AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads

Nathan Otterness
The University of North Carolina at Chapel Hill, USA
otternes@cs.unc.edu

James H. Anderson
The University of North Carolina at Chapel Hill, USA
anderson@cs.unc.edu

Abstract

Graphics processing units (GPUs) manufactured by NVIDIA continue to dominate many fields of research, including real-time GPU-management. NVIDIA's status as a key enabling technology for deep learning and image processing makes this unsurprising, especially when combined with the company's push into embedded, safety-critical domains like autonomous driving. NVIDIA's primary competitor, AMD, has received comparatively little attention, due in part to few embedded offerings and a lack of support from popular deep-learning toolkits. Recently, however, AMD's ROCm (Radeon Open Compute) software platform was made available to address at least the second of these two issues, but is ROCm worth the attention of safety-critical software developers? In order to answer this question, this paper explores the features and pitfalls of AMD GPUs, focusing on contrasting details with NVIDIA's GPU hardware and software. We argue that an open software stack such as ROCm may be able to provide much-needed flexibility and reproducibility in the context of real-time GPU research, where new algorithmic or analysis techniques should typically remain agnostic to the underlying GPU architecture. In support of this claim, we summarize how closed-source platforms have obstructed prior research using NVIDIA GPUs, and then demonstrate that AMD may be a viable alternative by modifying components of the ROCm software stack to implement spatial partitioning. Finally, we present a case study using the PyTorch deep-learning framework that demonstrates the impact such modifications can have on complex real-world software.

2012 ACM Subject Classification Computer systems organization → Heterogeneous (hybrid) systems; Computer systems organization → Real-time systems; Software and its engineering → Scheduling; Software and its engineering → Concurrency control; Computing methodologies → Graphics processors; Computing methodologies → Concurrent computing methodologies

Keywords and phrases real-time systems, graphics processing units, parallel computing

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.12

Funding Work was supported by NSF grants CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, ONR grant N00014-20-1-2698, and funding from General Motors.

1 Introduction

While the battle among chip manufacturers for dominance in the GPU (graphics processing unit) market has raged for years, the most prominent contenders for at least the past decade have been two companies: NVIDIA and AMD. Of these, NVIDIA is currently the clear leader, boasting nearly 77% of the market share for discrete desktop GPUs [37] at the time of writing.

Unfortunately, the new reality of autonomous vehicles means that artificial-intelligence and image-processing accelerators, including GPUs, cannot be relegated to the domains of datacenters and video games. When human lives directly depend on GPUs, software developers should seek the safest option irrespective of brand loyalty or sales numbers. In

© Nathan Otterness and James H. Anderson;
licensed under Creative Commons License CC-BY

short, the real-time GPU research community should not ignore either side in the NVIDIA-vs.-AMD battle.

Much of NVIDIA's dominance in the area of general-purpose GPU computing can be attributed to the popularity of CUDA, NVIDIA's proprietary GPU programming framework [32]. On top of CUDA itself, NVIDIA has heavily supported and encouraged the use of its GPUs in the area of artificial intelligence, as exemplified by its popular CUDA library supporting deep neural networks, cuDNN [33]. cuDNN was first released in 2007, and has since been adopted into many prominent deep-learning and image-processing frameworks such as Caffe [25], PyTorch [4], and TensorFlow [1].

In contrast, these popular deep-learning frameworks lacked AMD support for years, causing AMD to fall behind in research adoption. However, AMD has recently been improving its own response to CUDA with the ROCm software stack [12], which includes HIP, a GPU programming interface sporting near-identical features to CUDA. The introduction of HIP has allowed AMD to quickly develop versions of the aforementioned deep-learning and AI frameworks compatible with its own GPUs. For example, as of May 2018, the official PyTorch repository has shipped with scripts that allow it to be compiled for AMD GPUs, using ROCm.[1] Similarly, AMD maintains a fork of TensorFlow compatible with its own GPUs,[2] despite a lack of upstream support. Despite AMD's efforts, however, the dearth of research directed at AMD GPUs could lead one to question whether a serious competitor to NVIDIA even exists. In this paper we wish to rectify this lack of research by focusing on AMD GPUs, and how their features may be used to improve timing safety in a real-time setting.

Given that NVIDIA GPUs are well-established and better-studied, what do AMD GPUs have to offer real-time programmers? The answer requires considering one of the most important research questions pertaining to real-time GPU usage: *how can multiple applications safely and efficiently share a GPU?* Size, weight, and power concerns are already leading to increased centralization in embedded processors for autonomous systems, meaning that a wider range of devices and functions are managed by a smaller number of processors. Following this trend, future embedded applications will likely need to share a small number of low-power GPUs. Doing so inevitably increases hardware contention, which in turn, if not managed carefully, can lead to unsafe timing or extreme pessimism in analysis.

We claim that some features currently unique to AMD GPUs, especially *an open-source software stack* and *support for hardware partitioning*, have the potential to accelerate and aid the long-term viability of real-time GPU research. Of course AMD GPUs also have their own set of drawbacks, some of which may justifiably warrant continued research on NVIDIA GPUs regardless of the availability of an alternative. So, in this paper we also examine some drawbacks of AMD GPUs, namely, fewer supported platforms, less documentation, and instability stemming from less-mature software. To summarize: a greater variety of research platforms can only serve to help advance the field, even if some work is not possible on every alternative. After all, we do not seek to declare victory for one GPU manufacturer or the other, but to further the cause of developing safer GPU-accelerated software.

Contributions. In this paper, our primary goal is to shed light on an under-explored platform in real-time research: AMD GPUs. In doing so, we identify potential benefits and drawbacks of AMD GPUs, along with their implications for real-time GPU management. We especially contrast features of AMD GPUs with those offered by NVIDIA in the context of how different possible approaches could reshape real-time GPU research. Finally, we

---

[1] https://github.com/pytorch/pytorch/commit/cd86d4c5548c15e0bc9773565fa4fad73569f948

[2] https://github.com/ROCmSoftwarePlatform/tensorflow-upstream

describe our implementation of several modifications to AMD's software stack to expose a basic hardware-partitioning API, which we apply to improve timing predictability when multiple PyTorch neural networks share a single GPU.

Organization. The rest of the paper is organized as follows. Sec. 2 describes background and relevant work on real-time GPU management. Next, Sec. 3 further contrasts features of NVIDIA and AMD GPUs, and introduces several benefits and drawbacks of AMD's platform. Sec. 4 describes our case studies, including our modifications to the ROCm framework and PyTorch. Finally, Sec. 5 contains our concluding remarks and plans for future work.

2 Background and Related Work

We begin by providing essential information about GPU hardware, software, and prior research.

2.1 Overview of GPU Programming

Despite their different vendors, general-purpose programming on both NVIDIA and AMD GPUs requires following similar steps, so this section attempts to give a general overview while remaining vendor-agnostic, leaving a detailed comparison for Sec. 3.

Programs must carry out the following steps to run code on a GPU:

1. Copy input data from CPU memory to GPU memory.
2. Invoke a GPU kernel to process the data. (In the context of GPU programming, the term *kernel* typically refers to a section of code that runs on a GPU.) The program must also specify the number of parallel GPU threads to use.
3. Copy output data from GPU memory to CPU memory.

All three of these steps (including other details such as allocating and freeing GPU memory buffers) are carried out at the request of a CPU program, via a GPU-programming API. Typically, the program issues commands to one or more API-managed queues, called *streams*, after which the underlying runtime will execute the commands in FIFO order. On an NVIDIA-based system, the API of choice is typically CUDA, with HIP being a close analogue for AMD GPUs. We draw special attention to the API, because in a GPU-augmented safety-critical system, *the GPU API is safety-critical software.*

GPU hardware organization. Fig. 1 shows the high-level organization of the AMD RX 570 GPU, used as a test platform in this paper. Even though Fig. 1 shows an AMD GPU, NVIDIA GPUs exhibit a similar high-level layout, albeit with some different terminology.

A *discrete* GPU, such as the AMD RX 570, consists of self-contained DRAM and processors, and typically communicates with a host system using the PCI bus. This is in contrast to an *integrated* GPU, which shares components, possibly including memory, with the CPU. Both NVIDIA and AMD GPUs contain a shared L2 cache in addition to separate L1 caches assigned to independent compute units (CUs).

A GPU's parallel-computing capability comes from its collection of CUs (known as *streaming multiprocessors*, or SMs, in NVIDIA GPUs). Each CU is responsible for a subset of parallel threads associated with a kernel. In reality, a GPU's SIMT (single-instruction-multiple-thread) architecture means that GPU threads are organized into groups, where each thread in a group executes the same instruction at a given time on a different piece of data. This SIMT architecture simplifies GPU programming because it allows programmers to write GPU-accelerated programs using roughly the same logic as a traditional multithreaded application. This comes at the expense of potential inefficiencies in cases where GPU kernels

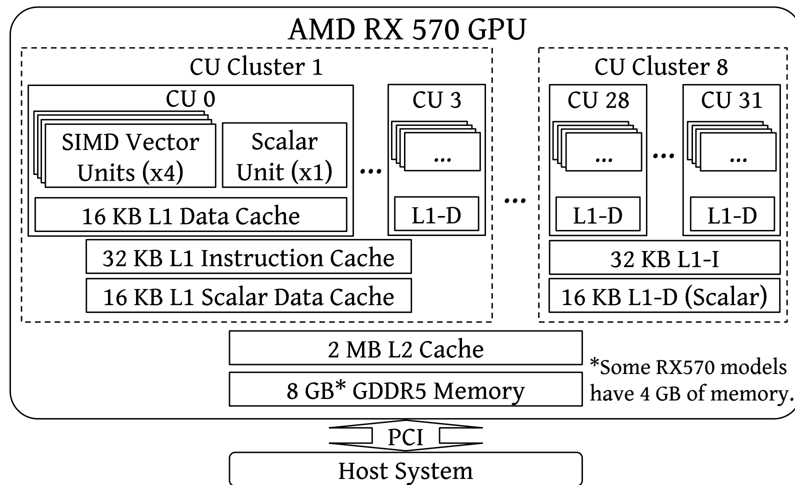

Figure 1 Organization of the AMD RX 570 GPU.

contain code with a heavily divergent control flow, but avoiding inefficient, branchy code is typically left as a responsibility of the programmer.

In addition to the memory and compute units described above, GPUs contain many other necessary pieces of hardware that are not represented in Fig. 1, including dedicated memory copy engines responsible for transferring data to and from CPU memory, and significant portions of hardware exclusively used for rendering graphics.

2.2 Related Work

This section gives an overview of prior research in real-time GPU management. We only give a brief overview of relevant work and topics here. We save further details concerning how several of these works implement real-time GPU management for Sec. 3, after we have described relevant GPU features in more detail.

Early real-time GPU-management systems focused on scheduling jobs on entire GPUs. Some of the earliest publications apply traditional non-preemptive real-time scheduling techniques to GPU computations [27, 28]. Subsequent works include a body of papers by Verner *et al.*, that assign tasks to GPUs in ways that improve worst-case response times [38, 39, 40]. An alternative approach, by Elliott *et al.*, treats GPU access as a resource-locking problem [17].

Other prior real-time GPU work seeks to reduce the impact of non-preemptivity on GPU computations, and does so by subdividing larger kernels and data transfers into chunks that individually require less time to complete, reducing worst-case blocking. This includes some work from the prior paragraph [27], in addition to several other papers [5, 30, 43].

Recent research has shifted from scheduling work on multi-GPU systems to *GPU sharing*, necessitated by the increasing prominence of embedded GPUs. Many of the works discussed in the previous paragraphs view GPUs as a black box, which leads to an obvious difficulty: *timing depends on unknown intra-GPU scheduling behavior*. This led those early works to avoid GPU sharing altogether, overriding the decision-making abilities of the “black box” hardware and software by only allowing it to schedule one job per GPU at a time.

There have been a few ways in which real-time GPU research has attempted to enable concurrent GPU sharing. In an attempt to provide a behavioral model for GPU-sharing

task systems, our group has used black-box experiments to infer additional details about the scheduling behavior of NVIDIA GPUs [3, 34]. In 2018, Jain *et al.* presented a work that enables partitioning L2 caches, DRAM, and GPU compute units in order to reduce interference during GPU sharing [24]. This was preceded by other approaches that partition compute units only [36].

Some real-time GPU-sharing research focuses on improving predictable access to memory on platforms with integrated GPUs where, as mentioned in Sec. 2.1, memory is shared between the CPU and GPU. These works propose techniques including scheduling-based approaches to isolate memory-sensitive GPU work [7, 18], and throttling-based techniques to limit worst-case memory contention [2, 22].

Finally, some researchers have opted to collaborate with NVIDIA in order to obtain access to internal closed-source code and documentation. One such effort, by Capodieci *et al.*, implements earliest-deadline-first (EDF) scheduling on an embedded NVIDIA GPU [6].

Prior real-time research using non-NVIDIA GPUs. Even though some of the concepts from prior research evaluated for NVIDIA GPUs would be applicable on other systems, very little real-time GPU research has specifically targeted non-NVIDIA systems. The only recent real-time GPU paper explicitly implemented using a non-NVIDIA system is by Lee *et al.*, and implements transactional GPU kernel support for a Samsung Exynos system-on-chip [31].

In fairness, and as mentioned in Sec. 1, AMD continues to lag behind NVIDIA in broader research adoption. Additionally, AMD's open-source ROCm software stack was first released in 2016 [12], and the `amdgpu` open-source Linux driver was released only slightly earlier. This has left relatively little time for a body of research to be established. In any case, to the best of our knowledge, this paper is the first that considers real-time GPU computing specifically using AMD GPUs.

3 Comparison between NVIDIA and AMD

We now provide further details regarding AMD GPUs. We assume that most readers will be more familiar with NVIDIA GPUs, so we often explain the behavior of AMD GPUs in contrast to those from NVIDIA. This section serves several purposes: **1)** familiarize readers with AMD GPU architectures and software, **2)** contrast AMD GPUs with NVIDIA GPUs, and **3)** explain the potential impact of various GPU features on real-time systems, especially in light of prior NVIDIA-centric research.

3.1 Software for GPGPU Programming

As mentioned earlier, safety-critical GPU software includes not only user-level applications, but also all of the underlying library and driver code. In the following paragraphs, we focus on the software used for general-purpose GPU (GPGPU) programming, how real-time work focusing on NVIDIA's CUDA framework is implemented, and how approaches can change with more openness.

The NVIDIA GPU software stack. Even though other GPU-programming frameworks may offer performance benefits [8], the CUDA toolkit remains the best-studied and most popular method for general-purpose programming on NVIDIA GPUs. In brief, *CUDA* collectively refers to the CUDA compiler, API, and runtime library, which in turn relies on the GPU driver. Due to the opaque nature of all parts of a CUDA program below the API level, a typical user has no ability to directly modify any components below the user-level program. Additionally, even documented portions of the CUDA API often lack important details or contain inaccurate information about timing [41].

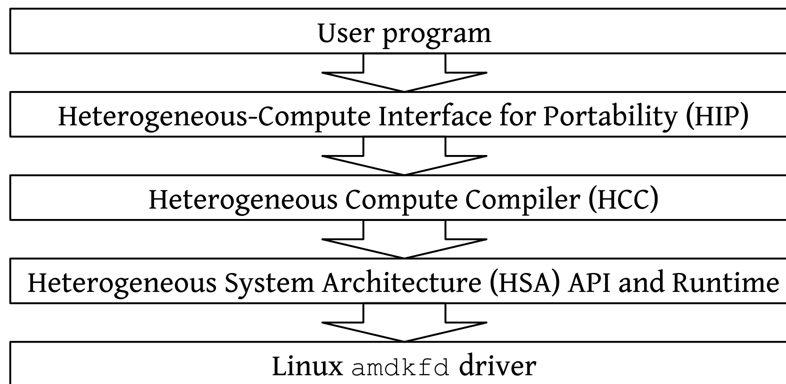

Figure 2 The ROCm software stack used for AMD GPGPU programming.

The AMD GPU software stack. AMD's ROCm (Radeon Open Compute) platform is largely analogous to the CUDA platform on NVIDIA systems. While the CUDA framework can be considered fairly monolithic (at least to its end users), ROCm is openly composed of several different modules and can support multiple programming languages [12]. However, the most important difference for the sake of real-time work is undoutedly the following:

▶ **Benefit 1.** *The ROCm software stack for computations on AMD GPUs is open source.*

Benefit 1 applies to every aspect of the ROCm stack used to program AMD GPUs, shown in Fig. 2. At the bottom of the stack is the `amdkfd` driver, which was added to the mainline Linux kernel in 2014 [29]. Above the driver lies ROCm's implementation of the Heterogeneous System Architecture (HSA) API, which was designed as a lightweight user-level API wrapping low-level functionality like memory-management, synchronization routines, kernel queues, *etc.*[20]. The next "layer" of the ROCm stack is the Heterogeneous Compute Compiler (HCC). HCC is a LLVM-based compiler capable of compiling code to target AMD GPUs, and uses the HSA API behind the scenes.

Finally, HIP (Heterogeneous-Compute Interface for Portability) is a programming language specifically designed to simplify porting CUDA applications to AMD GPUs. Technically, the HIP language is capable of targeting both AMD and NVIDIA GPUs: when targeting AMD the features of HIP are ultimately implemented on top of the HCC compiler, and when targeting NVIDIA it uses the CUDA compiler and a set of lightweight wrapper functions around the CUDA API. HIP's platform independence is facilitated by the fact that HIP, as a programming language, is nearly identical to CUDA. Converting CUDA to HIP code is only slightly more complicated than a find-and-replace of the term "`cuda`" with "`hip`" in source code. In fact, AMD provides a tool, `hipify`, that attempts to automatically transform CUDA code to HIP.

▶ **Benefit 2.** *Existing CUDA code can often be automatically converted to the cross-platform HIP programming language.*

As a concrete illustration of Benefit 2, Figs. 3 and 4 show the same kernel and CPU code fragment written in HIP and CUDA, respectively. In this simple case, the `VectorAdd` kernel code required no changes at all, with the only changes being the syntax for invoking the kernel and the `...DeviceSynchronize` function, which blocks until the kernel completes. For the sake of space, Figs. 3 and 4 omit `#include` directives and code for memory management

```
// Define a kernel setting vector c = a + b
__global__ void VectorAdd(int *a, int *b, int *c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
int main() {
  // ... Allocate vectors a, b, and c
  // Launch the kernel
  hipLaunchKernelGGL(VectorAdd, block_count, thread_count,
    0, 0, a, b, c);
  // Wait for the kernel to complete
  hipDeviceSynchronize();
  // Copy results from GPU, cleanup, etc.
}
```

Figure 3 Definition and launch of a vector-add kernel using the HIP programming language.

```
// Define a kernel setting vector c = a + b
__global__ void VectorAdd(int *a, int *b, int *c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
int main() {
  // ... Allocate vectors a, b, and c
  // Launch the kernel
  VectorAdd<<<block_count, thread_count>>>(a, b, c);
  // Wait for the kernel to complete
  cudaDeviceSynchronize();
  // Copy results from GPU, cleanup, etc.
}
```

Figure 4 Definition and launch of a vector-add kernel using the CUDA programming language.

or error checking, but all such changes are similar to the those already represented in the figures.

Of course, notable exceptions exist to the automatic conversion, such as GPU-architecture-specific inline assembly, which tends to be used for manual optimizations or accessing model-specific GPU registers.

The importance of openness. The appeal of a software stack like ROCm hinges on the answer to one question: why should developers care that ROCm is open source? Fortunately, the potential benefit of an open-source stack on real-time GPU research is easy to demonstrate. To see why, one only needs to revisit how prior real-time GPU management work was implemented.

TimeGraph [28] and *GPES* [43] require using the third-party open-source “Nouveau” driver for NVIDIA GPUs, which does not support CUDA. *RGEM* [27] uses a third-party CUDA implementation by PathScale, which has by now been discontinued for several years.[3] Papers that rely on reverse engineering [3, 24] require re-validation after every hardware or software change, and papers that intercept driver communication like *GPUSync* [17] are subject to a stable interface between the CUDA runtime libraries and the underlying device driver, but NVIDIA makes no such stability guarantees for these non-public implementation

[3] According to the PathScale website: `https://www.pathscale.com/`.

details. The key point here is that, when using NVIDIA GPUs, *working around the lack of source-code access is a prerequisite for meaningful research.*

Other prior work demonstrates what is possible with a greater amount of access to internal GPU details. For example, working in collaboration with NVIDIA enabled Capodieci *et al.* to implement the popular preemptive EDF real-time scheduler for GPU workloads, by directly modifying NVIDIA's source code [6]. In a second example, a significant reverse-engineering effort yielded enough internal details to enable Jain *et al.* to implement not only SM partitioning but also L2 cache and DRAM partitioning for NVIDIA GPUs [24]. Naturally, these two papers also worked around the lack of open-source access as well as those mentioned in the previous paragraph, but they also illustrate an additional conclusion: *improved access to GPU internals enables fundamentally more powerful management paradigms.*

The ROCm stack means that AMD GPUs are subject to almost none of the software-based limitations discussed above. Developers can modify the underlying device drivers and user-level HSA libraries, removing the need to use less-supported libraries or to enter non-disclosure agreements simply to add more functionality to existing code. Additionally, transformations of application-level source code could largely be rendered unnecessary given the ability to directly modify HIP or the HCC compiler. All of this means that a GPU-management system targeting AMD GPUs using the ROCm stack should not only be easier to implement, but also easier to use and easier for third parties (*i.e.*, other researchers) to validate.

3.2 Hardware Organization in AMD GPUs

We now shift our focus to the hardware organization of AMD GPUs, in this case exemplified by our test platform: the RX 570 represented in Fig. 1.

In 2011, AMD announced its *Graphics Core Next* (GCN) GPU architecture. For AMD, the GCN architecture was characterized by an increased focus on general-purpose GPU computing. This focus was represented in hardware by AMD's transition away from VLIW (very long instruction word) processors to SIMD (single instruction, multiple data) processors [10]. AMD's GCN architecture remains in use to this day, with progressive updates to accommodate new hardware. GCN's successor, codenamed "RDNA" [15], first appeared in consumer GPUs in 2019, but is not yet officially supported by the ROCm framework.[4] The RX 570, used in this paper, was released in 2017 and uses the fourth generation of the GCN architecture. Many of the interesting features visible in Fig. 1, including the L1 caches and computational elements, are actually specified by the GCN architecture and have remained relatively consistent since the first generation of GCN.

Processing in GCN GPUs. In AMD GPUs, parallel processors are divided into *Compute Units* (CUs), which are highly analogous to the *Streaming Multiprocessors* (SMs) in NVIDIA GPUs. The bulk of each CU's processing power lies within its four SIMD "vector units." Each vector unit has sixteen *lanes*, meaning that each is capable of carrying out a computation on sixteen different pieces of data in parallel. Since each CU contains four sixteen-lane vector units, each CU can carry out up to 64 threads' worth of computation at any given instant. This is highly similar to NVIDIA SMs, which generally have 64 or 128 "CUDA cores," each of which carries out a computation for a single thread. Each CU in a GCN GPU also contains a single *scalar unit*, generally used for instructions such as control-flow operations.

[4] At the time of writing, "Navi" or "RDNA" GPUs are not present in the official list at: `https://github.com/RadeonOpenCompute/ROCm#Hardware-and-Software-Support`.

Despite having 64 SIMD lanes per CU, each CU in the RX 570 can support up to 2,560 in-flight threads at a time.[5] This is similar to the SMs in NVIDIA GPUs, which each support up to 2,048 in-flight threads. Both NVIDIA and AMD GPUs use the large number of in-flight threads for *latency hiding*, swapping to a different group of 64 threads whenever the currently executing group of 64 stalls for some reason, *e.g.*, while waiting for memory.

GPU threads and thread blocks. We briefly return to Figs. 3 and 4 to call attention to the fact that both HIP and CUDA kernel code make use of GPU threads and *thread blocks*. This is seen in the `int i = ...` lines in the `VectorAdd` kernels in both figures, where the thread and thread-block indices are used to select a vector element to process. In both CUDA and HIP, thread blocks act as *schedulable entities*—a single thread block is never split across CUs (or SMs on NVIDIA GPUs), and all threads from a thread block must finish executing before a block is considered complete. After a block completes execution, its threads become available, allowing a new block to start executing on the same CU.

On both NVIDIA and AMD GPUs, the size of a thread block is specified by the programmer at runtime, when a kernel is launched. This can be seen in the `hipLaunchKernel...` line of Fig. 3 and the `VectorAdd<<<...>>>` line of Fig. 4. Blocks are limited to at most 1,024 threads on GPUs from both manufacturers, but programs requiring more than 1,024 threads can request multiple thread blocks.

CU management. Understanding the division of processing capacity into CUs is important: due to the fundamental division of modern GPUs' computing resources among CUs or SMs, partitioning tasks to non-overlapping sets of CUs intuitively should reduce possible interference when sharing a GPU among multiple real-time workloads. In fact, such an idea has received attention from both manufacturers. NVIDIA introduced a mechanism called *Execution Resource Provisioning* (ERP) with its Volta GPU architecture, but ERP only allows users to limit the number of SMs a task can use, not to assign tasks to specific SMs. Perhaps because of this limitation, to our knowledge, no prior real-time GPU management system has attempted to use ERP.

▶ **Benefit 3.** *AMD GPUs natively allow partitioning kernels among compute units.*

Compared to the limited options on NVIDIA systems, compute-unit management on recent AMD GPUs is vastly more powerful, leading to Benefit 3. Starting with the fourth generation of the GCN architecture (often referred to by the code name *Polaris*), AMD GPUs allow each queue of kernels to be associated with a *compute-unit mask*—simply a set of compute units upon which kernels in a queue are allowed to execute [11]. This offers an immediate advantage over the prior SM-partitioning systems on NVIDIA GPUs because the compute-unit mask is completely transparent to kernel code, meaning that general-purpose computations on AMD GPUs can be partitioned among compute units without requiring any source-code modification.

Unfortunately, AMD does not provide a function for setting the compute-unit mask in its HIP API, but we were able to add such a function without too much effort. In Sec. 4, we describe our implementation, and demonstrate the efficacy of the compute-unit-mask mechanism for improving predictability of GPU kernel response times.

Cache hierarchy in GCN GPUs. Readers familiar with NVIDIA GPUs have likely already noticed the difference between the cache hierarchy shown in Fig. 1 and the layout in NVIDIA's GPUs. Like NVIDIA GPUs, AMD GPUs contain an L2 cache that is shared

[5] This is the maximum number of in-flight threads according to the GCN documentation [10], but, as discussed in Sec. 4, we only ever observed a maximum of 2,048 in-flight threads in practice.

among all CUs, but the picture is slightly more complicated when it comes to L1 caches. In NVIDIA GPUs, each SM contains its own L1 data and instruction cache, but in GCN GPUs, each CU only contains its own L1 data cache. The L1 instruction cache is instead shared by a cluster of up to four CUs, as well as a separate L1 scalar data cache, which is a read-only cache used by the scalar unit.

This cache hierarchy means that although CU partitioning implies L1 data cache isolation, certain CU assignments may still be subject to L1 instruction cache interference. In practice, we saw little observable impact of this in our experiments, which we explain in more detail in the following section.

4 Case Studies

We next describe our experiments using the ROCm software stack on AMD GPUs, including our modifications to ROCm and experimental results. Naturally, any practical usage of a large software stack like ROCm is bound to encounter unforeseen complications, a few of which we also discuss in this section. All of our source code is available online, including modified ROCm code,[6] scripts and source-code patches for PyTorch,[7] and a microbenchmarking framework.[8]

4.1 Our Test Platform

We conducted our experiments on a system running Ubuntu 18.04 using Linux kernel version `5.4.0-rc8+`. Our host system contains 16 GB of memory and an Intel Xeon E5-2630 CPU, with eight physical cores supporting two hardware threads each.

We conducted our experiments using an AMD RX 570 GPU. The RX 570 is a mid-range GPU from AMD, costing approximately 180 US Dollars at the time of writing. We used an RX 570 containing 8 GB of GDDR5 memory, but models with 4 GB of memory are also available. The choice of ROCm-compatible platforms is limited, and most ROCm-capable systens will be similar to ours for the following reason:

▶ **Drawback 1.** *AMD's ROCm platform only supports Linux and discrete GPUs.*

Drawback 1 is easily learned by reading the list of supported systems on ROCm's website [12]. As its name implies, ROCm (again, *Radeon Open Compute*) seems motivated by an effort to recapture a portion of the compute-oriented market share currently dominated by NVIDIA. This has the implication that platforms less-suitable for high-performance computing are a lower priority at best.

We grant that the lack of Windows or MacOS support is unlikely to be a problem for researchers hoping to experiment with open-source software. Unfortunately, constraining support to discrete GPUs is a bigger limitation. Specifically, size and power constraints sometimes render discrete GPUs unsuitable for embedded applications, limiting the ability to directly test AMD-based GPU research in the real world. In contrast, Drawback 1 is not a problem for researchers choosing to target NVIDIA GPUs. Unlike AMD, NVIDIA offers several embedded-oriented GPUs, and its *Jetson* and *DRIVE* embedded platforms have been popular in real-time research [3, 6, 18, 22, 35, 42].

---

[6] Modified ROCm code: `https://github.com/yalue/rocm_mega_repo/tree/hip_modification_only`.
[7] PyTorch scripts and patch: `https://github.com/yalue/ecrts_2020_pytorch_experiment`.
[8] Microbenchmarking framework: `https://github.com/yalue/hip_plugin_framework`.

Currently, ROCm's kernel-level components such as the `amdgpu` or `amdkfd` Linux device drivers generally *do* support AMD's APUs (Application Processing Units), which include an integrated GPU, but AMD's APUs seem geared towards laptops or budget-oriented PCs rather than embedded applications.

4.2 Working with ROCm Source Code

While Sec. 4.1 focuses on our base hardware platform, it contains less information about our software choice apart from the underlying operating system. This is because our choice of the base ROCm software version and configuration bears a more in-depth explanation, which we cover in the following paragraphs. Recall that our proof-of-concept objective is to enable transparent compute-unit management in ROCm programs, but we can accomplish this in several different ways, depending on which of the "layers" depicted in Fig. 2 we wish to modify.

Our software platform is ROCm version 3.0. The source code for ROCm components (*e.g.*, HCC, HIP, *etc.*) are contained in separate repositories, while a central ROCm repository merely contains a list of revisions required for each ROCm version. We used this list to obtain an unmodified copy of the ROCm 3.0 source code.[9]

ROCm version 3.0 is up-to-date at the time of this paper's submission, but the code has undergone several revisions and releases since we began working. ROCm was currently on version 2.6 when we began our study of AMD GPUs in mid-2019, so it has already undergone four major revisions by the time of this paper's writing. This trend indicates a second potential drawback of conducting real-time research on AMD GPUs:

▶ **Drawback 2.** *Large portions of the ROCm code base undergo frequent, major changes.*

One can hardly fault AMD for the fact that ROCm continues to be under active development. The fact that continually updating the software is desirable for most users does not, however, negate the inconvenience for researchers. To date, our work has encountered two concrete consequences of the rapid pace of change in ROCm code.

First, our modifications target the HIP programming language, which in turn targets the HCC compiler. However, due to Drawback 2 the HCC compiler was announced as deprecated in March 2019, with support supposedly ended in January 2020. For the time being, HIP code still depends at least in part on the HCC compiler, but AMD's apparent plan is to replace all remaining uses of HCC with the AMDGPU support already present in the LLVM compiler framework [23].

Unfortunately, when the removal of HIP's dependency on the HCC compiler is complete, our CU-management modifications to HIP are likely to increase in complexity. This is due to the fact that HCC already contains an API for managing CU masks, so adding a CU-mask-management function to HIP does not currently require modifications to any lower levels of the ROCm stack. This would not be the case with the AMDGPU LLVM backend, because it does not currently contain a runtime library with functions for managing CU masks. So, supporting CU-management functionality in HIP will, in the future, also require adding CU-management APIs to the LLVM backend. While this is an added difficulty, access to source code means that it should at least remain possible.

A less major but equally time-consuming aspect of Drawback 2 is the effort required to work across the many components involved in the ROCm code base. In fact, a lack of clear

[9] The exact list we used can be found at `https://github.com/RadeonOpenCompute/ROCm/blob/roc-3.0.0/default.xml`.

documentation eventually caused us to abandon our efforts to compile the entire ROCm code base from source. Instead, we now install ROCm from AMD's pre-built debian repositories, and replace individual components with new versions we compile from source.[10] At one point, AMD maintained a repository of scripts capable of compiling a standalone version of ROCm,[11] but at the time of writing these scripts were last updated for ROCm 2.0, released in February 2019. While these scripts may provide a useful point of reference, old install scripts are a poor substitute for up-to-date human-readable instructions, especially when, as we have learned first-hand, questions arise about compatibility between various ROCm components' compile-time options, assumptions regarding installation directory structures, *etc.*

In fact, difficulties with ROCm's documentation are not limited to code compilation:

▶ **Drawback 3.** *There is no centralized, official source of ROCm documentation.*

Even though some may argue that Drawback 3 is mitigated by source-code availability, the entirety of ROCm source code is sufficiently large to make it an impractical source of "documentation." The userspace ROCm code required to run HIP contains over 200,000 lines of C and C++, and even that requires generous omissions from the overall line count. For example, there are many additional libraries and programming languages (including OpenCL) that are included in ROCm but not required to run basic HIP programs. The code relevant to our work is distributed among the components shown in Fig. 2 as follows:

- **HIP**: Roughly 46,000 lines, not including tests, samples, the `hipify` utility, or code for NVIDIA GPUs.
- **HCC**: Roughly 110,000 lines, only counting AMD-specific additions to the base LLVM code. Including all of the LLVM and `clang` compiler code brings the total closer to 1,375,000 lines.
- **HSA API and Runtime**:[12] Roughly 63,000 lines.
- **Drivers**: Roughly 223,000 lines. Of these, 23,000 are in the compute-specific `amdkfd` driver, and remainder are in the `amdgpu` driver, which also handles graphics.

So, even though code can serve as a definitive source of technical information, it is a poor source for high-level instructions or best practices, especially if it requires reading hundreds of thousands of lines.

To our knowledge, there is not a definitive "first-stop" location for ROCm documentation. Instead, there are documents from several different sources that may be useful to researchers working with the ROCm stack:
1) The HSA documentation [19] defines the HSA interface and communication protocol supported by the ROCm userspace libraries.
2) The ROCm website [12] contains an overview of the software system, but seems mostly geared towards "consumers" looking to install an unmodified version.
3) The GCN architecture manuals [14] and white papers [10, 11, 13] provide information about AMD GPU hardware.
4) The LLVM AMDGPU documentation [23] describes the LLVM backend for compiling code targeting AMD GPUs. This includes information about the executable-file format and

[10] We document our procedure in the README file with the code linked in Footnote 6.
[11] `https://github.com/RadeonOpenCompute/Experimental_ROC`
[12] We included both the ROCR-Runtime (`https://github.com/RadeonOpenCompute/ROCR-Runtime`) and ROCT-Thunk-Interface (`https://github.com/RadeonOpenCompute/ROCT-Thunk-Interface`) code in this number.

HSA interface, and is likely to be useful for researchers looking to modify or understand low-level details.

In summary, a reasonable amount of documentation exists, but ROCm offers no close analogue to a comprehensive document like NVIDIA's CUDA programming guide [16]. As mentioned, the lack of compilation instructions for the entire ROCm toolchain is one such example of Drawback 3, and instructions on the main ROCm website assume that users will install ROCm from the pre-compiled repositories [12]. Another facet of Drawback 3 is the fact that documentation may be updated in a piecewise manner, leading to confusing information such as webpages that still recommend using HCC[13] without noting that HCC is deprecated. Of course, the existence of our modifications to ROCm is itself a testament to the fact that none of the documentation-related difficulties are insurmountable.

4.3 Our Modifications to ROCm

We chose to add several different CU-management interfaces to ROCm, by modifying different layers of the stack shown in Fig. 2. Our modifications fell into two categories: we modified the `amdkfd` driver and HCC driver to add two different mechanisms for specifying an application-wide default CU mask, and we also added a new function to the HIP API that enables an application to set the CU mask associated with a HIP stream. While the evaluation we present in Sec. 4.4 only requires our HIP modification, our additions to HCC or the `amdkfd` driver may remain useful for setting process-wide CU masks without modifying application code. In this subsection, we describe all three modifications.

Modifications to the `amdkfd` driver. The `amdkfd` driver is responsible for setting up memory-mapped queues of GPU commands, which are directly shared between userspace and GPU hardware. The HSA API, mentioned in Sec. 3, issues commands to the GPU by writing them into the in-memory queues. User-level applications can control additional settings, such as CU masks, by issuing various ioctl commands to the driver. Even though existing ioctl commands are capable of changing existing queues' CU masks, being limited to per-queue settings makes partitioning an entire GPU-using application difficult, because the application may create new queues at any time in its execution.

To provide a more convenient interface, we added a new feature to the `amdkfd` driver: a `SET_DEFAULT_CU_MASK` ioctl. As the name of this command implies, it causes the driver to apply a default CU mask to any queues the calling process creates in the future. Implementing this change required straightforward changes to the driver code.[14] Using the modification requires a simple modification to the GPU-using applications themselves—they must be modified to issue our new ioctl command at the start of execution, before creating any ROCm queues.

Modifications to HCC. The driver modification outlined above provides one possible method for specifying a default process-wide CU mask, but it suffers from a couple of usability drawbacks. First, as mentioned, one must issue the new ioctl command from within the context of the GPU-using process, which, while a simple change, nonetheless requires modifying application source code. Second, setting a CU mask in the driver may lead to a discrepancy between what higher-level applications "think" a queue's CU mask is, and what the underlying CU mask actually has been set to. Our modifications to HCC allow users to specify a default CU mask with neither of these two drawbacks.

[13] This can still be seen on `https://rocm.github.io/languages.html` as of February 2020.
[14] We included a patch and further instructions in the code linked in Footnote 6.

Since HCC is implemented in userspace, we were able to implement a default-CU-mask modification using a well-known software feature: environment variables. Internally, HCC maintains a copy of each queue's expected CU mask, and uses this additional bookkeeping to potentially share a smaller pool of underlying driver-managed queues among multiple higher-level "queue" abstractions (the details of which, fortunately, are not necessary to understand our modification). In order to implement our environment-variable interface, we found the C++ constructor responsible for initializing one of the aforementioned HCC-managed queues, and modified it to initialize both the CU mask of the actual underlying HSA queue as well as the internal HCC-managed copy of the CU mask based on the contents of the (arbitrarily named) `HSA_DEFAULT_CU_MASK` environment variable. Using this modification is extremely simple: one only needs to set the environment variable before creating a process.

We are aware of no applications that do so, but it is conceivable that an application could circumvent our environment variable by directly interacting with the HSA API layer "below" HCC, as shown in Fig. 2. If necessary, we could even prevent such behavior by modifying AMD's HSA API implementation (known as `ROCR-Runtime` in the ROCm source code) in a manner similar to our HCC change, but, like the driver-based modification, this would risk HCC's bookkeeping becoming inconsistent with actual underlying CU-mask settings.

Modifications to HIP. The modification to HCC is convenient to use in practice, but offers little flexibility within the context of a single process, where we may wish to set different CU masks for different streams (this ended up being necessary for our PyTorch experiments, discussed in Sec. 4.4). To address this need, we chose to add a `hipStreamSetComputeUnitMask` function to the HIP API that sets the CU mask associated with an individual HIP stream.

Each HIP stream is implemented on top of a single underlying HCC-managed queue of GPU commands. Internally, this HCC-managed queue is represented by a C++ object that already provides a `set_cu_mask` method, so we implemented our new HIP function simply by calling this underlying method.

Our change to HIP was straightforward due to HIP's use of HCC, but recall that HIP supports alternate backends. One obvious example is HIP's `nvcc` backend, which targets NVIDIA GPUs by wrapping the CUDA API. NVIDIA GPUs have no concept of hardware-supported CU masking (at least judging by public documentation), so implementing `hipStreamSetComputeUnitMask` for NVIDIA is clearly impossible. As discussed in Sec. 4.2, we anticipate additional difficulties when HIP fully switches to a `clang`-based backend, since we will no longer be able to rely on the availability of the HCC-provided `accelerator_view` interface that provides CU-mask-management functions.

Final words on implementation. All of our CU-management implementation approaches worked as expected. Even though the remainder of our evaluation only required modifying HIP, the fact that so many approaches were viable, even for such a minor feature, shows how an open-source GPU-computation software stack can provide significant flexibility for research purposes. Recall that typical research using NVIDIA GPUs is only able to modify one "layer" in NVIDIA's analogue to Fig. 2: the user program!

In the remainder of this section, we discuss how we applied our modifications to improve timing predictability in a PyTorch application, and describe some additional details about AMD GPUs that we encountered in the course of our experiments.

4.4 Evaluation using PyTorch

We carried out a case study using PyTorch [4] to validate our approach using a complex piece of real-world GPU-accelerated software. We chose PyTorch in part due to its popularity; in 2019, it was used in the majority of papers presented at every major computer-vision

```
import torch
stream = torch.cuda.Stream("cuda:0") # Standard PyTorch API to create a stream
stream.set_cu_mask(0xffff0000)       # Invoke our new function
with torch.cuda.stream(stream):      # Make subsequent GPU code use the stream
    # ... evaluate the network, etc
```

Figure 5 Setting a CU mask in a PyTorch script. Note that PyTorch uses the term “cuda” regardless of the fact that it’s running on an AMD GPU.

conference [21]. Additionally, we wanted to design an experiment that accurately reflects a reality where a single GPU may need to concurrently process multiple independent streams of data, *e.g.*, on an autonomous vehicle with multiple cameras or sensors.

PyTorch, as its name may imply, focuses on providing an idiomatic Python interface for creating, training, and evaluating neural networks. This is a convenient interface for computer-vision and AI researchers, but the level of abstraction involved in a Python interface presents a hurdle for real-time management, where fine-grained control is more desirable.

Required modifications to PyTorch. Given PyTorch’s complexity and level of abstraction, it comes as no surprise that enabling CU mask support required additional modifications to the PyTorch source code. PyTorch is implemented using CUDA, but ships with a script for converting its CUDA source code to HIP. After obtaining the PyTorch source code and running the provided HIP-conversion script, we were able to add functionality for setting CU masks to the PyTorch API.[15] Fortunately, PyTorch already provides an interface for working with “CUDA” streams,[16] so we were able to add another function to PyTorch’s stream-management API to call the `hipStreamSetComputeUnitMask` function we added to HIP. Fig. 5 shows a snippet of a Python script illustrating the usage of the added functionality.

But why go to the trouble of modifying PyTorch source code, when our environment-variable-based HCC modification should already suffice to cause PyTorch to use a CU mask? In fact, our initial attempt was to do exactly that—and indeed, the environment-variable approach appears to correctly limit the CU mask for any given PyTorch process. Unfortunately, this was insufficient for our later experiments requiring contention for GPU resources. We found that AMD GPUs schedule work from separate processes in a different manner (namely, one that seems to prevent or reduce concurrent execution) than work coming from multiple streams within a single process. We observed similar behavior on NVIDIA GPUs in our prior work [3, 35, 41], so this analogous finding on AMD GPUs did not come as a particular surprise, but we chose to leave further investigation for future work. In the meantime, we found that using multiple threads and streams from within a single PyTorch script was sufficient for this paper, which only requires generating observable contention between competing GPU workloads.

Our test application. Our experiments were based on an example provided by PyTorch that trains a neural network to classify handwritten digits in the MNIST dataset.[17] We modified the original script in several ways. First, we changed it to only perform classification

[15] Our specific procedure for working with PyTorch source code, along with a patch containing our modifications, is included in the repository linked in Footnote 7.
[16] In order to maintain compatibility with existing scripts, PyTorch’s user-facing Python API continues to use the term “CUDA” even when compiled for HIP.
[17] Our experimental scripts are included in the link in Footnote 7. We based our experiments on the example from `https://github.com/pytorch/examples/tree/master/mnist`.

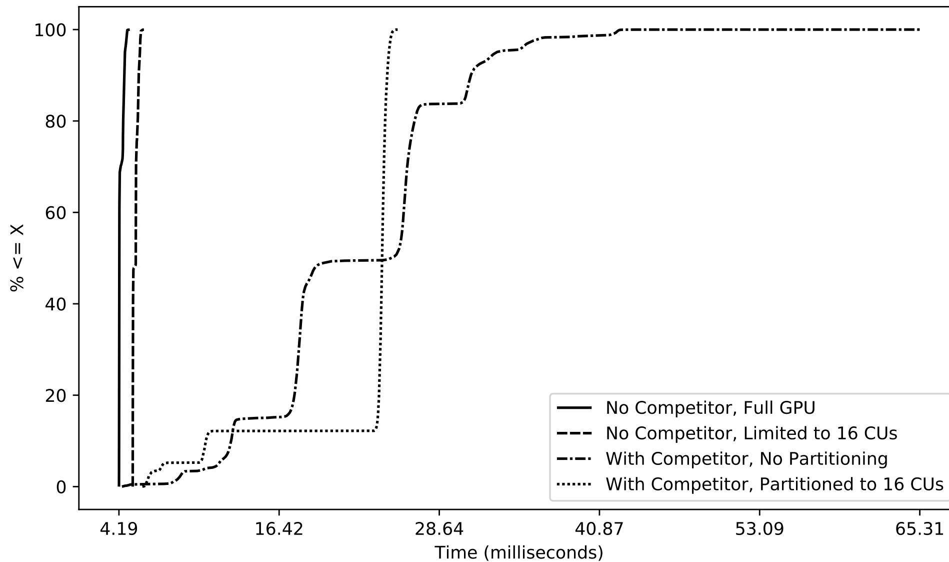

Figure 6 CDFs of times required for a forward pass of PyTorch's MNIST example, with and without a competitor.

using pre-trained network weights instead of training a new network. Second, we added an optional competitor thread[18] to provide additional contention for GPU resources. When active, the competitor thread evaluates random data using a separate neural network. The competing network was also based on the MNIST network, but used larger layer sizes and a larger number of layers to generate more GPU computations. Finally, we updated the manner in which the script loads the MNIST image data, which defaults to loading data on demand. To avoid noise involving data transfers or reads from storage, we pre-buffer all of the image data in GPU memory before conducting any measurements.

We instrumented the primary thread to record the duration of each "iteration," consisting of a single forward pass of the network, which classifies a batch of 64 different 32x32-pixel grayscale input images. We determined the duration by recording the time before the first layer of the network was evaluated, and again after a `stream.synchronize()` call returned, indicating the completion of all GPU operations. While some noise is certainly involved when recording times in Python code, any such noise will be negligible compared to the durations being measured in our application. Additionally, our script does not record times for the first several iterations while the GPU "warms up," which is a common practice in GPU research to avoid measurement noise while code or data is lazily loaded onto the device. In total, we record 36,504 time samples for every scenario (this arbitrary-seeming number is due to a combination of factors, including warm-up iterations and the way in which the MNIST dataset is subdivided into batches of 64).

Verifying CU-mask functionality. Our first experiment was designed to verify our ability to successfully limit a task's set of available CUs, and to test whether partitioning CUs improves response-time predictability in the presence of an adversarial workload. Fig. 6 shows the results from four scenarios. The first scenario is represented by the leftmost curve,

[18] Even though Python's threads are notorious for their lack of concurrency when using C/C++ modules, PyTorch takes measures to release the necessary Python locks within its C++ code.

and shows the response-time distribution when our single MNIST-evaluation PyTorch thread is allowed to execute on all 32 CUs with no competitor. As expected, this configuration exhibits the fastest response times, with a maximum observed response time of 5.02 ms. The second curve shows the response times of the same workload, but in this case it was limited to only use 16 CUs. As seen in Fig. 6, reducing the available CUs from 32 to 16 only increased observed response times by roughly one millisecond, indicating that, absent a competing workload, overheads or CPU operations comprise a significant portion the MNIST example network's overall execution time (we would expect the slowdown due to the reduction in resources to increase in proportion to the amount of GPU computation required).

Behavior changes drastically with the introduction of a competing thread. The curve in Fig. 6 with alternating dots and dashes shows the distribution of iteration times when the competitor thread is running, and both threads are allowed to access all 32 CUs on the GPU. The dotted curve shows the same scenario, but with the competitor and primary thread partitioned to separate groups of 16 CUs. Both distributions exhibit a very similar mean time of approximately 22 milliseconds, but the worst-case observed time without partitioning, 65 milliseconds, is significantly slower than the time when partitioning is applied, where the worst-case observed time was 25 milliseconds—only slightly worse than the same scenario's median time. While this improvement provides sufficient motivation to use our CU-masking API, the significant increase in mean and median times compared to the times without a competitor remains a concern for future work, as it indicates that threads also spend significant time contending for non-CU resources. Some of this contention may be related to Python threads or PyTorch internals, but there may also be other sources of contention for GPU resources that we can prevent with further modifications to ROCm.

4.5 Additional ROCm Experiments

In addition to our evaluation using PyTorch, we also conducted several microbenchmark experiments using a plugin-based framework we developed for earlier work on NVIDIA GPUs [34], and ported to HIP for this paper.[19] Porting our code to HIP required disabling two NVIDIA-specific features only available in kernel inline assembly: reading the `globaltimer` register, and obtaining the ID of the SM of the currently executing thread block. Losing access to these features is unfortunate, as both were essential for our prior experiments to infer scheduling rules for NVIDIA GPUs. However, even without them, the highly instrumented microbenchmarks remain useful for observing details at a finer granularity than a large system like PyTorch can provide. Most of the experiments we conducted with this framework simply confirmed the findings we already discussed in Sec. 4.4, but presenting them enables us to report a few additional results.

Revisiting in-flight threads per CU. Recall from Sec. 3.2 that, according to GCN documentation, AMD GPUs support 2,560 in-flight threads per CU. Our experiments, however, contradicted this number. We never observed 2,560 in-flight threads per CU in any experiment, regardless of thread block size, workload, or number of available CUs. Instead, we observed at most 2,048 in-flight threads in all situations. Fig. 7 contains one such example.

Fig. 7 was produced by our plugin framework concurrently running two microbenchmarks, `DWT2D` and `backprop`, that we adapted for our framework from the Rodinia benchmark suite [9]. The choice of these specific microbenchmarks was somewhat arbitrary (as we can observe the same effects with any of the others), however they issue kernels at slightly more

[19] Source code for this framework is linked in Footnote 8.

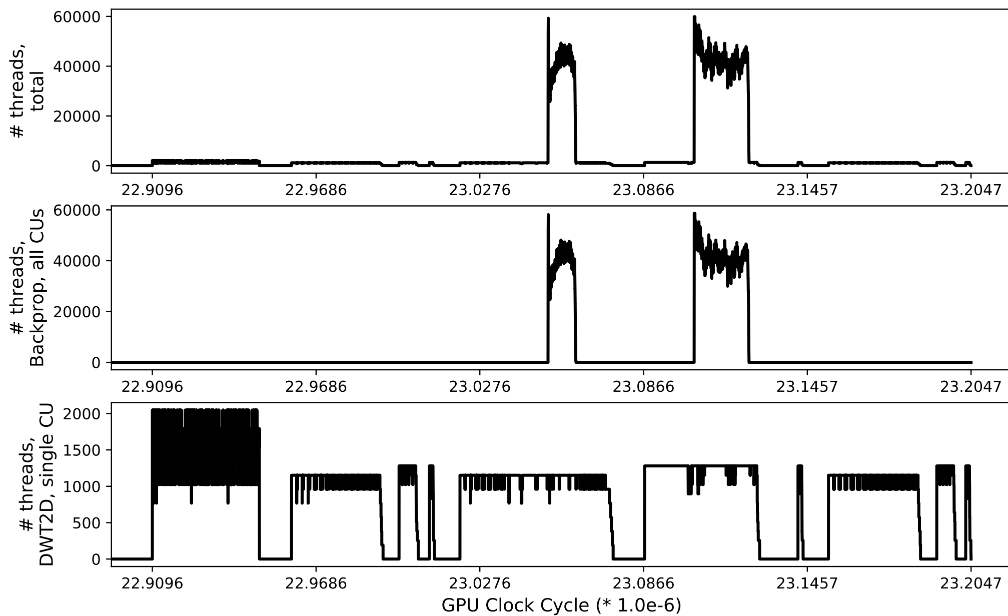

Figure 7 Timelines of the number of in-flight GPU threads during the concurrent execution of two benchmarks.

irregular intervals than other simpler microbenchmarks (*i.e.*, vector add), potentially allowing us to observe a wider variety of interactions. As with all of the plugins in our framework, we instrumented these microbenchmarks' kernels to record the GPU clock cycle when the first thread in each block starts running, as well as the cycle before the last thread in each block returns, and used this information to compute the number of in-flight GPU threads over the course of the microbenchmarks' execution.

Fig. 7 shows three timelines, each of which represents the active number of GPU threads at a given clock cycle. The top timeline shows the total number of active threads, and the other two show the number of threads running on behalf of the two microbenchmarks. These timelines contain two clear pieces of evidence contradicting a 2,560-thread limit. First, if each of the GPU's 32 CUs indeed supported 2,560 in-flight threads, we would expect to see a maximum of 81,920 concurrent total threads in the top timeline of Fig. 7, but we do not. Instead, the maximum total number of observed running threads never exceeds 65,536, consistent with a 2,048-thread limit (in fact, this particular experiment never even exceeds 60,000 total concurrent threads, but we observed a clear ceiling of 65,536 threads in many other microbenchmark experiments). The second major piece of evidence is shown in the timeline for `DWT2D` at the bottom of Fig. 7. We limited `DWT2D` to a single CU during this experiment, and its number of in-flight threads clearly never exceeds 2,048. (Note that its vertical axis uses a different scale.) Furthermore, `DWT2D` was using 256-thread blocks, which evenly divides a hypothetical 2,560-thread limit, so the lower-than-expected thread count cannot be a consequence of incompatible block sizes.

This observation likely relates to Drawbacks 2 and 3, where an aspect of the internal GCN architecture may not yet be adequately documented. Regardless of the reason, this observation underscores an important point for real-time researchers: *important hardware details may not be revealed through open-source software or public documentation.* This applies equally to NVIDIA GPUs, so we do not label this as an explicit "drawback" of

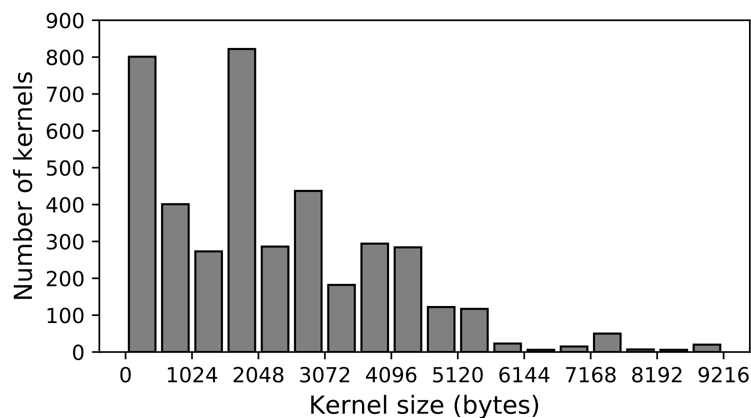

Figure 8 Distribution of GPU kernel sizes found in AMD's pre-compiled ROCm 2.6 repository.

working with AMD. It is quite possible that the "true" 2,560-thread limit can be reached in cases not covered by our microbenchmarks. Nonetheless, apparent discrepancies in a detail as fundamental as the number of supported parallel threads may lead researchers to assume an unsafe model of real-world behavior.

L1 instruction cache impact. For our final experiment, we revisit the question raised in Sec. 3 of the impact of sharing the L1 instruction cache between clusters of four CUs. As mentioned, we were unable to observe any L1 instruction cache interference in our experiments, even in the presence of an attempted adversarial workload, a kernel with 300 KB of code written using inline GCN assembly. There are two factors we suspect make GPU L1 instruction-cache interference only a minor concern in practice:

First, it is reasonable to assume that an L1 cache-line fetch retrieves multiple instructions. This, combined with the fact that well-optimized GPU kernel code tends to have few branches, means that GPU kernel code is unlikely to generate many L1 misses. Second, GPU kernels are unlikely to put much pressure on instruction caches simply by virtue of the small size of typical kernel code.

The second of these two points is supported by the data shown in Fig. 8. In order to obtain this data, we downloaded all of the pre-compiled packages from AMD's ROCm 2.6 respository, and, thanks to the LLVM documentation mentioned in Sec. 4.2 [23], wrote a program that parses ELF binaries containing GPU kernel code in order to report the size of each kernel. In total, we were able to find thousands of kernels in this repository, primarily from GPU-accelerated machine-learning libraries.

As shown in Fig. 8, the average kernel size in AMD's repositories was roughly 2 KB, meaning that up to sixteen such kernels could hypothetically execute concurrently without exhausting the 32 KB L1 instruction cache—an unlikely situation on a single cluster of four CUs. Since many kernels require executing hundreds or thousands of thread blocks, it is far more likely that only a single kernel would be running at a time on each CU cluster.

4.6 AMD GPU Performance

While less relevant for research involving timing predictability, we recognize that fast hardware is still necessary in real applications, and is likely to be of interest to readers. To this end, we ran our PyTorch script from Sec. 4.4 on several hardware platforms, without enabling CU partitioning or the competing thread. The resulting data is summarized in

	Min	Max	Median	Mean	Std. Deviation
NVIDIA Titan V	0.51	5.79	0.51	0.52	0.08
NVIDIA GTX 1060	1.40	1.63	1.42	1.42	0.01
NVIDIA GTX 970	1.38	2.77	1.40	1.40	0.02
AMD RX 570	4.19	5.02	4.21	4.32	0.18
CPU (Intel Xeon 4110)	5.54	18.43	8.60	8.40	1.78

Table 1 Times needed for a single forward pass of PyTorch's MNIST example on several different hardware devices. All times are in milliseconds.

Tbl. 1. Unsurprisingly, one of NVIDIA's flagship GPUs, the Titan V, exhibited the fastest performance by far. The other two NVIDIA GPUs also outperformed the RX 570 used in this paper, though even the RX 570 was faster than running the neural network solely on a CPU. It is possible that the RX 570's slower performance relative to its NVIDIA counterparts can be largely attributed to ROCm's lack of maturity compared to CUDA—a possibility strengthened by the fact that the RX 570 performs similarly to the NVIDIA GTX 1060 in graphical workloads [26]. Finally, even though AMD GPUs currently lag behind NVIDIA in our simple PyTorch benchmark, the slower performance is unlikely to prevent accurately evaluating *management principles*, which remain the focus of real-time research.

5 Conclusion

In this paper, we make the argument that the existence of an alternate platform—AMD GPUs—has the potential to reshape real-time GPU research. The thrust of our argument comes from two key points. First, we described how almost all prior real-time GPU research has, to some extent, worked around the limitations of NVIDIA's closed-source platform, often at the expense of long-term applicability. Second, as we demonstrated using case studies, many of the implementation drawbacks that hamper research on NVIDIA GPUs do not apply to AMD (despite AMD having its own collection of downsides).

In future work, we plan to continue our investigation of AMD GPUs, and, ideally, hope to implement portions of prior NVIDIA-focused real-time research using AMD's open-source software. We also plan to investigate the question of optimal CU assignment in more detail, as well as the impact of contention for non-CU hardware components in software running on AMD GPUs.

Establishing the "best" platform for any complex system will always involve a set of tradeoffs and personal preferences, but it is not necessary to prove that a system is the "best" in order for it to warrant further study. In a research setting, the question is still open as to whether the drawbacks discussed here are an acceptable trade for the benefits offered by an open-source GPU software stack, and the answer will likely vary from one project to another. Nonetheless, with this paper we have at least demonstrated that AMD GPUs warrant further consideration, and, despite some difficulties, offer significant benefits for developers looking for greater control over safety-critical software and hardware.

It is not our goal to provide a definitive answer to the NVIDIA-vs.-AMD question, or even to take a side in the battle. In fact, not only do we refrain from taking a side—we hope the battle for GPU dominance continues! The competition itself has already led to a major hardware vendor investing significantly in open-source GPU software, and it would be wonderful if others would follow AMD's example.

References

1. Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
2. Waqar Ali and Heechul Yun. Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms (Artifact). *Dagstuhl Artifacts Series*, 4(2):3:1–3:2, 2018. URL: `http://drops.dagstuhl.de/opus/volltexte/2018/8971`.
3. Tanya Amert, Nathan Otterness, James Anderson, and F. D. Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.
4. PyTorch Authors. PyTorch. Online at `https://pytorch.org/`, 2020.
5. Can Basaran and Kyoung-Don Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.
6. Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for GPU with preemption support. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.
7. Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. SiGAMMA: Server based integrated GPU arbitration mechanism for memory accesses. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 48–57, 2017.
8. Roberto Cavicchioli, Nicola Capodieci, Marco Solieri, and Marko Bertogna. Novel methodologies for predictable CPU-to-GPU command offloading. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.
9. Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.
10. AMD Corporation. AMD graphics core next (GCN) architecture. Online at `https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf`, accessed September 2019., 2011.
11. AMD Corporation. Radeon: Dissecting the polaris architecture (white paper). Online at `https://www.amd.com/system/files/documents/polaris-whitepaper.pdf`, acessed September 2019., 2016.
12. AMD Corporation. ROCm, a new era in open GPU computing. Online at `https://rocm.github.io/`, 2016.
13. AMD Corporation. Radeon's next-generation Vega architecture. Online at `https://www.techpowerup.com/gpu-specs/docs/amd-vega-architecture.pdf`, accessed September 2019., 2017.
14. AMD Corporation. "Vega" instruction set architecture: Reference guide. Online at `https://developer.amd.com/wp-content/resources/Vega_Shader_ISA_28July2017.pdf`, 2017.
15. AMD Corporation. Introducing RDNA architecture. Online at `https://www.amd.com/system/files/documents/rdna-whitepaper.pdf`, accessed February 2020., 2019.
16. NVIDIA Corporation. CUDA C programming guide. Online at `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2019.
17. Glenn A Elliott, Bryan C Ward, and James H Anderson. GPUSync: A framework for real-time GPU management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.
18. Björn Forsberg, Andrea Marongiu, and Luca Benini. GPUguard: Towards supporting a predictable execution model for heterogeneous SoC. In *Proceedings of the Conference on Design, Automation & Test in Europe*, 2017.
19. HSA Foundation. HSA platform system architecture specification. Online at `http://www.hsafoundation.com/?ddownload=5702`, 2018.
20. HSA Foundation. HSA runtime programmer's reference manual. Online at `http://www.hsafoundation.com/?ddownload=5704`, 2018.

21 Horace He. The state of machine learning frameworks in 2019. Online at `https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/`, October 2019.

22 Přemysl Houdek, Michal Sojka, and Zdeněk Hanzálek. Towards predictable execution model on ARM-based heterogeneous platforms. In *International Symposium on Industrial Electronics (ISIE)*, 2017.

23 LLVM Compiler Infrastructure. User guide for AMDGPU backend. Online at `https://llvm.org/docs/AMDGPUUsage.html`, 2019.

24 Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan (Raj) Rajkumar. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

25 Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

26 Will Judd. AMD Radeon RX 570 benchmarks: A capable 1080p workhorse. Online at `https://www.eurogamer.net/articles/digitalfoundry-2019-05-01-amd-radeon-rx-570-benchmarks-7001`, June 2019.

27 Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *IEEE Real-Time Systems Symposium (RTSS)*, 2011.

28 Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC*, 2011.

29 Michael Larabel. AMDKFD is present for linux 3.19 in open-source HSA start. *Phoronix.com*, 2014. Online at `https://www.phoronix.com/scan.php?page=news_item&px=MTg1MzE`.

30 Haeseung Lee and Mohammed Abdullah Al Faruque. Run-time scheduling framework for event-driven applications on a GPU-based embedded system. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2016.

31 Hyeonsu Lee, Jaehun Roh, and Euiseong Seo. A GPU kernel transactionization scheme for preemptive priority scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

32 NVIDIA. CUDA toolkit. Online at `https://developer.nvidia.com/cuda-toolkit`, 2019.

33 NVIDIA. NVIDIA cuDNN. Online at `https://developer.nvidia.com/cudnn`, 2019.

34 Nathan Otterness, Ming Yang, Tanya Amert, James Anderson, and F. D. Smith. Inferring the scheduling policies of an embedded cuda GPU. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2017.

35 Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James Anderson, F.D. Smith Smith, Alex Berg, and Shige Wang. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

36 Sujan Kumar Saha. Spatio-temporal GPU management for real-time cyber-physical systems. Master's thesis, UC Riverside, 2018.

37 Puja Tayal. NVIDIA loses some discrete GPU market share to AMD. Online at `https://articles2.marketrealist.com/2019/06/nvidia-loses-some-discrete-gpu-market-share-to-amd/`, June 2019.

38 Uri Verner, Avi Mendelson, and Assaf Schuster. Batch method for efficient resource sharing in real-time multi-GPU systems. In *International Conference on Distributed Computing and Networking*. Springer, 2014.

39 Uri Verner, Avi Mendelson, and Assaf Schuster. Scheduling periodic real-time communication in multi-GPU systems. In *IEEE International Conference on Computer Communication and Networks (ICCCN)*, 2014.

40 Uri Verner, Assaf Schuster, Mark Silberstein, and Avi Mendelson. Scheduling processing of real-time data streams on heterogeneous multi-GPU systems. In *ACM International Systems and Storage Conference*, 2012.

41 Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H Anderson, and F Donelson Smith. Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

42 Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F Donelson Smith, James H Anderson, and Jan-Michael Frahm. Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

43 Husheng Zhou, Guangmo Tong, and Cong Liu. GPES: A preemptive execution system for GPGPU computing. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.