be launched. When the EffiSha daemon changes its state to TORUN, the host thread may launch the kernel and the state of the kernel becomes RUNNING. When the EffiSha daemon changes its state to TOEVICT, the GPU kernel evicts from GPU voluntarily and the kernel goes back to the READY state. When the GPU kernel finishes, the kernel state turns to DONE and the GPU-stub gets reclaimed for other usage.

5.2 Basic Implementation of the Runtime

We now explain the basic implementation of the EffiSha runtime, and how it supports the state transitions for kernel scheduling. Our description follows the computation stages and state transitions shown in Figure 7. On the right side of Figure 7, there is the transformed GPU program code, which reveals the details of some macros showed in Figure 5; on the left side of Figure 7 are the states of the kernel corresponding to each of the computation stages.

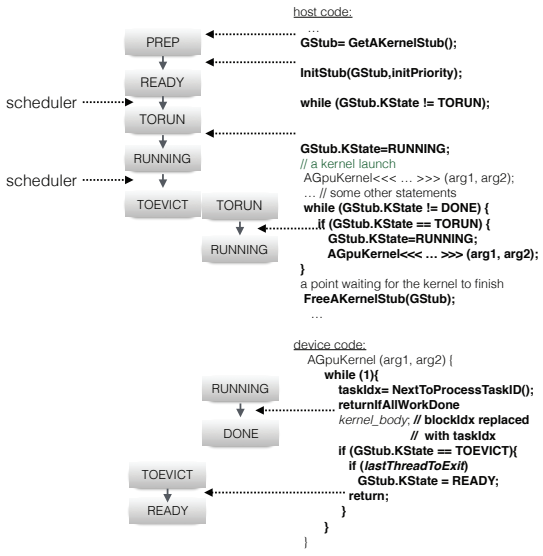

Figure 7. Changes of the KState of a kernel in each stage of the execution of a GPU program (bold font shows the code inserted by the EffiSha compiler).

The host thread calls the API "GetAKernelStub" to obtain a GPU-stub for the next kernel call. At the creation time, the KState in the stub equals "PREP". The API "initStub" sets the "iPriority" field of the GPU-stub with the initial priority (also called static priority) of the kernel; if it is "null", the kernel uses the default priority. The call also changes the state of the kernel to "READY". After that, the host thread starts to poll the *KState* in the stub until the EffiSha runtime daemon changes its value to "TORUN". The host thread then changes the *KState* to "RUNNING" and launches its kernel.

The bottom of Figure 7 shows the device code and state transitions. During the execution of a GPU kernel, when a GPU thread finds no more block-tasks left, it changes the field *KState* of the stub to "DONE". Otherwise, it processes its work, and then before it tries to process a new task, it checks whether the *KState* is now "TOEVICT". If so, the thread exits; the last thread to exit, changes the *KState* to "READY" such that the kernel can be relaunched later. The voluntary evictions require no saving of the GPU thread states as they happen only at the finish of some block-task. In the implementation, GPU-stubs are mapped to pinned memory such that GPU threads can also access it.

After launching the GPU kernel, the host thread continues its execution[3]. Right before it reaches the next waiting point, it gets into a *while* loop, in which, it repeatedly checks the *KState* until it becomes "DONE". During the check, the kernel may get evicted and later get scheduled by the EffiSha runtime to resume, in which case, the runtime changes *KState* to "TORUN"[4], and as soon as the host thread sees it, it relaunches the kernel.

6. Enhanced Design and Optimized Implementation

The basic design and implementation enables evictions and relaunches of GPU kernels, but is subject to two-fold limitations: applicability and efficiency.

6.1 Enhancing Applicability

The most important limitation in applicability is that the basic design cannot work with GPU streams. Modern GPU introduces streams for allowing concurrent executions of multiple kernels (of the same application) on a GPU. Each stream may contain a sequence of kernel launch events. Kernels in different streams can run concurrently on a GPU while kernels in the same stream get launched in serial. The basic design of EffiSha tracks the state of an individual kernel but is oblivious to streams. It cannot maintain the execution properties of multiple streams (e.g., letting multiple kernels from different streams run concurrently). Our enhanced design extends the basic design.

In our design, all streams of an application share a single priority. The choice comes from the property of current GPUs. On current GPUs, kernels from two different GPU applications cannot run concurrently on a GPU, and hence, at an eviction time, all streams of an application have to get evicted before another application can use that GPU. The single-priority restriction could get relaxed for future GPUs when more advanced co-run support is introduced.

In the enhanced design, a GPU-stub is for the entire GPU application rather than for each individual kernel. Its *KState* field now indicates the status of the GPU request from that whole application. Moreover, GPU-stub contains an array of queues besides *KState* and the initial priority. We call the queues *kernelQ*s. The host code is transformed such that a *kernelQ* is created in the GPU-stub right before the creation of a GPU stream. And right before the code for launching a GPU kernel, the ID of the kernel is added into the *kernelQ* of that kernel's stream[5]. As kernel launch commands are asynchronous, the host thread may execute multiple such commands in a sequence, leaving multiple kernel IDs in a *kernelQ*.

There are three other changes: (1) At a kernel launch point, the compiler does not make the code transformations as in the basic design. Instead, it replaces CUDA kernel launch calls with calls to a customized launch API, the implementation of which puts the ID of the to-be-launched ker-

[3] Here, we assume asynchronous kernel launches. If the original kernel launch is synchronous, the compiler changes it to asynchronous and adds a waiting point right after it.

[4] Through atomic operations, it first checks to ensure that *KState* is "READY" state.

[5] In CUDA, stream ID is part of the launching parameters. When it is omitted, the default stream (stream 0) is used. For CUDA 7, each host thread has its own stream 0; on earlier CUDA, one application has only one stream 0.

```
while (GStub.kernelQ[s]!=empty){
   if (GStub.KState == TORUN){
      GStub.KState = RUNNING;
      launch the kernels residing at the heads of all kernelQs.
   }
}
```

Figure 8. Implementation of the customized API for a synchronization on stream s.

nel into the corresponding *kernelQ* and records the kernel parameters for later launches or relaunches to use. That API does not actually launch the kernel as launch time is determined by the runtime scheduler. (2) When the current kernel is done, the device code dequeues that kernel from the *kernelQ* of its stream. It does not set *KState* to DONE unless all *kernelQ*s of the GPU-stub become empty. (3) At synchronization points, the compiler does not insert the code as those inserted in the basic design. Instead, it replaces all GPU device synchronization calls with our customized synchronization API. CUDA has two kinds of stream synchronizations: one is to wait for all the kernels in a particular stream to finish, the other for all streams of the program. For the former, the implementation of our API is as shown in Figure 8. As long as the runtime scheduler permits (indicated by *GStub.KState*), it lets all the streams of the program run on GPU until the *kernelQ* of the to-be-synchronized stream becomes empty (i.e., all kernels in that stream have finished). For the other kind of synchronizations, the implementation is similar except that the exit condition of the "while" loop is that *GStub.KState==DONE*. It is worth mentioning that this enhanced design also avoids some code analysis complexities facing the basic design: The code transformations in the basic design at synchronization points need to know which kernel invocation the points correspond to. It could be complex to figure out if the synchronization points are in a different device function or compilation unit.

6.2 Improving Efficiency

The second main issue of the basic implementation is that it adds some substantial runtime overhead, causing significant slowdowns (up to 97X) to a program execution (shown in Section 8). To make it work efficiently, we have developed three optimizations as follows.

Proxy on GPU In the basic implementation, *KState* is made visible to all three parties: the daemon, the host thread, and the GPU threads. That simplifies the coordinations among them, but is not quite efficient. The main issue is on the repeated accesses to *KState* by each GPU thread. Because the accesses have to go through the connections between CPU and GPU (e.g., through a PCIe link), they cause significant slowdowns to some kernels, especially those that have small block-tasks (e.g., 97x slowdown on a Triad kernel).

We optimize the implementation by creating a special kernel on a special stream for each GPU application. The kernel serves as the proxy of the EffiSha runtime, running on the GPU along with the other streams. In this implementation, there is a copy of *KState* on the GPU memory. The proxy keeps reading the *KState* on the host memory and updates the copy on the GPU accordingly. With that, the threads in other GPU streams just need to read the local copy to find out the time to get evicted. The other job of the proxy is to monitor the status of the kernel (e.g., by reading the block-task counter) and to update the *KState* on the host memory when necessary. The proxy only needs one GPU thread, consuming the minimum resource. It is launched whenever a kernel of the application gets launched.

Adaptive Checking The use of the proxy frees the other thread blocks from accessing the remote host memory repeatedly, which reduces the overhead substantially. But we still observe 15% overhead for kernels with fine-grained block-tasks. We add the feature of adaptive checking to further reduce the overhead. The idea is to let a GPU thread check the local copy of *KState* in every k iterations of the device-side *while* loop in Figure 7, where, k is set to $r * L/l$, L is the length of the scheduling time slice, l is the estimated length of each iteration of the device *while* loop, and the factor r is a positive integer, which helps amortize the influence of the errors in the estimation of l by making several checks per time slice (we use 10 in our experiments). There could be various ways to make the estimation, through performance models, offline or online profiling, or program analysis. Our implementation uses simple online sampling, which, at runtime, measures the length of the first iteration of the *while* loop executed by the first thread of each thread block and uses the average as the estimated value of l.

Together, these optimizations reduce the overhead by 11.06% as detailed in the evaluation section.

Asynchronous Checking In the enhanced design, at a device/stream synchronization point, there is a *while* loop which continuously checks *KState* or *kernelQ* to relaunch the kernels if they are evicted. One potential issue is that if there is a lot of CPU work existing between the original kernel launch point and the synchronization point, the kernel could have got evicted long before the host thread reaches the *while* loop, and hence suffers a long delay in the relaunch. The situation is rare according to our observations. Still, to alleviate the potential consequence, the EffiSha compiler inserts a custom signal handler into the GPU program. When the runtime daemon sees a long delay (about 100 microsec) for *KState* to turn into RUNNING from TORUN, it sends a signal to the host thread. The signal handling function changes *KState* to RUNNING and launches the kernel.

6.3 Setting Priority

The *iPriority* of a GPU-stub has a default value. A programmer may use an EffiSha API call (*setIPriority*) to alter the priority. The call is mainly to pass the info to the compiler, which uses it as the second parameter of the inserted call of

initStub to set the actual *iPriority* field of the GPU-stub at run time.

Similar to the priority control in Linux for CPU threads, only the superuser (root) may set a higher priority than the default. A caveat is that even though EffiSha runtime daemon respects the priority when scheduling kernels, the actual effect is also related with the priority of the host thread because (re)launches are through them. For instance, if when the EffiSha daemon changes the *KState* to TORUN, the host thread is in CPU waiting queue, the kernel cannot get relaunched until the host thread resumes running. A low-priority host thread could hence result in much (re)launch delay. To avoid it, a suggestion to superusers is to ensure that the host thread has a priority no lower than the kernel.

6.4 Memory Size and Dynamic Parallelism

As in many time-sharing systems, the evicted unfinished kernels and the next-to-run kernel may contend for the space of device memory. It is an issue only if the remaining memory cannot meet the requests of a GPU program. The solution is to let the compiler replace GPU memory allocation calls in the host code with a customized API. Through it, upon the failure of a memory allocation, the runtime scheduler puts that application into a waiting queue, and resumes it when an unfinished application has finished and hence released some space. If there are no unfinished applications, upon an allocation error, the API returns the error immediately as the shortage is due to the device limit.

Modern GPUs support dynamic parallelism (i.e., kernel launch inside a kernel). EffiSha gives no explicit treatment to dynamic parallelism, for two reasons. First, dynamic parallelism on GPU is rarely used in practice due to its large overhead (as much as 60X) [10, 11]. Second, a recent work [10] shows that dynamic parallelism in a kernel can be automatically replaced with thread reuse through *free launch* transformations, and yields large speedups. Therefore, for code with dynamic parallelism, one may apply *free launch* first to remove the dynamic parallelism and then apply Effisha.

7. Scheduling Policies

The support provided by EffiSha opens up the opportunities for the development of various preemptive schedulers for GPU. We demonstrate it by implementing the following two scheduling policies.

1) Static priority-based preemptive schedule. In this schedule, the EffiSha runtime daemon schedules kernels purely based on their initial priorities. The scheduler maintains a waiting queue for applications that are waiting to use GPU. The applications in the queue are ordered in their priorities. When the GPU becomes available, the scheduler always chooses the head of the queue (i.e., the one with the highest priority) to launch. If a new coming kernel has a higher priority than a currently running kernel, the scheduler immediately has the current kernel evicted and has the new kernel launched. Otherwise, the currently running kernel runs to completion.

2) Dynamic priority-based preemptive schedule. The schedule policy is similar to the CPU scheduling policy used by recent Linux systems. It maintains two waiting queues, one active queue and one inactive queue. This schedule uses dynamic priority. The dynamic priority of the GPU request from an application (including all of its streams) equals to its initial (static) priority when its kernels just arrive, but gets increased by one for every millisecond the request waits in the active queue. The increase is capped at 20. In each of the two queues, requests are ordered by their dynamic priorities. A new coming request is put into the active queue (positioned based on its priority).

Every time a request is scheduled to be met, it gets a time slice (or called *quota*). The length of the time slice is weighted by the priority; it is set to $(p+1)/2$ ms, where p is the priority of the kernel. At the expiration of its time quota, the GPU request goes into the inactive waiting queue and its dynamic priority is reset to its initial (static) priority. When a GPU becomes available, the scheduler chooses to meet the request at the head of the active waiting queue. When the active queue becomes empty, the two queues switch the role: The inactive queue becomes active, and the old active queue becomes the inactive queue. The dynamic priority can help prevent starvations facing the static priority-based scheduler.

8. Evaluation

We implement the EffiSha system based on Clang [12], a source-code analysis and transformation tool. We conduct a set of experiments, including comparisons to PKM [6], a prior kernel slicing-based software solution. We try to answer the following main questions:

1) *Preemption.* Can EffiSha indeed enable preemptions? How much delay does the restriction on preemption granularity cause for an eviction to take place?

2) *Priority.* Can an preemptive scheduler based on EffiSha indeed support the different priorities of different kernels? Can it help improve the overall responsiveness and throughput?

3) *Overhead.* How much time overhead does EffiSha add to GPU executions? What is the breakdown?

4) *Comparison.* How doe EffiSha compare with the state-of-art software approach?

8.1 Methodology

Our experiments use all the programs in the level-1 folder of the SHOC CUDA benchmark suite [9]. Table 1 lists the benchmarks.

We design an experimental setting to emulate the scenarios where the requests for GPU from different applications may have different priorities, and some requests may arrive while the GPU is serving for other applications. In our experimental setting, the 11 benchmarks issue their requests

Table 1. Benchmarks

Benchmark	Description
BFS	Breadth-first Search in a graph
FFT	1D Fast Fourier Transform
MD	Molecular dynamics
MD5Hash	MD5 Hash
NeuralNet	A Neural Network
Reduction	Summation of numbers
Scan	An exclusive parallel prefix sum of floating data
Sort	A radix sort on unsigned integer key-value pairs
Spmv	Sparse matrix-dense vector multiplication
Stencil2D	2D 9-point stencil computation
Triad	A version of the stream triad

Table 2. Kernel lengths and priorities.

Benchmark	Standalone	Priorities		
	time (ms)	random	group	SJF
NeuralNet	14.25	2	2	1
Sort	5.46	17	2	4
Reduction	2.06	17	2	7
MD5Hash	3.29	22	5	6
MD	13.8	22	5	2
Scan	1.41	7	5	8
Triad	1.22	3	8	9
Stencil2D	28.4	20	8	0
FFT	1.17	24	11	10
Spmv	4.57	7	11	5
BFS	5.99	1	11	3

for GPU (i.e., launching a GPU kernel) in turns. Another application issues the request 3ms after the previous application issues its request. The launch order is chosen arbitrarily, as shown as the top-down order in Table 2. (The priority settings are explained later.)

We run the experiments on a machine equipped with an NVIDIA K40m GPU running CUDA 7 and an Intel Xeon E5-2697 processor. Without noting otherwise, in the experiments, the baseline of the comparisons is always the executions of the original benchmarks without any modifications[6].

8.2 Support for Preemptions and Priorities

We run the 11 benchmarks in all the settings under the preemptive schedule. The programs all run correctly, and kernels get evicted as expected by the scheduling policies.

We further conduct the following experiment to validate the appropriate support of priorities by EffiSha-based preemptive scheduling. We launch the 11 benchmarks for 1000 times and measure the total number of evictions for each kernel. Every launch is assigned with a random priority from 1 to 9 (the higher the more important the kernel is). Initially, all the kernels are launched at the same time. A kernel gets invoked again a certain time period (which is called "wait_time") after the finish of its previous invocation. We change the wait_time from 1ms to 64ms. We use the static-priority–based preemptive scheduling policy for this experiment; the fixed priority makes it easy to analyze the results.

[6] We observed that when multiple kernels request a GPU, the K40m GPU, by default, periodically switch its tenants. The details are not disclosed officially; our observation is that it uses an SM-draining mechanism. It offers no software controllability.

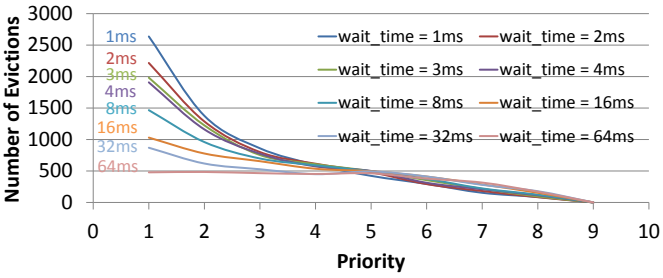

Figure 9. Total Number of Evictions for Different Priorities

Figure 9 shows the number of evictions that happen on the kernels at each level of priority. Because every launch gets a random priority, the total lengths of the kernels at the various priorities are similar. However, the kernels at lower priorities gets clearly more evictions than those at higher priorities. When "wait_time" is 1ms, the launches of kernels are dense, and there are a large number of evictions for kernels at the low priorities. As "wait_time" increases, fewer conflicts happen among the requests for GPU, and the numbers of evictions drop. For a given "wait_time", the higher the priority of a kernel is, the less frequently the kernel gets evicted. These observations are consistent with the objective of the priority-based scheduling policy, confirming the feasibility enabled by EffiSha for scheduling GPU kernels to accommodate kernels of different priorities. Extra evidences are shown in the experiments on the dynamic priority-based scheduling (which will be presented later in Section 8.4).

8.3 Eviction Delay

Since the preemption happens at the end of a block-task, there could be some delay between the setting of the eviction flag and the exit of the kernel from GPU. We call it *eviction delay*. The delay indicates how quickly the running kernel can act on an eviction request. This part reports the eviction delay of each kernel observed in our experiments. To measure it, we keep each kernel doing the same work on GPU as it does in its normal runs, but repeatedly. At some moment, the eviction flag is set to true. An inserted cudaStreamSynchronize() call in the host code makes the host thread wait for the kernel to exit to measure the latency. Figure 10 shows the observed eviction delay of each kernel. *MD* has the longest eviction delay (0.4ms) because of its relatively large block-task, while *BFS, FFT, NeuralNet* and *Triad* have delays less than 0.015ms. The average delay is about 0.08ms, much smaller than the typical length of a context switch period in Linux.

Besides eviction delay, there are some other overhead in the preemptive executions, including relaunch overhead, state checking overhead, and so on. We next report the experimental results of EffiSha-enabled priority-based preemptive scheduling, which shows that these delays do not prevent the scheduler from working profitably. And Section 8.5 analyzes the delays in detail.

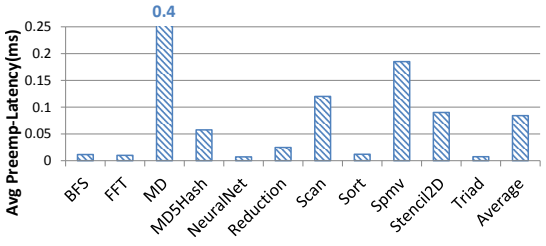

Figure 10. Eviction delay of each kernel.

8.4 Benefits from Preemptive Scheduling

The efficient preemption enabled by EffiSha opens the opportunities for the design of various preemptive schedules for GPU kernels. In this part, we use the aforementioned dynamic priority-based preemptive schedule as an example to illustrate the substantial potential benefits of such schedules. The reported timing results include all runtime overhead.

Metrics Following common practice [13], we use two metrics to characterize co-run performance:

1) Average normalized turnaround times (ANTT). NTT is defined as follows [13]:

$$NTT_i = {T_i}^{MP}/{T_i}^{SP},$$

where, ${T_i}^{SP}$ and ${T_i}^{MP}$ are the execution time of a kernel in its standalone run and its co-run. NTT is usually greater than 1, the smaller the more responsive the kernel is. ANTT is the average of NTTs of a set of programs.

2) System overall throughput (STP). STP is defined as follows. STP is from 0 to n (the number of programs); the larger, the better.

$$STP = \sum_{i=1}^{n} {T_i}^{SP}/{T_i}^{MP}.$$

Results In our experiments with the scheduling, we launch the benchmarks one by one by following the scheme described in Section 8.1. We use three settings of priority assignments. The first is to assign priorities to the kernels in a shortest job first (SJF) manner, in which, the priority of a kernel is based on its length (i.e., "standalone time" in Table 2); the longer, the lower, as shown in the "SJF" column in Table 2.

It is well known that an SJF schedule minimizes the total waiting time of a set of jobs, giving superior responsiveness [14]. However, without priority-based preemption support, the default scheduler on GPU fails in harvesting the benefits. As the "default" bars in Figure 11 show, the NTT of the kernels range from 2 to 15, and the high-priority kernels (towards the right end in the graph) tend to suffer the largest NTTs. It is because they have the shortest lengths (i.e., smallest T^{SP}) and hence their NTTs are more sensitive to waiting times. The EffiSha-enabled preemptive scheduler substantially reduces the NTTs for most of the kernels. As

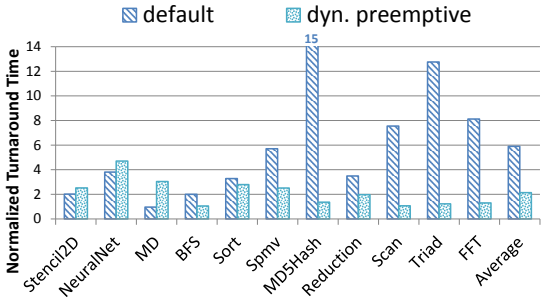

Figure 11. Normalized turnaround time; SJF priorities are used. The benchmarks are put in an increasing order of their priorities (from left to right).

the "preemptive" bars in Figure 11 show, the NTTs are lowered to below two for most of the kernels. The reduction is especially significant for the high-priority kernels which receive a high priority treatment in the preemptive scheduler. For instance, when *FFT* requests GPU, the preemptive scheduler immediately raises up the preemption flag and the currently running kernel gets evicted quickly; the *FFT* kernel gets to run and finish on GPU soon after its request is issued. Its NTT is lowered from 8.1 to 1.15. The NTTs of some kernels on the left end in Figure 11 increase slightly, for the low priority treatment they get in the scheduling.

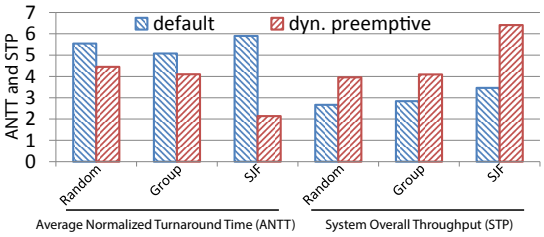

Figure 12. Average normalized turnaround time (ANTT) and overall system throughput (STP) under three priority settings.

On average, the preemptive scheduler cuts the ANTT by 65% (from 6 to 2), and improves the overall throughput by a factor of 1.8 (from 3.5 to 6.3), as reported in Figure 12. Figure 12 also reports the overall results under two other settings of priorities. The "random" scheme assigns an application with a random integer between 0 and 30 as the priority of its kernel. The "group" scheme assigns adjacent two or three applications with the same priority. The priorities of the kernels in both settings are shown in Table 2. Although the benefits of the preemptive scheduler in these settings are not as large as in the "SJF" setting, the improvements are still substantial: 18% ANTT reduction and 1.35X throughput enhancement in the "random" case, and 24% and 1.42X in the "group" case.

8.5 Overhead and Comparison

EffiSha may introduce several sources of overhead. We categorize the overhead into two classes:

1) Transformation overhead: Overhead incurred by code changes, including the usage of persistent kernels, the creation of the GPU proxy, the checks on the eviction flag, and other changes and their effects that the EffiSha compiler makes to the host code and kernels.

2) Scheduling overhead: Overhead incurred by preemptive scheduling. It mainly includes the time taken by kernel evictions and resumptions, and their side effects on the runtime executions.

We measure the overhead of each class respectively.

Transformation Overhead Transformation overhead refers to the overhead brought by the transformed code and the introduced runtime daemon and proxy. For this measurement, we run each of the transformed benchmarks alone; during the execution, the daemon (and proxy) works as normal, except that there is no kernel evictions since no other kernels request the GPU. We compare the execution time with that of the original unmodified kernels to compute the overhead.

Figure 13 reports the transformation overhead in three cases: the basic EffiSha design, the design with the use of proxy, the design with the use of proxy and all other optimizations that have been mentioned in Section 6. The first group of bars in Figure 13 show the overhead of the basic EffiSha implementation. The overhead differs for different applications, due to the differences in their memory access intensities and the length of each block-task. But overall, the overhead is large (up to 97X). The overhead is primarily due to the fact that every GPU thread has to frequently poll the state variable on the host memory through the PCIe bus.

The second group of bars in Figure 13 show that the usage of proxy cuts the overhead dramatically. The average overhead reduces from 10X to 15%. However, for *stencil2D*, the overhead is still over 100%. The reason is that every block-task in that kernel is short. Since the polling of the flag is done in every block-task, the overhead is large. The third group of bars indicate that the adaptive polling method helps address the problem. The overhead of *stencil2D* is reduced to 8%, and the average overhead becomes 4%. We note that the host-side overhead due to the code transformations is negligible.

Compare with Kernel Slicing-based Methods Before EffiSha, kernel slicing is the main approach designed for increasing GPU kernel schedule flexibility. PKM [6] is a representative of the approach. PKM partitions an original kernel into many smaller kernels, with each processing a predefined number (K) of the block-tasks in the original kernel. Preemptions are not supported in PKM, but the reduced size of a kernel could help reduce the time that a kernel needs to wait before getting access to GPU.

A challenge to PKM is to decide the appropriate value of K. A large K leads to slow response to other kernel launch requests, while a small K reduces the thread-level parallelism and incurs additional kernel launching overhead. We measure the overhead of PKM under a spectrum of responsiveness requirements (the objective kernel launch delay ranges from 1ms to 0.0625ms) by tuning K till meeting each level of responsiveness requirement. Figure 14 reports the observed PKM execution overhead. As the objective delay decreases, the overhead of PKM becomes increasingly significant (average 45%, up to 98%, slowdown for 0.0625ms delay requirement).

To compare the overhead of PKM with EffiSha, we tune the K value of PKM such that the launch delay for each kernel is the same as that in EffiSha. The launch delay of EffiSha for each kernel is showed in Figure 10. The last bar in Figure 13 shows the overhead of PKM. As we can see, the average overhead in EffiSha with all optimizations is 4% (up to 8% for $Triad$). For PKM, the average overhead is 58% (up to 140%). Compared to PKM, EffiSha has a much lower overhead. Moreover, EffiSha does not require fine-tuning to determine the appropriate K value as PKM does.

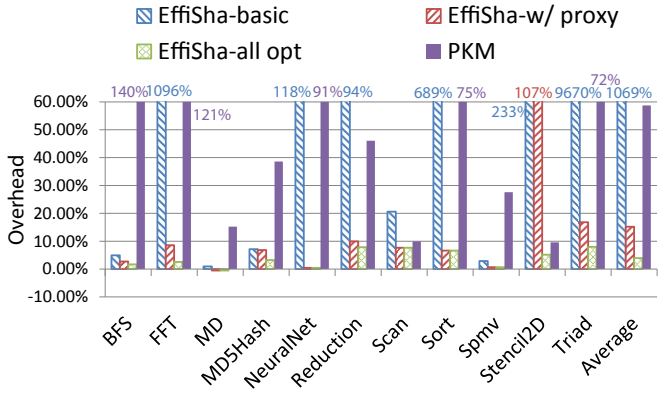

Figure 13. Transformation overhead.

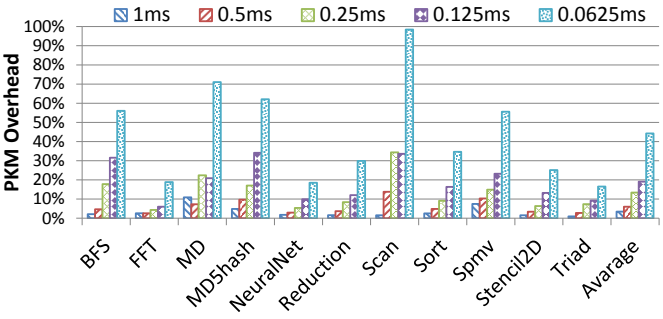

Figure 14. PKM overhead.

Scheduling Overhead We measure scheduling overhead as follows. We launch the original unmodified benchmarks one by one in the scheme mentioned in Section 8 without using EffiSha, and record the time spanning (denoted as T)

from the start of the first kernel to the end of the last kernel. (Notice that because the time between the launches of two adjacent kernels is short, at any moment during the period of T, the GPU is working on some kernel.) We then redo the measurement but with each of the two preemptive schedulers deployed. Let T' be the measured time got in one of the experiments. The scheduling overhead in that experiment is computed as $(T' - T)/T$.

Figure 15 reports the overhead of the two preemptive schedulers under each of the three priority settings. The "dynamic priority" scheduler has overhead larger than the "static priority" scheduler has for its extra complexities. But all the overhead is less than 5%.

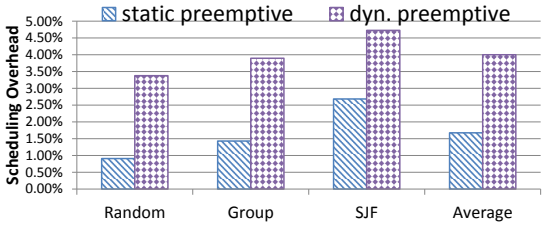

Figure 15. Scheduling overhead.

9. Discussions

To use EffiSha or a scheduler based on it, there is no need to change either the hardware or the OS. One just need to create an alias in the system to redirect the invocations of the default GPU compiler to a customized script, which first invokes the EffiSha source-to-source compiler and then calls the default GPU compiler on the transformed code. To add preemption support to a GPU library, the library developers would need to do the recompilation. An alternative is to implement the EffiSha code transformations at the assembly (PTX) level, which could avoid the need for the source code of a GPU program or library.

Upon a preemption, the memory data of the preempted kernel remain on the GPU memory. There is a possibility that the next kernel on GPU could observe some leftover data on the memory on GPUs that has no virtual memory support. But it is important to note that as observed in prior work [15], such security issue exists even on the default GPU executions. Upon the finish of a default GPU kernel execution, the data typically remain on the GPU memory before the parent CPU process terminates and are hence subject to the same potential issues. The virtual memory support on latest GPUs could help alleviate the security concern.

The sharing of the memory space among multiple GPU applications could cause potential space contention, causing difficulties to some large-footprint kernels to run. Swapping some data of an evicted kernel to the host memory could be an option to alleviate the problem. Extensions of portable data placement optimizers (e.g., PORPLE [16–18]) to both host and device memory could facilitate the process. It is left to study in the future.

10. Related Work

With GPUs becoming an important computing resource, there is a strong interest in making GPUs first-class resource to be managed by operating systems (OS). Gdev [19] integrates the GPU runtime support into the OS. It allows GPU memory to be shared among multiple GPU contexts and virtualizes the GPU into multiple logic ones. GPUfs [20] enables GPUs with the file I/O support. It allows the GPU code to access file systems directly through high-level APIs. GPUvm [1] investigates full- and para-virtualization for GPU virtualization. Wang and others [21] proposed OS-level management for GPU memory rather than letting it managed by applications. Chen and others have recently studied runtime QoS control in data centers equipped with accelerators [22]. None of these studies have considered kernel preemptions.

Some hardware schemes have been proposed to reduce preemption overhead on GPU. Tanasic and others [2] have proposed a hardware-based SM-draining scheme. Park and others [4] have proposed to select different hardware-based preemption techniques based on the progress of a thread block. Lin and others have proposed the use of compression to reduce kernel switch overhead [5]. Wang and others [23] have designed simultaneous multi-kernel (SMK), a hardware method to support incremental kernel evictions. Although these schemes have shown high promises, they require new hardware support. In a certain sense, our solution could be regarded as a software-based approach to materialize SM-draining efficiently. Without hardware extensions needed, it enables preemption on existing GPU devices, and meanwhile, offers software with the rich controllabilities of kernel preemptive scheduling.

Existing proposals of software schemes to enable preemption is to use kernel slicing as mentioned in the introduction. The scheme has been studied in real-time community, represented by the recent proposals of GPES [7], and PKM [6]. A challenge in this scheme is to select the appropriate slice size. As Section 8 has shown, there is an inherent overhead-responsiveness dilemma. For real-time systems where the set of workload is predefined, one could use profiling to find slice sizes suitable to these applications. But For general-purpose systems (e.g., data centers and cloud) with a large, dynamically changing set of workload, the approach is difficult to use. In comparison, our solution requires no kernel slicing and hence is not subject to such a dilemma. Moreover, with Kernel Slicing, the preemption boundaries are still at kernel boundaries; but for EffiSha, the preemption boundaries are at the boundary of a basic block. It is designed to fit the needs of general-purpose systems.

The problem EffiSha addresses is to enable flexible time sharing of GPU among different applications. Orthogonal to

it is the support of spatial sharing of GPU. The problem there is how to maximize the GPU efficiency by better controlling the resource usage of each kernel when multiple GPU kernels run at the same time on one GPU. Techniques, such as Elastic Kernel [24] and Kernelet-based co-scheduling [25]. A recent work proposes to partition SMs among kernels for efficient resource usage through SM-centric program transformations [26]. There are also some hardware proposals for a similar purpose [27]. Spatial sharing and time sharing are orthogonal in the sense that both are needed to make GPU better serve for a multi-user shared environment.

Another layer of scheduling is about the tasks within a GPU kernel or between CPU and GPU [8, 28–32]. They are complementary to the scheduling of different kernels on GPU.

Persistent threads have been used in numerous previous studies. For instance, Chen and Shen have used it in *Free Launch* [10] to mitigate the large overhead of dynamic kernel launches. Effisha leverages persistent threads to gain the controllability of GPU task scheduling.

11. Conclusion

In this work, through EffiSha, we demonstrate a software solution that enables preemptive scheduling for GPU without the need for hardware extensions. Experiments show that EffiSha is able to support kernel preemptions with less than 4% overhead on average. It opens up the opportunities for priority-based preemptive management of kernels on existing GPUs. Enabled by EffiSha, a dynamic priority-based preemptive scheduler demonstrates significant potential for reducing the average turnaround time (18-65% on average) and improving the overall system throughput of program executions (1.35X-1.8X on average) that time-share a GPU.

We thank the anonymous reviewers for the helpful comments. This material is based upon work supported by DOE Early Career Award (DE-SC0013700), the National Science Foundation (NSF) under Grant No. 1455404, 1455733 (CAREER), 1525609, 1216569, and 1618509, and Google Faculty Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE, NSF, or Google.

References

[1] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "Gpuvm: Why not virtualizing gpus at the hypervisor?," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (Berkeley, CA, USA), pp. 109–120, USENIX Association, 2014.

[2] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *Proceeding of the 41st annual international symposium on Computer architecuture*, pp. 193–204, IEEE Press, 2014.

[3] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged scheduling for fair, protected access to computational accelerators," in *Proceedings of the international conference on Architectural support for programming languages and operating systems*, 2014.

[4] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared gpu," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 593–606, ACM, 2015.

[5] Z. Lin, L. Nyland, and H. Zhou, "Enabling efficient preemption for simt architectures with lightweight context switching," in *the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC16)*, 2016.

[6] C. Basaran and K. Kang, "Supporting preemptive task executions and memory copies in gpgpus," in *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, 2012.

[7] H. Zhou, G. Tong, and C. Liu, "Gpes: a preemptive execution system for gpgpu computing," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pp. 87–97, IEEE, 2015.

[8] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workload s," in *Innovative Parallel Computing*, 2012.

[9] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *GPGPU*, 2010.

[10] G. Chen and X. Shen, "Free launch: Optimizing gpu dynamic kernel launches through thread reuse," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[11] Y. Yang and H. Zhou, "Cuda-np: Realizing nested thread-level parallelism in gpgpu applications," *SIGPLAN Not.*, vol. 49, pp. 93–106, Feb. 2014.

[12] "http://clang.llvm.org."

[13] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE micro*, no. 3, pp. 42–53, 2008.

[14] A. S. Tanenbaum, *Modern Operating Systems*. Pearson, 2007.

[15] R. D. Pietro, F. Lombardi, and A. Villani, "Cuda leaks: A detailed hack for cuda and a (partial) fix," *ACM Trans. Embed. Comput. Syst.*, vol. 15, pp. 15:1–15:25, Jan. 2016.

[16] G. Chen, B. Wu, D. Li, and X. Shen, "Porple: An extensible optimizer for portable data placement on gpu," in *Proceedings of the 47th International Conference on Microarchitecture*, 2014.

[17] G. Chen and X. Shen, "Coherence-free multiview: Enabling reference-discerning data placement on gpu," in *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, (New York, NY, USA), pp. 14:1–14:13, ACM, 2016.

[18] G. Chen, X. Shen, B. Wu, and D. Li, "Optimizing data placement on gpu memory: A portable approach," *IEEE Transactions on Computers*, vol. PP, no. 99, 2016.

[19] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class gpu resource management in the operating system," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, (Berkeley, CA, USA), pp. 37–37, USENIX Association, 2012.

[20] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "Gpufs: Integrating a file system with gpus," *ACM Trans. Comput. Syst.*, vol. 32, pp. 1:1–1:31, Feb. 2014.

[21] K. Wang, X. Ding, R. Lee, S. Kato, and X. Zhang, "Gdm: Device memory management for gpgpu computing," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, (New York, NY, USA), pp. 533–545, ACM, 2014.

[22] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[23] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing," in *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture*, HPCA'16, 2016.

[24] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU Concurrency with Elastic Kernels," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[25] J. Zhong and B. He, "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling," *CoRR*, vol. abs/1303.5164, 2013.

[26] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations," in *Proceedings of the International Conference on Supercomputing*, ICS '15, 2015.

[27] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The Case for GPGPU Spatial Multitasking," in *International Symposium on High Performance Computer Architecture*, 2012.

[28] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao, "Dynamic load balancing on single-and multi-gpu systems," in *IPDPS*, 2010.

[29] S. Xiao and W. chun Feng, "Inter-block gpu communication via fast barrier synchronization," in *IPDPS*, 2010.

[30] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, 2009.

[31] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the gpu," in *Proceedings of the Conference on High Performance Graphics*, 2010.

[32] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. D. okter, and D. Schmalstieg, "Whippletree: Task-based scheduling of dynamic workloads on the gpu," *ACM Transactions on Computer Systems*, vol. 33, no. 6, 2014.