GPES: A Preemptive Execution System for GPGPU Computing

Husheng Zhou, Guangmo Tong, and Cong Liu
Department of Computer Science, The University of Texas at Dallas

***Abstract*—Graphics processing units (GPUs) are being widely used as co-processors in many application domains to accelerate general-purpose workloads that are computationally intensive, known as GPGPU computing. Real-time multi-tasking support is a critical requirement for many emerging GPGPU computing domains. However, due to the asynchronous and non-preemptive nature of GPU processing, in multi-tasking environments, tasks with higher priority may be blocked by lower priority tasks for a lengthy duration. This severely harms the system's timing predictability and is a serious impediment limiting the applicability of GPGPU in many real-time and embedded systems. In this paper, we present an efficient GPGPU preemptive execution system (GPES), which combines user-level and driver-level runtime engines to reduce the pending time of high-priority GPGPU tasks that may be blocked by long-freezing low-priority competing workloads. GPES automatically slices a long-running kernel execution into multiple subkernel launches and splits data transaction into multiple chunks at user-level, then inserts preemption points between subkernel launches and memory-copy operations at driver-level. We implement a prototype of GPES, and use real-world benchmarks and case studies for evaluation. Experimental results demonstrate that GPES is able to reduce the pending time of high-priority tasks in a multi-tasking environment by up to 90% over the existing GPU driver solutions, while introducing small overheads.**

I. Introduction

Graphics processing units (GPUs) are being widely used as co-processors in many domains to achieve acceleration. They are particularly capable of executing data-parallel applications, due to their highly multi-threaded architecture and high-bandwidth memory. The NVIDIA GTX 770, which is a representation of the latest discrete GPUs, integrates 1536 processing cores on a chip and reaches up to 3000 GFLOPS. Along with the support of the CUDA [19] programming model developed by NVIDIA, GPUs can be easily used for general-purpose computing in addition to dedicated graphics applications, a.k.a. GPGPU. For instance, for the emerging autonomous driving cyber-physical systems, GPUs are proven to be a good option to accelerate many real-time computation intensive workloads such as real-time motion planning [15] and object recognition [9]. It has also been shown in [7] that GPU-accelerated systems can achieve ten times speedup for software routers.

However, the NVIDIA proprietary driver is mainly designed for accelerating particular high-performance applications, which may not be efficiently applicable for GPGPU in real-time multi-tasking environments. Once pieces of GPU-accelerated code, a.k.a. kernels, from different applications are loaded onto the GPU, they are dispatched by hardware scheduler. Such hardware-based scheduling will harm the response time of high-priority GPGPU tasks, since the hardware scheduler does not consider task priorities. Consequently, due to the asynchronous and non-preemptive nature of GPU processing, in multi-tasking environments, a task with higher priority or urgency (e.g., with a shorter deadline) may be blocked by lower priority tasks that already started running on GPUs. This severely harms the system's timing predictability and is a serious impediment limiting the applicability of GPGPU in many real-time systems. Specifically, GPGPU computing is a triple-step procedure: host-to-device input memory-copying, performing computation, device-to-host output memory-copying. GPUs are exclusive to other tasks when being occupied by certain tasks, which could easily cause priority inversions and thus deadline misses.

In order to apply GPGPU to real-time workloads, additional supports are needed to address such challenges, which is the focus of this paper. We present a GPGPU preemptive execution system (GPES) to better achieve preemptivity and prioritization for GPGPU computing. Previous work such as Gdev [13] and TimeGraph [12] provide fairness and prioritization capabilities for multiple GPU applications. However, they can not address the blocking issues of large memory-copy and long-running computation. RGEM [11] introduces a responsive execution model which splits a memory-copy into multiple chunks and launches the kernels of different GPGPU tasks in order according to their priorities. However, in RGEM, only data transmission can be preempted. Once all input data are ready, the kernel execution is still in burst mode. Thus, a kernel from a high priority task can still be blocked by a low-priority kernel that already started execution till its completion. As discussed in RGEM [11], it leaves the preemption support for GPGPU kernel execution as future work, seeking solutions at firmware implementation of microcontroller. Actually our experiments show that the controllability of kernel execution is entirely implemented in the GPU hardware. Reverse-engineering of firmware heads to a wrong direction.

The contribution of this paper is to develop an efficient and comprehensive GPGPU solution, namely GPES, for real-time computing. We implement the prototype of GPES in an NVIDIA GTX 480 GPU based on the NOUVEAU open-source driver [4]. GPES consists of a user-space library and an OS-space module. In the user-space library, we re-implement a set of CUDA APIs to perform kernel slicing and data slicing, which divide a kernel into multiple subkernels and also partition a data transaction into chunks. Specifically, we introduce a novel binary rewriting technique to transparently rewrite the kernel code for relatively simple kernels; while for those complex kernels we develop a source-to-source transformation technique and compile the transformed kernel code into CUDA binaries. In the OS module, we insert GPU-to-CPU interrupts

978-1-4799-8603-3/15/$31.00 ©2015 IEEE

at the boundaries between subkernel launches and data chunk transactions for higher priority tasks, as they may serve as the preemption points. GPES uses the IRQ handler prepared for GPU-to-CPU interrupts to switch contexts between different applications according to their priorities. We use real-world benchmarks and a video processing case study for evaluation. Experimental results demonstrate that GPES is able to reduce the pending time of high-priority tasks by up to 90% over the existing GPU driver solutions (specifically, NVIDIA and Gdev [13]), while introducing reasonably small overheads.

The rest of this paper is organized as follows. Sec. II presents the system model. Sec. III describes a motivational measurements-based case study. Sec. IV highlights some of the design and implementation details that deserve articulation. Sec. V discusses our implementation methodology and experimental results. Sec. VI describes related work. Sec. VII concludes.

II. System Model

In this paper, we target the GPGPU computing architecture composed of a discrete GPU and a multi-core CPU. We use CUDA as the underlying programming model. Current operating systems (OS) treat discrete GPUs as I/O devices. All communications between GPU and CPU are via the PCI bus. GPGPU applications typically follow the following execution flow: (i) initializing the GPU device, (ii) allocating GPU device memory, (iii) transferring data from host memory to device memory, (iv) launching the computation work (kernel) on GPU, (v) copying results back to host memory, and (vi) freeing device memory and closing device. GPES is designed based on the NVIDIA GF100 architecture [20] (note that, it is also applicable to other NVIDIA GPU architectures [22], [23]). We define some terminologies that will be frequently used in the rest of this paper.

Context: Context conceptually represents separate virtual address space in the GPU hardware. GPU resource management is context-based. In order to use the GPU for computation, an application must explicitly create a context. Different applications are running on different contexts. This is necessary because the GPU memory controller does not support multiple virtual address spaces simultaneously. To be convenient, in the rest of this paper, it is assumed that kernels and memory-copy operations from different applications have different contexts. If multiple kernels from different contexts are loaded at once, they can co-exist on the same GPU and will be dispatched in turn by the hardware scheduler. Hence, they can execute in an interleaved manner through automatic context switching performed by hardware, but not simultaneously.

MP/SM, SP: Streaming multiprocessor (MP or SM) is the internal unit of NVIDIA GPU hardware that performs the actual computations. NVIDIA GPU consists of several SMs, each of which is further divided into shader processors (SP). The number of SMs and SPs is product-specific. Low-end GPUs typically have one SM, and high-end GPUs have 15 or 16 SMs. Take NVIDIA GTX 480 for example, which is based on GF110 architecture, it has 15 SMs.

Thread, Warp, Block, Grid: The NVIDIA CUDA [19] programming model consists of four levels of hierarchy. In Fermi or Kepler [22] architecture, 32 threads make up a warp. Warps are the basic units of execution on the GPU. Threads in each warp are executed together. A group of warps stitch together to form a block. These blocks are combined to form a grid. A grid is corresponding to an execution kernel, thus in the rest of these paper, kernel launch and grid launch are interchangeable forms. When executing a kernel, the corresponding entire grid is mapped to one GPU device, blocks are mapped to SMs (MPs), and internally, computations are scheduled warp by warp. Notice that, in Fermi and Kepler architectures, grids from different kernels can execute on the same GPU device simultaneously, which is so called concurrent kernels. But such grids (kernels) must come from the same context. In CUDA programming, the programmer can control the number of threads within a block and the number of blocks within a grid.

Channel: Each GPU context is associated with a GPU hardware channel. Internally, a channel is managed by channel engine which is a subarea of the MMIO (memory-mapped I/O) region. The channel engine maintains the status of GPU contexts including FIFO queues of GPU commands.

Command: Typically the GPU is controlled by the CPU using commands. The operating system and GPU driver maintain GPU *command buffers* which are accessible to both CPU and GPU. The CPU writes commands to them, while the GPU reads the commands from them. There are hundreds of commands defined by the architecture (e.g., Fermi or Kepler). For example, when copy data from the host to the device memory, we send a set of commands to the GPU, specifying the source and the destination virtual addresses together with the mode of direct memory access (DMA). A single GPU command is composed of GPU method and the values passed to the instructions. It represents atomicity operation. Commands are usually grouped as non-preemptive regions called *command group*. A tuple of size and command group address forms the packet written to command buffer.

We assume that tasks (computation or memory-copying operation) within the same GPGPU application have the same priority. The priorities of tasks may be propagated from CPU or dynamically assigned by GPES. GPGPU applications (with or without deadlines) may or may not execute periodically. GPES makes scheduling decisions at driver level before it offloads GPU tasks to the GPU device.

III. A Case Study

Due to the asynchronous and non-preemptive nature of GPU processing, in multi-tasking environments, tasks with high priorities may be blocked by low-priority tasks. Such priority inversions may occur due to either kernel execution blocking or data transfer blocking. In a real-time system, this may cause deadline misses.

We conduct a measurements-based case study to show the impact of the non-preemptive kernel execution and data transfer blocking on real applications in practice, using two best available GPGPU drivers in a multi-tasking environment: the NVIDIA proprietary driver [21], and the Nouveau open source driver [4] plus Gdev [13] which is a GPGPU run-time and resource management engine that manages GPUs as first-class computing resources. We measure the performance of running a video processing application *heartwall* [3] competing with *mmul* (matrix multiplication). *Heartwall* processes

TABLE I: Jitter and tardiness of video processing application when competing with matrix multiplication under the NVIDIA proprietary driver (NV) and the Nouveau open source driver plus the Gdev module (Gdev)

	small	medium	large
NV tardiness	1169.64 ms	4387.7 ms	6130.77 ms
NV jitter	1188.13 ms	4523.22 ms	6483.90 ms
Gdev tardiness	426.24 ms	2678.97 ms	6726.78 ms
Gdev jitter	1197.11 ms	5095.33 ms	12705.99 ms

a medical video frame by frame. A single frame processing consists of a memory-copying operation and a kernel launch. We assign the highest CPU priority (nice) to the *heartwall* application by viewing it as a high priority task, and assign low CPU priority to the *mmul* application as low priority task. The average processing time (memory-copy and kernel launch) of single iteration in *heartwall* is 380ms. It is set to execute periodically at an interval of 500ms. *Mmul* has variable processing time depending on its data size. It is configured to execute repeatedly with three sizes: small (256KB), medium (4MB) and large (16MB). Each combination executes for 500 seconds in total to impose high workloads on the entire system. We report the relative jitter and tardiness in the same manner as [14]. Jitter is the deviation from the true periodicity of a periodic frame playback, which quantifies the smoothness of a video. If a frame i starts displaying at time t_i and the actual period between two frames is p, then its relative jitter is $|t_i-(t_{i-1}+p)|$. Tardiness represents the delay of completion.

As listed in Table I, the jitter and tardiness on both drivers are significant, particularly when competing against *mmul* application with large data sizes. It is clear from this case study that current existing GPGPU drivers lack mechanism to make high-priority tasks preemptive when competing with long-freezing low-priority tasks. This lack of support motivates us to develop GPES, as describe next.

IV. System Design and Implementation

In this section we present the design and implementation of GPES, which aims to make tasks on GPU more preemptive and interruptible in multi-tasking environments. We implement GPES based on Gdev [13] which is open-source and publicly available. The software stack of GPES consists of a kernel transformer, a user-space library and an OS module. As depicted in Fig. 1, the shadowed rectangles represent the components of GPES. The kernel transformer performs automatic source-to-source transformation to kernel source code. The GPES library is a wrapper of driver APIs and provides CUDA API interfaces, where kernel execution slicing and data slicing are implemented. These two components are implemented at user space. The GPES module performs the actual functionality of memory-copy, kernel launch, scheduling, and interrupt handling. GPES module is implemented at OS space.

GPES is implemented on top of the existing GPGPU programming framework of CUDA: source code of application is categorized into CPU code and GPU code; CPU code is compiled into executable file by gcc, whereas GPU code is compiled into object file (cubin) by nvcc [19]. The executable file executes on CPU and loads cubin file onto GPU. In the following sections, we highlight some of the implementation details that deserve articulation.

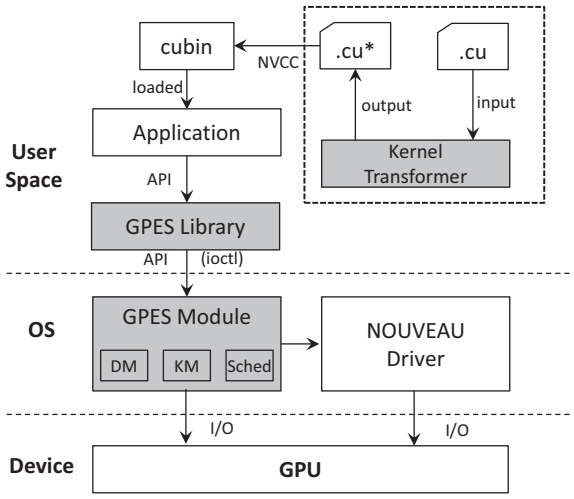

Fig. 1: GPES framework

A. Kernel Execution Slicing

A long-running kernel can prevent other kernels from accessing GPU computing resources. To avoid this, one of our techniques is to slice the execution of a large kernel into smaller subkernels, so that high priority kernels can preempt control of the GPU after the completion of a subkernel. An ideal approach is to insert "preemption points" and control the mapping from blocks to SMs, where we force the GPU to execute part of the blocks each time. But unfortunately, currently the CUDA programming model does not provide this level of controllability on SMs. The scheduler which dispatches logical blocks to hardware SMs is entirely implemented in the GPU hardware. NVIDIA has not disclosed the details of implementation to public. A kernel is submitted in the form of a grid. Once a grid is offloaded to the GPU device, the execution is non-interruptible.

To achieve kernel execution slicing, we develop an alternative approach: workloads partitioning. In the following subsections, we introduce "source-to-source transformation" to better explain our idea and further introduce a novel technique "just in time kernel code rewriting" to make the kernel execution slicing totally transparent to applications.

1) Source-to-Source Transformation: To better support parallel computing, GPU hardware maintains continuous indexes for all blocks in one grid, a.k.a, *blockIdx*. For example, if a grid consists of 256 blocks in one dimension, the blockIdx ranges from 0 to 255. In order to make a long-running kernel interruptible, we convert a large kernel into multiple subkernels, each of which is launched with a *blockRange*. BlockRange is defined as the number of blocks to be executed in this subkernel, which is bounded by a start blockIdx and an end blockIdx. Slicing blockRange forces each subkernel to complete part of the computing workloads. Thus, the execution time of each subkernel is much shorter than the original kernel. At the end of each subkernel launch, we setup an "interrupt point" to allow higher priority kernels from other GPGPU applications to preempt the control of GPU.

As mentioned in Sec. II, programmer can control the number and the shape of blocks in one grid. Such blocks are grouped up to three dimensions. For readability, here we

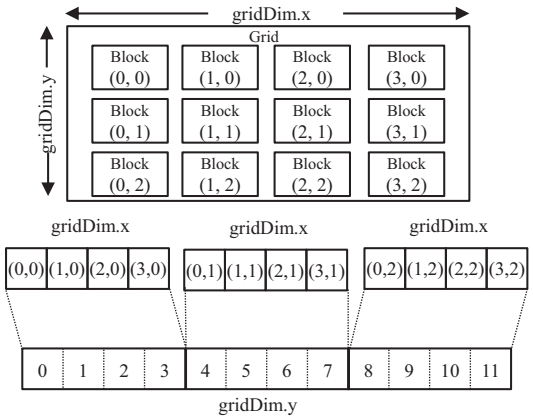

Fig. 2: A two dimensional grid is flattened to one dimension

```
1  __global__ void add(uint32_t *a, uint32_t *b,
2                      uint32_t *c, uint32_t n,
3                      int block_from, int block_to)
4  {
5    int flatId = gridDim.y * blockIdx.x + blockIdx.y;
6    if (flatId < block_from || flatId > block_to)
7      return;
8    int i = blockIdx.x * blockDim.x + threadIdx.x;
9    int j = blockIdx.y * blockDim.y + threadIdx.y;
10   if (i < n && j < n) {
11     int idx = i * n + j;
12     c[idx] = a[idx] + b[idx];
13   }
14 }
```

Fig. 3: Kernel code transformation

consider for example a two-dimensional grid whose blocks are organized as a 16×16 rectangle. The blockIdx can be represented as a pair of (blockIdx.x, blockIdx.y), such as (0, 0), (1, 0), (2, 0). In order to assign a *blockRange* to each subkernel, our kernel transformation technique flattens N-dimensional blocks to one dimensional blocks, which makes the slicing easier. Fig. 2 shows that a two dimensional grid is flattened to one dimension, where each block is associated with a flattened index. GPES executes an equal subset of blocks per subkernel launch, till all blocks are executed.

We implement kernel execution slicing technique through source-to-source transformation before the GPU code is compiled into cubin object file. This procedure is automatically performed by the kernel transformer. As shown in Fig. 3, the shadowed parts represent patched lines of code compared to the original kernel code in the *madd* (matrix addition) benchmark. Two parameters *block_from* and *block_to* are added to represent the range of flattened blocks to be executed. If a kernel launches more than one dimensional grid, lines of kernel-independent flattening code are inserted as shown on lines 5–7. After flattening the block indexes, each thread identifies its block index (flatId) and checks if it will continue to perform the computation. Thus, GPES actually performs computation of (*block_to*− *block_from*+1) blocks for each subkernel launch and skip the rest of computation. The computation of one kernel can thus be divided into several subkernels. Note that these patched lines of code are universal to all kernels and thus can be automatically applied by GPES.

To efficiently utilize the transformed subkernels, we re-implement several functions in the openCUDA [10] library. For example, we re-implement function *cuLaunchGrid* which is the CUDA API of launching a kernel, the pseudo-code is shown in Algorithm 1. The variable *SP* represents the number of subkernels that the original kernel will be sliced into, *range* denotes the number of blocks to be executed in each subkernel launch. Two additional arguments *block_from* and *block_to* are added to the subkernel's parameter buffer, which indicate the bounds of blocks. Function *cuLaunchGrid_v0* implements the actual subkernel launch. Till now, without touching the CPU source code, one kernel launch is converted to a number of subkernel launches specified by *SP*. Notice that, *SP* is a predefined value. Intuitively, a large SP value implies a small execution duration of each subkernel launch, which may also result in more overheads. The impact due to different *SP* values will be studied in Sec. V-B.

Algorithm 1 Customized cuLaunchGrid

```
1  cuLaunchGrid(f, grid_width, grid_height) {
2    range = grid_width * grid_height / SP;
3    rest = (grid_width * grid_height) % SP;
4    while(i < SP) {
5      block_from = i * range;
6      block_to = (i + 1) * range;
7      if ( i == SP - 1)
8        to += rest;
9      update_param_buf(f, block_from, block_to);
10     update_param_size(f);
11     cuLaunchGrid_v0(f, grid_width, grid_height);
12     i++;
13   }
14 }
```

2) Just In Time Kernel Code Rewriting: Source-to-source transformation must be completed at compile time. It has to access the GPU source file. Furthermore, once compilation is done, the granularity of partitioned block range can not be changed at run-time. To transparently complete the kernel transformation without accessing the application's source code, we introduce a novel technique – "just in time kernel code rewriting". Through this approach, no source code of incoming applications is needed, as all transformations are performed transparently at run-time. Thus, it serves as a better alternative to replace the source-to-source transformation technique.

Through analyzing the changes between the generated GPU binary before and after the source-to-source transformation, we figure out that the additional instructions introduced in *mmul*'s kernel assembly code (a.k.a, SASS – shader assembly) are at the beginning, as shown in Fig. 4 lines 4–7. In order to do kernel binary code rewriting, two questions need to be solved: (i) how to insert the additional instructions into the original kernel code, and (ii) what impacts my such insertions introduce? Answering these questions is not easy, because NVIDIA does not disclose much of the detail of its architectures and instructions. But fortunately, there are some third-party projects that have revealed useful information [8], [16], [13], such as the layout of GPU instructions, and the memory organization for a kernel launch. Currently two types of instructions are used in NVIDIA architecture: 4-byte and

```
1  MOV R1, c [0x1] [0x100];
2  S2R R0, SR_CTAid_X;
3  S2R R2, SR_CTAid_Y;
4  IMAD.U32.U32 R3, R0, c [0x0] [0x18], R2;
5  ISETP.GT.AND P0, pt, R3, c [0x0] [0x40], pt;
6  ISETP.LT.OR P0, pt, R3, c [0x0] [0x3c], P0;
7  @P0 EXIT;
8  S2R R4, SR_Tid_Y;
9  S2R R3, SR_Tid_X;
10 IMAD.U32.U32 R2, R2, c [0x0] [0xc], R4;
11 IMAD.U32.U32 R10, R0, c [0x0] [0x8], R3;
12 ISETP.LT.AND P0, pt, R2, c [0x0] [0x38], pt;
13 ISETP.LT.AND P0, pt, R10, c [0x0] [0x38], P0;
14 @!P0 EXIT;
15 ISETP.EQ.U32.AND P0, pt, RZ, c [0x0] [0x38], pt;
16 @P0 BRA 0x128;
   ... ...
```

Fig. 4: Changes in GPU SASS due to kernel transformation.

8-byte. In our experiment environment, we always use 8-bytes instructions. Each 8-byte GPU instruction is usually composed of na (specify the instruction name), mod (operation mode), pr (predicate bits), re0 (destination register), re1 (second register operand), imme (32-bit immediate value), and nb (specify instruction name). To achieve the same goal of slicing kernel execution as source-to-source transformation, we must carefully follow this instruction format and insert our *range-selection* instructions.

To transparently rewrite the kernel code and reform the kernel, we add two parameters to the kernel (*block_from* and *block_to*) as discussed in the previous subsection. We re-implement *cuModuleLoad* and *cuModuleGetFunction* which are CUDA APIs loading cubin object file to get the kernel information (e.g., kernel code, kernel size, parameter size). After loading the GPU binary into memory, GPES reallocates a memory space for binary codes with additional 32 bytes for four *range-selection* instructions, and performs binary rewriting. When these four instructions are inserted, other instructions will be shifted. If there are unconditional branches, an offset-fixing action thus needs to be performed. For example at Fig. 4 line 16, the original branch destination is 0x108. Four additional instructions with 8 bytes each instruction causes 32 bytes (0x20) shift. For some complex kernels, we have to insert more than four *range-selection* instructions, since the registers used in inserted instructions may be further used for computation. In such cases, we need to introduce two more instructions to temporarily store and restore the register values. Additionally, GPUs maintain special registers for block indexes and parameters which are not explicitly revealed in SASS code. Thus, only modifying SASS code is not enough. As kernel's parameters are stored in constant memory, we have to modify the size of kernel's constant memory. The size of constant memory should be enlarged by 8 bytes since we add two additional int type parameters for block range-selection.

B. Data Transfer Slicing

Though data transfer and computation use different engines, memory copying of one application can not perform simultaneously with kernel launching of another process, since they belong to different contexts and GPU can hold only one context at a time. Thus, in a multi-tasking environment, a large memory copy operation of a low-priority task can also stall high-priority tasks. To prevent this, GPES seeks to split a non-preemptive memory transfer into multiple smaller chunks to make it preemptive. At the boundary of the each chunk, a preemptive point is inserted.

Two CUDA APIs *cuMemcpyHtoD* and *cuMemcpyDtoH* are used to perform memory-copying between host memory and GPU device memory. To realize our idea, we re-implement these functions. Every single memory-copy operation is divided into multiple ones. Then each time GPES transfers only one chunk of data. At driver level, GPES is aware of all memory-copy requests from all user space applications. GPES maintains a queue of such memory-copy requests. Request with the highest priority will be put at the head of queue. Once the current memory-copy is done, the memory-copy request at the head of queue will be performed. GPES setups a fence at the end of each transfer, the fence will raise an interrupt to notify the completion of the current transfer, and wake up the schedule thread.

Intuitively, the chunk size impacts the granularity of preemption. The overhead introduced by fine-grained data transfer has been well studied by [5], [11]. Our preliminary experiments showed similar results. Thus, in our implementation, if data chunk size in host-to-device memory-copying is no large than 4MB, we use direct I/O write; else we use DMA engine to transfer data (according to [5], [11], and our experiments). The threshold of device-to-host memory-copying is set to 4KB. The impact due to sliced chunks will be studied in Sec. V-B.

C. Context Switch Scheduling

A GPU can hold only one context at any time. GPES uses interrupts to trigger context switches. GPES's context switch module is implemented at driver level, which uses two scheduler threads to perform computation scheduling and memory-copy scheduling separately. The scheduler threads are waken up by GPU interrupts generated upon the completion of any computation or memory-copy operation.

Different from TimeGraph [12] which is a GPU command scheduler integrated in GPU device driver to protect critical GPU applications from interference, GPES is API driven, which means interrupts are setup only when the interrupt function is called, and the scheduler is invoked only when computation or data transmission requests are submitted; while TimeGraph is command driven, interrupts are inserted in between command groups, causing the scheduler to be invoked whenever GPU commands are flushed. The scheduling overhead of GPES is thus much less.

Gdev is also API driven, and uses interrupts to invoke scheduler. Although GPES is implemented on top of Gdev, it uses very different interrupt schema to be more compatible with the kernel execution slicing and data slicing techniques. GPES setups interrupts for both kernel execution and memory-copying, which can make low-priority memory-copy operations yield to requests from other applications with higher priority; whereas Gdev does not use interrupts for memory-copy operations since Gdev synchronizes memory-copy with computation, causing the memory-copy operation non-preemptable. Furthermore, Gdev performs scheduling at the granularity of context-level. It creates a scheduling entity for each context. If the arriving scheduling entity has the

same context as the current entity, Gdev will not stall it. Gdev allows multiple continuous kernels belong to the same context to be launched simultaneously in order to utilize the feature of concurrent kernel execution. However, such schema prevents the newly arriving high-priority tasks to preempt at the boundary between such two continuous kernels. Different from Gdev, GPES creates scheduling entity for each kernel launch and memory-copy chunk. If there is a schedule entity occupying GPU, new arriving schedule entities will be stalled regardless of whether it has the same context. Another difference between Gdev and GPES is that the priorities of GPU contexts are propagated from the OS to Gdev; whereas GPES not only supports such mechanisms but can also adaptively assign priorities to specific schedule entities. The impact due to adaptive priority assignment is shown in Appendix.

When one kernel is sliced into multiple subkernels, "interrupt points" inserted between subkernels will raise multiple interrupts. All interrupts from the GPU that are caught in the IRQ handler are relayed to GPES. When GPES receives an interrupt, it references the fence identity to verify which kernel launch or memory-copy operation raised the interrupt. At each interrupt point, scheduling entity at the head of queue is popped out and set active, so that the application with the highest priority may preempt. The goal of the GPES scheduler is to correctly schedule computation and data transmissions for each GPU context based on priority.

D. Challenges and Limitations

Our GPES prototype implementation has several limitations. First, it does not yet support texture and 3-D processing. Thus when choosing benchmarks, we only choose CUDA-based image processing samples instead of OpenGL graphics. Another limitation of slicing kernel into multiple subkernels at block level is that, we can not handle global synchronizations (if any) due to the global barrier originally deployed in the kernel. Furthermore, the research area of binary rewriting itself is still an open area. Our GPU binary rewriting technique can only handle simple kernels with simple semantics and a few unconditional branches. More sophisticated binary analysis techniques such as alias analysis will be introduced to make the binary rewriting function more reliable. We leave such further improvements as future work.

V. Evaluation

In this section, we present the experimental results used to evaluate the effectiveness of GPES.

A. Experimental Setup

Our experiments are conducted with the Linux kernel 3.3.1 on NVIDIA GeForce GTX 480 graphics card and Intel i7 4770K processor. Benchmarks are chosen from Gdev test samples and Rodinia benchmark suits [3]. Table II lists benchmarks used in experiments. All benchmarks are written in CUDA driver API and compiled by NVCC 4.0 [21].

Because GPES focuses on the scheduling, it does not implement data swapping which is adopted in Gdev [13] as a core component to support excessive memory resource demands. We thus choose benchmark combinations that would not overload the GPU device memory. Furthermore, Gdev virtualizes a physical GPU into multiple logical GPUs, providing isolation and fairness among virtualized GPU devices. With different goals, GPES seeks to improve prioritization and preemptivity, and does not implement such virtualization. For fairness, we set the number of Gdev's virtual device to one and execute all benchmarks on this virtual GPU.

TABLE II: Benchmarks used in evaluation

NAME	Description	Structure
mmul	Matrix multiplication	Single kernel
madd	Matrix addition	Single kernel
heartwall	Medical imaging	1 kernel per loop
backprop	Back propagation	2 dependent kernels
bfs	Breadth-first search	Single kernel
hotspot	Physics simulation	1 kernel per loop
lud	LU Decomposition	3 dependent kernels per loop
nn	K-nearest neighbors	Single kernel
srad2	Image processing	2 kernels per loop

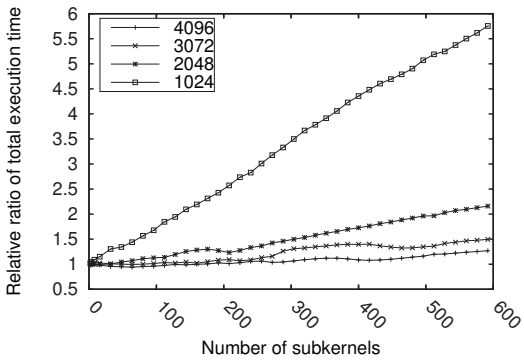

Fig. 5: Relationship between execution time and number of subkernels

B. Overhead due to kernel slicing

As mentioned in Sec. IV-A, one kernel launch can be sliced into multiple launches. Each launch offloads a subkernel with related parameters to GPU computing engine. Each launch of subkernel will introduce overheads caused by initialization and hardware dispatching. Also, each thread will execute at least four additional instructions to identify its *blockIdx*. All these extra work will introduce overheads.

In order to evaluate the performance of our implementation of kernel slicing, we slice the kernel of *mmul* benchmark into multiple subkernels, and record its execution time. As shown in Fig. 5, the x-axis is the number of subkernels that we slice the original kernel into; the y-axis is the relative ratio of the total execution time with slicing divided by the execution time of the original kernel without slicing (abbreviated as *relative ratio*); four curves in this figure represent *mmul* with 4 different input sizes (e.g., 1024×1024). We observe from Fig. 5 that kernel slicing introduces reasonable amount overheads. For example, when the number of subkernels reaches 400, the *relative ratio* of *mmul* instance with 4096×4096 input size is only 1.04. We also observe that instances with larger inputs tolerate more fine-grained slicing. Essentially, there is a tradeoff between the granularity of kernel slicing and overhead, which often depends on different applications and devices. To accurately identify an appropriate value of subkernel granularity, an offline profiling is need. However, GPES makes kernel slicing decisions online to handle dynamically coming applications. In our experiments, we pick the number of blocks executed

in each subkernel launch to be no less than 120, since with such subkernel granularity, the slowdown of all benchmarks introduced by kernel slicing can be limited to less than 5%. This is reasonable because our GTX 480 GPU can hold at most 120 blocks simultaneously. If the number of blocks to be executed on GPU is less than 120, the computation resources may not be fully utilized. In summary, the kernel slicing technique adopted by GPES is efficient and applicable, with acceptable overhead.

C. Overhead due to data slicing

We also measured overheads caused by our implementation of data slicing using the *memcpy* benchmark. The *memcpy* benchmark performs memory-copying from host to device and then transfer the data back using DMA without doing any kernel computation. Fig. 6 (a) illustrates the impact of data slicing on host-to-device (HtoD) memory-copying time. The x-axis is the number of chunks, y-axis is the total HtoD time from the first chunk to last chunk. The curves represent *memcpy* instances with different input data. We observe that the total HtoD time increases with the increased chunk number. However, such increases are reasonably small when the number of chunks is no greater than 128. When the number of chunks is greater than 128, the HtoD time substantially increases. This is because the overhead introduced by using the DMA engine is non-trivial. Fig. 6 (b) reveals the same trend as in set (a) on the device to host memory-copying time. We observe that the total DtoH time of instances with smaller inputs (1M, 8M, 64M) increases with the increased number of chunks. But when the 1M instance is sliced into 256 chunks (4KB per chunk), it stops increasing. This is because it hits the threshold of 4KB where direct I/O performs better than DMA engine when transferring data with size less than 4KB from device to host. However such fine-grained slicing is not encouraged since it introduces non-trivial overhead compared to memory-copy without slicing. Meanwhile, the instances with 512MB input performs almost consistently regardless of the growth of the number of chunks, since large chunks can benefit from DMA transaction. With 1024 chunks and 512MB input data, the total DtoH time with slicing (310ms) is only 6.8% more than the time without slicing (290ms). In the experiments, we pick the HtoD chunk size no less than 4MB and the DtoH chunk size no less than 512KB, since such granularity provides good tradeoff between fine grained preemption and overhead.

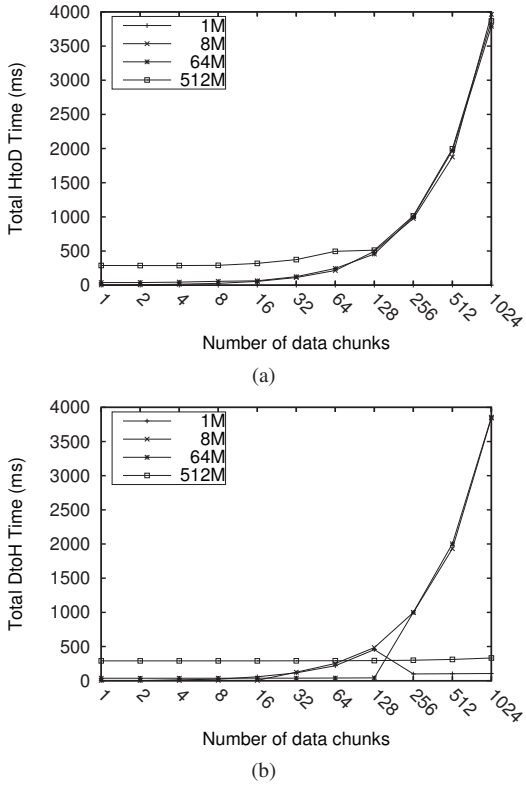

Fig. 6: Relationship between memory-copy time and number of chunks. (a) Host to device (b) Device to host

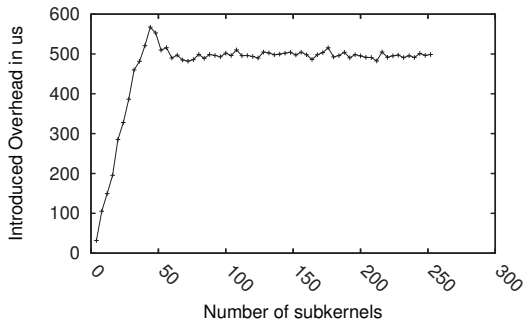

Fig. 7: Additional context switch overhead via kernel execution slicing

D. Overhead of context switching

There are two stages that would introduce context switching overheads in computation scheduling. One happens at each new kernel launching time, when GPES locks the computation scheduling thread, then performs context switch scheduling, and finally unlock the computation thread. During this stage, GPES checks current status of GPU. If the previously offloaded kernel is not returned, GPES will stall the launch request; otherwise it sets the incoming context active. The other stage happens when an interrupt indicating kernel completion is caught. Between the locking and unlocking operations to the computation scheduling thread, GPES draws out the context with the highest priority at the request queue and sets it to active. Similarly, there are also two stages which introduce context switching overheads in scheduling memory-copying operations.

We evaluate the context switching overhead in GPES by recording the time slots when scheduling threads are suspended. We use the *mmul* benchmark with 16MB input to evaluate the context switching overhead caused by kernel slicing, as illustrated in Fig. 7. The x-axis is the number of subkernels, the y-axis is the overhead introduced by context switch scheduling in μs. We observe that the overhead increases with the increased number of subkernels till the number reaches 50 subkernels, after which the overhead stops increasing. The reason is that with a large number of subkernels, each subkernel executes very fast such that no incoming requests will be stalled. Thus, no additional overhead will be introduced. The result of context switching overhead caused by data slicing is similar to Fig. 7. Thus, we conclude that our context switching overhead is trivial (less than 600 μs) compared to the execution time of kernel launch and memory-copying.

E. Multi-Tasking Performance

In this subsection, we discuss the performance of high-priority applications when competing with low-priority applications under GPES. Following are some terms that will be used in the description.

Response time: The time elapsed between the context creation and context destruction. We recorded this time by measuring the time slots between *cuCtxCreate* and *cuCtxDestroy* are called by an application.

Compute-Occupying time: The *compute-occupying time* of a specific kernel is the number of time slots when it is offloaded to GPU till its interrupt indicating completion is captured by GPES. The *compute-occupying time* of a CUDA application is the sum of all its kernels.

Copy-Occupying time: The time period that an application occupies the GPU copy engine.

Occupying time: Sum of *compute-occupying time* and *copy-occupying time*.

Pending time: The difference between *response time* and *occupying time*.

Impact of kernel execution slicing. In the first set of two experiments shown in Fig. 8, we evaluate the performance of high-priority applications with different kernel structures when competing with low-priority computation-intensive applications under GPES and Gdev. Benchmark *mmul* is chosen as the low-priority application (LP) in all these three experiments, which is considered as computation-intensive application. GPES only performs kernel execution slicing since the memory-copying time is relatively small compared to kernel execution time. We execute two instances of *mmul* with the same input size and priority repeatedly to interfere with high-priority applications.

In the first experiment, we choose *madd, bfs, nn* with small input sizes as high-priority applications (HP). These benchmarks are all single-kernel applications. They execute periodically with an interval of 5,000ms. This configuration can avoid the interference among high-priority tasks whereas each high-priority task can compete with at least one low-priority task. We report the average pending time of the three high-priority applications in Fig. 8 (a). The x-axis is the input data size of LP; the y-axis is the relative ratio of HPs' average pending time divided by the standalone execution time of LP without kernel execution slicing, denoted as normalized average pending time; the four curves represent Gdev, GPES with each kernel sliced into two subkernels (GPES+2SP, for short), GPES with eight subkernels (GPES+8SP), GPES with 32 subkernels (GPES+32SP). We observe that with our kernel execution slicing technique, the average pending time is reduced dramatically compared to Gdev under all scenarios. For example, when the input data of LP reaches 100MB, the normalized average pending time of Gdev is 0.98, while the normalized average pending time of GPES with 32 subkernels is less than 0.05, which is more than 90% reduction.

In the second experiment, we choose *backpro, heartwall, lud* as HPs. All these benchmarks have dependent kernels, which means that the second kernel launch must be performed after the first launch. There is a memory-copying operation

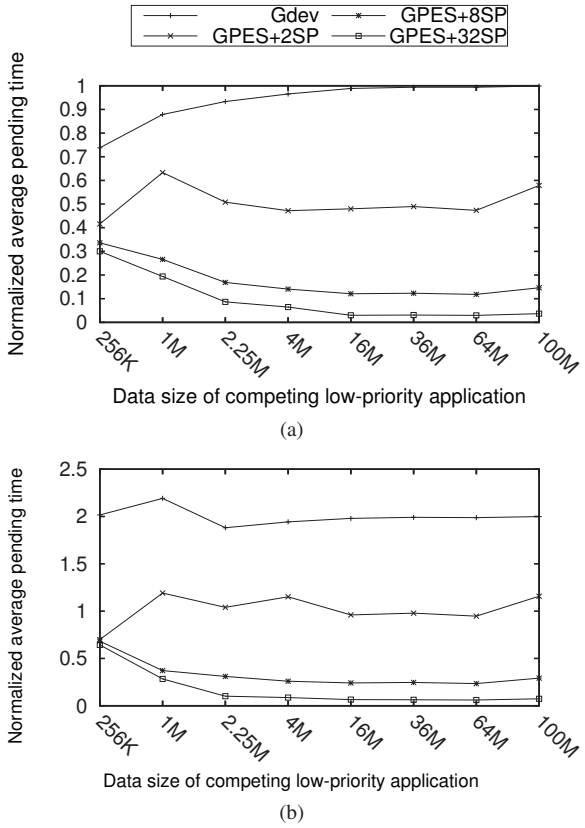

Fig. 8: Impact of kernel execution slicing. (a) Single kernel (b) Dependent kernels

between two kernel launches. As seen in Fig. 8 (b), we observe that the average pending time is almost doubled compared to the previous case (Fig. 8 (a)). For example, when being interfered by LPs with 100MB input, the normalized average pending time of Gdev and GPES with two subkernels are 2.0 and 1.2 respectively. The reason is that the two dependent kernels of HP are interleaved by a subkernel of LP, causing extra pending time. Nonetheless, GPES significantly outperforms Gdev in reducing the average pending time of HPs.

Impact of data slicing. In the second set of three experiments, we evaluate the impact of data slicing on applications when competing with low-priority data-intensive applications under GPES and Gdev. Benchmark *memcpy* is used as the competing LP, which does not contain any kernel launch. Three sets of benchmarks are chosen as HPs: computation-intensive set, data-intensive set, and mixed set. LP is configured to execute repeatedly, while each instance of HP executes for every 5,000ms to avoid the interference among high-priority tasks and make sure each high-priority task can compete with the low-priority task. The results of these three experiments are depicted in Fig. 9, where x-axis is the input data size of LP; the y-axis is the relative ratio of HPs' average pending time divided by the standalone execution time of LP without data execution slicing, denoted as normalized average pending time; the four curves represent Gdev, GPES with each memory-copying data sliced into four chunks (GPES+4CK), GPES with 32 chunks (GPES+32CK), GPES with 256 chunks (GPES+256CK).

In the first experiment, we choose *mmul, lud, heartwall* as HPs. These benchmarks are all computation-intensive applica-

tions. Heartwall is configured with only one iteration; mmul is configured with 2048×2048 matrix; lud is configured with 1024×1024 matrix. We observe from Fig. 9 (a) that the data slicing technique is efficient to reduce the average pending time in all scenarios compared to Gdev. For example, when the data size of LP reaches 512MB, the average pending time with Gdev is 0.63, whereas GPES with 256 chunks is 0.25.

In the second experiment, we use *nn, backpro, bfs* configured with large data size input as HPs. These benchmarks are data-intensive applications. Specifically, we modify nn to read all data from one data file instead of originally from thousands of data files, in order to reduce the file I/O latency. We observe from Fig. 9 (b) that the average pending time is greater compare to (a), which is caused by the fact that large data transfers often cause longer-time GPU initialization (e.g., memory allocating) which is non-preemptive. Nonetheless, the data slicing technique of GPES is still very efficient in reducing the average pending time in all scenarios compared to Gdev.

In the third experiment, we mix all the six benchmarks used in previous two experiments together with the same execution configuration. We observe from Fig. 9 (c) that GPES is still superior to Gdev.

Video case study. Recall the case study used in Sec. III, the jitter and tardiness increase fast under NVIDIA proprietary driver and Gdev. For the same video processing application, we conduct an evaluation using GPES to slice host-to-device memory-copying into 4MB chunks and device-to-host memory-copying into 512KB chunks, and slice kernel into 120 blocks per subkernel launch. The result is shown in Fig. 10. We observe that the jitter and tardiness can be significantly reduced compared to Gdev. For example, when the data size of competing application reaches 36MB, the tardiness and jitter under Gdev are 11,253ms and 21,398ms respectively; whereas GPES is able to reduce these values to 1,388ms and 2,580ms. The jitter and tardiness under GPES are much lower than Gdev in all cases, and the reduction can reach up to 80%. GPES can thus prevent video applications from being interfered by low-priority competing applications containing long-freezing memory-copying or kernel execution.

We have also conducted experiments evaluating GPES against Gdev for non-real-time and security-concerned settings. Due to space constraints, we put the results in the Appendix.

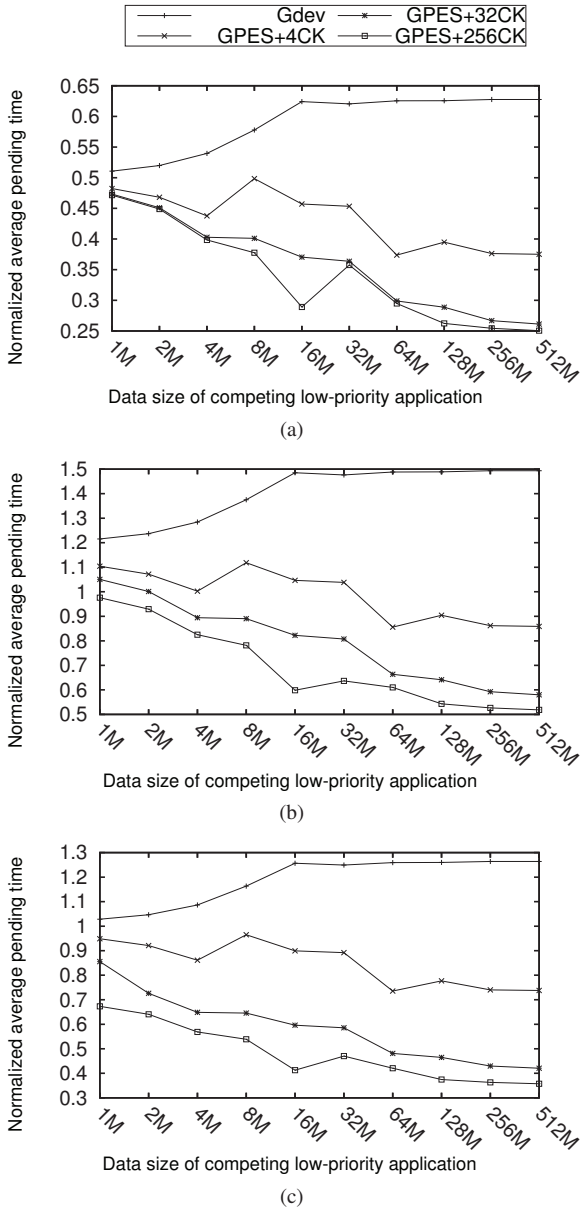

Fig. 9: Impact of data slicing (a) Computation-intensive set (b) Data-intensive set (c) Mixed set

VI. Related Work

Kernel transformation. Lee *et al.* propose SKMD [17] which transparently translates a single OpenCL [6] kernel into variations and executes them on multiple GPUs simultaneously. Elastic Kernel [24] rewrites the kernel source code and reshapes the Kernel Grid to use $N:1$ logical-to-physical mapping scheme. We implement kernel source transformation to slice a kernel into multiple subkernels and share the same idea of flattening workgroups as SKMD, but the goal of our technique is fundamentally different from those techniques: SKMD transforms kernels to distribute the workloads of a single kernel on multiple devices; Elastic Kernel uses kernel transformation to enable concurrent execution of different kernels; whereas we slice kernels to make the long-running kernel interruptible for better preemption. Furthermore, the source-to-source transformation technique is just a small (optional) part of our system, because we implement a novel and better kernel code rewriting technique as an alternative.

GPU scheduling A number of runtime systems have been developed to perform GPU task scheduling. Qilin [18] provides an adaptive mapping to automatically partition tasks on a CPU and a GPU. StarPU [1] runtime system provides programmers with a portable interface for dynamically mapping tasks onto heterogeneous processors (CPUs and GPUs). The aforementioned runtime systems are implemented at user-level and focus on heterogeneous systems without considering interference among multiple applications on the same GPU.

PTask [25] focuses on eliminating performance interference of GPU sharing. TimeGraph [12] and Gdev [13] provide prioritization and isolation capabilities in GPU resource management. GDM [26] enhances GPU memory management by introducing a staging area in host memory for each process.

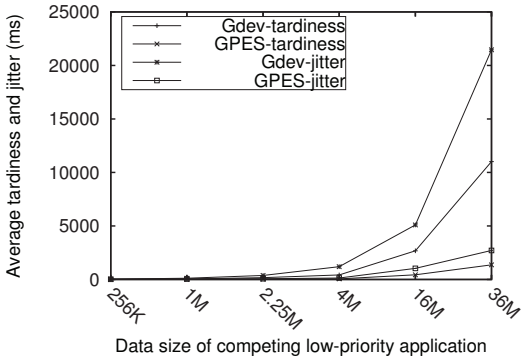

Fig. 10: Jitter and tardiness of image processing case under Gdev and GPES.

These work are implemented at OS-level, and also propose scheduling algorithms for different applications sharing GPU resources. Specifically, Gdev and Timegraph make enhancements on kernel scheduling between contenting applications. However, they can not handle the priority inversions caused by long-running or non-terminating kernels. Furthermore, GPES utilizes different interrupt schema compared to these work. More detailed differences between GPES and Gdev and TimeGraph have been described in Sec. IV-C.

RGEM [11] and PKM [2] are two GPGPU engines which provide responsive and preemptive support for GPGPU tasks in a multi-tasking environment. However, they are implemented at user-level, thus lacking the view of the whole operating system. Moreover, in order to utilize their engine, GPGPU applications are required to be rewritten with the interfaces they provide. This may put much burden on end-programmers. Also, they have to know all applications before hand and then compile them into one single process, which does not reflect a real multi-tasking environment with dynamically coming applications. In contrast, GPES does not need the source code of GPGPU applications. Also, it can transparently provide preemption and prioritization support for dynamically coming applications. To the best of our knowledge, GPES is the first piece of work which supports preemptive computation and memory-copying in a practical multi-tasking environment.

VII. Conclusion

In this paper, we present GPES, a GPGPU preemptive execution system to make long-running low-priority GPGPU applications interruptible and preemptable in a multi-tasking environment. We implement a prototype system based on open-source GPGPU drivers. Our system introduces several techniques that slice a long-running kernel into several smaller subkernels and slice data transmissions into chunks with acceptable overheads. In order to achieve better preemptivity, GPES also implements new interrupt handling and context switching schemes. Experimental results demonstrate that GPES can achieve much better performance compared to the state-of-art open-source driver, and performs consistently well across different applications. The incurred overheads due to the proposed techniques are reasonably small, which makes GPES a practical and efficient solution for real-time GPGPU computing.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *Proc. of Euro-Par*, pages 863–874, 2009.

[2] C. Basaran and K.-D. Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *Proc. of IEEE ECRTS*, pages 287–296, 2012.

[3] 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 *Proc. of IEEE International Conference on Workload Characterization*, pages 44–54, 2009.

[4] FREEDESKTOP. Nouveau Open-Source Driver. http://nouveau.freedesktop.org.

[5] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro. Data transfer matters for gpu computing. In *Proc. of IEEE International Conference on Parallel and Distributed Systems*, pages 275–282, 2013.

[6] K. O. W. Group. OpenCL-The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl, 2008.

[7] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *Proc. of ACM SIGCOMM*, pages 195–206. ACM, 2011.

[8] Y. Hou, J. Lai, and D. Mikushin. AsFermi: An assembler for the NVIDIA fermi instruction set. http://code.google.com/p/asfermi, 2011.

[9] Y. Iida, M. Hirabayashi, T. Azumi, N. Nishio, and S. Kato. Connected smartphones and high-performance servers for remote object detection. In *Proc. of IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*, 2011.

[10] S. Kato. Implementing open-source cuda runtime. Technical report, 2013.

[11] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *Proc. of IEEE RTSS*, pages 57–66, 2011.

[12] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. of USENIX Annual Technical Conference*, pages 17–30, 2011.

[13] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt. Gdev: First-Class GPU Resource Management in the Operating System. In *Proc. of USENIX Annual Technical Conference*, pages 401–412, 2012.

[14] C. J. Kenna, J. L. Herman, B. B. Brandenburg, A. F. Mills, and J. H. Anderson. Soft real-time on multiprocessors: Are analysis-based schedulers really worth it? In *Proc. of IEEE RTSS*, pages 93–103, 2011.

[15] J. Kim, B. Andersson, D. d. Niz, and R. R. Rajkumar. Segment-Fixed Priority Scheduling for Self-Suspending Real-Time Tasks. In *Proc. of IEEE RTSS*, pages 246–257, 2013.

[16] M. Koscielnicki. envytools. git://0x04.net/envytools.git, 2012.

[17] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Proc. of IEEE PACT*, pages 245–256, 2013.

[18] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. of ACM MICRO*, pages 45–55, 2009.

[19] NVIDIA. Compute unified device architecture programming guide.

[20] NVIDIA. NVIDIA GF100 Whitepaper. http://www.nvidia.com/object/IO_86775.html, 2010.

[21] NVIDIA. CUDA 4.0. http://developer.nvidia.com/cuda-toolkit-40, 2011.

[22] NVIDIA. NVIDIA Kepler Architecture. http://www.nvidia.com/object/nvidia-kepler.html, 2014.

[23] NVIDIA. NVIDIA Maxwell Architecture. https://developer.nvidia.com/maxwell-compute-architecture, 2014.

[24] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *Proc. of ACM SIGPLAN*, pages 407–418. ACM, 2013.

[25] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proc. of ACM SOSP*, pages 233–248, 2011.

[26] K. Wang, X. Ding, R. Lee, S. Kato, and X. Zhang. GDM: Device memory management for GPGPU computing. In *Proc. of ACM SIGMETRICS*, pages 533–545, 2014.

APPENDIX

Non-real-time setting. For applications that do not have predefined priorities, GPES can still reduce the overall pending time. We conduct an evaluation using three benchmarks: mmul (1024×1024), srad (35 iterations), nn (file size 1024). These three benchmarks execute in different orders which reflects different priorities under Gdev and GPES. The overall pending time are depicted in Fig. 11. The x-axis indicates the combination of three benchmarks. For example, mmul.srad.nn indicates mmul is firstly loaded, srad is the second and so on. We observe that the overall pending time in most scenarios under GPES is much less than Gdev. For example, in the combination of mmul.srad.nn, the overall pending time under GPES is 53% less than Gdev. However, as shown in the last bar of Fig. 11, the overall pending time under GPES is larger than Gdev. This is because it represents the best ordering under both GPES and Gdev. But in this case, GPES introduces extra overhead due to slicing.

Defending against DOS Attacks. In many systems, a malicious GPGPU application can attack the system by submitting large numbers of kernels or an extremely large kernel which consists of many threads, causing denial-of-service (DOS) to normal GPGPU applications. GPES can mitigate such attack. By enforcing slicing large kernel with a lot of threads into smaller ones and each time executing a range of threads, GPU control can be regained by normal applications when an interrupt is issued.

In order to demonstrate how GPES mitigates such DOS attack, we have hand-coded two malicious GPGPU applications: LARGE and INFI. LARGE is a malicious application with a very simple kernel but consisting of a large number of threads. INFI is a malicious application issuing kernels repeatedly. We co-run each of our benchmarks (heartwall, madd, mmul, hotspot, nn) with LARGE and INFI, compare the makespan under GPES and Gdev. We execute the malicious application first, and then start running our benchmarks. Makespan is the elaspe from the benchmarks's start to its completion. The results are shown in Fig. 12, where the y-axis denotes the ratio of the makespan of running the benchmark alone without any malicious applications divided by the makespan with malicious applications. With GPES, the LARGE is sliced into 1024 subkernels, and all benchmarks successfully complete execution in acceptable time, averagely slowed down by 49% compared to standalone executions. With Gdev, all benchmarks are slowed down by 90% when co-running with LARGE or 100% (non-terminated) when co-running with INFI.

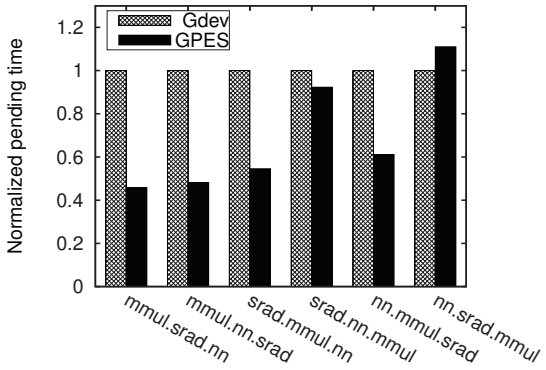

Fig. 11: Normalized pending time under Gdev and GPES.

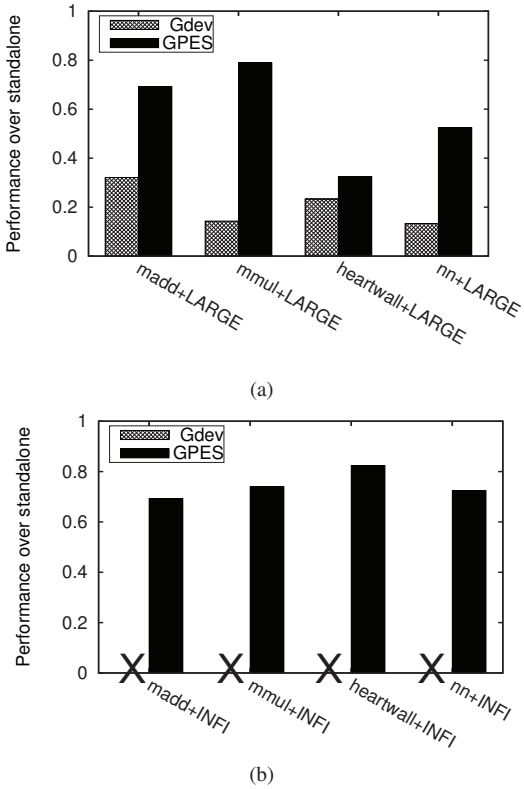

Fig. 12: Defending against malicious applications (a) LARGE (b) INFI. (A 'X' mark means the normal application does not terminate and the performance can not be measured.)