A Software Framework for Efficient Preemptive Scheduling on GPU

Guoyang Chen, Xipeng Shen, Huiyang Zhou
North Carolina University
{gchen11,xshen5,hzhou}@ncsu.edu

Abstract

Modern GPU is 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 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. EffiSha consists of a set of APIs, an innovative preemption-enabling code transformation, a new runtime manager, and a set of preemptive scheduling policies. The APIs redirect GPU requests to the runtime manager, which runs on the CPU side as a daemon and decides which GPU request(s) should be served and when a kernel should get preempted. The preemption-enabling code transformation makes GPU kernels voluntarily evict from GPU without the need to saving and restoring kernel states. We demonstrate the benefits of EffiSha by experimenting a set of preemptive scheduling policies, which show significantly enhanced support for fairness and priority-aware scheduling of GPU kernels.

1. Introduction

As a massively parallel architecture, GPU has attained broad adoptions in accelerating various applications. It has become an essential type of computing resource in modern computing systems, ranging from supercomputers, data centers, work stations, to smartphones.

Most of these systems are multitasking and parallel 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 simulteneously by a single node in the data center. How to manage GPU usage fairly and efficiently in such environments is important for the responsiveness of the applications, the utilization of the GPU, and the fairness and 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 oblivous to the priorities of kernels. Numerous studies [10, 12, 15, 16] have shown that the problematic way to manage GPU causes serious unfairness, response delays, and low GPU utilizations.

Some recent work [12, 16] proposes some hardware extensions to help alleviate the issue, but the increased complexity make their practical adoptions uncertain.

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 [10, 15] 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 [4, 22] 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. The prior proposals (e.g., kernel slicing [4, 22]) help reduce the length of a kernel and hence the delay, but is subject to performance loss due to the increased GPU kernel launching overhead and the reduced parallelism in smaller kernels. Our measurement shows that the PKM [21] has 58% performance loss with small preemption latency.

In this work, we propose a simple yet effective way to solve the fundamental dilemma that has been facing the prior solutions. The key is the first software approach to enabling efficient preemptions of GPU kernels. Kernels need not be sliced anymore. They get evicted when it is time to switch kernels on GPU. The dilemma between schedule flexibility and performance loss is hence resolved.

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 [12, 16].

Our solution is pure software based, consisting of some novel program transformations and runtime machinary. We call our compiler-runtime synergistic framework EffiSha (for <u>effi</u>cient <u>sha</u>ring). As a pure software framework, it is immediately deployable in today's systems.

EffiSha consists of a set of APIs, an innovative code transformation, and a new runtime managing scheme. The APIs are at two levels. The high-level APIs allow users to express their intended priorities of a GPU kernel if they want to, while the low-level APIs are used by compilers to pass information about the kernel to the runtime.

The novel code transformation, named *preemption enabling transformation*, is the key to making efficient GPU preemption possible. It converts a GPU kernel into a special form such that preemptions at certain points in the kernel would need to save and restore little if any data. The transformed kernel is subject to certain restrictions on where in the kernel preemptions may happen. Our experiments validate that the restriction does not prevent EffiSha from enalbing flexible preemptive GPU scheduling.

The EffiSha runtime translates the preemption opportunities into efficient scheduling capability. It features an innovative proxy design. It includes a runtime daemon process that runs on the CPU side and a runtime proxy that runs on the GPU side. The daemon executes the (preemptive) scheduling policy of GPU kernels, and informs the running GPU kernel when it is time for it to get evicted. The proxy is a special GPU thread block of the running GPU kernel. It is created as part of the GPU kernel through the earlier mentioned *preemption enabling transformation*. (Each GPU kernel has its own proxy.) The idea of creating such proxies is essential for keeping the interactions between GPU kernels and the CPU daemon efficient.

To demonstrate the benefits of EffiSha, we construct the first GPU scheduler that supports preemptive GPU kernel scheduling. We implement two scheduling policies on it. The results on 11 GPU benchmarks show that the enabled preemptive scheduling improves the responsiveness (in terms of average turnaround time) of GPU kernels over the default executions, and produces schedules consistent with the different priorities of kernels. With the same small-preemption-latency requirements, PKM [21] shows 58% slowdown while EffiSha only has 4% slowdown. It demonstrates the promise of EffiSha for opening up new opportunities for sharing GPUs in a more fair, efficient, and flexible manner.

Overall, this work makes the following major contributions:

- It presents EffiSha, a complete framework that, for the first time, makes beneficial preemptive GPU scheduling possible without special hardware extensions.
- 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.
- It exemplifies the usage for EffiSha by constructing and experimenting a preemptive GPU scheduler. .

2. Background

Modern GPUs exploit massive thread/data-level parallelism to achieve high computational throughput. They employ the single-instruction multiple-thread (SIMT) programming models. All the threads/work items share the same kernel code and the thread identifiers are typically used to determine their corresponding workloads. For a kernel, its threads/work items are organized in a grid of thread blocks/workgroups. Depending on the resource availability, a GPU dispatches one or more thread blocks to its streaming multiprocessors (SMs) or compute units. All the threads in the same thread block must be executed on the same SM. Each thread block is executed independently upon others and the order of thread block dispatching is decided by the hardware to achieve load balance among SMs. Warps are formed from the threads in a thread block and each warp is executed in the single-instruction multiple-data (SIMD) manner. Threads in the same thread block can communicate and synchronize with each other through fast on-chip shared memory. As a result, a thread block forms a basic unit of collaborative execution and is also referred to as a cooperative thread array (CTA).

```
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.

1. kernel (traditional GPU)
2. predefined # of block-tasks (kernel slicing based work[20][21])
3. single block-task (EffiSha)
4. arbitrary segment of a block-task (impractical)

Figure 2. Four levels of GPU scheduling granularity.

With GPUs being widely adopted for general purpose computing, there is an eminent need for the devices to be shared among multiple applications. In the past, GPU does 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 finishes before it can use the GPU. In a recent generation of GPU (NVIDIA K40m), we find that it can support a certain degree of eviction and time-sharing among different kernels from different applications. The details of the scheduling is not disclosed. Our empirical exploration shows that when a new kernel arrives, after a certain time, the currently running GPU kernel drains all its active thread blocks and lets the new kernel to run on the GPU. This new feature improves the flexibility of time sharing of GPU. However, still, it is the hardware that makes the full decision on how these kernels are scheduled and executed. It is entirely oblivious to the priorities of the kernels or the requirement of the quality of service to them. In comparison, our proposed preemption scheme empowers the user to realize flexible scheduling policies for efficiency, fairness, user responsiveness, or quality of service.

3. Granularity for GPU Scheduling

Scheduling granularity determines the time when a kernel switch can happen on GPU. This section explains the choice made in this work.

We first introduce the term *block-task*. In a typical data-parallel GPU program, each thread block processes a small portion of the entire data set. A *block-task* refers to the work of a thread block in such a kernel. The execution of the kernel is hence a collection of block-tasks. The ID of a thread block is taken as the ID of its block-task. For example, the matrix addition example in the Figure 1 consists of B block-tasks (B is the total number of thread blocks) with IDs equaling 0, 1, ..., and (B-1).

Figure 2 lists four granularities for GPU scheduling. At the top is an entire kernel execution; kernel switches on GPU can happen only at the end of the entire kernel. This is what traditional GPU supports. All previously proposed software solutions have tried to support level-2 granularity. The *preemptive kernel model (PKM)* [21], 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.

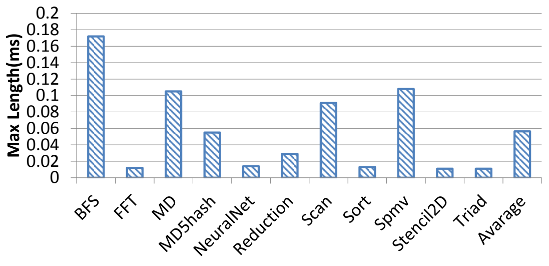


Figure 3. The maximum length of a task in SHOC benchmarks (the largest input in the benchmark suite are used).

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 dillemma 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. However, it is impractical on existing GPUs as it would require saving and restoring the state of massive GPU threads, which incurs tremendous overhead [12, 16].

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 much 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 delay of kernel evictions (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 [6]. All block-tasks are shorter than 0.8 milliseconds. The actual eviction delay is even much shorter than the task length, only 0.08ms on average (detailed in Section 8). Considering that in modern Linux, a context switch happens once every 1-10ms in typical cases [], the level-3 granularity is quite small. For the rare cases where a block-task is too large, it may be possible to develop some compiler techniques to reduce the granularity by splitting a kernel or reforming tasks; it is left for future explorations.

4. Overview of EffiSha

EffiSha is the first framework that offers the level 3 scheduling granularity efficiently. This section provides an overview of it.

As Figure 4 shows, it 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 we have developed. It transforms a given GPU program into a form amenable for runtime management and scheduling. First, 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. Second, the compiler 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 with the property that no (or a minimum amount of) 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.

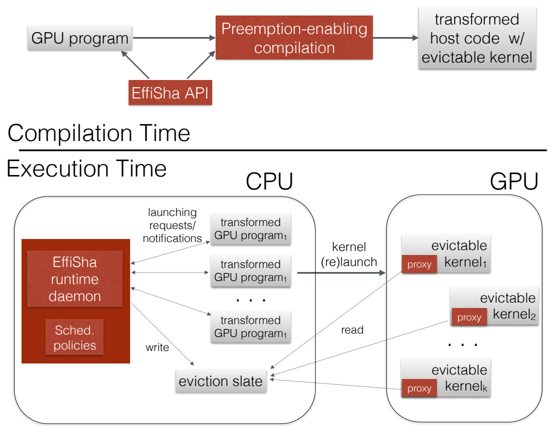


Figure 4. Overview of EffiSha.

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

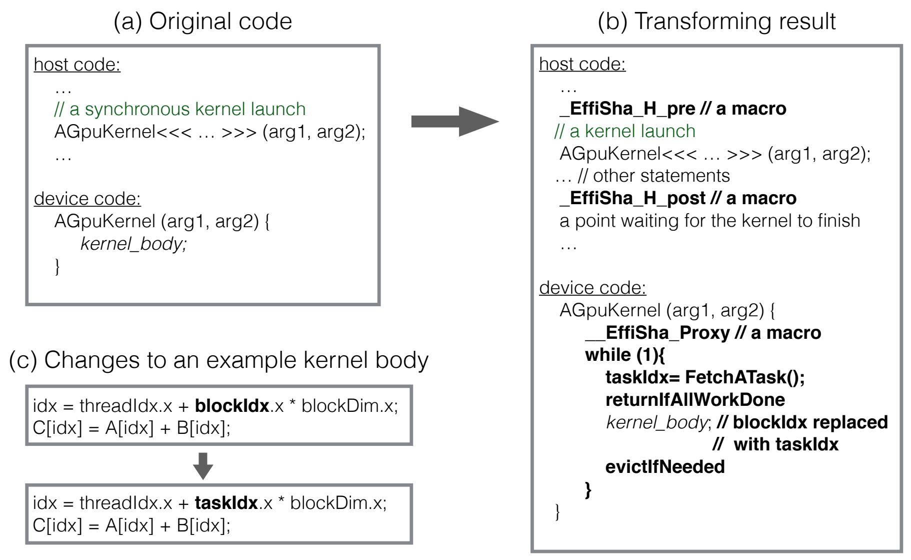
The EffiSha daemon receives all the GPU requests from the applications. These requests could be of different priorities. The EffiSha daemon organizes the requests in queues. Based on the scheduling policy that we have designed, it decides when to evict the current running kernel(s) and which kernel(s) to launch next. The interplay between the daemon and the “proxy” of the daemon efficiently notifies the executing GPU kernels when it is time for one or multiple of them to evict. 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 one of the kernels to start or resume their executions on GPU, it notifies the hosting process of the kernel, which launches or re-launches those kernels accordingly.

The design of EffiSha ensures a very low overhead of a GPU kernel preemption. It exploits the enabled preemptive schedules to achieve improved responsiveness, fairness, and GPU utilization. We next use an example to first explain how the essential machinary of EffiSha works, and then describe the implementations in the compiler module and the runtime.

5. *Preemption-Enabling* Code Transformation

Figure 5 illustrates the effects of the *Preemption-Enabling* transformation to a GPU program. It converts the GPU kernel to 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). As the execution of a GPU kernel follows the single-program multiple-threads (SPMT) model: All GPU threads run the same kernel code; the differences in their behaviors are determined by their thread IDs, as shown by the usage of



```
host code:
   …
   // a synchronous kernel launch
   AGpuKernel<<< … >>> (arg1, arg2);
   …

device code:
   AGpuKernel (arg1, arg2) {
       kernel_body;
   }
```

```
host code:
   …
   _EffiSha_H_pre // a macro
  // a kernel launch
   AGpuKernel<<< … >>> (arg1, arg2);
  … // other statements
   _EffiSha_H_post // a macro
   a point waiting for the kernel to finish
   …

device code:
   AGpuKernel (arg1, arg2) {
       __EffiSha_Proxy // a macro
       while (1){
          taskIdx= FetchATask();
          returnIfAllWorkDone
          kernel_body; // blockIdx replaced
                       //  with taskIdx
          evictIfNeeded
       }
   }
```

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

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

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

"blockIdx" in the matrix addition example in the top part of Figure 5 (c). The *preemption-enabling* transformation changes it into a form indexed by block-task IDs.

An important observation is that the ith block-task does not have to be processed by the ith thread block; it keeps its integrity as long as it is indexed with i (its ID), regardless of which thread block is processing it. Based on the observation, one step of 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—and a thread block can even process multiple block-tasks. The flexibility is important for the remaining part of the transformation.

The second part of the transformation is illustrated in the device code in Figure 5 (b). It adds a *while* loop around the body of the kernel. In each iteration of the loop, a thread block grabs the ID of a yet-to-process block-task and works on that block-task. The loop iterates until all block-tasks are finished or it is time for the kernel to get evicted. These threads, as they stay alive across block-tasks, are also called persistent threads [8]. Because in the new form each thread block executes many (rather than one) block-tasks, much fewer thread blocks need to be created at the launch of the kernel—as long as the number of thread blocks is enough to keep the GPU fully utilized.

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 to restore threads states for processing those block-tasks. 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 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., in the case of atomic data summation), they can find the values on the memory. No extra data saving or restoring is needed upon evictions.

The placement of eviction points offer the level-3 scheduling granularity as Section 3 has discussed. This level of scheduling granularity offers more flexibilities than previous software-based support of GPU scheduling does.

In addition to the described code changes, the *Preemption-Enabling* transformation adds two predefined macros respectively to the points before and after the kernel invocation in the host code. It also injects into the kernel code a proxy of the EffiSha runtime daemon. For their close connections with the runtime components of EffiSha, we postpone the explanation of these features to the next section while explaining the EffiSha runtime.

6. EffiSha API and Runtime

The main functionality of the EffiSha runtime is to manage the usage of GPU, making a kernel get launched, evicted, or re-launched at appropriate times. With the undisclosed GPU driver as a black box, it is hard for the EffiSha daemon to directly manipulate a GPU kernel inside the context of another process. As a result, the runtime management has to involve 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 smoothly and efficiently.

Our description of the design starts with a data structure named *kernel-stubs* and the set of possible states that a GPU kernel could have and their transitions. They are essential for understanding the runtime cooperations.

6.1 Kernel-Stubs and State Transitions

The EffiSha daemon hosts a number of *kernel stubs*. Each holds a record for a GPU kernel that is yet to finish. As Figure 6 (a) shows,

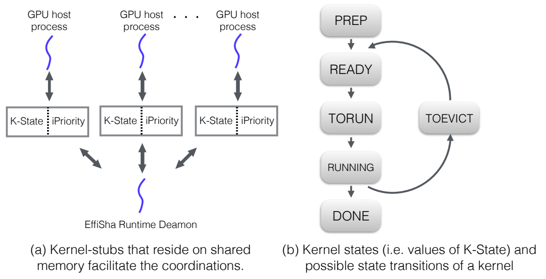

GPU host process
GPU host process
. . .
GPU host process
K-State
iPriority
K-State
iPriority
K-State
iPriority
EffiSha Runtime Deamon
(a) Kernel-stubs that reside on shared memory facilitate the coordinations.
PREP
READY
TORUN
RUNNING
DONE
TOEVICT
(b) Kernel states (i.e. values of K-State) and possible state transitions of a kernel

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

a kernel 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 kernel stub is a small piece of (CPU-side) shared memory created by the EffiSha daemon. When an application requests a kernel stub for a GPU kernel call (through an EffiSha API call), the daemon allocates a kernel 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 than the use of shared memory because the stubs are frequently read by the host thread as we shall see in the next subsection.

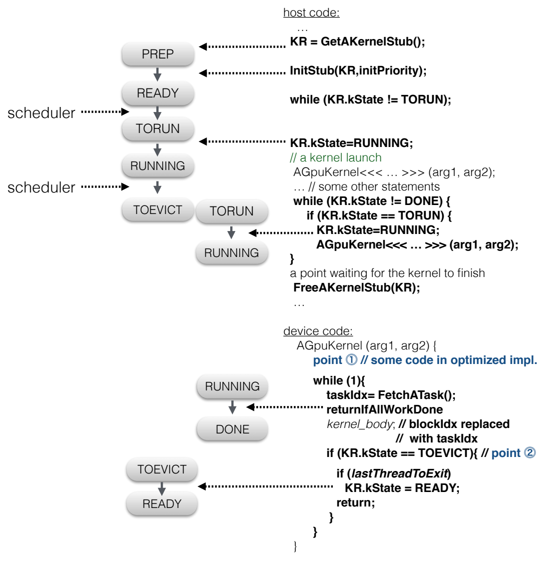
The field *kState* in a kernel stub can have one of seven values, corresponding to all the seven states that a GPU kernel can possibly have. Figure 6 (b) shows the set of states and the possible transitions among them. When a kernel stub gets created, it is before the host thread reaches the kernel invocation; the kernel is in a preparation state (PREP). The kernel then transits to the READY state, indicating that it 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 kernel stub gets reclaimed for other usage.

6.2 Basic Implementation of the Runtime

We now describe the basic implementation of the EffiSha runtime, and explain how the runtime supports the state transitions for kernel scheduling. We leave optimizations of the implementation to the next subsection.

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.

The host thread calls the API "GetAKernelStub" to obtain a kernel 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 stub with the initial priority (also called static priority) of the kernel; if it is "null", the kernel inherits the priority of the host thread. 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.



```
host code:
   …
KR = GetAKernelStub();

InitStub(KR,initPriority);

while (KR.kState != TORUN);


KR.kState=RUNNING;
// a kernel launch
 AGpuKernel<<< … >>> (arg1, arg2);
 … // some other statements
 while (KR.kState != DONE) {
    if (KR.kState == TORUN) {
       KR.kState=RUNNING;
       AGpuKernel<<< … >>> (arg1, arg2);
}
a point waiting for the kernel to finish
 FreeAKernelStub(KR);
 …

device code:
   AGpuKernel (arg1, arg2) {
      point ① // some code in optimized impl.

      while (1){
         taskIdx= FetchATask();
         returnIfAllWorkDone
         kernel_body; // blockIdx replaced
                      //  with taskIdx
         if (KR.kState == TOEVICT){ // point ②

            if (lastThreadToExit)
              KR.kState = READY;
            return;
          }
      }
  }
```

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 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-task left, it changes the kState field of the stub to "DONE". Otherwise, it processes its work, and then before grabing a new task, it checks whether the kState is now "TOEVICT". If so, the thread exits; the last thread to exit, changes kState to "READY" such that the kernel can be re-launched later. The voluntary evictions require no saving of the GPU thread states thanks to the independence of block-tasks mentioned in Section 5. In the implementation, kernel 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[1]. 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 runtim to resume, in which case, the runtime changes kState to "TORUN", and as soon as the host thread sees it, it re-launches the kernel.

6.3 Optimizations

The basic implementation enables evictions and re-launches of GPU kernels through coordinations among the EffiSha runtime deamon, the host CPU thread, and the GPU threads. 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 the PCIe link), they cause significant slowdowns to some kernels, especially those that have small block-tasks (e.g., 100x slowdown on a Triad kernel).

We optimize the implementation by creating a special thread block for each GPU kernel. The thread block does not process any

[1] 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.

```
if (this is the first thread block){
  local_kState = KR.kState;
  while (local_kState!=DONE){
     if (KR.kState==TOEVICT){
        local_kState = TOEVICT;
        break;
     }
  }
  return;
}
```

Figure 8. Codelets used for creating a proxy of the EffiSha runtime on GPU.

block-tasks for the kernel, but serves as the proxy of the EffiSha runtime, running on the GPU along with the other thread blocks of the kernel. In this implementation, there is a copy of kState on the GPU memory. The proxy thread block keeps reading the kState on the host memory and updates the copy on the GPU accordingly. With that, the other thread blocks of the GPU kernel 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 to "DONE" or "READY" accordingly.

The proxy is implemented through the help of the EffiSha compiler. The compiler adds the code in Figure 8 to the point $\textcircled{1}$ right above the device *while* loop in Figure 7. The check of the thread block ID ensures that thread block 0 serves as the proxy and the remaining thread blocks serve as normal workers. Notice that the check of the ID is outside the block-task while loop and adds almost no overhead. In addition, the read of kState at the point $\textcircled{2}$ is replaced with the read of the local copy of kState.

We note that for the implementation to work well, the proxy should not be put by the hardware into the waiting queue while other thread blocks of the kernel are actively running. Otherwise, the local copy of kState would not get updated promptly. The condition is ensured by the preemption-enabling transformation described in the previous section. Recall that by making each thread block process many block-tasks, the transformation brings the flexibility for controlling the number of thread blocks. For the proxy to work actively, the compiler just needs to set the total number of thread blocks properly (based on the register and shared memory demands per thread block).

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 host code in Figure 7, we have showed that there is a *while* loop before the waiting point in the code, which continuously checks *kState* to help re-launch the kernel if it is evicted. One potential issue is that if there is a lot of CPU work existing after the kernel launch and before the waiting point, the kernel could have got evicted long before the host thread reaches the *while* loop, and hence suffers a long delay in the re-launch. This issue occurs rarely in our experiments because the amount CPU work before the waiting point is usually quite small compared to the scheduling time slice. Still, to alleviate the potential issue in some rare cases, 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.

6.4 Setting Priority

By default, a kernel's *iPriority* inherits the priority of the host thread. However, a programmer may use an EffiSha API call (*setIPriority*) to customize the priority of a kernel. 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 kernel 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. If a normal user sets the priority higher than the host thread, the EffiSha runtime ignores the value and uses the priority of the host thread instead.

A caveat is that even though the EffiSha runtime daemon respects the priority when it schedules 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 re-launched until the host thread resumes running. A low-priority host thread could hence result in lots of delay to the kernel (re)launch. To avoid the problem, a suggestion to superusers is to ensure that the host thread has a priority no lower than the kernel.

7. Scheduling Policies

The support provided by EffiSha opens up the opportunities for investigating the effects of different preemptive scheduling policies on GPU kernel executions. In this work, we exemplify it by constructing two preemptive scheduling policies.

- Priority-based immediate eviction (*piv*). In this schedule, the EffiSha runtime daemon schedules kernels purely based on their initial priorities. The scheduler maintains a waiting queue for kernels that are waiting to use GPU. The kernels 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 kernel 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.

  This scheduling policy allows high-priority kernels to get executed as soon as possible, but leaves low-priority kernels facing the risk of starvation when there is a long-running kernel or continuous arrivals of new kernels with higher priority than some waiting kernels.

- Dynamic priority-based round-robin policy (dprr): 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 a kernel equals to its initial (static) priority when that kernel just arrives, but get increased by one for every millisecond the kernel waits in the active queue. The increase is capped at 20. In each of the two queues, kernels are ordered by their dynamic priorities. A new coming kernel is put into the active queue (positioned based on its priority).

Everytime a kernel is scheduled to work, 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 kernel 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 the head of the active waiting queue to run. 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.

8. Evaluation

We implement a prototype EffiSha system, which uses Clang [1] as the base for the development of the source-to-source compilation. We conduct a set of experiments, trying to answer the following main questions:

- *Preemption.* Can EffiSha enable preemptions without hurting the correctness of the executions?
- *Priority.* Can schedulers based on EffiSha indeed support the different priorities of different kernels?
- *Overhead.* How much time overhead does EffiSha add to GPU executions?
- *Delay.* As mentioned, EffiSha restricts where preemptions can happen in a GPU kernel. How much delay does the restriction cause for an eviction to take place?
- *Facilitation.* Can EffiSha facilitate the construction and comparison of different preemptive schedulers on GPU?

8.1 Methodology

Our experiments use the programs in the level-1 folder of the SHOC CUDA benchmark suite [6]. Table 2 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 for GPU (i.e., launching a GPU kernel) in turns. Another application issues the request 3ms after the previous application issues its request. We set the order of the 11 applications as the top-down order in Table 3—which was arbitrarily chosen.

We experiment with three different priority settings, as the three rightmost columns in Table 3 show. 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 "SJF" (shortest job first) scheme decides the priority of a kernel based on its length (i.e., the "standalone time" column in Table 3); the longer, the lower.

We run the experiments on a NVIDIA K40m GPU. Table 1 shows the information of the machine. Without noting otherwise, in the experiments, the baseline of the comparisons is always the executions of the default benchmarks without going through any changes by EffiSha.

Table 1. Machine Description.

Name	**GPU card**	**Processor**	**CUDA**
K40m	NVIDIA K40m	Intel Xeon E5-2697	7.0

Table 2. 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 single and double precision stencil computation
Triad	A version of the stream triad

Table 3. Kernel length 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

8.2 Soundness and Support of Priorities

We run the 11 benchmarks in all the settings under the four preemptive schedules. 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 (ranging from 1 to 9). 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. The scheduling policy *piv* is used; its direct usage of the static priority of kernels makes it ideal for studying the relations between priorities and the actual measurements.

Figure 9 shows the number of evictions that happen on the kernels at each level of priority. 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 accommondate kernels of different priorities.

8.3 Overhead

EffiSha may introduce several sources of overhead, categorized into two classes:

- Overhead incurred by code changes, including the usage of persistent kernels, the creation of the GPU proxy, the checks on

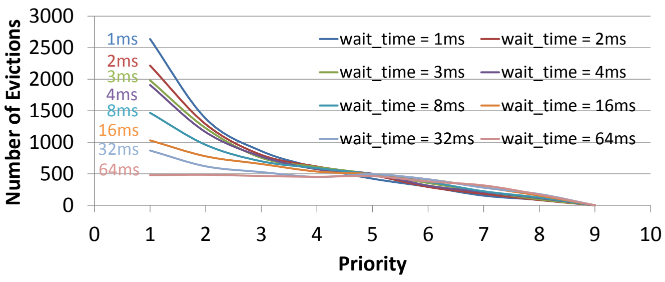


Figure 9. Total Number of Evictions for Different Priorities

the eviction flag, and other changes that the EffiSha compiler makes to the host code and kernels.

- Overhead incurred by preemptive scheduling. It mainly includes the time taken by kernel evictions and resumptions, and any side effects of them to the runtime executions.

We measure the overhead of either class respectively.

Transformation overhead. The first is to measure the overhead brought by its 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 kernels (i.e., without EffiSha transformations) to compute the overhead. We call such overhead the *transformation overhead*.

Figure 10 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 6.3. The first group of bars in Figure 10 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 10 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%.

Here we compare EffiSha to a prior work, PKM [21]. PKM partitions the kernel grid into subgrids and launches subgrids as separate kernels. This way, the GPU kernel can be preempted at the subgrid granularity. The challenge of PKM, however, is the subgrid configuration: large subgrids lead to high preemption latency (i.e., slow response to preemption requests) while small subgrids reduce the thread-level parallelism and incur additional kernel launching overheads. In Figure 11, we show the execution overhead of Kernelet for different subgrid configurations, which are tuned to meet different preemption latency requirements, from 1ms to 0.0625ms. As the preemption latency decreases, the overhead of PKM becomes significant(average 45%(up to 98%) slowdown for response time 0.0625ms).

To compare the overhead of PKM with EffiSha, we tuned the subgrid value so that the response time is same as EffiSha for each application. The response time of EffiSha for each application is showed in Figure 13. The last bar in Figure 10 shows the overhead of PKM. As we can see, the average overhead of EffiSha with all optimizations is 4%(up to 8% for $Triad$). For PKM, with the same preemption latency as EffiSha for each application, the average overhead is 58%(up to 140%). We can see that, compared to PKM, EffiSha has much lower overhead and does not require fine-tuning to determine the subgrid configuration.

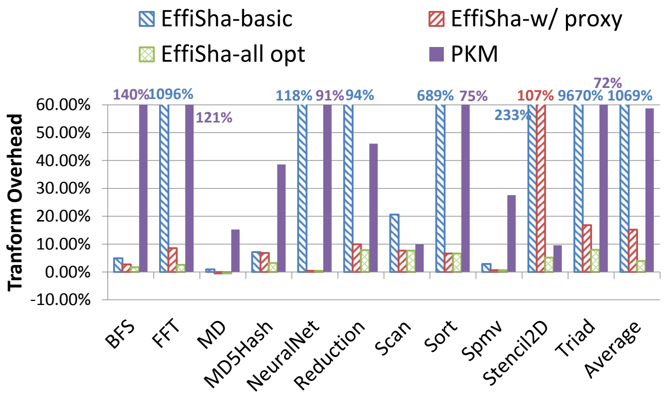


Figure 10. Transformation overhead.

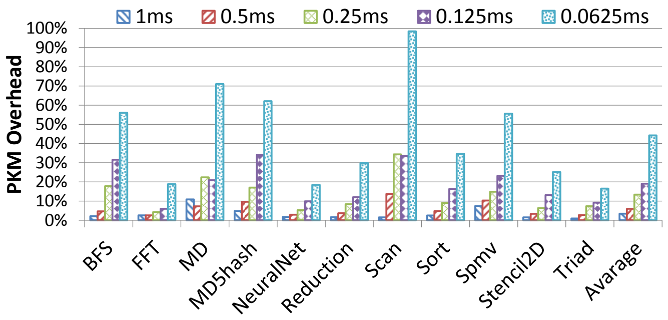


Figure 11. PKM overhead.

Scheduling Overhead In the second way, we measure the overhead when EffiSha is actually used in some preemptive schedulers. Our measurement is as follows. We run the 11 benchmarks (with the 3ms delay of each kernel's starting time) on the default GPU, and record the time spanning from the start of the first kernel to the end of the last kernel, and denote the time as T. (Notice that because the time between the launches of two adjacent kernels (i.e., the $\delta = 3ms$) 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 then computed as $(T' - T)/T$.

Figure 12 reports the overhead of the four schedulers under each of the three priority settings. The scheduler "dprr" has the larger overhead than "piv" for its more operations in the more complex scheduling algorithm. But all the overhead is less than 5%.

8.4 Delay

To see how quickly the running kernel can act on an eviction request, we measure the preemption latency for each kernel, which refers to the time between the setting of the eviction flag and the exit of the kernel from GPU. To measure it, we keep each kernel doing

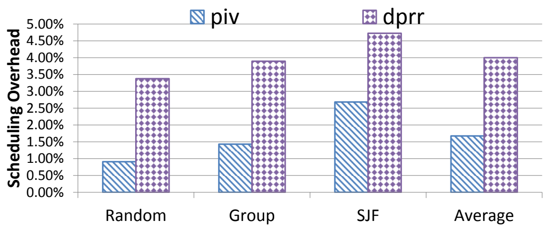


Figure 12. Scheduling overhead.

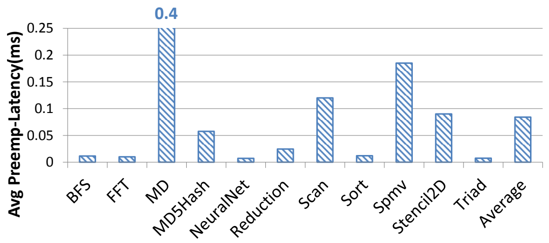


Figure 13. Preemption latency of each kernel.

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 13 shows the preemption latency of each kernel. The average latency is about 0.08ms. *MD* has the longest preemption latency (0.4ms) because of its large block-task. *BFS, FFT, NeuralNet* and *Triad* have a less than 0.015ms latency.

8.5 Facilitating Scheduler Designs

Thanks to the feasibility brought by EffiSha, one can now compare different preemptive schedulers for GPU kernels. In this part, we exemplify it by reporting some observations in our comparisons of the two preemptive schedulers and the default executions.

One obvious difference from the default executions is that all these schedulers show a certain consideration of kernel priorities, while the default executions do not. As this difference has been mentioned in earlier part of the paper, our comparisons in this part concentrate on the implications of the different preemptive schedulers to the workload turnaround time and the overall system throughput.

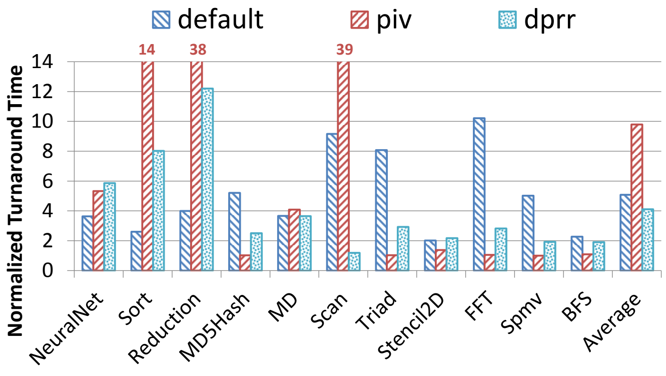
Normalized turnaround time. Figure 14 reports the normalized turnaround times (NTT) for the kernels in the three settings. Here, NTT is defined as follows [7]

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

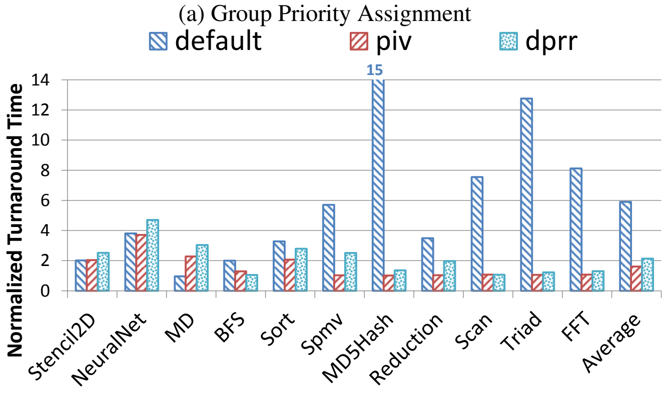
where, T_i^{SP} and T_i^{MP} are the execution time of a kernel in its standalone run and its co-run. The value of NTT is usually greater than 1, the smaller the more responsive the kernel is.

In each of the graphs in Figure 14, we order the benchmarks from left to right in an increasing order of priority.

Our discussion starts with the default scheduling results on the "Group Priority Assignment" (Figure 14 (a)). The average NTT is 5.08. Programs *Scan, Triad, FFT* have the largest NTT, which are 9.16, 8.07 and 10.8 respectively. The reason is that the kernel execution times of *Scan, Triad, FFT* are 1.41ms, 1.22ms, 1.17ms, which are relatively small compared to other applications. They wait for 11ms, 8ms and 10ms for the default scheduler to schedule them to GPU. Programs *MD, Spmv, Stencil2D* wait for much longer times (36ms, 18ms and 28ms), but because their execution times are much longer, their NTT are only 3.7, 5.0 and 2 respectively.



(a) Group Priority Assignment



(b) SJF Priority Assignment

Figure 14. Normalized turnaround times. The benchmarks are put in an increasing order of their priorities (from left to right).

The *piv* scheduler allows applications with a high priority to finish as soon as possible. As we can see, for all benchmarks on the left side of *Scan*(including) , NTTs under *piv* are larger than the *default*. For all benchmarks on the right side of *Scan*, their NTTs under *piv* are smaller than the *default*. Overall, its average NTT is much larger than the *default*. The reason is that some low-priority benchmarks, although they are ready to run at an early time, have to wait for the higher-priority kernels to finish and hence are subject to a very long turnaround time. For example, *Reduction, Sort, Scan* wait for 77ms, 73ms and 53ms to be scheduled to SMs while their execution times are only 2.06ms, 5.46ms, and 1.41ms, which leads to the large NTTs (38,14,39).

The policy *dprr* provides more chances for all applications to share GPU than *piv* does.

We also tried to assign random priorities to the kernels, and the insights we obtained are similar to what we have already mentioned. Results are omitted for the interest of space.

The results on the SJF setting (Figure 14(b)) have some differences. The high priority benchmarks *MD5Hash, Scan, Triad, FFT* suffer large NTTs in the default execution. All the benchmarks on the right hand side of *Sort* get the perfect NTT (1). On average, both piv and dprr can significantly reduce NTT compared to the default execution.

System throughput. Figure 15 shows the overall system throughput for the five schedulers. Following prior work [7], the system throughput (STP) is defined as

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

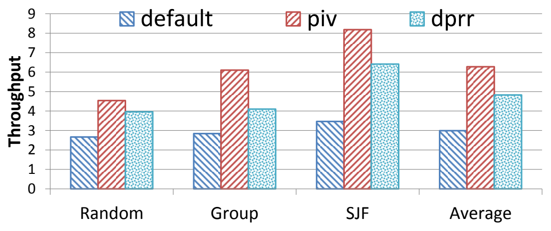


Figure 15. Overall system throughput.

The range of STP is from 0 to n, where, n is the number of kernels (11 in our case). The larger, the better.

The support of priority-based preemption does not throttle the system throughput. All the preemptive schedulers happen to get a higher throughput than the default.

9. 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 (OSes). Numbers efforts have been reported along this direction. Gdev [9] 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 [13] enables GPUs with the file I/O support. It allows the GPU code to access file systems directly through high-level APIs. GPUvm [15] investigates full- and para-virtualization for GPU virtualization. Wang et al. [18] proposed OS-level management for GPU memory rather than letting it managed by applications.

One critical issue that prevents GPUs from being CPU-like first-class computing resource is the lack for preemption. Preemptive multitasking/multiprogramming on CPUs is achieved with context switching. But it incurs too much overhead on GPUs due to the nature of massive concurrent threads. To reduce this overhead, different hardware and software schemes have been proposed. Tanasic and others [16] proposed a hardware-based SM-draining scheme, leveraging the fact that a completed thread block does not need to store its context. When a preemption request is received by an SM, no more thread blocks are allowed to be dispatched to it. After the SM drains all the resident thread blocks, it can be preempted. To further reduce the preemption latency, Park and others [12] proposed to use different preemption techniques depending the progress of a thread block. When the thread block is at an idempotent point, it can be simply flushed and marked as unexecuted. When the thread block is close to finish, SM-draining is used. At other points, the context switching is used to enable immediate response. Although these schemes show high promises, they require new hardware support. In comparison, our proposed scheme is pure software and enables preemption on existing GPU devices.

Besides our proposed approach, another software scheme to enable preemption is to use kernel slicing. It slices a long-running kernel into multiple subkernels, splits large data transaction into multiple chunks, and then inserts preemption points between subkernel launches and memory copy operations. The scheme has been studied in real-time community, represented by the recent proposals of GPES [22] and PKM [4]. 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 centerns 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. 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 controling the resource usage of each kernel when multiple GPU kernels run at the same time on one GPU. Techniques, such as Elastic Kernel [11] and Kernelet [21], have been proposed to adjust the resource usage of each of the co-running kernels such that they can run efficiently together. A recent work proposes to partition SMs among kernels for efficient resource usage through SM-centric program transformations [19]. There are also some hardware proposals for a similar purpose [2]. 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. Spatial sharing is often limited to a small number of kernels (mostly two in prior studies), and on current GPUs, co-running kernels must come from the same application. In a general shared environment like data centers and clouds, time sharing is needed even if spatial sharing is used.

Another layer of scheduling is about the tasks within a GPU kernel or between CPU and GPU. A number of studies have explored this direction [3, 5, 8, 14, 17, 20]; some of them have used persistent threads to facilitate the scheduling. These studies are complementary to the scheduling of different kernels on GPU.

10. Conclusions

In this work, through EffiSha, we demonstrate a software solution that enables preemptive scheduling for GPU without the need for hardware extensions. To use EffiSha, the programs need to go through the compilation-based transformation. For practical usage, this explicit step can be avoided if the transformation is integrated into the native compiler by the vendors. Experiments show that EffiSha is able to support preemption with less than 4% overhead on average. It opens up the opportunities for priority-based preemptive management of kernels on existing GPUs.

References

[1] http://clang.llvm.org.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, (3):42–53, 2008.

[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] 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, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2342821.2342858.

[10] 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.

[11] 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.

[12] 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*, pages 593–606. ACM, 2015.

[13] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: Integrating a file system with gpus. *ACM Trans. Comput. Syst.*, 32(1):1:1–1:31, Feb. 2014. ISSN 0734-2071. doi: 10.1145/2553081. URL http://doi.acm.org/10.1145/2553081.

[14] 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*, 33 (6), 2014.

[15] 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, pages 109–120, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL http://dl.acm.org/citation.cfm?id=2643634.2643646.

[16] 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*, pages 193–204. IEEE Press, 2014.

[17] 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.

[18] 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, pages 533–545, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2789-3. doi: 10.1145/2591971.2592002. URL http://doi.acm.org/10.1145/2591971.2592002.

[19] 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.

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

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

[22] 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*, pages 87–97. IEEE, 2015.