EDGE: Event-Driven GPU Execution

Tayler Hicklin Hetherington, Maria Lubeznov, Deval Shah, Tor M. Aamodt
Electrical & Computer Engineering
The University of British Columbia
Vancouver, Canada
{taylerh, mlubeznov, devalshah, aamodt}@ece.ubc.ca

***Abstract*—GPUs are known to benefit structured applications with ample parallelism, such as deep learning in a datacenter. Recently, GPUs have shown promise for irregular streaming network tasks. However, the GPU's co-processor dependence on a CPU for task management, inefficiencies with fine-grained tasks, and limited multiprogramming capabilities introduce challenges with efficiently supporting latency-sensitive streaming tasks.**

This paper proposes an event-driven GPU execution model, EDGE, that enables non-CPU devices to directly launch pre-configured tasks on a GPU without CPU interaction. Along with freeing up the CPU to work on other tasks, we estimate that EDGE can reduce the kernel launch latency by 4.4× compared to the baseline CPU-launched approach. This paper also proposes a warp-level preemption mechanism to further reduce the end-to-end latency of fine-grained tasks in a shared GPU environment. We evaluate multiple optimizations that reduce the average warp preemption latency by 35.9× over waiting for a preempted warp to naturally flush the pipeline. When compared to waiting for the first available resources, we find that warp-level preemption reduces the average and tail warp scheduling latencies by 2.6× and 2.9×, respectively, and improves the average normalized turnaround time by 1.4×.

***Index Terms*—GPU, Multiprogramming, Networking**

I. Introduction

To address slowdowns in Moore's Law [41] and Dennard scaling [12], the use of specialized processors in the datacenter is growing. For example, Google [11], Amazon [4], Facebook [35], Microsoft [39], and Oracle [48] use graphics processing units (GPUs) in their datacenters to accelerate computationally expensive and highly parallel applications, such as deep learning. Many recent works have shown potential for the GPU to branch out from the more traditional scientific or deep learning applications to a broader class of streaming applications relevant to the datacenter. Examples include server and packet processing applications [10], [19], [21], [22], [30], [31], [59], network function virtualization (NFV) [23], [70], databases [6], [66], and webservers [2].

However, limitations with GPU architecture and system-level integration introduce challenges with efficiently supporting latency-sensitive streaming tasks. First, GPUs are typically coupled to a CPU in a co-processor configuration, which may unnecessarily involve the CPU for GPU task management and I/O. For instance, many of the GPU networking works mentioned above employ a persistent CPU and/or GPU software runtime to orchestrate the communication of data and control between a network interface (NIC) and GPU, which can increase latency and complexity for tasks only requiring the GPU. Second, GPUs optimize for throughput over latency, preferring larger tasks to efficiently utilize the GPU hardware resources and amortize task launch overheads. Consequently, GPU networking applications often construct large packet batches to improve throughput at the cost of queuing and processing latencies. Lastly, GPUs have limited multiprogramming support, which reduces the ability to efficiently share the GPU between longer running tasks and latency-sensitive streaming tasks. For example, recent works use GPUs to accelerate NFV operations (e.g., IP forwarding/encryption); however, task launch overheads and challenges with the co-execution of concurrent GPU tasks can degrade performance and impact service-level objectives (SLO) [23], [70]. Others have shown that GPUs are viable candidates for deep learning inference workloads in datacenters, but the smaller batch sizes necessary to meet SLO requirements limit GPU resource-efficiency [24]. In such cases, GPU resource sharing can improve resource-efficiency at the cost of unpredictable latencies and poor performance isolation.

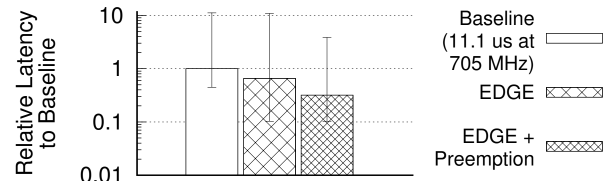


Fig. 1: Combined kernel launch and warp scheduling latency.

This paper proposes improved GPU support for fine-grained, latency-sensitive streaming tasks in a heterogeneous environment. Specifically, it proposes an event-driven extension to existing GPU programming models called EDGE. EDGE enables external devices to efficiently launch tasks of any size on the GPU without a CPU and/or GPU software framework. EDGE is a form of GPU active messages [13] that adopts a similar programming model to the Heterogeneous System Architecture (HSA) [17]. In EDGE, tasks are communicated directly to the GPU through in-memory buffers instead of through a GPU driver running on the CPU. The CPU is only responsible for configuring a communication plan between an external device and the GPU. The CPU pre-registers *event kernels* on the GPU to be triggered in response to external events and pre-configures the parameter memory that each subsequent event kernel will access. Once configured, the CPU is not required on the critical path for the management of data or control. External devices directly trigger GPU event

kernels by writing to special doorbell registers. The basic device requirements for supporting EDGE are discussed in Section III-D. We propose GPU hardware extensions to internally manage and launch event kernels, which can reduce the kernel launch latency over the baseline CPU-launched kernels by an estimated 4.4×. Reducing the kernel launch overheads can enable the use of smaller GPU kernels in certain streaming applications to improve end-to-end latencies.

EDGE reduces the overall time required to begin executing kernel instructions on the GPU by reducing both kernel launch overheads and scheduling latency. Once launched on the GPU, event kernels of any size are treated as regular host or device launched kernels. However, EDGE also provides an optimized path for fine-grained kernels – a single warp – to share GPU resources with concurrently running GPU tasks. We propose a lightweight warp-level preemption mechanism, which borrows a subset of resources from a running kernel. We evaluate multiple optimizations to reduce the warp preemption latency by 35.9× compared to waiting for a preempted warp to naturally flush the pipeline. On a set of networking event kernels and traditional GPU background kernels, we find that warp-level preemption reduces the average/tail event warp scheduling latencies and improves average normalized turnaround time (ANTT) over waiting for free resources by 2.6×, 2.9×, and 1.4×, respectively. Figure 1 shows that EDGE with warp-level preemption reduces the overall launch time by 3.14× versus a baseline GPU without EDGE and 2.06× versus an EDGE-enabled GPU lacking warp-level preemption.

This paper makes the following contributions:

- It proposes an event-driven extension to the GPU programming model and set of hardware changes that reduce the GPU's dependence on the CPU for task management. These reduce kernel launch overheads and enable non-CPU devices within a heterogeneous system to directly initiate computation on the GPU.
- It proposes a warp-level GPU preemption mechanism that enables the fine-grained co-execution of GPU event kernels with concurrently running applications. When the GPU is fully occupied, warp-level preemption reduces latency of scheduling an event warp and improves throughput versus waiting for resources to become available.
- It enables the effective use of smaller GPU kernels for streaming applications. Smaller kernels can be overlapped to reduce the end-to-end task latency while maintaining high throughput.

This paper is organized as follows: Section II elaborates on the background and motivation for EDGE, Section III presents the design of EDGE, Section IV describes the warp-level preemption mechanism, Section V provides our experimental methodology, Section VI evaluates EDGE, Section VII discusses related work, and Section VIII concludes.

II. Background and Motivation

This section briefly summarizes relevant GPU background material, discusses existing GPU kernel launching and resource sharing techniques, and motivates EDGE.

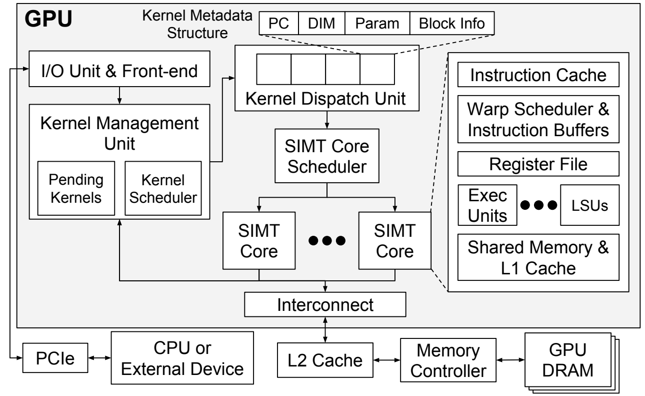


Fig. 2: Baseline GPU Architecture.

A. GPU Architecture and Programming Model

Figure 2 presents a high-level view of the GPU architecture assumed in this study [25], [62], resembling an NVIDIA or AMD discrete GPU. GPUs are highly parallel, throughput-oriented accelerators containing multiple single-instruction, multiple-thread (SIMT) cores, kernel and thread schedulers, caches, and high-bandwidth off-chip global memory. The SIMT cores contain an instruction cache, warp schedulers, instruction buffers, a large register file, multiple execution and load-store units (LSUs), a local on-chip memory, which can be configured as an L1 data cache or shared scratchpad memory, and (not shown) address translation units.

GPU applications typically consist of a CPU side, responsible for communicating data and tasks with the GPU, and a GPU side, which executes the parallel functions (kernel). GPU SIMT cores execute instructions in lock-step in groups of scalar threads (e.g., 32 or 64 threads), referred to as warps. Multiple warps are grouped together into thread blocks (TBs), which are the schedulable unit of work on the GPU. Groups of TBs form the work for a kernel. GPUs are programmed in parallel APIs, such as CUDA or OpenCL. The CPU communicates with the GPU through a driver running on the CPU, typically connected to the host system via PCIe for discrete GPUs. Asynchronous operations through multiple streams (hardware queues) enable data transfers and (parallel) kernel executions to be overlapped. NVIDIA HyperQ provides 32 such queues.

In the baseline [25], [62], the CPU launches a kernel by passing the kernel parameters, metadata (kernel dimensions, shared memory size, and stream), and function pointer to a GPU driver on the CPU. The GPU driver then configures the kernel's parameter memory, constructs a kernel metadata structure (KMD), and launches the kernel into hardware queues in the I/O & front-end unit. Internally, a kernel management unit (KMU) stores the pending kernels and schedules kernels to the kernel dispatch unit (KDU) based on priority when a free entry is available. Finally, a SIMT-core scheduler configures and distributes TBs from the KDU to the SIMT-cores based on available resources for the TB contexts. Current NVIDIA GPUs support up to 32 concurrently running kernels in the KDU and 1024 pending kernels in the KMU. CUDA dynamic

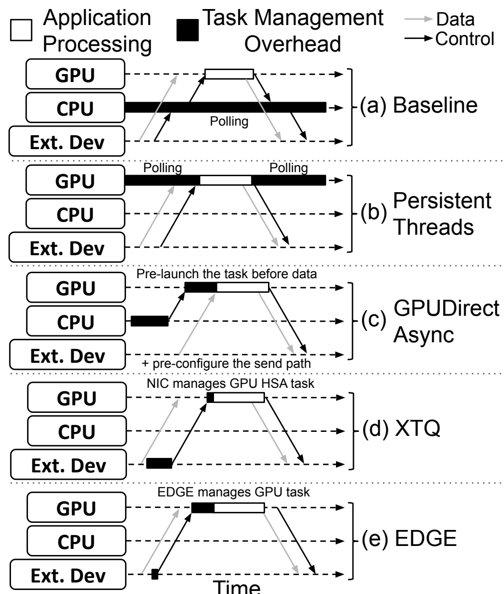
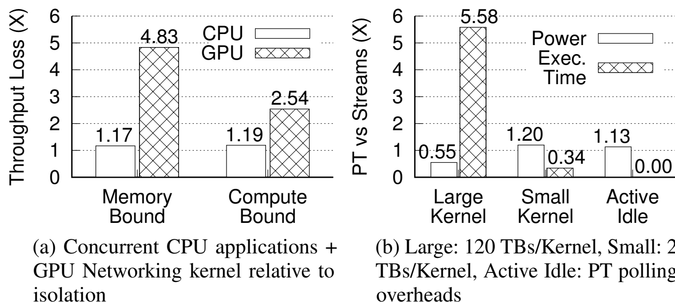


Fig. 3: External GPU kernel launching approaches.



(a) Concurrent CPU applications + GPU Networking kernel relative to isolation

(b) Large: 120 TBs/Kernel, Small: 2 TBs/Kernel, Active Idle: PT polling overheads

Fig. 4: Evaluation of Baseline (Fig. 3a) and Persistent Threads (Fig. 3b) on an NVIDIA GTX 1080 Ti / Intel i7-2600K.

parallelism (CDP) [44] enables GPU threads to internally launch sub-kernels on the GPU to exploit irregular parallelism in applications.

GPU data may be explicitly transferred via memory copies, accessed in shared/unified memory, or accessed directly through remote direct memory access (RDMA), such as GPUDirect [45].

B. GPU Kernel Launch Overheads

The baseline described above incurs overheads for streaming applications as it requires the CPU to configure, launch, and handle the completion of every GPU kernel. As illustrated in Figure 3a, in a streaming environment where the tasks originate from an external device, such as a NIC, the CPU may poll both the external device and GPU to improve performance. If the CPU is not responsible for any of the task processing, including the CPU on the critical path can increase the kernel launch latency and limit the potential for workload consolidation. The latter is important for increasing utilization and reducing costs in datacenters [7].

We measure this impact in Figure 4a, which evaluates the performance for a set of memory and compute bound CPU tasks (Spec2006) concurrently running with a GPU UDP ping benchmark (swaps packet header source/destination). In this experiment, data is transferred directly to the GPU from an Ethernet NIC via GPUDirect. Polling CPU threads were required for efficient packet I/O and kernel launch/completion management. We find that, relative to running in isolation, the CPU memory and compute bound applications run 1.17× and 1.19× slower with GPU-ping, while the peak GPU-ping packet rate is reduced by 4.83× and 2.54×, respectively. This performance impact is caused by the CPU having to switch between the CPU applications and GPU task management, even though all of the actual GPU-ping processing is performed on the GPU. Hence, reducing the GPU's reliance on the CPU can improve system efficiency and lower the kernel launch latency when consolidating workloads onto independent devices.

There are multiple alternatives for launching GPU tasks besides the baseline described above. Figure 3b illustrates persistent threads (PT) [20], which is a software-only approach to manage GPU tasks via polling GPU threads. PT pre-configures a large number of continuously running TBs that poll in-memory work queues and perform the application processing. PT replaces the GPU driver and hardware task schedulers with a GPU software scheduler, relaxing the requirement for the CPU to be involved. The polling TBs can improve efficiency for smaller kernels by reducing kernel launch overheads, but can reduce efficiency for larger tasks, as the software schedulers are less efficient than hardware. Figure 4b evaluates a PT framework relative to the baseline for a continuous stream of matrix multiplication kernels on two matrix sizes; small (2 TBs/kernel) and large (120 TBs/kernel). Power is measured using the NVIDIA Management Library [46]. With the small kernel, PT spends more time performing the matrix multiplication relative to the kernel launch overheads, which decreases execution time by 2.9× with only 20% higher power; hence lowering energy. However, the increased overhead from the TB software scheduler on larger kernels increases execution time by 5.6× while lowering power by 45%; hence increasing energy. Additionally, PT polling indefinitely consumes GPU resources, which limits the potential for other kernels to run and increases power consumption relative to the baseline by 13% (in an active power state, p2) when no tasks are pending (*Active Idle*).

Figure 3c illustrates GPUDirect Async [52], which removes the CPU from the critical path for receiving and sending data between an Infiniband NIC and the GPU. For receiving, GPUDirect Async pre-configures the receive buffers and pre-launches the kernels to be asynchronously executed when the data arrives, reducing launch overheads and freeing up the CPU when waiting for data. Similarly for sending, GPUDirect Async pre-configures the transmit buffers to enable the GPU to directly send data to the NIC, freeing up the CPU from managing the transmit on behalf of the GPU. However, the CPU is still required to manage each kernel launch.

HSA [17] utilizes user-level in-memory work queues and doorbell registers to configure and launch GPU tasks through

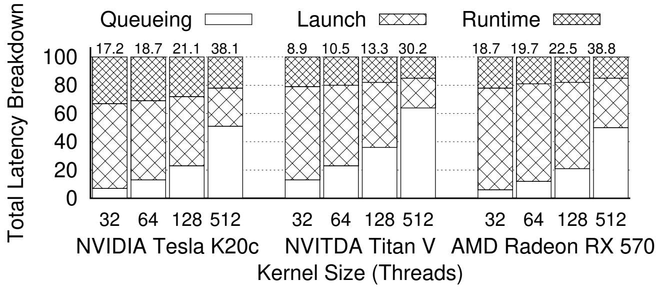


Fig. 5: Latency breakdown for fine-grained streaming GPU kernels. Absolute latencies above the bars are in microseconds.

standard memory operations. An HSA packet processor (typically a hardware unit on the GPU) processes the work queues to launch kernels. While HSA does not require the CPU or GPU driver to launch GPU tasks, HSA still requires each GPU task to be separately configured, which increases the amount of data needed to specify the GPU task.

Figure 3d illustrates XTQ [33], which proposes an extension to Infiniband NICs to internally manage pre-registered HSA tasks through RDMA. RDMA writes specify a kernel ID, which the NIC uses to configure and launch the corresponding GPU kernel via an HSA work queue. XTQ lowers kernel launch overheads by implementing the HSA logic in the Infiniband NIC to bypass the CPU on the critical path. As XTQ exploits RDMA on Infiniband, it is more targeted to remote task launching than fine-grained streaming applications. XTQ also pushes the kernel management functionality to the NIC, not the GPU, requiring other external devices to reimplement similar XTQ functionality to launch tasks directly on the GPU.

Figure 3e illustrates EDGE. EDGE adopts a similar programming model and execution flow as HSA and XTQ. However, EDGE pushes the kernel configuration and management logic to the GPU, reducing complexity for external devices wanting to launch tasks directly on the GPU. Additionally, EDGE provides optimizations for fine-grained streaming GPU tasks.

Another approach to reduce kernel launch overheads is to use integrated GPUs. These GPUs reside on the same die as the CPU and share a common physical memory, inherently reducing kernel launch and data transfer overheads relative to discrete GPUs. However, integrated GPUs typically have lower peak processing capabilities than discrete GPUs, which may limit their effectiveness in the datacenter. Furthermore, for tasks not originating from the CPU, such as a NIC, the physically shared CPU-GPU memory provides reduced benefits.

C. *Fine-Grained GPU Kernels*

Latency-sensitive streaming GPU applications, such as packet processing, may benefit from using smaller kernels (fewer threads). Many highly-parallel GPU packet processing applications batch packets together and process each packet with a separate GPU thread [19], [21], [22]. Thus, the kernel size is variable. Larger batches, and hence larger kernels, are typically constructed to improve throughput, increase GPU utilization, and mitigate kernel launch overheads. However, this comes at the cost of increased queuing and processing latencies. If the kernel launch overheads were low, the smallest batch size that can benefit from GPU parallelization is the size of a warp (e.g, 32 packets). Decreasing the batch size can reduce the packet queuing latency and the time to process each packet batch, while multiple small packet batches can be pipelined on the GPU to maintain high throughput.

Additionally, irregular applications are often represented as multiple tasks with low degrees of parallelism, which can also benefit from using smaller kernels. Pagoda [68] lists several example applications, such as DCT, fractal analysis, and beamforming. However, the kernel launch overheads for smaller kernels have a higher contribution to the end-to-end latency, as they are amortized over less work.

To estimate the launch overheads described above, we measure the latency breakdown for a continuous stream of GPU IPv4 forwarding kernels with different kernel sizes at 8 Gbps (Figure 5). We evaluate two discrete NVIDIA GPUs and an AMD GPU using CUDA and ROCm [1], respectively. Each GPU thread processes a separate packet. Section VI-A discusses how the launch overheads were computed. When the kernel is small, the kernel launch overheads contribute to a larger fraction of the total latency than the packet queuing and kernel runtime. The opposite behavior is seen when the packet batch sizes are larger. If the kernel launch overheads are lowered, the end-to-end latency for the smaller packet batches can be reduced.

D. *GPU Multiprogramming*

Current GPUs have limited support for compute multiprogramming, although they have long supported preemptive multiprogramming for graphics [42]. Recent NVIDIA GPUs [43], [47] and the HSA specification [17] describe support for instruction-level kernel preemption for higher-priority compute tasks; however, our experience with the most recent NVIDIA and HSA-enabled AMD GPUs show that higher-priority kernels wait for the first available resources (e.g., a TB completion) instead of immediately preempting.

There has been a large body of research towards improving preemption/context switching and multiprogramming on GPUs [9], [28], [38], [49], [53], [57], [64], [65], [69]. These works tend to target resource sharing, spatial multitasking, and external/self-initiated preemption at the TB, SIMT core, or full kernel level with hardware, software, and/or compiler support. Many of these works identify that saving and restoring the context for a full TB or kernel is high, and evaluate alternative techniques to preempt at the TB granularity. For example, both FLEP [65] and EffiSha [9] propose using a compiler to transform large multi-TB kernels into a smaller set of persistent thread-like TBs and a CPU/GPU runtime to schedule kernels to initiate preemption. Preemption occurs when a persistent-TB completes one or more executions of one kernel and identifies that it should be preempted to execute a different kernel. However, if the size of a higher-priority kernel is small, such as a single warp, preempting or waiting for a full TB/kernel

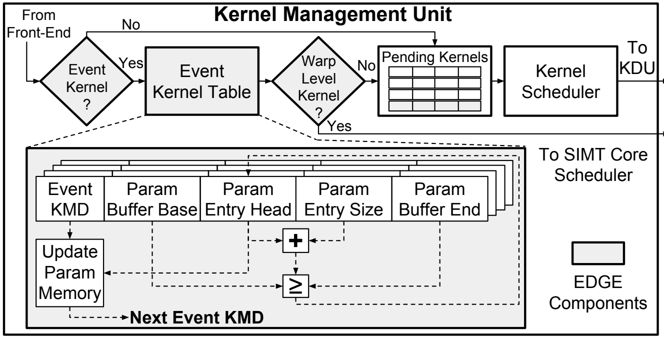


Fig. 6: EDGE event kernel table (EKT) architecture.

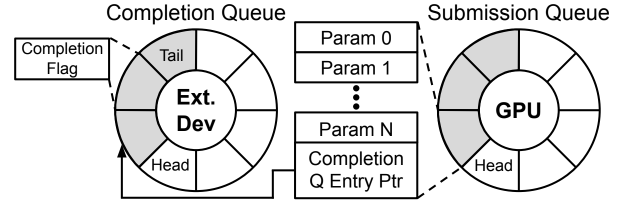


Fig. 7: EDGE event submission and completion queues.

to complete can lead to high preemption latencies (Figure 12) and increase the impact on the kernel being preempted. For small kernels, EDGE instead targets partial-TB preemption at the warp granularity, reducing preemption overheads and enabling the lightweight co-execution of fine-grained streaming tasks with concurrently running applications. For larger kernels, EDGE can make use of these previously proposed preemption techniques.

III. Event-Driven GPU Execution

EDGE provides an alternative path for devices to directly and efficiently launch tasks on the GPU. The basic device requirements for supporting EDGE are discussed in Section III-D. Supporting an event-driven execution model on the GPU has four main requirements: the GPU needs to know which task to run on a given event, which data each event task should operate on, when to execute the task, and how to indicate that the task is complete. This section describes how each of these requirements are addressed in EDGE.

A. Event Kernels and the Event Kernel Table

EDGE introduces a new type of GPU kernel, the *event kernel*, which is internally launched by the GPU in response to an internal or external event. Event kernels are configured once and remain on the GPU, such that any device capable of communicating with the GPU is able to launch an event kernel. EDGE manages event kernels in an *event kernel table* (EKT) within the kernel management unit (KMU), as shown in Figure 6. When a kernel operation is received from the GPU's front-end, traditional kernels flow directly to the pending kernel queues, while event kernels access the EKT. EDGE supports registering a new event kernel, removing an existing event kernel, and launching an event kernel from the EKT.

The EKT consists of a small on-chip SRAM buffer and control logic. The EKT is responsible for storing the pre-configured event kernel metadata structures (event KMDs), which contain the necessary information to describe the event kernel, such as the kernel function pointer, kernel dimensions, and pointer to the kernel parameter memory. Multiple different event kernels can be stored, with the size of the EKT dictating the maximum number of separate event kernels. Event kernels are assigned a unique identifier upon registration (Section III-E), which can be used to look up the corresponding event kernel from the EKT. The remaining portions of the EKT are described in Section III-C.

B. Event Kernel Parameter Memory

Part of the baseline GPU kernel launch overheads are a result of configuring and communicating the kernel parameters with the GPU. EDGE exploits the opportunity that the same kernel will have the same parameter memory structure (same types, number, and order of parameters). As such, the parameter memory could be pre-configured (e.g., setting constants and GPU memory buffers) for the maximum number of concurrent instances of each event kernel, removing the need to repeatedly configure it on the critical path in streaming applications.

EDGE adopts a similar approach to HSA [17] and PT [20], utilizing user-level, in-memory task queues to interact with the GPU. In-memory task queues replace expensive API calls to the GPU driver with simple memory load and store instructions. However, pre-allocating and configuring the parameter memory once instead of before each kernel launch requires that both the GPU and external device know which entries in the task queues are free, complete, or contain valid tasks to launch. EDGE reduces this complexity with in-order circular task queues. Consequently, the external device does not need to query a runtime or the GPU for the next valid entry in the task queue, nor does it need to specify which entry contains a valid task when launching an event. When the GPU receives a signal to launch an event kernel, the parameter memory to operate on is simply the next entry in the queue.

A pair of circular task queues manage task submission and completion in EDGE, as shown in Figure 7. Each entry in the submission queue contains the pre-configured event kernel parameters and pointer to the corresponding entry in the completion queue. The size of each submission queue entry is dependent on the number and size of event kernel parameters. The completion queue maintains a set of flags specifying if an entry is free or complete, which is polled by the external device. The external device is responsible for managing its own head/tail logic for the circular completion queue and clearing the completion flag after consuming the results of an event kernel. As is described in Section III-C, the GPU only requires a single head pointer. To minimize access latency for each device, the submission and completion queues are stored in GPU global memory and external device memory, respectively.

C. Launching Event Kernels

Once registered with the GPU, devices can directly launch event kernels by signalling the GPU. Similar to HSA, EDGE achieves this through writes to memory-mapped doorbell registers. Each entry in the EKT is tied to a unique doorbell register, which is returned upon event kernel registration. EDGE requires modifications to the GPU's front-end (Figure 2) to identify doorbell register writes and trigger reads from the corresponding entries in the EKT. Launching an event kernel is therefore reduced to a single read from the on-chip EKT buffer, significantly lowering kernel launch overheads.

Each EKT entry contains an event KMD and metadata describing the event kernel's parameter memory (Figure 6). *Param Buffer Base* stores the base address of the submission queue. *Param Entry Head* stores the address of the next valid parameter memory to use from the in-order submission queue. *Param Entry Size* stores the size of each entry in the submission queue. Finally, *Param Buffer End* stores the address of the last entry in the submission queue. The EKT also contains logic for implementing the circular buffer in hardware. When an EKT entry is read, the current head is incremented by the parameter memory size and stored back into the head. The head pointer is reset to the base pointer when exceeding the maximum number of entries in the submission queue.

As shown in Figure 6, traditional GPU kernels flow directly to the pending kernel table, whereas event kernel requests access the EKT. An event registration request stores the EKT and metadata in an EKT entry. An event launch request reads an entry from the EKT to prepare the event kernel for scheduling. Event kernels larger than a single warp are transferred to separate event kernel queues in the KMU to be scheduled by the kernel scheduler, similar to device launched GPU kernels in NVIDIA CDP [63]. As will be described in Section IV, small event kernels (a single warp) are instead transferred directly to the SIMT core scheduler for fast scheduling. This enables EDGE to increase the total number of concurrent warp-level kernels on a GPU (more details in Section IV-C). Upon completion, a simple hardware unit or GPU software completion routine can mark the corresponding event completion flag as done.

D. Device Requirements for Supporting EDGE

There are two main requirements for devices to support EDGE. First, the device must be able to read and write directly to GPU memory. This is satisfied by any device able to communicate over the PCIe bus with the GPU (e.g., DMA). Directly accessing GPU memory is required to populate the input GPU buffers, set any event kernel arguments, write to the doorbell registers, and read from the output GPU buffers. Second, the device requires logic for interacting with the circular event submission and completion queues (Section III-B). For example, the device needs to manage internal head and tail pointers and check/update the event kernel status in the completion queue entries. If the device contains an internal command processor, this logic can be implemented in software running on the processor. Otherwise, this logic should be implemented in hardware.

TABLE I: EDGE API extensions.

eventDoorbellReg = **edgeRegisterEvent**<<<pType0,... pTypeN>>>(&eventSubQueue, kernelPtr, gridDim, TBDim, shrdMem, maxNumEvents)
edgeUnregisterEvent(eventDoorbellReg)

E. EDGE API

Table I presents the EDGE extensions to the GPGPU API (e.g., CUDA or OpenCL). The API enables the CPU to register and unregister event kernels with the GPU. Similar to HSA, all other EDGE operations, such as configuring parameter memory and launching event kernels, are done through generic memory operations.

Registering an event kernel is similar to the baseline CUDA kernel launch API. The main differences are that the kernel launch is delayed by storing the kernel metadata structure on the GPU (*kernelPtr, gridDim, TBDim, shrdMem*) and the parameter memory structure is specified (*pType0...pTypeN*) instead of passing the actual parameters. *edgeRegisterEvent* allocates the event submission queue in GPU memory based on the structure of event kernel parameters and maximum number of in-flight events, which is stored in *eventSubQueue*. The same GPU driver logic as the baseline kernel launch API can be used to verify the kernel pointer, dimensions, and shared memory requirements. A unique doorbell register, *edgeDoorbellReg*, is assigned to the event kernel and returned. The doorbell register address is also used as an identifier to unregister the event kernel and free the corresponding submission queue memory. Under successful operation, the new event kernel is stored in an event kernel table entry and the parameter buffer values are initialized to the first submission queue entry. As event kernels are associated with a doorbell register, not a kernel name, separate event kernels (potentially with the same name) can be concurrently registered with a unique set of parameters. The CPU is then responsible for allocating the event completion queue in external device memory (using an available API for the external device), pre-allocating all of the GPU input/output buffers for the event kernels, and assigning the corresponding completion queue entry pointers, GPU buffer pointers, and any constant parameters in each submission queue entry.

If *edgeRegisterEvent* fails, a NULL value is returned. This can happen if any inputs are invalid, if the event kernel table is full, or if there is not enough GPU memory to allocate the event submission queue. In the event of a failure, any modified state is reverted and allocated memory is freed.

Figure 8 presents an example of the EDGE API (Figure 8c) compared to the baseline (Figure 8b) for a streaming GPU application where tasks originate from an external device (e.g., a NIC or FPGA). EDGE does not require any modifications to the GPU kernel code from the baseline (Figure 8a). In both the baseline and EDGE, the parameter memory is pre-configured for the maximum number of in-flight kernels, while EDGE also assigns the corresponding completion queue pointers. The main difference is that the baseline requires one or more CPU threads

```
__global__ void kernel(arg0type arg0, arg1type arg1, ... , argNtype argN) { ... }
// EDGE event kernels can update the completion queue flags in GPU hardware or with a
// system-level function that automatically runs after the event kernel completes
```

(a) GPU Code (Same for the baseline and EDGE)

```
/* All code below runs on the CPU. Device code (e.g., device driver/processing) is not shown */
main():
  // Allocate/configure parameter memory for the maximum number of in-flight kernels
  gpuParams = configureBaseParamMem(maxInflightKernels);
  // Configure the device with the pre-allocated parameter memory
  configureDeviceWithGPUArgs(gpuParams, maxInflightKernels);
  ...
  // Start one or more polling CPU threads to manage the GPU kernel launch and completion
  startCPULaunchCompleteThreads();

configureBaseParamMem(maxInflightKernels):
  paramBuffer = allocParamBuffers(maxInflightKernels, paramSize);
  for (i=0 ... maxInflightKernels):
    paramBuffer[i].arg0 = gpuMalloc(...);
    paramBuffer[i].arg1 = some_consant;
    ...
    paramBuffer[i].argN = gpuMalloc(...);
  return paramBuffer;

cpuThreadsLaunchKernels():
  while (true):   // CPU threads poll the device for new tasks to launch
    // Get the next task from the external device and launch the kernel. The external device (not shown)
    // writes to the GPU buffers directly (e.g., RDMA) and updates any arguments
    arg0, arg1, ..., argN = getNextTaskFromDevice();
    kernel<<<gridDim, TBDim, shrdMem, nextThreadStream()>>>(arg0, arg1, ... argN);
    gpuEventList.append(new gpuKernelEvent); // Record information to track the kernel's completion

cpuThreadsCompleteKernels():
  while (true):                          // CPU threads poll the GPU for completed kernels
    gpuKernelEvent = getNextKernelEvent();  // Check if any kernels have completed
    if (gpuKernelEvent == "done"):
      // Signal the device to consume the output buffers populated by the GPU
      notifyDeviceOfTaskCompletion(gpuKernelEvent);
      gpuEventList.remove(gpuKernelEvent);
```

(b) Baseline

```
/********************************* Configuration code below runs on the CPU *********************************/
main():
  // Register the event kernel with the EDGE API
  eventDoorbellReg = edgeRegisterEvent<<<arg0type, arg1type, ..., argNtype>>>(
                                &eventSubQ, kernel, gridDim, TBDim, shrdMem, maxInflightKernels);
  // Allocate the event completion queue in device memory and set all to free (external device-side function)
  eventCompQ = allocateInDeviceMemory(maxInflightKernels);
  // Configure the parameter memory for the maximum number of in-flight kernels
  configureEdgeParamMemory(eventSubQ, eventCompQ, maxInflightKernels);
  // Configure the device with the pre-allocated parameter memory and EDGE components
  configureDeviceForEdge(eventSubQ, eventCompQ, eventDoorbellReg, maxInflightKernels);

configureEdgeParamMemory(paramBuffer, compQ, maxInflightKernels):
  for (i=0 ... maxInflightKernels):
    paramBuffer[i].arg0 = gpuMalloc(...);
    paramBuffer[i].arg1 = some_consant;
    ...
    paramBuffer[i].argN = gpuMalloc(...);
    paramBuffer[i].compQptr = &compQ[i];  // Store EDGE completion queue pointer

/******* Code below runs on any device (e.g., internal processor or hardware implementation) *******/
deviceSideLaunchKernels():
  while (true):
    newData = getDataForGPUTask();    // External device generates data for the GPU to operate on
    if (eventCompQ[head] == "free"):     // Check that the GPU can handle the new task
      // Directly write any data to the pre-allocated GPU buffers (e.g., RDMA) and update any args
      writeToInputArgs(eventSubQ[head], newData);
      // Mark the task as busy and increment head
      eventCompQ[head] = "busy"; head = (head + 1) % maxInflightKernels;
      writeToDoorbellReg(eventDoorbellReg);  // Signal the GPU to launch the new event kernel

deviceSideCompleteKernels():
  while (true):
    if (eventCompQ[tail] == "done"):                    // Check if the GPU has completed the task at tail
      result = readFromOutputArgs(eventSubQ[tail]);  // Handle the completion of the task
      ...
      // Mark the task as free and increment the tail
      eventCompQ[tail] = "free"; tail = (tail + 1) % maxInflightKernels;
```

(c) EDGE

Fig. 8: Pseudo code comparison between the baseline and EDGE for an example streaming GPU application.

to poll the external device for new tasks to launch and poll the GPU for completed tasks to consume through the standard GPU APIs, whereas EDGE implements the kernel launching and completion management directly on the external device (Section III-B, Section III-C). As described in Section III-D, the required device-side EDGE logic may be implemented as code running on an internal processor or as dedicated hardware. While not shown, the baseline also requires device-side logic for configuring the device to populate/consume the GPU buffers (e.g., through a device driver or device API).

IV. Event Kernel Preemption and Priority

Event kernels are treated like regular host or device launched kernels. They can be of any shape (kernel dimensions) or size (number of threads), utilize the baseline GPU hardware task and thread schedulers, and have a configurable priority. The ability to increase the priority can help to maintain a level of quality of service (QoS) in an enviornment where other GPU tasks may be run concurrently, such as the datacenter.

As described in Section II-C, certain GPU applications, such as packet processing, may beneft from using smaller kernels. If the GPU is fully occupied executing other tasks, however, the latency benefits of using smaller kernels is reduced. To address this, we propose a fine-grained partial preemption technique when the event kernel is the size of a single warp. In EDGE, a lower-priority background kernel is only partially preempted to obtain the required resources to run the event kernel. This form of partial preemption for warp-level event kernels can enable the fine-grained co-execution of multiple kernels to reduce impacts on the system QoS compared to full preemption or waiting for TBs from concurrently running kernels to complete.

A. Spatial Multitasking through Hardware Reservation

A simple coarse-grained approach to ensure event kernels have enough resources to execute is to statically reserve a fraction of the resources for the event kernel. For example, the event kernel registration could reserve specific GPU SIMT cores to handle the event kernel. While this indefinitely lowers the amount of resources available for normal tasks to run on the GPU, the time to schedule an event kernel and the contention for shared resources (e.g., caches, memory bandwidth) can be significantly reduced. In Section VI, we evaluate a fair-sharing approach, where background kernels and event kernels receive an equal number of reserved SIMT cores. Evaluating a dynamic resource reservation technique is left to future work.

B. Thread Block-Level Event Kernels

In the baseline GPU, TBs from higher-priority (HP) kernels interrupt TBs from lower-priority (LP) kernels at the TB execution boundaries. Specifically, the SIMT core scheduler (Figure 2) stops scheduling new LP TBs and prioritizes HP TBs from the KDU as LP TBs complete. Once all TBs from the HP kernel have been scheduled, the scheduler resumes scheduling TBs from the LP kernel. We refer to this as *draining* partial preemption, which is similar to the TB draining technique presented in [57]. While this process enables the HP kernel to begin executing as soon as enough resources become available, the scheduling latency for the HP kernel is dependent on the

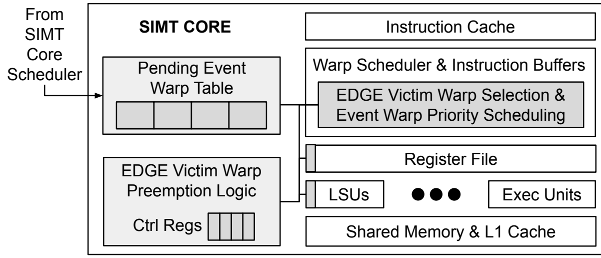


Fig. 9: EDGE SIMT core architectural modifications.

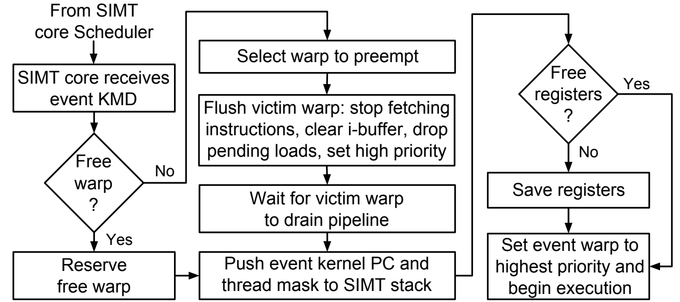


Fig. 10: EDGE SIMT core preemption logic.

remaining execution time of the LP TBs. This can lead to high variability in the event kernel scheduling latencies. We also evaluate TB draining in Section VI.

C. Warp-level Events

There are several simplifications when scheduling a kernel containing a single warp compared to multiple TBs, which can be exploited to improve efficiency. The SIMT core scheduler (Figure 2) is responsible for scheduling TBs from running kernels in the KDU to SIMT cores with enough resources to handle the TB. The scheduler configures SIMT core control registers to indicate which TBs from the kernel are currently executing and to track when TBs complete. Subsequent TBs are then scheduled or the completed kernel is removed. For a kernel with a single warp, many of these requirements are removed. For example, when the warp completes, the kernel completes. Consequently, the management of warp-level kernels does not need to be centralized across all SIMT cores. Similar to the global KDU, EDGE proposes minor hardware extensions to the SIMT cores to internally manage warp-level kernels, as shown in Figure 9. This increases the maximum number of concurrent warp-level kernels on the GPU without the overheads for managing kernels of any size and shape. As shown in Figure 6 and Figure 9, warp-level event kernels are transferred directly to the SIMT core scheduler, which selects a SIMT core to handle the warp-level event and transfers the event kernel to a small Pending Event Warp Table residing in the SIMT core.

D. Warp-level Preemption

GPUs contain multiple SIMT cores and warp contexts, each potential candidates to handle the warp-level event kernel. For example, the NVIDIA GeForce 1080 Ti contains 28 SIMT cores with up to 64 warps per SIMT core, for a total of 1792 warps. GPUs schedule TBs to SIMT cores until the first required resource is depleted (registers, shared memory, warps, TB contexts). However, this can lead to underutilization if one resource is more heavily used than others. Depending on the resource requirements of the warp-level event kernel and the current SIMT core utilization, the event warp may be able to immediately run on a free warp context. However, if there are no available resources to process the warp-level event kernel, the kernel must either block until sufficient resources are available or preempt a running warp.

We propose a fine-grained, warp-level preemption mechanism that can temporarily borrow the necessary resources from a running TB to execute the warp-level event kernel. The main challenge with preempting a warp is to minimize the preemption latency. Compared to full TB or kernel-level preemption, warp-level preemption requires saving significantly fewer resources, which lowers preemption overheads. Additionally, GPUs exploit fine-grained multithreading to hide the effects of long latency operations by seamlessly switching between multiple warps. If a warp is temporarily preempted to process a warp-level event kernel, the preempted task can still make progress by scheduling other warps from the TB. Figure 9 highlights the high-level architectural modifications and Figure 10 presents the EDGE SIMT core logic to support warp-level preemption, as described below.

EDGE preempts warps at an instruction granularity instead of waiting for a warp to complete all of its instructions, as is done in TB draining (Section IV-B). To correctly save a warp's context and to minimize complexity, any pending instructions are flushed from the pipeline prior to starting the event kernel. Depending on the state of the selected warp to preempt, called the *victim warp*, or the instructions currently in the pipeline, this preemption latency can be quite large with high variability.

We identified four main causes of large scheduling latencies: First, low scheduling priority for the victim warp; second, pending instructions in the instruction buffer (i-buffer); third, victim warps waiting at barriers; and fourth, in-flight loads.

EDGE with warp-level preemption tackles these sources of increased latency by employing, respectively, the following *flushing optimizations*: First, to ensure that the victim warp completes promptly, the victim warp's priority is temporarily increased. Second, flushing any non-issued instructions from the i-buffer limits the number of instructions the warp-level event kernel needs to wait for. Third, a victim warp waiting at a barrier is a perfect candidate for interrupting, since the warp is currently sitting idle. This is referred to as *barrier skipping*. However, we need to ensure that the victim warp is correctly re-inserted into the barrier when the event kernel completes. Finally, in-flight loads can be dropped and replayed for victim warps. Dropping loads involves releasing the MSHR entry for the pending load, releasing the registers reserved in the scoreboard, and rolling back the program counter to re-execute the load instruction once the victim warp is rescheduled after the event kernel. We also need to ensure that no independent instructions for the victim warp have completed out of order following the load instruction, due to separate execution pipelines [58]. Section

VI-B evaluates the benefits of these flushing optimizations on preemption latency.

Once the victim warp has been properly flushed, any resources overwritten by the event warp need to be saved. Unlike a regular warp, event warps do not require TB barrier management, since there is only a single warp. Additionally, the special thread / TB dimension registers are always initialized to the same values. The event warp can use the victim warp's SIMT stack [18] to efficiently save the program counter and warp divergence state by pushing on a new entry containing the program counter of the event kernel's start address and a full active thread mask. When the event kernel completes, the event kernel entry can be popped off the SIMT stack to resume execution of the preempted victim warp.

Any registers used by the event warps need to be saved and restored, which is achieved through preallocated regions in global GPU memory. However, as shown in Figure 10, if the SIMT core register file is underutilized and contains enough free registers, the event warp instead reserves new registers. In EDGE, we avoid changing the resource requirements of a victim warp during preemption and only allow event warps to preempt victim warps with the same or lower number of registers (even if there are enough free registers in the register file). This can limit the opportunities for event-warp preemption and is discussed more in Section VI-C. Other works have also shown that GPU multiprogramming can benefit from underutilized GPU resources through resource partitioning/sharing [64], [67].

To further improve QoS for the event kernel, once an event kernel is running, EDGE sets its warp priority to the highest level in the warp scheduler.

E. EDGE Overheads and Limitations

The EKT (Figure 6) is modeled as a small on chip buffer with a single read/write port. Each row contains five 8B values for the kernel pointers and parameter buffer pointers, six 4B values for the kernel dimensions, and a 2B value for the shared memory requirements, for a total width of 66B. Assuming the EKT has 32 entries (maximum of 32 different event kernels), the total storage overhead for the EKT is 2.1KB – less than 1% of a single SIMT core's register file. The EKT also contains a single 64-bit adder and comparator for the circular buffer logic. EDGE requires one or more Pending Event Kernel queues in the KMU, which can be achieved by adding pending kernel queues or reusing the existing hardware queues for host/device launched kernels. EDGE also adds logic to the GPU front-end for identifying writes to the doorbell registers and to the SIMT core scheduler for selecting a SIMT core to handle a warp-level event kernel. The SIMT core modifications (Figure 9) include the small event warp table to manage event kernels and the logic for warp-level preemption. A four-entry event warp table adds 160B per SIMT core. Supporting warp-level preemption requires logic to prioritize warps in the warp scheduler, manipulate valid bits to flush entries from the instruction buffer, release scoreboard/MSHR entries, and control registers to drop in-flight loads and to indicate if the victim warp should return to a warp barrier after completing the event kernel.

TABLE II: Gem5-GPU Configuration

Component	Configuration
Num SIMT core / frequency	16 / 700MHz
Max TBs / threads per SIMT core	16 / 2048
Num registers per SIMT core	65536
L1 $ / shared memory size per SIMT core	64KB / 48KB
L2 $ size	1MB
GPU base warp scheduler	Greedy-then-Oldest
Gem5 CPU / memory configuration	O3CPU / Gem5 fused

A limitation with warp-level preemption is shared memory, which is allocated at a TB level. Consequently, there is no warp-local region of shared memory for the event warp. Using shared memory would require saving and restoring the entire shared memory for the victim warp's TB and blocking the TB until the event warp completes. To keep overheads low, we restrict preemption for event warps not requiring shared memory. Event kernels requiring shared memory are treated as regular event kernels.

V. Methodology

EDGE is implemented in Gem5-GPU v2.0 [50] with GPGPU-Sim v3.2.2 [5]. Although GPGPU-Sim is loosely based on an NVIDIA architecture, it implements a generic SIMT execution model and memory hierarchy similar to both NVIDIA and AMD GPUs. As such, we expect that the proposed techniques in EDGE will be relevant to both NVIDIA and AMD architectures. Additionally, we observed similar trends for kernel launch overheads on both NVIDIA and AMD GPUs (Figure 5).

Gem5-GPU was modified to include support for CUDA streams, concurrent kernel execution, concurrent TB execution from different kernels per SM (CDP changes in GPGPU-Sim [63]), and kernel argument memory using Gem5's Ruby memory system. The Gem5-GPU configuration used in this work is listed in Table II. We modified the baseline Gem5-GPU architecture to include a timing model for the EKT, updates to the SIMT core scheduler, EDGE SIMT core controller, victim warp selection and flushing, and the fine-grained preemption mechanisms. A static latency is included to account for the kernel dispatch time from the KMU to running on a SIMT core (discussed in Section VI). Gem5-GPU's CUDA runtime library was extended with the EDGE API in Table I to manage event kernels on the GPU.

We evaluate multiple traditional GPU kernels and networking-based event kernels. The traditional GPU applications are Back Propagation (BP), Breadth First Search (BFS), K-means (KMN), LU Decomposition (LUD), Speckle Reducing Anisotropic Diffusion (SRAD), and Stream Cluster (SC) from Rodinia [8], Matrix Multiply (MM) from the CUDA SDK examples, and two convolution kernels, filterActs_YxX_color and filterActs_YxX_sparse (CONV1/2), from Cuda-convnet [32] using a layer configuration similar to LeNet [34]. These kernels contain varying thread block runtimes, which can benefit thread block draining or preemption depending on the duration and completion rate. Additionally, the input data

size for the traditional GPU kernels was set such that event kernels are not optimistically biased to having free TBs/warps available based on the kernel's implementation. The event kernels are MemC [22], IPv4 and IPv6 IP forwarding [26], and IPSec [68]. MemC, a key-value store and UDP packet processing kernel, is evaluated on *GET* requests using 16B keys and 2B values. Shared memory is removed from MemC. The hashtable contains 16k entries and is warmed up with 8k *SETs*. The IPv4/6 forwarding tables are constructed using the same prefixes as in [26]. IPSec implements the 3DES algorithm [15] with 256B packet payloads. Each event kernel is configured for 32 requests per event-warp except for IPSec, which launches a single event-warp per packet.

VI. Evaluation

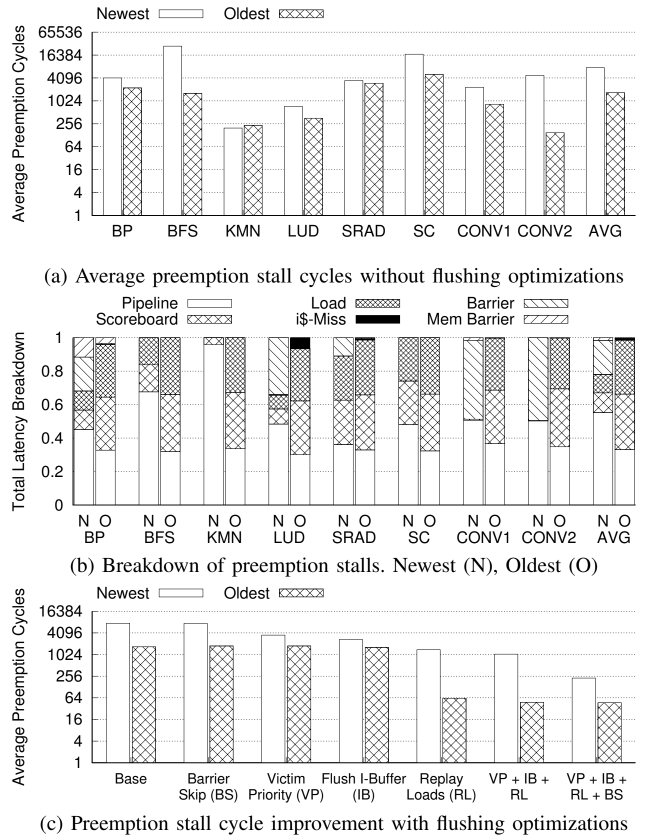
This section evaluates EDGE and warp-level preemption.

A. Event Kernel Launch Latency

We first evaluate the reduction in kernel launch latency with EDGE relative to the baseline CUDA kernel launching approach. We measure the baseline launch latency on an NVIDIA Titan V with both the NVIDIA profiler and CPU TSC timers on a microbenchmark that launches an empty kernel (single TB and warp), using the CUDA driver API. The timers are placed immediately before the kernel launch and after device synchronization, and the synchronization instruction overhead (measured with the profiler) is subtracted from the total time. Averaging over 100k kernel launches, we find that the baseline kernel launch latency is ∼5us. Furthermore, the asynchronous CUDA launch API call alone takes ∼2us, which limits the maximum event rate a single CPU thread can support.

We estimate the kernel launch latency for an event kernel already configured and residing on the GPU in EDGE using results from DTBL [62]. DTBL measures the kernel dispatching latency (in GPU cycles) to schedule a kernel residing in the KMU through the KDU to the SIMT cores using the clock() instruction at the end of one kernel and the start of another kernel dependent on the first. This dependency restricts the second kernel from being scheduled until the first kernel completes, effectively measuring the launch latency for a kernel residing in the EKT (Figure 6). DTBL estimates an average of 283 GPU cycles on the NVIDIA K20c. We also tried to reproduce these results; however, additional instructions inserted by the NVIDIA compiler, such as the kernel parameter loading, address resolution, and the latency to store the result of clock() at the end of the first kernel, introduce overheads unrelated to the kernel dispatch latency. We contacted an author of DTBL (who was working at NVIDIA) and confirmed that they verified this dispatch latency using internal tools. As such, we use an event kernel dispatch latency of 300 GPU cycles to account for the additional overheads of loading an event kernel from the EKT to the KMU.

To account for the PCIe overheads when writing to the doorbell register and completion queue, we add 700ns (PCIe round-trip latency) to EDGE's kernel launch overhead [16], [36], [40], [61]. With 300 GPU cycles for the event kernel dispatch latency (at 705 MHz), we estimate that EDGE can reduce the kernel launch latency by ∼4.4× (1.1us) compared to the average baseline CUDA launch latency (5us).



(a) Average preemption stall cycles without flushing optimizations

(b) Breakdown of preemption stalls. Newest (N), Oldest (O)

(c) Preemption stall cycle improvement with flushing optimizations

Fig. 11: Event warp preemption latency evaluation

B. Warp-Level Preemption

A key requirement for preemption is low and predictable latency. However, this is challenging when the warp to preempt (victim warp) can be in any state of execution in the pipeline. Warp-level preemption requires flushing the victim warp's instructions from the pipeline. ALU operations (as modeled in GPGPU-Sim) range from 4 to 330 cycles depending on the operation and precision, while long latency memory operations (e.g., DRAM accesses) can take 1000's of cycles. Waiting for the victim warp to naturally flush the pipeline may result in high scheduling latencies. Figure 11a measures the average preemption latency for an event kernel with low resource requirements on two warp selection policies: Newest and Oldest, where the most or least recently launched warps are selected, respectively. With the baseline greedy-then-oldest warp scheduling policy, selecting older warps tends to have a much lower preemption latency, since older warps are prioritized. As such, the victim warp selection alone can have a large impact on scheduling latency, ranging from 2-8k cycles on average. Evaluating methods to optimize the warp selection to reduce preemption latency is an area for future work.

Figure 11b presents a breakdown of the factors contributing to the event warp preemption latency, which are not mutually

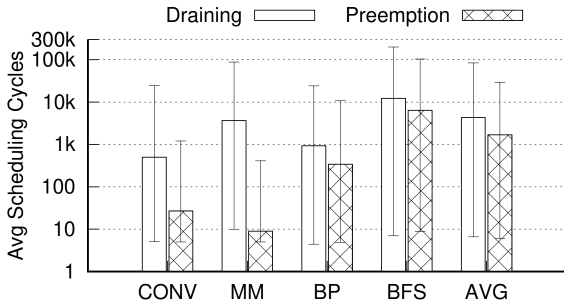


Fig. 12: Average, min, and max event kernel scheduling latency.

exclusive. For example, an instruction that is in the pipeline may also have registers reserved in the scoreboard. *Pipeline* means that a warp instruction is in the pipeline, *Scoreboard* means that the instruction has registers reserved in the scoreboard, *Load* indicates the warp is waiting for in-flight loads, *I$-Miss* is an instruction cache miss, and *Barrier* and *Mem Barrier* indicate that warps are waiting at thread or memory barriers, respectively. The trend for Oldest policy is very similar – victim warps are waiting for load instructions to complete. The Newest policy has more diversity, such as waiting for thread barriers. This is related to the implementation of the evaluated benchmarks, which, for example, have thread barriers near the start of the kernel after loading data from global to shared memory.

Figure 11c measures the improvements in preemption latency when applying each of the preemption optimizations discussed in Section IV-D. The average preemption latency is reduced by 35.9× and 33.7× for the Oldest and Newest policies, respectively. At 705 MHz, the average preemption latency is ∼70*ns* (50 cycles) and ∼310*ns* (220 cycles), respectively. For Oldest, the largest benefit comes from dropping and replaying load instructions, which is explained by the large fraction of stalls on pending loads in Figure 11b. For Newest, many benchmarks are also stalled on barrier instructions. However, only applying the barrier skipping does not result in a large reduction because BFS and SC, which are not blocked on thread barriers, dominate the average calculation. After reducing the preemption latency with the other optimizations, thread barriers have a larger contribution to the average preemption latency.

C. Preemption vs. Draining

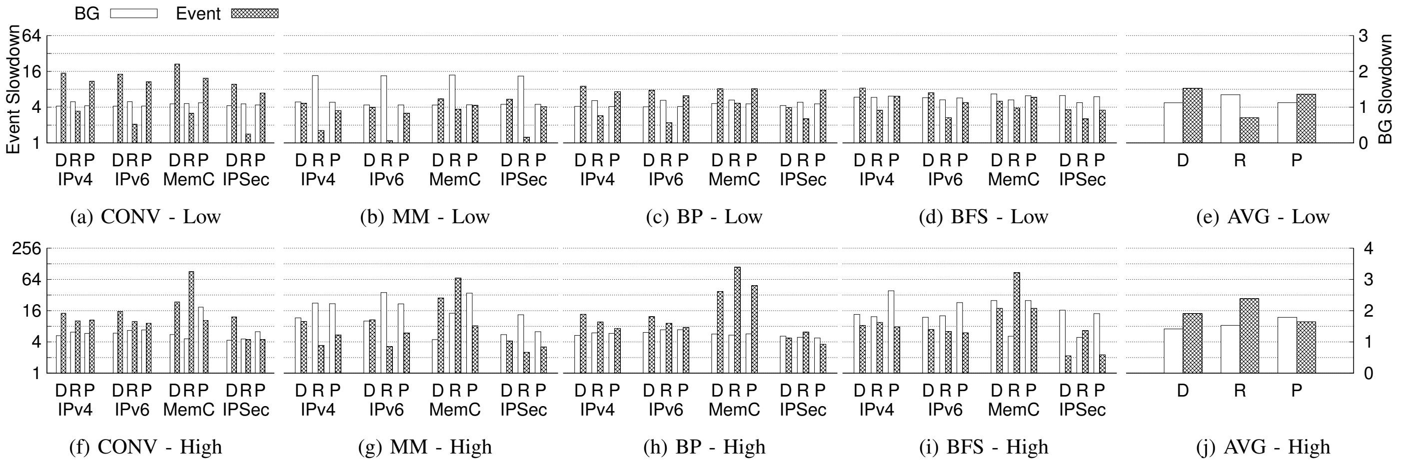
With exclusive access to the GPU, the main benefits of EDGE come from the reduced kernel launch overheads and increased independence from the CPU for task management. However, when the GPU is shared between other applications, the warp-level preemption mechanism can reduce the end-to-end latency for fine-grained streaming kernels over waiting for free resources (draining). As described in Section IV-B, the baseline GPU schedules TBs from running kernels in the KDU to the SIMT cores based on available resources and priority. Depending on the implementation and active state of any concurrently running kernels, TB draining can result in long scheduling latencies with high variability.

Figure 12 evaluates the benefits of preemption over draining for a set of warp-level (32 threads) event kernels (IPv4, IPv6, MemC, and IPSec) and background kernels (CONV, MM, BP, and BFS) that fully occupy the GPU. For each combination of event and background kernel, we launch an event kernel at random times over three runs of the background kernel and measure the total number of preemption cycles before the event kernel can begin executing with TB draining and preemption. To increase the total number of event kernel preemption measurements per background kernel, we skip the actual execution of the event kernel once it is ready to be scheduled. As described in Section II, the use of smaller kernels corresponds to smaller network packet batch sizes, which can help to lower queuing and processing latencies. Figure 5 shows that despite having low absolute values, the launch overheads of small kernels are significant relative to the kernel runtime. For example, the single warp runtime (in isolation) is approximately 2300 GPU cycles for IPv4/6 and 23000 GPU cycles for MemC and IPSec.

Warp-level preemption requires that the victim warp uses enough registers to support the event kernel (Section IV-D). However, depending on the kernel implementations, this may not always be the case. If the register usage condition is satisfied, preemption is initiated and the registers are either saved or new registers are allocated from the register file if free. This flow restricts event warps from using more registers than the victim warp, even if available in the register file. We include two background kernels, BP and BFS, which use fewer registers per thread than MemC and IPSec. Consequently, the preemption latency for these kernel combinations will be similar to draining (IPv4/6 can preempt all kernels).

As shown in Figure 12, warp-level preemption reduces the average and tail event kernel scheduling latencies over draining by 2.6× and 2.9×, respectively. When the background kernels have enough registers to be preempted by all event kernel (e.g., CONV/MM), the average and tail latencies via preemption improve significantly to 115.7× and 68.4×, respectively. In all experiments, we found that there were always enough registers available in the underutilized register file (avoids register saving/restoring), even for MemC and IPSec with BP and BFS, which were restricted from preemption.

Allowing MemC and IPSec to allocate a larger number of free registers from the underutilized register file (Section IV-D) improves the average and tail scheduling latencies by 96.4× and 53.8× over draining, respectively. Given the potential to reduce scheduling latency with preemption, it may be beneficial to consider more aggressive approaches to ensure preemption is possible. For example, EDGE could reserve a portion of the register file for event warps. Alternatively, the maximum number of registers per event kernel could be limited to ensure that it fits within a background kernel's footprint, trading off scheduling latency for lower runtime performance due to increased register spilling. Another option is to preempt more than one warp from the background kernel, such that the event kernel can use the registers from multiple contiguous warps. Evaluating techniques to improve the opportunities for



(a) CONV - Low (b) MM - Low (c) BP - Low (d) BFS - Low (e) AVG - Low

(f) CONV - High (g) MM - High (h) BP - High (i) BFS - High (j) AVG - High

Fig. 13: Slowdown of the background kernels (BG) and event kernels (Event) for Draining (D), Reservation (R), and Preemption (P). Low = 2 overlapping events. High = 16-32 overlapping events. (Event slowdown: left y-axis — BG slowdown: right y-axis).

fine-grained preemption is left for future work.

D. Event Kernels and Multiprogramming

This section evaluates a complete multiprogramming use-case with streaming event kernels and longer running background tasks. We use the same combination of background kernels and fine-grained event kernels as the previous section. This section also evaluates a static fair resource-sharing approach, referred to as *reserved*, which equally partitions SIMT cores between the event kernels and background kernels.

Figure 13 presents the performance impact when running multiple event kernels alongside a background kernel under different event rates and scheduling techniques. The performance impact is measured as the kernel slowdown in the multiprogrammed case relative to isolation. The time to schedule an event kernel is included in the total event runtime. The event kernel slowdown is shown on the left y-axis ($\log_2$) and the background kernel slowdown is shown on the right y-axis. The top row evaluates a *low* event kernel rate, where a maximum of two event kernels are concurrently run on the GPU. The bottom row evaluates a *high* event kernel rate, where a maximum of 32 IPv4/6 and 16 MemC/IPSec event kernels are concurrently run on the GPU. We double the number of concurrent IPv4/6 kernels as their execution times are relatively short. Figure 13(a)-(d) and (f)-(i) present the results for each combination of event kernel and background kernel, while (e) and (j) present the average results across all kernels.

The ideal slowdown for all kernels is one. However, scheduling overheads and contention over shared resources negatively impact performance. For the low event rate, preemption is able to improve the average event kernel performance by 1.3× over draining, while both techniques have negligible impact on the background kernel's performance due to low event utilization. The main benefits for preemption come from the reduction in scheduling latency. SIMT core reservation behaves as expected given the low utilization of half the GPU. Event kernels benefit from zero scheduling latency and reduced contention for shared resources, such as the SIMT cores, instruction/data caches, and memory bandwidth, whereas the background kernels lose half of their compute resources. However, as can be seen in Figure 13(a)-(d), the background kernel performance does not decrease linearly with the number of reserved SIMT cores. This is largely a factor of whether the application is compute-intensive (e.g., MM) or memory-intensive (e.g., CONV). Overall, reservation under low event rates improves event kernel performance and reduces background kernel performance relative to preemption by 2.3× and 1.3×, respectively.

The higher event rate places significantly more stress on the system, leading to larger performance impacts for both event and background kernels with all scheduling techniques. MemC, the more heavy-weight event kernel, incurs the largest overheads (37/109/48× for D/R/P) due to the increased interference from other MemC kernels. Such slowdowns are prohibitively high, indicating that MemC requires more exclusive access to GPU resources under high rates to maintain SLOs. Draining helps to reduce the background kernel interference on the event kernel's execution under high rates, since it blocks entire background TBs instead of warps. This can improve the runtime of the event kernel, limiting the benefits of faster scheduling. IPSec experiences the lowest slowdown of the event kernels, as it is more compute-intensive and better optimized for the GPU's SIMT architecture. Each thread in a warp works on the same packet, instead of a separate thread per packet, which reduces the potential branch and memory divergence, and hence the extent of interference from the background kernel.

Unlike the low event rate, reservation performs the worst for event kernels at high rates. This is because the event kernels are limited to the reserved SIMT cores, increasing the total number of event kernels per SIMT core. On average, we find that preemption improves event kernel performance over draining and reservation by 1.4× and 2.8×, respectively. However, the event kernel improvements come at the cost of the background kernel performance, which is decreased by 1.3× and 1.2×, respectively. While preemption only blocks the *execution* of a single background warp, it blocks the *completion* of an entire

TB waiting on the tail warp. Conversely, draining allows the background TBs to complete before beginning event execution, which maintains higher utilization for the background kernel.

As described in Section VI-C, we restrict MemC and IPSec from preempting BP and BFS due to register utilization, which results in similar performance for preemption and draining. BFS also shows lower benefits from preemption for IPv4/6. While the other background applications launch one (CONV/MM) or two (BP) kernels, BFS launches multiple kernels at each stage in the search algorithm. Due to the coarse-grained kernel launch synchronization, BFS TBs must complete before the next kernel can be launched. This results in more free resources being available on the GPU at a given time, which both preemption and draining are able to exploit to begin running immediately.

Depending on the scheduling delay and runtime slowdown, each technique is able to execute a different number of event kernels within a single background task. We also compute the ANTT [14] to measure the overall system performance in a multiprogrammed environment. As there are significantly more event kernels than the single background task, the ANTT closely follows the event kernel slowdowns. With high event rates, we find that preemption is able to improve the ANTT over draining and reservation by 1.4× and 2.7×, respectively.

E. Summary

Overall, EDGE provides three main benefits – it removes the requirement for a CPU or persistent GPU runtime to manage GPU tasks, reduces the kernel launch latency for event kernels of any size, and can improve total system throughput and end-to-end latency for fine-grained streaming kernels in a multiprogrammed environment. As highlighted in Section VI-D, different combinations of applications and event rates can have large variations in performance/fairness between the evaluated scheduling techniques. There are many possibilities and trade-offs for managing resource sharing and execution priorities. We believe that a dynamic approach would provide the greatest flexibility (e.g., dynamically scaling the amount of reserved resources based on the event rate and using preemption when free resources are unavailable). Furthermore, dynamically limiting the scheduling of background TBs/warps while event kernels are executing can significantly improve event kernel performance, since the contention for shared resources, such as the memory system, are reduced. We plan to evaluate such dynamic approaches in future work.

VII. Related Work

Multiple related works for reducing GPU kernel launch overheads, reducing the GPU's dependence on the CPU for task management, and improving GPU multiprogramming support are discussed in Section II-B and Section II-D.

Previous works have evaluated CPU/GPU PT runtimes for improving GPU efficiency and independence from the CPU driver and GPU hardware task schedulers for GPU task management and I/O [9], [10], [31], [54], [68], [69]. Most of these works evaluate TB-level PT frameworks, while Pagoda [68] presents a warp-level PT framework targeted towards fine-grained kernels. GPUrdma [10] even removes CPU interaction for performing I/O operations with an Infiniband NIC. However, these works require persistently running GPU threads, which as discussed in Section II-B, can trade off flexibility for efficiency. Others have pushed towards improving GPUs as a first-class computing resource to support efficient GPU multiprogramming [27], [29], [51], [55], [60]. These works utilize OSes, Hypervisors, CPU/GPU runtime environments, GPU compilers, and/or GPU hardware techniques to virtualize GPU resources to improve resource sharing. Our proposed event kernels can make use of these resource virtualization techniques to improve fairness in a multiprogrammed environment, while EDGE enables the direct initiation of such event kernels from external devices, removing the requirement for CPU involvement on the critical path or polling GPU runtimes.

Recent work [56] also proposes an event-driven GPU programming model consisting of GPU *callback* functions, which are invoked on events, such as file read completions. This framework, similar to GPUfs [54] and GPUnet [31], contains CPU and GPU event loops, which poll for RPC events from either device, or require re-launching kernels from the host upon RPC completion. Considering event kernels as callback functions, EDGE can remove the requirement for continuously running GPU event loops, while minimizing the impact on performance required to re-launch full kernels.

VIII. Conclusion

This paper improves GPU support for fine-grained, latency-sensitive streaming applications in the datacenter, such as networking. We propose an event-driven GPU execution model, API, and set of hardware extensions (EDGE) that reduce the GPU's dependence on a CPU and enable external devices to directly launch pre-configured tasks on a GPU. We estimate that EDGE can reduce the kernel launch overheads by 4.4× compared to the baseline CPU-launched approach. We also propose a warp-level preemption mechanism to reduce the scheduling latency for fine-grained, streaming GPU tasks. EDGE includes multiple optimizations that reduce the average warp preemption latency by 35.9× over waiting for a preempted warp to naturally flush the pipeline. For a set of warp-level network event kernels and concurrently running traditional GPU kernels, we find that warp-level preemption is able to reduce the average/tail scheduling latencies and ANTT over waiting for resources to become available by 2.6×, 2.9×, and 1.4×, respectively. When preemption is always possible, the average and tail scheduling latencies improve to 115.7× and 68.4× over draining, respectively.

IX. Acknowledgements

The authors would like to thank the reviewers for their insightful feedback. The authors would also like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) and The University of British Columbia for funding this research.

REFERENCES

[1] I. Advanced Micro Devices, "Hardware to play rocm," 2018. [Online]. Available: https://rocm.github.io/hardware.html

[2] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck, "Rhythm: Harnessing data parallel hardware for server workloads," in *ACM SIGPLAN Notices*, vol. 49, no. 4. ACM, 2014, pp. 19–34.

[3] B. Aker, "libMemcached," http://libmemcached.org/libMemcached.html.

[4] Amazon, "Amazon ec2 p3 instances - accelerate machine learning and high performance computing applications with powerful gpus," 2019. [Online]. Available: https://aws.amazon.com/ec2/instance-types/p3/

[5] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[6] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 94–103.

[7] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[8] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[9] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "Effisha: A software framework for enabling effficient preemptive scheduling of gpu," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2017, pp. 3–16.

[10] F. Daoud, A. Watad, and M. Silberstein, "Gpurdma: Gpu-side library for high performance networking from gpu kernels," in *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2016, p. 6.

[11] J. Dean, "Large scale deep learning," Keynote GPU Technical Conference 2015, 03 2015. [Online]. Available: http://www.ustream.tv/recorded/60071572

[12] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[13] T. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*. IEEE, 1992, pp. 256–266.

[14] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE micro*, vol. 28, no. 3, 2008.

[15] P. FIPS, "46-3. data encryption standard (des)," *National Institute of Standards and Technology*, vol. 25, no. 10, pp. 1–22, 1999.

[16] M. Flajslik and M. Rosenblum, "Network interface design for low latency request-response protocols," in *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, 2013, pp. 333–346.

[17] H. Foundation, "Hsa runtime programmer's reference manual v1.1.1," 2016. [Online]. Available: http://www.hsafoundation.com/standards/

[18] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 407–420.

[19] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "Apunet: Revitalizing gpu as packet processing accelerator." in *NSDI*, 2017, pp. 83–96.

[20] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–14.

[21] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4. ACM, 2010, pp. 195–206.

[22] T. H. Hetherington, M. O'Connor, and T. M. Aamodt, "Memcachedgpu: Scaling-up scale-out key-value stores," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 43–57. [Online]. Available: http://doi.acm.org/10.1145/2806777.2806836

[23] Y. Hu and T. Li, "Enabling efficient network service function chain deployment on heterogeneous server platform," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 27–39.

[24] P. Jain, X. Mo, A. Jain, H. Subbaraj, R. S. Durrani, A. Tumanov, J. Gonzalez, and I. Stoica, "Dynamic Space-Time Scheduling for GPU Inference," *arXiv preprint arXiv:1901.00041*, 2018.

[25] S. Jones, P. A. Cuadra, D. E. Wexler, I. Llamas, L. V. Shah, J. F. Duluk Jr, and C. Lamb, "Technique for computational nested parallelism," Dec. 6 2016, uS Patent 9,513,975.

[26] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using gpus in software packet processing." in *NSDI*, 2015, pp. 409–423.

[27] S. Kato, S. Brandt, Y. Ishikawa, and R. Rajkumar, "Operating systems challenges for gpu resource management," in *Proc. of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2011, pp. 23–32.

[28] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "Rgem: A responsive gpgpu execution model for runtime engines," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. IEEE, 2011, pp. 57–66.

[29] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, "Gdev: First-class gpu resource management in the operating system." in *USENIX Annual Technical Conference*. Boston, MA;, 2012, pp. 401–412.

[30] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon, "Nba (network balancing act): A high-performance packet processing framework for heterogeneous processors," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. ACM, 2015.

[31] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein, "Gpunet: Networking abstractions for gpu programs." in *OSDI*, vol. 14, 2014, pp. 6–8.

[32] A. Krizhevsky, "Cuda-convnet," https://github.com/dnouri/cuda-convnet, 2015.

[33] M. LeBeane, B. Potter, A. Pan, A. Dutu, V. Agarwala, W. Lee, D. Majeti, B. Ghimire, E. Van Tassell, S. Wasmundt *et al.*, "Extended task queuing: active messages for heterogeneous systems," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 933–944.

[34] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[35] K. Lee and Facebook, "Introducing big basin: Our next-generation ai hardware," https://code.facebook.com/posts/1835166200089399/introducing-big-basin-our-next-generation-ai-hardware/, 2017.

[36] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 137–152.

[37] G. Memik, W. H. Mangione-Smith, and W. Hu, "Netbench: a benchmarking suite for network processors," in *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, 2001, pp. 39–42.

[38] J. Menon, M. De Kruijf, and K. Sankaralingam, "igpu: exception support and speculative execution on gpus," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 72–83.

[39] Microsoft, "Gpu optimized virtual machine sizes," 2018. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-gpu

[40] D. J. Miller, P. M. Watts, and A. W. Moore, "Motivating future interconnects: a differential measurement analysis of pci latency," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2009, pp. 94–103.

[41] G. E. Moore, "Cramming more components onto integrated circuits, electronics,(38) 8," 1965.

[42] NVIDIA, "Nvidia's next generation cuda compute architecture: Fermi," 2009. [Online]. Available: https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[43] ——, "Nvidia tesla p100," 2016. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[44] NVIDIA, "Cuda c programming guide," 2017. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_Dynamic_Parallelism_Programming_Guide.pdf

[45] NVIDIA, "Gpudirect," 2017. [Online]. Available: https://developer.nvidia.com/gpudirect

[46] NVIDIA, "Nvidia management library (nvml)," https://developer.nvidia.com/nvidia-management-library-nvml, 2017.

[47] NVIDIA, "Nvidia tesla v100 gpu architecture," 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[48] Oracle, "Using nvidia gpu cloud with oracle cloud infrastructure," 2019. [Online]. Available: https://docs.cloud.oracle.com/iaas/Content/Compute/References/ngcimage.htm

[49] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared gpu," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 593–606, 2015.

[50] J. Power, J. Hestness, M. Orr, M. Hill, and D. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *Computer Architecture Letters*, vol. 13, no. 1, Jan 2014. [Online]. Available: http://gem5-gpu.cs.wisc.edu

[51] 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*. ACM, 2011, pp. 233–248.

[52] D. Rossetti, "Gpudirect: integrating the gpu with a network interface," in *GPU Technology Conference*, 2015.

[53] L.-W. Shieh, K.-C. Chen, H.-C. Fu, P.-H. Wang, and C.-L. Yang, "Enabling fast preemption via dual-kernel support on gpus," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 2017, pp. 121–126.

[54] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "Gpufs: integrating a file system with gpus," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 485–498.

[55] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "Gpuvm: Gpu virtualization at the hypervisor," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2752–2766, 2016.

[56] Y. Suzuki, H. Yamada, S. Kato, and K. Kono, "Towards multi-tenant gpgpu: Event-driven programming model for system-wide scheduling on shared gpus," in *Proceedings of the Workshop on Multicore and Rack-scale Systems*, 2016.

[57] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 193–204.

[58] I. Tanasic, I. Gelado, M. Jorda, E. Ayguade, and N. Navarro, "Efficient exception handling support for gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 109–122.

[59] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "Gaspp: A gpu-accelerated stateful packet processing framework." in *USENIX Annual Technical Conference*, 2014, pp. 321–332.

[60] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A holistic approach to resource virtualization in gpus," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–14.

[61] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. Le Moal, T. Bunker, J. Xu, S. Swanson *et al.*, "{DC} express: Shortest latency protocol for reading phase change memory over {PCI} express," in *Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, 2014, pp. 309–315.

[62] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 528–540, 2016.

[63] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured gpu applications," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 51–60.

[64] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 358–369.

[65] B. Wu, X. Liu, X. Zhou, and C. Jiang, "Flep: Enabling flexible and efficient preemption on gpus," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 483–496.

[66] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, "Kernel weaver: Automatically fusing database primitives for efficient gpu computation," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 107–118.

[67] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 230–242.

[68] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: Fine-grained gpu resource virtualization for narrow tasks," in *ACM SIGPLAN Notices*, vol. 52, no. 8. ACM, 2017, pp. 221–234.

[69] L. Zeno, A. Mendelson, and M. Silberstein, "Gpupio: The case for i/o-driven preemption on gpus," in *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. ACM, 2016, pp. 63–71.

[70] Z. Zheng, J. Bi, H. Wang, C. Sun, H. Yu, H. Hu, K. Gao, and J. Wu, "Grus: Enabling latency slos for gpu-accelerated nfv systems," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018, pp. 154–164.

APPENDIX
ARTIFACT APPENDIX

A. Abstract

EDGE is built on top of Gem5-gpu (https://gem5-gpu.cs.wisc.edu/). We evaluated EDGE on Ubuntu 14.04 running on a cluster of machines, however, there are no specific software or hardware requirements. This section provides detailed instructions on how to compile EDGE, as well as scripts for running the experiments and collecting the data. We mainly target Figure 12 and Figure 13, as they contain the main results in this paper. Also, Figure 1 is based on the data in Figure 12, averaged across all background tasks and event kernels used in our evaluation. In addition to the setup instructions, we also provide a VirtualBox VM with all dependencies installed and code compiled to run the experiments.

B. Artifact Check-List (Meta-Information)

- **Program:** Mixed, provided.
- **Compilation:** gcc4.7 + gcc4.4 + cuda3.2
- **Run-time environment:** Ubuntu 14.04 + gem5-gpu dependencies (provided).
- **Metrics:** Latency (min, max, average).
- **Output:** Provided scripts.
- **Experiments:** Provided scripts.
- **How much disk space required (approximately)?:** The main directory requires 5G. Extra storage is also required for installing dependencies. The size of the provided VM is 12G.
- **How much time is needed to prepare workflow (approximately)?:** Less than 1 hour.
- **How much time is needed to complete experiments (approximately)?:** Less than 1 day on a cluster of machines. We did not attempt to run all of the experiments on a single machine, but we estimate it would take about a week.
- **Publicly available?:** Yes.
- **Licenses (if publicly available)?:** BSD-3 (Gem5, GPGPU-Sim, Rodinia, MemC benchmark, CONV benchmarks), Apache License Version 2.0 (IPv4/6 forwarding benchmark), NVIDIA-SDK license (MM benchmark)
- **Archived (provide DOI)?:** https://zenodo.org/badge/197230707.svg

C. Description

1) How delivered: We have created GitHub repository for EDGE, as well as ready-to-use VirtualBox VM with everything installed and compiled. The username for VM is "maria" and the password is "asdfqwer". The EDGE GitHub repository can be found here: https://github.com/marialubeznov/event_driven_gpu_execution. The EDGE VM can be downloaded here: http://www.ece.ubc.ca/~mlubeznov/edge.

2) Hardware Dependencies: In general, there are no strict hardware dependencies. However, each experiment in this paper may take over an hour to run on our modified version of Gem5-gpu, and the complete paper data includes over one hundred experiments. Consequently, we recommend using a cluster of machines to run all of the experiments in EDGE. We have provided run scripts for both a single machine and a cluster of machines (using the PBS job scheduler).

3) Software Dependencies: Our artifact has been tested on Ubuntu 14.04. It does not require root access, but it has some dependencies that need to be installed (listed below). These dependencies are essentially a union of Gem5 and GPGPU-Sim dependencies. Also, Gem5-gpu requires gcc4.7 to build the simulator and gcc4.4 to build the benchmarks.

- NVIDIA CUDA 3.2
- gcc 4.4 for benchmark/edge directory
- gcc 4.7 for Gem5 and benchmark/libcuda directories
- libnuma
- python 2.7
- SCons any recent version.
- zlib any recent version. Need the "zlib-dev" or "zlib1g-dev" package to get the zlib.h header file as well as the library itself.
- swig
- makedepend
- bison
- flex

4) Data Sets: All of the benchmarks used in this paper are included in the GitHub repository and VM. Most of the benchmarks are taken from open source projects and others are prepared by us. Where appropriate, the data sets provided with the benchmarks are used.

- Rodinia [8].
- Cuda-convnet [32]: https://github.com/dnouri/cuda-convnet (Convolution kernels on random input data).
- MemcachedGPU [22]: https://github.com/tayler-hetherington/MemcachedGPU (GET request kernel on request traces generated by Memaslap [3]).
- IPv4/6 Forwarding [26]: Unique IPv4 and IPv6 prefixes from RouteViews (provided with the benchmark).
- IPSec [68]: Network packets of varied sizes generated by NetBench [37] (provided with the benchmark).

D. Installation

First install the dependencies listed above.

```
# Clone git repo
sudo apt-get install git
git clone https://github.com/marialubeznov/
    event_driven_gpu_execution.git
#Install gcc versions
sudo apt-get install build-essential
sudo cat "deb␣http://dk.archive.ubuntu.com/ubuntu/␣
    trusty␣main␣universe" >> /etc/apt/sources.list
sudo apt-get update
sudo apt-get install g++-4.4
sudo apt-get install g++-4.7
sudo update-alternatives --remove-all gcc
sudo update-alternatives --install /usr/bin/gcc gcc
    /usr/bin/gcc-4.4 10
sudo update-alternatives --install /usr/bin/gcc gcc
    /usr/bin/gcc-4.7 20
sudo update-alternatives --install /usr/bin/g++ g++
    /usr/bin/g++-4.4 10
sudo update-alternatives --install /usr/bin/g++ g++
    /usr/bin/g++-4.7 20
#Update gcc version
sudo update-alternatives --config gcc
sudo update-alternatives --config g++
#Install dependencies:
sudo apt-get install python-dev
sudo apt-get install scons
sudo apt-get install zlib1g-dev
sudo apt-get install swig
sudo apt-get install xutils-dev
sudo apt-get install flex bison
sudo apt-get install libnuma-dev
#Install cuda 3.2
wget http://developer.download.nvidia.com/compute/
    cuda/3_2_prod/toolkit/cudatoolkit_3.2.16
    _linux_64_ubuntu10.04.run
wget http://developer.download.nvidia.com/compute/
    cuda/3_2_prod/sdk/gpucomputingsdk_3.2.16_linux.
    run
chmod +x cudatoolkit_3.2.16_linux_64_ubuntu10.04.run
     gpucomputingsdk_3.2.16_linux.run
```

```
./cudatoolkit_3.2.16_linux_64_ubuntu10.04.run
./gpucomputingsdk_3.2.16_linux.run
cd <sdk_install_path>/NVIDIA_GPU_Computing_SDK/C/
    common
make
cd <edge_benchmark_install_path>/
    event_driven_gpu_execution/benchmarks/edge/
ln -s ../common
```

Update `$LOCAL_GEM5_PATH/set_env` with relevant paths:

- `LOCAL_GEM5_PATH` path to the EDGE git clone directory.
- `ZLIB_PATH` path to directory containing libz.so (if not default).
- `CUDAHOME, CUDA_HOME`
- `NVIDIA_CUDA_SDK, NVIDIA_CUDA_SDK_LOCATION`

Compile Gem5-gpu and the benchmarks.

```
cd $LOCAL_GEM5_PATH
source set_env
cd benchmarks/libcuda
#set gcc version to 4.7
make
cd $LOCAL_GEM5_PATH/gem5
#set gcc version to 4.7
./build_command
cd $LOCAL_GEM5_PATH/benchmarks/edge/<name>/
#set gcc version to 4.4
make
```

E. Evaluation and Expected Results

Running the experiments required for Figure 12 and Figure 13 can be done using the following set of scripts (present in all 4 benchmark directories).

Single machine:

```
run_no_overlap.sh
run_low_util.sh
run_high_util.sh
```

Cluster:

```
run_<benchmark_name>_no_overlap.pbs
run_<benchmark_name>_low_util.pbs
run_<benchmark_name>_high_util.pbs
run_cluster_no_overlap.sh
run_cluster_low_util.sh
run_cluster_high_util.sh
```

The results presented in Figure 12 may not exactly match the absolute values reported in the paper, since for each combination of event kernel and background kernel, we launch the event kernels at random times and average over three runs of the corresponding background kernel. However, the trend of the results should closely match the results presented in the paper. The results presented in Figure 13 can be replicated.

After completing all the experiments above, the data can be collected using the following scripts:

```
#Figure 12
$LOCAL_GEM5_PATH/benchmarks/common/GetDataFig12.sh <
    background task>  > <background task>_no_overlap
bash $LOCAL_GEM5_PATH/benchmarks/common/
    process_data_fig12.sh
#Figure 13
$LOCAL_GEM5_PATH/benchmarks/common/GetDataFig13.sh <
    background task> <type>  > <background_task>_<
    type>
bash $LOCAL_GEM5_PATH/benchmarks/common/
    process_data_fig13.sh
```

Where `type = {low_util, high_util}`

The *type* keywords represent the low and high event kernel rate experiments, respectively. The background task is provided as an index corresponding to the mapping in Table III.

TABLE III: Benchmark index mapping.

0	1	2	3
Convolution	Matrix multiply	Backprop (rodinia)	BFS (rodinia)

The main results from these experiments are reported as a latency or runtime metric. As multiple event kernels are launched during each experiment, we report three latency metrics (minimum, maximum, and average) and the total number of launched event kernels. The runtime slowdowns are computed using the average kernel runtimes when an event kernel is run concurrently with a background kernel and the kernel runtime in isolation.

F. Experiment Customization

The GPU configuration can be configured in:

```
$LOCAL_GEM5_PATH/gem5gpu/configs/gpu_config/gpgpusim
    .fermi.config.template
```

The command format for launching the benchmarks directly is:

```
$LOCAL_GEM5_PATH/gem5/build/X86_VI_hammer_GPU/gem5.
    opt $LOCAL_GEM5_PATH/gem5-gpu/configs/se_fusion.
    py -c $LOCAL_GEM5_PATH/benchmarks/edge/<
    benchmark_name>/<benchmark_exe_name> -o '<
    benchmark_options>'
```