FLEP: Enabling Flexible and Efficient Preemption on GPUs

Bo Wu
Colorado School of Mines
bwu@mines.edu

Xu Liu
College of William and Mary
xl10@cs.wm.edu

Xiaobo Zhou
University of Colorado, Colorado Springs
xzhou@uccs.edu

Changjun Jiang
Tongji University, China
cjjiang@tongji.edu.cn

Abstract

GPUs are widely adopted in HPC and cloud computing platforms to accelerate general-purpose workloads. However, modern GPUs do not support flexible preemption, leading to performance and priority inversion problems in multi-tasking environments.

In this paper, we propose and develop FLEP, the first software system that enables flexible kernel preemption and kernel scheduling on commodity GPUs. The FLEP compilation engine transforms the GPU program into preemptable forms, which can be interrupted during execution and yield all or part of the streaming multi-processors (SMs) in the GPU. The FLEP runtime engine intercepts all kernel invocations and determines which kernels and how those kernels should be preempted and scheduled. Experimental results on two-kernel co-runs demonstrate up to 24.2X speedup for high-priority kernels and up to 27X improvement on normalized average turnaround time for kernels with the same priority. FLEP reduces the preemption latency by up to 41% compared to yielding the whole GPU when the waiting kernels only need several SMs. With all the benefits, FLEP only introduces 2.5% runtime overhead, which is substantially lower than the kernel slicing approach.

CCS Concepts • **Software and its engineering → Multiprocessing / multiprogramming / multitasking**

Keywords Preemption; Multi-tasking; GPGPU; Kernel scheduling

ASPLOS '17, April 08-12, 2017, Xi'an, China
© 2017 ACM. ISBN 978-1-4503-4465-4/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037742

1. Introduction

Recent years have seen rapid adoption of GPUs in various types of platforms because of the tremendous throughput powered by massive parallelism. Applications offload compute intensive workloads as kernels to the GPU for acceleration. Typically, one application cannot fully utilize the GPU resources, and hence co-locating GPU applications on the same machine may dramatically improve system efficiency [9, 31, 34, 23, 25].

However, in such a multi-tasking environment, different applications may compete for GPU access. It is critical to coordinate the kernel executions in an effective and efficient manner. Unfortunately, unlike CPUs, modern GPUs lack a key capability for efficient and fair multi-tasking: *preemption support*. As such, high-priority kernels cannot be immediately scheduled due to the GPU occupancy by a low-priority kernel, which is known as priority inversion [13, 20]. Moreover, in a system with user-facing applications and throughput-oriented applications co-located [11, 17, 9], long-running kernels may block short-running kernels, seriously degrading system's fairness and responsiveness.

Nvidia has been improving the support for GPU sharing in recent generations of GPUs. Starting from the Fermi architecture [28], Nvidia GPUs can concurrently run up to 16 kernels from the same process. The Kepler GPU architecture further enhances the concurrency support through Hyper-Q [29], which enables simultaneous kernel launchs from multiple threads. Furthermore, the introduction of Multi-Process Service (MPS) [30] enables kernels from different processes to share the same GPU. However, those architecture features do not solve the problems caused by the lack of preemption support. The key reason is that the streaming multi-processors (SMs) of the GPU can only support up to a certain number of active threads, while a kernel typically launches hundreds of thousands of threads, much more than what the SMs can host at one time. Once starting its execution, a kernel occupies the GPU until all its threads finish execution. Hence, concurrent kernels can simultaneously share the GPU only when their aggregate demand for hardware resources is low, which is uncommon in practice.

Supporting preemption in GPU architectures is non-trivial due to its dramatically different design philosophy from CPUs. The large amount of active threads and sizable register files impose unique challenges for lightweight preemption. Previous works [35, 32] proposed hardware extensions to enable preemptive programming on GPUs by exploiting the special semantics of GPU programs. However, the NVIDIA Pascal GPU architecture is the only one that claims to support preemption, though no publicly available information shows the availability of software-level preemption control, let alone the evaluation of the preemption overhead. Existing software solutions typically rely on kernel slicing [41, 19, 5], where the compiler or programmer slices the original kernel into many sub-kernels. The GPU can be preempted at the end of each sub-kernel. These approaches face a challenge of determining the granularity of the sub-kernels. Too large granularity may lead to unsatisfactory preemption latency, while too small granularity results in non-trivial runtime overhead due to excessive kernel invocations. Moreover, the kernel slicing-based techniques always preempt the whole GPU, but the waiting kernel may only need multiple SMs. In this case, preempting just enough SMs to run the waiting kernel substantially reduces preemption latency.

In this paper, our goal is to develop a software system to enable flexible kernel preemption and kernel scheduling for GPU applications in a transparent and efficient manner. To this end, we have developed FLEP, the first compilation and runtime software system that supports the following features:

- **Transparency:** FLEP automatically transforms existing GPU programs and does not require the user to integrate the co-running kernels into the same GPU context.
- **Flexibility:** FLEP supports temporal preemption, which yields the whole GPU, and spatial preemption, which yields part of the SMs, depending on the configuration of the waiting kernel. Moreover, FLEP can be configured to preempt and schedule kernels with different goals.
- **Efficiency:** FLEP incurs marginal runtime overhead and is hence practical for real-world usage.

The key components of FLEP are a compilation engine and a runtime engine. The goal of the compilation engine is to transform the input program so that: 1) the GPU kernels can yield an arbitrary number of SMs; and 2) the CPU code can receive orders from the runtime to preempt its kernels. The major challenge is to circumvent the uncontrollable hardware thread scheduler and reduce the runtime overhead incurred by the transformed code. To address it, FLEP features a novel way to use persistent thread blocks with SM awareness, which can yield a specific number of SMs informed by the CPU and control the task granularity to amortize communication overhead.

The FLEP runtime aims at determining and enforcing the optimal kernel schedules through preemption support. It predicts the duration of kernel invocations based on plug-in performance models, tracks the kernels' execution status, and decides whether to preempt the running kernel and the order to schedule waiting kernels given a scheduling policy. We explore two scheduling policies with different goals to show the FLEP's flexibility and efficiency in controlling runtime overhead.

The primary contributions of this work are as follows:

- We propose compilation techniques to transform GPU programs to enable both temporal and spatial preemption for the first time.
- We present a runtime engine with two scheduling policies to determine kernel preemption and kernel scheduling to achieve different goals.
- We implement a prototype system, FLEP, which consists of the offline and online components to enable flexible and efficient GPU kernel preemption and scheduling.
- We evaluate FLEP on eight benchmarks covering multiple domains. The experiments on 28 co-running benchmark pairs demonstrate up to 24.2X speedup for high-priority kernels. For 28 pairs of co-running benchmarks with the same priority, FLEP improves the average normalized turnaround time, which represents average responsiveness, by 8.1X on average. When fairness is the major goal, FLEP demonstrates near-optimal weighted fairness with well controlled performance degradation. For a set of three-kernel co-runs, FLEP delivers up to 20.2X improvement on average normalized turnaround time, but kernel reordering, a common approach to manage kernel co-runs, only yields 2.3% improvement. FLEP's flexible spatial preemption reduces the preemption overhead by up to 41% compared to all-SM preemption when the waiting kernels do not launch enough threads to occupy the whole GPU. With all the benefits, FLEP only introduces around 2.5% runtime overhead, which is much lower than the kernel slicing approach.

2. Background and Motivation

In this section, we provide the background on the basic GPU program execution model and the special features of the GPU hardware thread scheduler, which is necessary to understand the proposed techniques. We focus on the Nvidia GPUs and use the CUDA terminologies, but the other widely used GPU programming model, OpenCL [3], is very similar. We then motivate this research by exposing the problems of lacking flexible preemption support. Based on the co-running results of a set of benchmarks, we show the priority inversion problem and the potential to improve average program responsiveness.

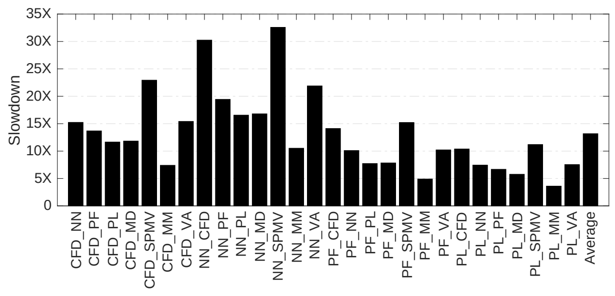
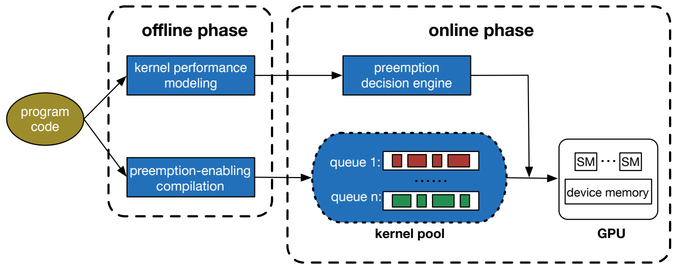


Figure 1: Slowdown of high-priority kernels.



offline phase
online phase
program code
kernel performance modeling
preemption-enabling compilation
preemption decision engine
queue 1:
queue n:
kernel pool
SM
SM
device memory
GPU

Figure 3: Overview of the FLEP framework.

2.1 GPU Program Execution Model

The GPU, as a massively parallel compute engine, has thousands of streaming processors (SP). Hundreds of them compose a streaming multi-processor (SM), which can concurrently execute up to a certain number of threads (e.g. 2048 in the Nvidia Kepler architecture). When a kernel is launched, the GPU creates a large number of threads, which form many thread blocks. A thread block is also called a cooperative thread array (CTA), in which the threads can communicate through on-chip shared memory. All the CTAs, once launched, are managed by a hardware scheduler. When an SM has sufficient hardware resources (e.g., registers or shared memory), the scheduler dispatches a CTA to it. A dispatched CTA is called an active CTA, whose threads are grouped into warps, the basic unit to be scheduled by the warp scheduler for SIMD execution on the SPs. Note that once a kernel starts execution, it blocks all other kernels until all of its CTAs are scheduled. In other words, modern GPUs do not support preemption.

GPUs are typically used together with CPUs to form a heterogeneous system. Each process running on the CPU can leverage the GPU for acceleration by following several steps. The CPU initializes the data and allocates space in the GPU device memory. The CPU transfers data to the GPU memory. The CPU invokes the GPU kernel to process the transferred data. Finally, the GPU sends the computed results back to the CPU memory (i.e., the host memory). All the operations on the GPU, including the data allocation, data transfer, and kernel invocation, are through sending commands. A GPU may support multiple streams through which the commands can be sent to the GPU. The commands in the same stream are executed in FIFO order, while commands in different streams can be executed in parallel.

In early GPUs, the streams are mainly used for overlapping data transfer and kernel execution. Starting from the Kepler architecture, Nvidia GPUs support concurrent execution of kernels from different processes using MPS. Each MPS instance (i.e., an independent GPU program) is automatically allocated a distinct stream. If the kernel command from one stream cannot fully utilize the GPU, the co-running kernels from a different stream can use the remaining resources.

2.2 Demand for Flexible Preemption

The non-preemption feature of GPUs has two major problems. First, the kernel of a high-priority process has to wait even if the GPU is running a kernel from a low-priority process, causing the well-known priority inversion problem [13, 20]. Second, a short-running kernel may have to wait until a long-running kernel finishes, which significantly degrades overall performance and fairness. To illustrate how serious the problem is, we co-run 28 application pairs. For each pair of programs denoted as A_B, we run program A on a small input and B on a large input. We invoke A's kernel immediately after B's kernel is launched on the GPU. Figure 1 shows the slowdown of A's kernel because of the waiting for B's kernel to finish. The performance degradation due to the waiting is up to 32.6X. We will detail the benchmarks and experiment setting in Section 6.

The results motivate temporal preemption, which forces the running kernel to yield the GPU, such that a high priority or short-running kernel can start execution quickly, depicted by Figure 2. Existing software-based solutions to enabling temporal preemption on GPUs are all based on kernel slicing. They require the programmer or leverage a compiler to slice the original kernel into many sub-kernels, each processing part of the data input. One complexity of using kernel slicing is to determine the optimal granularity. On the one hand, too coarse-grained slicing (i.e., one sub-kernel processes one large portion of the input) may not show enough response time improvement. On the other hand, too fine-grained slicing introduces non-trivial overhead. For example, the Kepler GPU supports concurrent execution of 120 active CTAs of size 256, so we slice the kernel such that each sub-kernel launches 120 CTAs. As a result, we observe over 10% overhead for several benchmarks (details in Section 6). Note that this overhead exists even if the kernel is never preempted.

Moreover, a high-end GPU typically has more than 10 SMs, but current solutions yield all the SMs for preemption, while the waiting kernel may only need several SMs. Therefore, an ideal solution should also support spatial preemption, which asks the running kernel to only yield part of the SMs, as depicted in Figure 2 (b). The major advantage of spatial preemption is to reduce preemption overhead, which is very important when a kernel may be preempted many

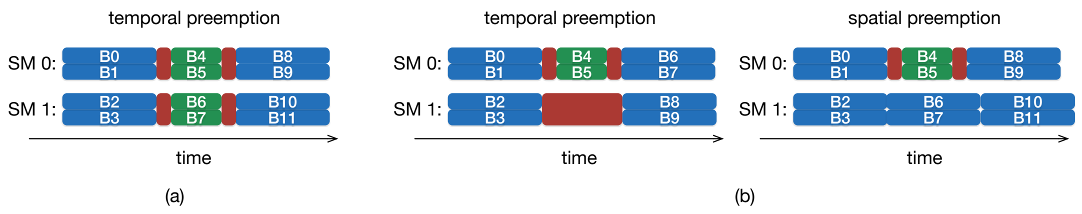


Figure 2: The GPU has two SMs, each of which can support two concurrent CTAs. The blocks in blue represent CTAs from the kernel being preempted (K1). The blocks in red represent preemption overhead. The blocks in green represent the CTAs from the preempting kernel (K2). Figure (a) demonstrates the idea of temporal preemption. Kernel k1 yields the whole GPU, which incurs some overhead. K2's four CTAs fully occupy the GPU, whose execution is followed by some preemption overhead and the resumed execution of K1. In (b), the left-hand sub-figure demonstrates temporal preemption, when k2 only needs one SM, but both SMs are preempted. The overhead on SM 1 is substantial, as it does not have to be preempted. The right-hand sub-figure demonstrates the idea of spatial preemption, in which only one SM is preempted.

times by micro kernels. Such a scenario is common in cloud computing platforms, where the GPU may need to process a large number of short queries from user-facing interactive applications.

3. Overview of FLEP

The goal of FLEP is to enable flexible preemption and determine during runtime which kernels and how these kernels should be preempted. Specifically, FLEP handles two use scenarios. First, the CPU processes with GPU acceleration have different priorities. FLEP assigns the GPU invocations the same priority as their parent processes have. The kernel invocations of higher priority always preempt those of lower priority. Second, when all active kernels (the running kernel and those that are ready to run) have the same priority, FLEP uses a performance model to determine whether the running kernel should be preempted and which kernel should be scheduled next.

Figure 3 shows the overview of FLEP. The framework consists of an online phase and an offline phase. In the offline phase, FLEP completes two tasks. The first task is to build for each kernel a performance model to be used in the online phase to determine the preemption and scheduling order of the kernels with the same priority. The second task is to automatically compile the GPU program to be preemptable. The kernel invocation statement on the CPU side is transformed such that once it is executed, the kernel invocation command, together with its parameters, is intercepted by FLEP's runtime, which determines when to schedule this kernel. The kernel code is also transformed so that it can be notified by its host CPU code to yield all or part of the SMs.

The online phase also has two tasks. The first task is kernel duration prediction and kernel execution logging. It uses the performance model to predict the execution time of each intercepted kernel invocation. As a kernel runs, FLEP records its elapsed time and update the remaining execution time. The second task is making kernel preemption and scheduling decisions. FLEP manages kernels buffered in priority queues, the number of which is equal to the number of distinct priorities of the intercepted kernel invocations. All kernels with the same priority are buffered in the same queue. Whenever a new kernel arrives or a kernel finishes processing, FLEP determines whether to preempt the running kernel, if there is one, and the next kernel to schedule.

In the next two sections, we describe the design and implementation details of the two phases.

4. Offline Phase

In this section, we present the compilation engine design and the performance modeling methodology for GPU kernels.

4.1 Compiler-based Program Transformation

GPU Kernel Transformation The goal of the GPU kernel transformation is to make the GPU kernel execution preemptable. Specifically, the CPU can interrupt the GPU kernel's execution for other kernels to occupy the GPU, and resume the kernel's execution later to finish the remaining tasks. This is challenging due to the way GPU schedules threads. A kernel launch typically launches a large number of CTAs, which are buffered in a FIFO queue managed by a hardware thread scheduler. That means once the scheduler distributes the first CTA of a kernel to an SM, no CTAs from other kernels have a chance to be scheduled until all CTAs from that kernel are dispatched.

The technique, persistent threads [16], limits the number of launched CTAs to the maximum the GPU can simultaneously host. If we view the computation completed by a CTA in the original kernel as a task, persistent threads breaks the 1-to-1 mapping between tasks and CTAs, and instead assigns more tasks to a CTA. The FLEP compiler leverages this idea to enable a naive form of preemptable kernels, as shown in Figure 4 (a). Instead of launching N CTAs as in the original kernel configuration, FLEP configures the kernel to only launch $num\_SMs * max\_CTAs\_per\_SM$ CTAs, where num_SMs is the number of SMs in the GPU and $max\_CTAs\_per\_SM$ is the maximum number of active CTAs an SM can host. The value for $max\_CTAs\_per\_SM$ depends on the hardware resource usage of a CTA, including

```
kernel_temporal_preemption(
                    ...  //original parameters
                    volatile bool *temp_P)
{
   while(1) {
      if(*temp_P == true) return;

      if((task = pull_task()) == NULL)
         return;

      process(task);
   }
}
```

(a)

```
kernel_temporal_preemption_enhanced(
                    ...  //original parameters
                    volatile unsigned int *temp_P,
                    unsigned int L)
{
   while(1) {
      if(*temp_P==true) return;

      for(int i=0; i<L; ++i) {
         if((task = pull_task()) == NULL)
            return;

         process(task); }
   }
}
```

(b)

```
kernel_spatial_preemption(
                    ...  //original parameters
                    volatile unsigned int *spa_P,
                    unsigned int L)
{
   while(1) {
      if(hostSM_ID < *spa_P) return;

      for(int i=0; i<L; ++i) {
         if((task = pull_task()) == NULL)
            return;

         process(task); }
   }
}
```

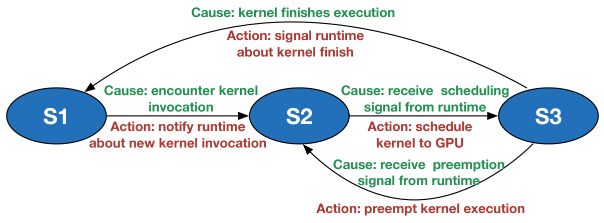
(c)

Figure 4: The three versions of transformed kernels in pseudocode. Figure (a) shows how to enable temporal preemption based on persistent threads. A CTA should quit execution once it finds out the value pointed by $temp\_P$ is changed to 1 by the CPU. Figure (b) shows how to amortize the overhead of the access to $temp\_P$ by processing L tasks. Figure (c) shows how to enable spatial preemption through retrieving the host SM IDs for each CTA.

registers, shared memory, and number of threads, which are either given during runtime or can be derived through a linear scan of the compiled kernel code. It is hence guaranteed that all launched CTAs are active. When a CTA starts execution, it enters a loop, each iteration of which tries to grab a new task to process. A task here refers to the computations that should be done by a CTA in the original kernel. To enable preemption, we introduce a variable, $temp\_P$, which can be read and updated by both the CPU and the GPU. On Nvidia GPUs, such variables can be allocated in pinned memory, a special non-pageable memory in the host memory space. When the CPU decides to preempt this kernel, it sets $temp\_P$ to be true. At the beginning of next iteration of the while loop, the kernel code checks the value of $temp\_P$, and quits execution if it is true. Hence, all SMs are released and the CTAs from the waiting kernel can be scheduled.

This basic transformation enables *temporal preemption*, in which the kernel yields all SMs when $temp\_P$ is set to true. But it may introduce non-trivial runtime overhead. The major source comes from the access to the variable $temp\_P$ allocated in the host memory in each iteration. To reduce the overhead, we further transform the kernel to the form in Figure 4 (b). In this new form, the variable tmp_P is checked once when the CTA processes L tasks. The overhead is thus amortized to L tasks, and hence we name L the amortizing factor. In this work, we always include this optimization for kernel transformation. FLEP can automatically find the smallest value for L through offline tuning (trying different values from small to large) such that the runtime overhead introduced by the transformation is less than 4%.

As explained in Section 2, it is oftentimes not necessary to yield all the SMs for preemption, as the waiting kernel may only launch a small number of CTAs. In this case, we only need to preempt just enough SMs to host those CTAs. To enable such *spatial preemption*, FLEP transforms the target kernel to the form shown in Figure 4 (c). The key technique is to retrieve the ID of the host SM for each active CTA. The variable spa_P shared between the CPU and the GPU now



Cause: kernel finishes execution
Action: signal runtime about kernel finish
S1
Cause: encounter kernel invocation
Action: notify runtime about new kernel invocation
S2
Cause: receive scheduling signal from runtime
Action: schedule kernel to GPU
S3
Cause: receive preemption signal from runtime
Action: preempt kernel execution

Figure 5: State machine for the transformed CPU code.

contains two kinds of information. First, a non-zero value indicates that the kernel should be preempted. Second, the value guides the kernel to yield just part of the SMs. All the CTAs, which run on SMs of ID smaller than spa_P, should quit execution; All the other CTAs keep running until all tasks of the victim kernel are processed. Spatial preemption is equivalent to temporal preemption if spa_P's value is larger than or equal to the number of SMs in the GPU.

Since the transformation treats a CTA as a worker, all threads in the same CTA should be synchronized to process the same task or yield the host SM. The transformed kernel uses a global variable to track the number of finished tasks, which is updated through expensive atomic operations. Fortunately, a CTA only needs to access this variable once when it tries to grab a new task to process. Hence, one optimization is to just use one thread (e.g., the first thread) in each CTA to read the value for $temp\_P$ or spa_P and pull tasks. The information is then stored in shared memory, followed by a synchronization inside the CTA, which is then visible to all the threads in the same CTA. The host SM ID can be retrieved extremely fast on Nvidia GPUs, as a register named `%smid` stores the ID.

CPU Code Transformation The FLEP compiler transforms the CPU code to be preemption-aware, meaning that 1) the kernel invocations should be managed by the runtime and 2) the runtime can signal the CPU to preempt its launched GPU kernel. In the original program, when the CPU invokes a kernel, the GPU driver creates a bunch of CTAs, which wait in a queue to be scheduled to SMs. As explained in Section 2, once the CTAs are managed by the hardware scheduler, we can no longer control the scheduling order. In this work, the transformed CPU code notifies the FLEP runtime for kernel invocations and leaves the kernel scheduling decisions to the runtime. Recall that the FLEP compiler transforms the GPU kernels in a way that they can only be preempted by their host CPU programs, not the runtime. Hence, the FLEP compiler transforms the CPU code, such that it can receive preemption signals from the runtime, and preempts its running kernels.

Figure 5 shows the state machine to demonstrate the functionality of the transformed CPU code. The CPU execution can be in one of the three states: $S1$ (CPU code execution), $S2$ (waiting for scheduling decisions), and $S3$ (waiting for GPU execution). When no GPU kernels are active, the CPU is in $S1$, preparing data for the GPU or processing data transferred back from the GPU. When the CPU encounters a kernel invocation statement, it sends the kernel's information (i.e., kernel's name and configuration parameters) to the runtime without launching the kernel, and enters the $S2$ state. If the runtime decides to schedule this kernel to the GPU, it signals the CPU to launch the kernel. The CPU then enters state $S3$, meaning that the GPU is executing its kernel. There are two out-going edges from $S3$. If the kernel finishes execution, the state transitions back to $S1$ for the CPU to process the results. If the CPU receives a preemption signal from the runtime, it preempts the kernel execution by setting the shared variable described in the GPU kernel transformation to the the appropriate value. The state transitions to $S2$ and the CPU waits for its turn to use the GPU to finish the remaining tasks.

Compilation Infrastructure We implement the FLEP compiler as a source-to-source compiler using the Clang LibTooling library (version 3.9) [1], which provides a CUDA frontend and a set of APIs to easily manipulate the abstract syntax tree. The transformation only needs one simple pass to transform both CPU and GPU code. FLEP then invokes the NVCC compiler provided by Nvidia to compile the transformed program to generate the binary. We did not choose to implement the techniques directly in Clang because its backend does not support the `%smid` register yet.

4.2 Performance Modeling for Kernels and Preemption Overhead Estimation

FLEP's preemption capability can help improve the system's average responsiveness by preempting long-running kernels to reduce the waiting time for short-running kernels. To evaluate this capability, FLEP needs to predict the duration of kernel executions. There are many sophasicated performance models for GPU kernels which may need to access performance counters or are built on complicated analytical analysis. Since the performance model will be used online by FLEP, we follow the performance modeling methodology used by Chen et al. [9] and build lightweight kernel-specific models through linear regression. The model only needs four parameters: grid size, CTA size, input size, and the size of used shared memory, which can be easily obtained given a kernel invocation. For each kernel, we use 100 randomly generated data inputs and L2-norm penalty for training the linear regression model. The inputs are the features and the output is the duration of the kernel.

Note that in this work, we do not aim at building the most accurate performance models. Our goal is to use reasonable and lightweight models to evaluate FLEP's capability of making preemption and scheduling decisions. We make FLEP highly flexible, which can easily integrate other performance models. In Section 6, we show that the linear regression model, despite its simplicity, helps FLEP substantially improve the performance of a set of co-running benchmarks.

Other than kernel duration, the preemption overhead also plays an important role for making online scheduling decisions. Instead of building a model to predict the preemption overhead, we profile the overhead of 50 runs with different inputs and use the average as an estimate of the online preemption overhead.

5. Online Phase

In this section, we describe how FLEP manages the kernel invocations and track their execution status. We also explain how FLEP makes preemption and scheduling decisions given kernels with the same priority and those with different priorities.

5.1 Duration Prediction and Execution Logging

When a kernel is invoked by the CPU, the kernel's name, priority and features for performance modeling (as explained in Section 4) are sent to FLEP's runtime. FLEP feeds the features into the kernel-specific performance model to predict the invocation's duration. Meanwhile, FLEP creates a triplet to record the kernel's execution status, which contains three fields: predicted duration (T_e), waiting time (T_w), and the predicted remaining execution time (T_r). Whenever a kernel is active but not running on the GPU, its T_w accumulates. T_r's initial value is the same as T_e. Its value decreases when it runs on the the GPU. T_e, once initialized, is never updated. T_w and T_r are updated only in the following three cases: 1) a new kernel arrives; 2) a kernel is preempted; and 3) a kernel finishes execution. The recorded timing information is critical to make scheduling decisions, as will be discussed in the remainder of this section.

5.2 Preemption and Scheduling Decisions

We design and implement two scheduling policies with different preferences. FLEP can be configured to use either one.

```
0:  // Executed when a new kernel (Kn) is invoked
1:
2:  if (kernel Kr is running on GPU) {
3:     if (Kr.priority < Kn.priority) {
4:        Preempt(Kr)
5:        Enqueue(Kr.priority)
6:        Schedule_to_GPU(Kn) }
7:     else if (Kr.priority > Kn.priority) {
8:        Enqueue(Kn, Kn.priority) }
9:     else //Kr.priority == Kn.priority {
10:       Schedule_for_queue(Kn.priority)
11:    }
12: } else {
13:    Enqueue(Kn, Kn.priority)
14:    Schedule_for_queue(Kn.priority)
15: }
16:
------------------------------------------------------------
17: //Executed when one kernel is finished
18:
19: //find the non-empty queue of highest priority hp
20: Schedule_for_queue(hp)
21:
------------------------------------------------------------
22: //Definition of the key scheduling function
23:
24: schedule_for_queue(p){
25:
26:    //find the kernel Ks in the queue of priority level p
27:    //with smallest Tr
28:    if (kernel Kr is running on GPU){
29:       if(Kr.Tr > Ks.Tr + preemption_overhead) {
30:          Preempt(Kr)
31:          Schedule_to_GPU(Ks) }
32:    }  else {
33:          Schedule_to_GPU(Ks) }
34: }
```

Figure 6: Online algorithm to make preemption and scheduling decisions.

5.2.1 Highest priority first scheduling with performance degradation minimization (HPF)

Figure 6 shows the algorithm of HPF' design. HPF makes new scheduling decisions in two cases. **Case 1:** A new kernel (K_n) is invoked (lines 0–15). HPF first checks whether a kernel K_r is running on the GPU. If the new kernel's priority is higher, HPF preempts K_r and schedule K_n. Since HPF always preempts the running kernel if any higher-priority kernels are waiting, it knows K_n is the only kernel whose priority is higher than that of K_r. If K_n's priority is lower than K_r's, HPF enqueues K_n to the appropriate queue according to its priority. When the two kernels' priorities are equal, HPF invokes a function, $Schedule\_for\_queue$, to determine the scheduling order of kernels with the same priority. We discuss this function later. **Case 2:** A kernel is finished (lines 17–20). HPF identifies the first non-empty queue with the highest priority and invokes $Schedule\_for\_queue$ to schedule kernels in that queue.

To make decisions for kernels with the same priority, HPF concerns average performance degradation. Given a kernel K, its performance degradation is defined as $(T_w^K + T_e^K)/T_e^K$, where T_w^K is the overall waiting time when the kernel is finished. The goal is to minimize the sum of the performance degradation for all N kernels defined as $\sum_i^N (T_w^{K_i} + T_e^{K_i})/T_e^{K_i}$. Muthukrishnan et al. [26] proved that shortest remaining time scheduling algorithm yields average performance degradation no worse than twice that of the optimal schedule. Hence, we implement shortest remaining time scheduling in the function $Schedule\_for\_queue$ as shown by lines 22–34. HPF first identifies the kernel K_s, whose T_r (remaining execution time) is the shortest and schedule it to the GPU. To speed up this operation, when a kernel is enqueued, the runtime makes sure that the kernels are ordered based on their T_r values. Hence, K_s is always at the beginning of the queue. If a kernel, K_r, is running on the GPU, HPF needs to determine whether to preempt it by comparing K_r's remaining execution time with the sum of K_s's remaining time and the preemption overhead. HPF preempts K_r only if K_r's remaining execution time is larger. We involve the preemption overhead to make the preemption decision, because it influences all the other kernels' waiting time.

5.2.2 Fairness first scheduling under an overhead constraint (FFS)

FFS tries to ensure that a kernel gets its share of the GPU proportional to its priority. To achieve this goal, FFS implements weighted round-robin scheduling. In each round, a kernel i runs on the GPU for an epoch of length $T \times W_i$, where W_i is the weight assigned to the kernel's priority and T is a parameter to control context switch frequency. Since FLEP enables preemption through compiler transformation, too frequent context switching (i.e. a small value for T) introduces excessive preemption overhead. To determine T, FFS introduces a configurable threshold $max\_overhead$, which represents the maximum performance degradation the user would like to sacrifice for fairness. FFS calculates the minimum value for T such that the following constraint holds.

$$\frac{\sum_i Oi}{T\sum_i W_i} \leq max\_overhead$$

where O_i represents the preemption overhead to preempt kernel i.

6. Evaluation

6.1 Methodology

Benchmarks We select 8 benchmarks shown in Table 1 from three benchmark suites, including SHOC [10], Rodinia [6], and CUDA SDK [27]. The benchmarks cover multiple domains, such as machine learning, scientific computing, and dynamic programming. We give a brief description of the benchmarks. CFD is a finite volume solver for fluid dynamics applications. NN computes the 10 nearest neighbor result for a given point. PF finds the path on a 2-D grid using dynamic programming. PL estimates the location of a target object based on a Bayesian framework. MD computes the forces among a large number of atoms. SPMV multiplies a sparse matrix with a dense vector. MM multiplies two dense square matrices. VA adds two dense vectors.

We make sure the selected benchmarks have dramatically different characteristics. The kernel size varies from 6 lines of code to 130 lines of code. This is important to evaluate the efficiency of FLEP. For example, the benchmark VA only has 6 lines of code in the kernel without a loop struc-

Table 1: Benchmarks and Kernel Execution Time on Three Inputs

Benchmark	Source	Description	Lines of code in kernel	exe. time (large)	exe. time (small)	exe. time (trivial)	amortizing factor
CFD	Rodinia [6]	finite volume solver	130	11106 (us)	521 (us)	81 (us)	1
NN	Rodinia [6]	nearest neighbor	10	15775 (us)	728 (us)	55 (us)	100
PF	Rodinia [6]	dynamic programming	81	7364 (us)	811 (us)	57 (us)	150
PL	Rodinia [6]	Bayesian framework	24	5419 (us)	952 (us)	83 (us)	100
MD	SHOC [10]	molecular dynamics	61	15905 (us)	938 (us)	90 (us)	1
SPMV	SHOC [10]	sparse matrix vector multi.	23	5840 (us)	484 (us)	68 (us)	2
MM	CUDA SDK [27]	dense matrix multiplication	74	2579 (us)	1499 (us)	73 (us)	2
VA	CUDA SDK [27]	vector addition	6	30634 (us)	720 (us)	49(us)	200

ture. Hence, controlling the runtime overhead introduced by FLEP is a critical task. Through offline tuning as explained in Section 4, we choose different amortizing factors for different benchmarks, as shown in the last column of Table 1. For each benchmark, we use three inputs (large, small, and trivial) to evaluate different preemption scenarios. Table 1 lists the execution times of the benchmarks on the inputs, when each benchmark runs alone on the GPU. Both large and small inputs need all SMs on the GPU for optimal performance. The small inputs only need part of SMs to evaluate spatial preemption. For example, if the GPU has 15 SMs, which can simultaneously host 120 active CTAs, the trivial input may only create 40 CTAs and only needs 5 SMs.

Evaluation Strategy for Temporal Preemption We evaluate FLEP with all benchmarks using the large and small input sets in two scenarios. In the first scenario, we emulate a multi-tasking environment, in which applications have different priorities. More specifically, we evaluate pairs of co-running applications in the form of A_B, where benchmark A runs the large input with a high priority, and B takes the small input with a low priority. To evaluate FLEP's effectiveness to avoid priority inversion with the HPF policy, we use as the metric the speedup with preemption support provided by FLEP over the original turnaround time (i.e., waiting time plus execution time) for the high priority benchmarks. To evaluate the FFS policy, we continuously monitor the GPU share percentage of the two kernels and throughput degradation.

In the second scenario, we assign the co-running benchmarks the same priority. Based on the performance model and runtime decision engine, FLEP determines whether to preempt the running kernel and which kernel to run next. We use System Throughput (STP) and Average Normalized Turnaround Time (ANTT) as metrics defined by Eyerman [14] for evaluating the performance of multi-programmed applications. STP shows the overall system throughput, and ANTT shows average responsiveness. Note that unlike most previous studies, in all the experiments we leverage MPS to co-run the applications in different GPU contexts instead of integrating the kernels into the same program. Unless noted otherwise, we use MPS-based executions as the baseline for all co-run experiments. We average the results from 10 repetitive runs.

Evaluation Strategy for Spatial Preemption For each benchmark pair A_B, we run A on the large input and B on the trivial input with a higher priority. We focus on preemption overhead to show the advantage of only yielding part of the SMs with the other SMs keeping running A's CTAs. We run benchmarks A and B using MPS to get the aggregate kernel execution time T_{org}. For the co-run with preemption support, we first launch the kernel of A followed by the kernel of B. We record T_{FLEP} as the time period between the launch of A's kernel and the finish of both A and B's kernels. The preemption overhead is defined as $(T_{FLEP} - Torg)/T_{org}$.

Machine Environment The experiments are carried out on a machine equipped with one Intel Xeon E3-1286 V3 CPU (3.70 GHz and 4 cores) and an Nvidia K40 GPU with 15 SMs. The host memory is 128 GB, and the GPU device memory is 12 GB. The OS is Ubuntu Linux 14.04 with kernel version 3.13.0. The CUDA toolkit version is 7.0. In all experiments, we enable MPS for benchmark co-runs on the GPU.

6.2 Prediction Accuracy for Kernel Duration

Figure 7 shows the average prediction errors when the benchmarks run large and small inputs. We observe reasonably accurate results, an average of 6.9% deviation from the real execution time. The accuracy varies from 2.7% to 12.2% due to the dramatically different runtime behaviors of the benchmarks. NN, MM, and VA have regular parallelism and memory access patterns. The degree of control and divergence is minimal and hence the difference between inputs mainly lies in data size. For example, VA (vector addition) has perfect spatial locality, which results in perfect memory coalescing and predictable memory performance. On the contrary, the other benchmarks' runtime behaviors largely depend on the input features. For example, in MD the memory access pattern is determined by the neighborhood relations among the simulated atoms. The training data may not be representative enough for the data used in the experiments. SPMV is even more difficult to predict, as its control

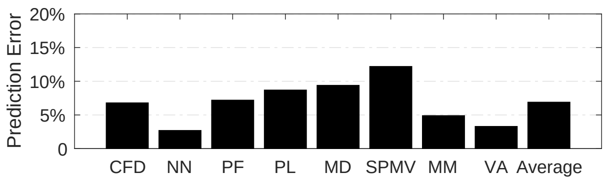


Figure 7: Kernel duration prediction errors.

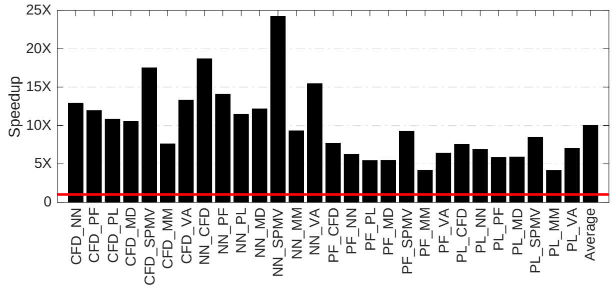


Figure 8: Performance improvement for high-priority kernels over their execution in MPS-based co-runs.

and memory access patterns are sensitive to the non-zero element distribution in the input matrix. Nevertheless, as later sections will show, the prediction helps FLEP make preemption decisions, which dramatically improves the overall performance and responsiveness.

6.3 Results on Temporal Preemption

6.3.1 Two-kernel co-runs with HPF scheduling

We select CFD, NN, PF, and PL to run on the large inputs as low-priority workloads. We pair each of them with each other benchmark running on the small inputs as high-priority workloads. For each co-run pair, we launch the high-priority workloads immediately after the low-priority workload starts to run. Recall that in the default co-runs based on MPS, the low-priority workload creates a large number of CTAs, whose execution blocks the whole GPU until finished. FLEP with the HPF scheduling policy, however, preempts the low-priority running kernel, if there is any waiting kernels with higher priority. Figure 8 shows substantial performance improvements for the high-priority workloads. We observe on average 10.1X speedup, with up to 24.2X improvement for SPMV when it coruns with NN. The smallest speedup is 4.1X for MM with PF as co-runner. The large variance in speedups is because of the different execution times shown in Table 1. Running alone on the GPU, NN needs 15775 us to process the large input, while SPMV on the small input only runs for 484 us. Hence, preempting NN greatly reduces the waiting time for SPMV. However, PL's execution time is only around 3.6 times longer than that of MM, leading to an upper bound of 4.6X speedup from temporal preemption.

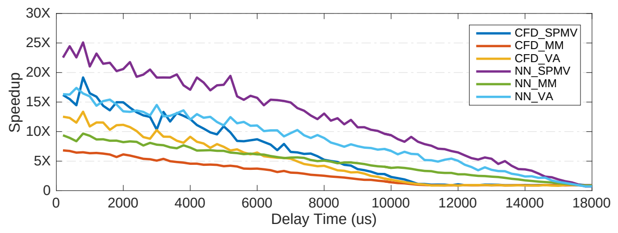


Figure 9: Performance improvement for high-priority kernels over their execution in MPS-based co-runs when delay is introduced between kernel invocations.

We further evaluate the cases in which the high-priority kernel's invocation does not immediately follow that of the low-priority kernel. We introduce a delay between the two invocations in each co-run, which is varied to show the influence on the speedup of the high-priority workloads. Figure 9 demonstrates that as the delay increases, the speedup decreases almost linearly. The results align well with the intuition, because during the delayed time, the low-priority finishes some work, which virtually reduces the waiting time for the high-priority workload in the default co-run. Hence, the performance improvement potential is less. We observe a plateau close to 1 at the end of each performance curve, because the delay time is longer than the execution time of the low-priority workload. The high-priority workload, once launched, can thus be immediately executed, yielding no performance improvement potential.

In the evaluation of the equal-priority setting, we give the co-running benchmarks the same priority. For each of MD, MM, SPMV, and VA, we run it on the small input together with each of the other 7 benchmarks on the large inputs. Figure 10 shows the ANTT improvement over the MPS-based co-runs. FLEP enhances ANTT by 8X on average for the 28 benchmark pairs. When the long-running kernel is invoked, FLEP immediately schedules it to the GPU. After that, the short-running kernel is invoked, and FLEP decides to preempt the running kernel and schedule the short-running ones for improvement on average responsiveness. Note that the improvement is less than that of the performance improvement for high priority workloads in the previous results. The reason is that ANTT's definition involves both co-running kernels, but the long-running kernel does not benefit from the preemption. Figure 11 shows the STP degradation due to the overhead introduced by FLEP. Higher bars indicate lower throughput. The average STP degradation is around 5.4%. For systems, whose responsiveness is critical, trading small degradation on system throughput for substantial improvement on ANTT is desirable.

6.3.2 Three-kernel co-runs with HPF scheduling policy

In real-world deployment, more than two applications may share the same GPU. We hence go beyond benchmarks pairs and evaluate the co-runs of three benchmarks. We randomly

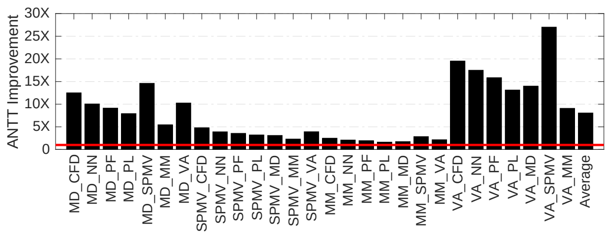


Figure 10: Improvement on average normalized turnaround time.

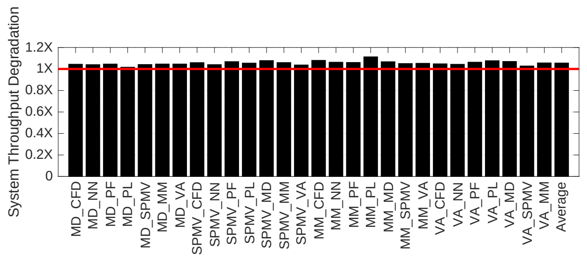


Figure 11: Performance degradation on system throughput.

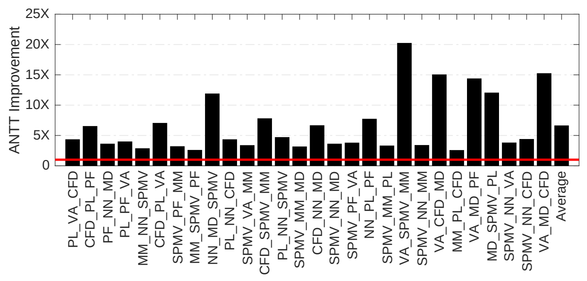


Figure 12: ANTT improvement on three-benchmark co-runs.

choose 28 benchmark triplets. For a triplet represented by A_B_C, we launch the kernel from benchmark A on the large input, followed by those from B and C on the their corresponding small inputs. Figure 12 shows the improvement on ANTT from FLEP over the default co-runs enabled by MPS. The improvement is up to 20.2X for the triplet VA_SPMV_MM. Thanks to FLEP's awareness of the kernel duration, FLEP decides to preempt VA's execution and schedule SPMV, the shortest kernel among the three, which also dramatically reduces the waiting time for MM. After SPMV's kernel finishes, FLEP schedules MM before VA, as the remaining execution time of VA is significantly longer than MM. On average, FLEP provides 6.6X improvement.

When multiple kernels are in the wait queue, existing frameworks [23, 25], despite providing no preemption support, can schedule shorter kernels to improve turnaround time. We hence evaluate kernel reordering but only observe around 2.3% improvement on ANTT due to the blocking long kernels launched at the very beginning.

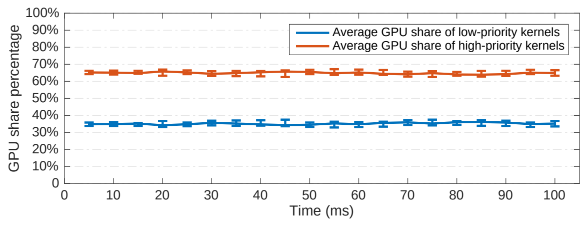


Figure 13: Average GPU share for high and low priority kernels with FFS.

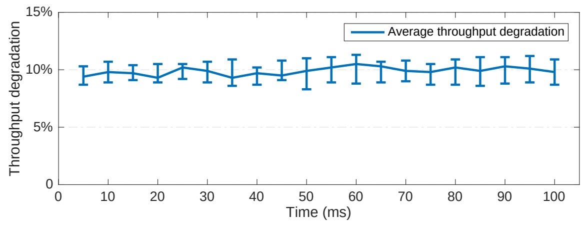


Figure 14: Average throughput degradation with FFS.

6.3.3 Two-kernel co-runs with FFS scheduling policy

We use the same co-run pairs as in the experiments for the HPF scheduling. The only change is that each benchmark keeps invoking the same kernel in an infinite loop. Since HPF always favors high-priority kernels, in this scenario the low-priority kernels do not have any share of the GPU. Figure 13 shows FFS's enforcement of weighted fairness when the weight ratio between the high and low priorities is 2:1. The curves represent average GPU share across all co-run pairs. We observe roughly 2/3 share of the GPU time for high-priority workloads and around 1/3 share for low-priority workloads, which is the goal for the weighted fairness control. The error bars are very narrow, indicating slight differences across the co-run pairs. Figure 14 presents the throughput degradation with $max\_overhead$ empirically selected as 10%. The results show that FLEP keeps the performance degradation close to the threshold with small variation across the co-runs. Note that FLEP allows users to configure this parameter for the desired level of fairness granularity.

We elide the results for three-kernel co-runs with FFS due to space limit, because they are similar to those of the two-kernel co-runs.

6.4 Results on Spatial Preemption

To evaluate spatial preemption, for each co-run pair, we invoke a low-priority kernel to run the large input followed by a high-priority kernel invocation with the trivial input. FLEP preempts just enough SMs to host all CTAs launched by the high-priority kernel. We co-run each benchmark with all the other benchmarks and average the preemption overhead. Figure 15 shows the reduction on preemption overhead from spatial preemption compared to temporal pre-

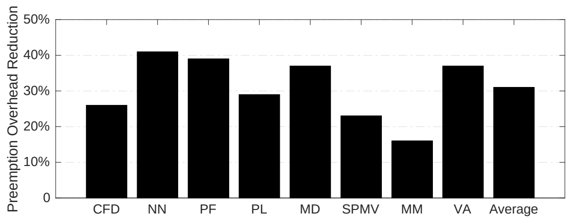


Figure 15: Preemption overhead reduction through spatial preemption.

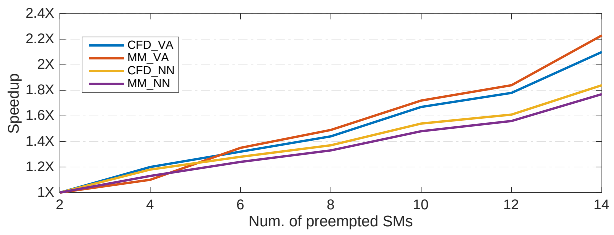


Figure 16: Performance improvement over baseline spatial preemption.

emption, which preempts all SMs. We observe an average of 31% reduction (up to 41% for NN), which makes significant difference if small kernels keep preempting long-running low-priority kernels. There are two plausible reasons for the significant reduction. First, spatial preemption avoids unnecessary SM preemption. In contrary, temporal preemption interrupts all SMs' execution. Second, the spatial co-running of kernels with different characteristics (e.g., memory-intensive and compute-intensive) may better utilize the hardware resource.

Spatial preemption, however, has the side effect of sacrificing the performance of high-priority workloads. Suppose a high-priority workload launches 8 CTAs. Although yielding one SM is enough to host all of them, yielding 8 SMs to run one CTA on each results in better performance thanks to the mitigated contention on the SMs. Figure 16 shows the performance improvement from yielding more than enough SMs to host the launched CTAs on four co-run pairs as case studies. Both NN and MD need two SMs to host all CTAs; so the baseline setting preempts two SMs. On the one hand, we observe that NN and MD's performance indeed improves as we increase the number of preempted SMs. On the other hand, the largest speedup over the baseline is only around 2.22X, indicating that the side effect may not be serious. FLEP's flexibility enables fine-tuning of the number of SMs to yield for the user to strike the desired trade-off between preemption overhead and performance degradation of the high-priority kernels.

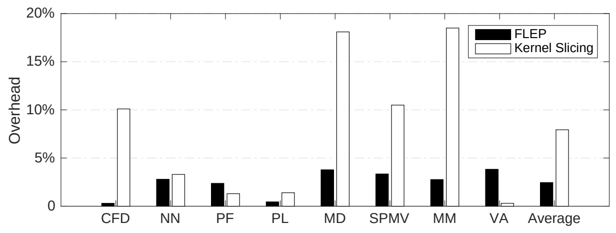


Figure 17: Comparison of the overhead introduced by FLEP and kernel slicing for single-kernel runs.

6.5 Overhead of Single-kernel Runs with Preemption Enabled

Figure 17 shows the overhead introduced by FLEP and kernel slicing. The results demonstrate that without being preempted, the transformed kernels by FLEP run almost as fast as the original kernels with only 2.5% overhead on average. To achieve the same preemption granularity, kernel slicing introduces an average overhead of 8%. For NN, PF, and PL, FLEP and kernel slicing have comparable overhead. However, kernel slicing performs much worse for CFD, MD, SPMV, and MM, because kernel slicing incurs an excessive number of kernel invocations. VA is the only benchmark, for which kernel slicing substantially outperforms FLEP, as FLEP has to use a large amortizing factor (200). In this case, kernel slicing only needs to slice the kernel in a coarse-grained manner, hence its negligible overhead.

7. Limitations

FLEP currently only supports CUDA on Nvidia GPUs. To our best knowledge, OpenCL, or any other GPU programming model, does not yet provide an interface to retrieve SM IDs online on either Nvidia GPUs or AMD GPUs, but FLEP relies on this functionality to enable spatial preemption. However, this is not a limitation for FLEP to support only temporal preemption, as long as the underlying GPU satisfies the requirements to run persistent threads. This work shows that online SM ID retrieval is a useful feature to manage kernel co-runs.

The FLEP compiler works on program source code, which may not be available in the cloud computing environment. However, previous work shows that extracting the GPU code from executables and transforming it through compilers is doable [41]. In our future work, we will investigate in this direction to make FLEP practical in more scenarios.

Choosing the appropriate amortizing factor is important for controlling the preemption overhead, which is cumbersome as a required offline step. Moreover, a too large amortizing factor means slower responsiveness for preemption, because a CTA needs to do more work before checking whether it should yield the execution. Future communication technology between the CPU and GPU, such as

NVLink [2], can dramatically reduce the communication latency and hence the overhead incurred by FLEP.

8. Related Work

Managing GPUs in a multi-tasking environment is critical for achieving high performance and fairness. The problem has drawn significant attention in several closely related research fields. We group existing work into three categories:

Compiler- or runtime-based: Several frameworks have been propsoed to enable preemption on GPUs through kernel slicing [41, 19, 5]. The basic idea is to slice long-running kernel invocations into multiple short-running ones. GPU applications can be preempted when short-running kernel invocations are finished. Kernel slicing has two major problems. First, it is difficult to determine the most appropriate length of each slice. Second, the approach introduces overhead even if the kernel is never preempted. In contrary, FLEP does not slice kernels and is shown to introduce negligible overhead. An independent work done by Chen et. al. proposed similar compiler techniques as in FLEP to circumvent the limitation of the kernel slicing approach [8, 7], but they did not enable spatial preemption.

Pai and others observed that carefully scheduling multiple kernels to co-run together can significantly improve GPU utlization [31]. A similar approach is used in several other works [40, 24, 15]. A major drawback of this line of work is that the co-running kernels are artificially synthesized into the same GPU context, which is not realistic in real-world settings. FLEP does not have this assumption and can work with applications with separate GPU contexts.

When multiple GPU applications co-run together on the same GPU, the device memory and PCIe bus may becomes critical resources if the combined working set is too large. Wang and others [37, 38] implemented a GPU memory management framework to treat the device memory as a data cache, and coordinate the device memory usage of the co-running applications. Baymax proposed by Chen and others [9] built an analytical model to guide data transfers from different applications. Kehne et al. [22] proposed GPUSwap, which modifies the GPU driver to enable device memory oversubscription in a transparent way. FLEP currently assumes the combined working set can fit into the device memory. We plan to integrate GPUSwap into FLEP to handle large working sets.

Hardware-based: Tansaic et al. [35] first proposed hardware extensions to enable preemption on GPUs. They leverage the special semantics of the GPU programming model and design two preemption techniques: switch and drain. Chimera [32] extended this work by introducing a third technique to flush the GPU cores, and utilizes all three techniques to achieve a required latency. Our work is a software-only framework that works on commodity GPUs.

Adraens et al. [4] showed the benefits from a hardware extension to co-run memory-bound and compute-bound GPU kernels in a spatial manner. Wang and others [39] also showed similar benefits from such fine-rained hardware partitioning. Jog et al. [18] evaluated the efficiency of the memory system when GPU kernels simultaneously run on different SMs, based on which they propose new memory scheduling policies to improve the overall memory throughput.

OS-based: Some studies modified the operating system or the GPU drivers to manage GPUs as first class computing resources [21, 33, 20]. The focus is on performance isolation and interference elimination. As GPUs are increasingly being adopted in data centers, full GPU virtualization becomes an important research area. Tian et al. [36] implemented gVirt to virtualize Intel GPUs for graphics workloads. Dong and others [12] further improved gVirt's performance through hybrid page tables. Suzuki et al. implemented GPUvm for full virtulization of Nvidia GPUs on the Xen hypervisor [34]. All of those works assume that the GPU is non-preemptive, while FLEP, by providing preemption service, complements them to provide more opportunities for performance optimization.

9. Conclusions

In this paper, we presented FLEP, the first practical, transparent, and flexible software system to enable preemption on commodity GPUs. FLEP works on GPU programs in different GPU contexts, and does not require any manual code change. The FLEP compiler automatically transforms both the CPU and GPU part of the program to support kernel invocation interception and runtime kernel preemption. The FLEP runtime leverages kernel-specific performance models to predict the duration for kernel invocations, based on which preemption and kernel scheduling decisions are made. FLEP improves the performance of the high-priority kernels by 10X on average in 28 co-runs. When the co-running kernels have equal priority, FLEP preempts the long-running kernels and improves the ANTT by 8X for two-kernel co-runs and 6.6X for three-kernel co-runs, respectively, on average. When the waiting kernels do not generate enough CTAs to occupy all SMs, FLEP only preempts part of the SMs to reduce the preemption overhead by up to 41% compared to temporal preemption. FLEP demonstrates that preemptive programming can be efficiently supported on commodity GPUs.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. The effort of this project is funded by National Science Foundation Grants 1464216 and 1618912.

References

[1] clang: a C language family frontend for LLVM. `http://clang.llvm.org/` ; accessed 23-02-2016.

[2] NVLink Communication Protocol. `https://en.wikipedia.org/wiki/NVLink`.

[3] OpenCL. `http://www.khronos.org/opencl/`.

[4] J. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for GPGPU spatial multitasking. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 79–90, 2012.

[5] C. Basaran and K. Kang. Supporting preemptive task executions and memory copies in gpgpus. In *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 287–296, 2012.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[7] G. Chen, X. Shen, and H. Zhou. A software framework for efficient preemptive scheduling on gpu. Technical report, North Carolina State University, 2016.

[8] G. Chen, Y. Zhao, X. Shen, and H. Zhou. Effisha: A software framework for enabling efficient preemptive scheduling of gpu. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'17, 2017.

[9] Q. Chen, H. Yang, J. Mars, and L. Tang. Baymax : Qos awareness and increased utilization of non-preemptive accelerators in warehouse scale computers. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[10] 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.

[11] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[12] Y. Dong, M. Xue, X. Zheng, J. Wang, Z. Qi, and H. Guan. Boosting gpu virtualization performance with hybrid shadow page tables. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 517–528, Santa Clara, CA, July 2015. USENIX Association.

[13] G. A. Elliott and J. H. Anderson. Real-world constraints of gpus in real-time systems. In *17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2011, Toyama, Japan, August 28-31, 2011, Volume 2*, pages 48–54, 2011.

[14] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[15] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent gpgpu kernels. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012. USENIX.

[16] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing*, page 14, May 2012.

[17] U. Hoelzle and L. A. Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[18] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das. Anatomy of gpu memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, pages 223–234, New York, NY, USA, 2015. ACM.

[19] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 - December 2, 2011*, pages 57–66, 2011.

[20] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[21] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class gpu resource management in the operating system. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 401–412, Boston, MA, 2012. USENIX.

[22] J. Kehne, J. Metter, and F. Bellosa. Gpuswap: Enabling oversubscription of gpu memory through transparent swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*, pages 65–77, Istanbul, Turkey, Mar. 14–15 2015.

[23] T. Li, V. K. Narayana, and T. A. El-Ghazawi. Reordering GPU kernel launches to enable efficient concurrent execution. *CoRR*, abs/1511.07983, 2015.

[24] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen. Efficient GPU spatial-temporal multitasking. *IEEE Trans. Parallel Distrib. Syst.*, 26(3):748–760, 2015.

[25] C. Margiolas and M. F. P. O&#039;Boyle. Portable and transparent software managed scheduling on accelerators for fair resource sharing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 82–93, New York, NY, USA, 2016. ACM.

[26] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online scheduling to minimize average stretch. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, Washington, DC, USA, 1999.

[27] NVIDIA. Cuda software development toolkit v7.0 . https://developer.nvidia.com/cuda-toolkit-70.

[28] NVIDIA. Nvidia's next generation cuda computer architecture: Fermi. Technical report.

[29] NVIDIA. Next generation cuda computer architecture kepler gk110. Technical report, 2012.

[30] NVIDIA. Sharing a gpu between mpi processes: multi-process service (mps) overview. Technical report, 2013.

[31] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 407–418, New York, NY, USA, 2013.

[32] 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*, ASPLOS '15, pages 593–606, New York, NY, USA, 2015. ACM.

[33] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[34] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. Gpuvm: Why not virtualizing gpus at the hypervisor? In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 109–120, Philadelphia, PA, June 2014. USENIX Association.

[35] 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*, ISCA '14, pages 193–204, Piscataway, NJ, USA, 2014. IEEE Press.

[36] K. Tian, Y. Dong, and D. Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 121–132, Philadelphia, PA, June 2014. USENIX Association.

[37] 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.

[38] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. *Proc. VLDB Endow.*, 7(11):1011–1022, July 2014.

[39] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel: Fine-grained sharing of gpgpus. *IEEE COMPUTER ARCHITECTURE LETTERS*, PP(99):748–760, 2015.

[40] B. Wu, G. Chen, D. Li, X. Shen, and J. S. Vetter. Enabling and exploiting flexible task assignment on GPU through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*, pages 119–130, 2015.

[41] H. Zhou, G. Tong, and C. Liu. GPES: a preemptive execution system for GPGPU computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA, April 13-16, 2015*, pages 87–97, 2015.