Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices"

Nathan Otterness[1] and James H. Anderson[1]

[1]University of North Carolina, Chapel Hill, North Carolina, USA.

Contributing authors: otternes@cs.unc.edu; anderson@cs.unc.edu;

Abstract

Graphics processing units (GPUs) have been the target of a significant body of recent real-time research, but research is often hampered by the "black box" nature of GPU hardware and software. Now that one GPU manufacturer, AMD, has embraced an open-source software stack, one may expect an increased amount of real-time research to use AMD GPUs. Reality, however, is more complicated. Without understanding where internal details may differ, researchers have no basis for assuming that observations made using NVIDIA GPUs will continue to hold for AMD GPUs. Additionally, the openness of AMD's software does not mean that their scheduling behavior is obvious, especially due to sparse, scattered documentation. In this paper, we gather the disparate pieces of documentation into a single coherent source that provides an end-to-end description of how compute work is scheduled on AMD GPUs. In doing so, we start with a concrete demonstration of how incorrect management triggers extreme worst-case behavior in shared AMD GPUs. Subsequently, we explain the internal scheduling rules for AMD GPUs, how they led to the "worst practices," and how to correctly manage some of the most performance-critical factors in AMD GPU sharing.

Keywords: real-time systems, graphics processing units, parallel computing

1 Introduction

Real-time systems research seeks to enable predictable timing on computing hardware that is becoming ever more varied and sophisticated. While different hardware components of a full computational platform may require

different management techniques, some truths hold for all aspects of real-time research. One such fact is that, for any given hardware component, techniques for managing and predicting timing are only as powerful as the *models* they are predicated upon.

This fact is reflected by the recent research attention given to graphics processing units (GPUs) in real-time systems. Early work viewed GPUs as black boxes, capable of predictably handling one computational task at a time. As later work published more details and management techniques, sharing a GPU among multiple real-time tasks without compromising timing predictability became increasingly possible. While this progress is encouraging, describing research trends in this way fails to capture part of the difficulty in working with GPUs as opposed to some better-established hardware components, *i.e.*, CPUs. Namely, even in the face of different hardware-implementation details, the *models* of CPU execution assumed in the real-time literature are often applicable to a wide variety of CPUs. The same cannot be said for any current model of GPU execution.

The foremost difficulty with establishing a widely applicable model of GPU behavior is that adequate details about GPU behavior can be difficult to obtain. The second difficulty is related: it is often unknown whether details will remain the same between old and new GPU models from the same manufacturer (Amert et al, 2017), much less GPUs from different manufacturers entirely. In summary, models of GPU execution suffer from both a *lack of information and a lack of commonality*.

Both of these problems are related, though. Working around the lack of commonality requires establishing a baseline of identical expectations and capabilities. If such a common baseline diverges from default behavior in some GPUs, perhaps sufficient additional management can be applied to create a common abstraction. Stated as a question: can we build a sufficiently powerful model using only the attributes that all GPUs have in common? And, if not, how can we work around the differences? Even if an ideal solution requires new hardware, there is no way to determine which hardware changes are necessary if we do not sufficiently understand the behavior of current hardware. In all cases, solving the problem of commonality must begin with solving the problem of information.

"Open source" and the information problem. Following the above argument, one would think that an open-source platform, like AMD's GPU-compute software, would be in an unrivaled position of prominence within real-time GPU research. Unfortunately, mere source-code access is far from an ideal solution to the information difficulties discussed above. Depending on how one counts, AMD's open-source code base contains from tens of thousands to millions of lines of code (Otterness and Anderson, 2020), and even accounts for ten percent of the lines of code in the entire Linux kernel source tree (Larabel, 2020). This is certainly a rich source of information, but without external documentation it is a set of encyclopedias without an index: mostly useful to those who already know where to look.

For NVIDIA GPUs, which offer very little official information about internal behavior, this lack of knowledge has been partially filled by black-box experiments and reverse-engineering efforts of real-time researchers (Amert et al, 2017; Olmedo et al, 2020). This paper fills an information gap for AMD GPUs that is similar to that filled by prior work on NVIDIA. Unlike this prior work, however, we do not rely on black-box tests or reverse engineering in this paper. Instead, we collected the details in this paper from several sources, including public presentations, white papers, and specific source code references.[1] Our investigation is particularly focused on AMD GPUs' *compute-unit masking* feature, which allows partitioning GPU-sharing tasks to separate compute resources. Compute-unit masking is highly desirable to prevent unpredictable contention when sharing GPU hardware (Otterness and Anderson, 2020), but apart from some software "tricks" (*i.e.* Jain et al (2019)), such partitioning remains almost exclusive to AMD GPUs[2] and is inextricable from internal hardware-scheduling behavior.

Contributions. Our contributions are twofold. First, using disparate sources of information, we present an end-to-end view of how general-purpose computations are scheduled on AMD GPUs. Second, we present the first and only (to our knowledge) public documentation on how to effectively use the compute-unit masking feature of AMD GPUs.

Organization. In Section 2, we begin with an overview of GPU programming and prior work on GPU scheduling. Following this, we present our motivating experimental example in Section 3, including our hardware and software test platforms. We describe AMD's GPU-compute scheduling in Section 4. We discuss practical applications of compute-unit masking and future research in Section 5 and finally conclude in Section 6.

2 Background

An understanding of basic GPU usage for general-purpose computation is necessary to understand the experiments and scheduling details discussed later in this paper. Similarly, the context of this paper is motivated by the current state of real-time GPU research. We cover both of these topics in this section.

2.1 GPU Programming

Programs that offload work to a GPU typically use the following pattern (omitting setup such as allocating memory):

1. Copy input data from CPU memory to GPU memory.

[1] We originally learned many of these details in a private conversation with an AMD engineer, to whom we are extremely grateful. This simplified our search for corresponding information in the publicly available material.

[2] NVIDIA introduced partitioning support, known as MIG (multi-instance GPU) in its most recent top-end GPUs (NVIDIA Corporation, 2020). MIG is arguably more powerful than AMD's CU masking, as, unlike CU masks, MIG allows memory partitioning. Unfortunately, it is unclear if or when MIG will be supported in NVIDIA's consumer-oriented GPUs.

```
// Code for the GPU kernel.  Sets 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() {
  // [Omitted: allocate and initialize vectors a, b, and c]

  // Create a stream
  hipStream_t stream;
  hipStreamCreate(&stream);

  // [Omitted: Copy input data to the GPU]

  // Launch the kernel
  hipLaunchKernelGGL(block_count, thread_count, 0,
    stream, a, b, c);

  // Wait for the kernel to complete
  hipStreamSynchronize(stream);

  // [Omitted: Copy results from the GPU, cleanup, etc.]
}
```

Fig. 1: HIP code that defines and launches a GPU kernel.

2. Invoke a piece of GPU code, called a *kernel*.
3. Wait for the GPU code (the kernel) to finish executing.
4. Copy the resulting data from GPU memory to CPU memory.

From userspace, all of these steps are carried out using a higher-level API for controlling the GPU. For example, the well-known CUDA API provides this functionality for NVIDIA GPUs. CUDA is unsupported on AMD GPUs, so in this paper we instead use the HIP API, which is highly similar to CUDA but supported by AMD. Note also that we use the term *kernel* to refer to a section of code that runs on the GPU (this is common terminology in GPU literature). When it is necessary to refer instead to the operating-system notion of "kernel code," we use alternate terminology such as "driver code."

Threads, blocks, and streams. In this paper, we use the term *thread* to refer to a single logical thread of computation on the GPU. When invoking a kernel, the (CPU) program specifies a number of parallel GPU threads with which to execute the kernel code. From a programming standpoint, threads are an abstraction designed to simplify writing software that takes advantage of the GPU's parallel-processing hardware. In hardware, each GPU thread will be executed using a single lane in one of the GPU's SIMD (single instruction, multiple data) vector-processing units.

Figure 1 illustrates the concepts of *blocks* and *streams* as used in a real application. The beginning of the code in Figure 1 defines a kernel for setting vector `c` to the sum of input vectors `a` and `b`. The kernel is launched using the `hipLaunchKernelGGL` function in `main`. Of particular interest are the `block_count` and `thread_count` arguments when launching the kernel. These two arguments control the number of parallel threads created to execute the kernel code. A *thread block*, or simply *block* consists of up to 1,024 parallel threads (specified by `thread_count`). The number of threads in a block is limited to 1,024, but billions of blocks may be requested per kernel—effectively an unlimited quantity for practical applications.

As shown in Figure 1, threads in the `VectorAdd` kernel are able to use the special `blockIdx` and `threadIdx` variables to access their per-thread block and thread indices, meaning that threads and blocks serve the essential purpose of distinguishing between GPU threads at runtime. In the example kernel, each thread uses this information to compute a unique index into the vectors. However, thread blocks also play another role: they serve as schedulable entities when dispatching work to the GPU's computation hardware. We cover this role in detail in Section 4.

The other detail present in the `hipLaunchKernelGGL` invocation is the `stream` argument. This makes use of the *HIP stream* created earlier using the `hipStreamCreate` API call. HIP streams are queues of GPU operations, such as kernel launches or memory transfers. Operations enqueued in a HIP stream execute in FIFO order, meaning that each enqueued operation must complete before the next begins. In Figure 1, the `stream` argument indicates that the kernel-launch operation is to be enqueued in the stream created earlier in `main`. Specifying a stream argument to `hipLaunchKernelGGL` is actually optional; kernel launches default to being enqueued in a special "null stream." While an explicit non-default stream will not provide any benefit in Figure 1's single-kernel example, user-created streams have a performance benefit in complex applications: work enqueued in separate streams may execute concurrently, whereas work in a single stream (*e.g.*, the null stream) is guaranteed to be serialized. Given the obvious connection between execution order and GPU scheduling, this basic understanding of the HIP stream API is essential background for our discussion in Section 4.

2.2 Prior Work

The past decade of academic literature includes a handful of papers dedicated to reverse engineering or revealing various aspects of NVIDIA GPU behavior. In 2013, Peres used reverse engineering to infer power-management controls for NVIDIA GPUs (Peres, 2013). Later that year, Fuji *et al.* reverse engineered and modified the microcontroller firmware for the NVIDIA GTX 480 to improve response times for some workloads (Fujii et al, 2013). In 2016, Mei and Chu used microbenchmark experiments to infer the cache and memory layout for several generations of NVIDIA GPUs (Mei and Chu, 2016). Later, Jia *et al.* published successive papers using microbenchmarking to infer instruction-set

details about both NVIDIA's Volta (Jia et al, 2018) and Turing (Jia et al, 2019) GPU architectures.

While potentially applicable in a real-time setting, the work mentioned in the previous paragraph all focuses on aspects of performance other than *predictable timing*. This aspect has been the focus of prior work from the real-time literature. Rather than attempting to summarize the significant, growing collection of *all* real-time GPU research, we restrict our focus to the narrower set of papers directly related to our current topic: those that heavily feature new information about internal GPU scheduling policies.

Early GPU-management work tended to treat GPUs as black boxes. In this early work (*e.g.*, Kato et al (2011)), separate real-time tasks acquire exclusive ownership over one or more GPUs. This requires interacting with GPU internals to some extent, *e.g.*, when implementing modified schedulers, but the principle of exclusivity is agnostic to the underlying GPU architecture. The principles outlined in Section 1 explain the motivation behind such an approach: treating GPU-internal behavior as a black box is certainly sufficient for establishing *commonality* across many GPUs. Unfortunately, treating entire GPUs as black boxes can also lead to *capacity loss*: the GPU's computational capacity may be sufficient to support several concurrent tasks, but this type of scheduling treatment mostly prevents truly concurrent GPU sharing.

While GPUs have become only more powerful since 2011 (when Kato et al (2011) was published), the demands made of GPUs have increased apace with, or faster than, the hardware's capabilities. Nowhere is this more apparent than with AI-oriented embedded GPUs, such as NVIDIA's Jetson TX2. Avoiding capacity loss is essential for effective usage of embedded GPUs, which not only are less powerful, but also must often be shared by entire task systems. This motivates some of our group's prior work, in which we used black-box experiments to infer the queueing structure used to schedule work on the Jetson TX2 GPU (Amert et al, 2017). This information led to useful models (Yang et al, 2018), and yielded software tools capable of re-testing queuing behavior on future NVIDIA GPUs. Nonetheless, strictly black-box experiments are unable to adequately expose some details and require re-validation after hardware or software updates.

Another notable contribution, by Capodieci et al (2018), focuses on the NVIDIA Drive PX embedded-GPU system-on-chip, prominently aimed at the automotive market. This work focuses on the implementation of preemptive deadline-based schedulers for GPU workloads, but it also serves as a rich source of information about (the unmodified) GPU scheduling behavior. Unlike the other prior work we discuss, this paper leverages NVIDIA collaboration, eliminating any need for black-box experiments or reverse engineering. So, while the scheduling internals described in Capodieci et al (2018) are certainly useful for enriching models, the implementation approach depends on NVIDIA's willingness to collaborate, which unfortunately may remain out of reach for other groups.

A 2019 paper by Jain et al (2019) contains one of the most comprehensive reverse-engineering efforts of NVIDIA GPUs' memory hierarchy to date. Jain *et al.* focus on partitioning: methods to divide a single GPU's compute and memory resources into non-interfering regions. Not only do Jain *et al.* rely partially on reverse engineering, they also rely on modifications to an open-source subset of NVIDIA's Linux driver. Strong partitioning somewhat obviates the need for a deeper understanding of scheduling behavior, which even Jain *et al.* acknowledge depends on closed-source, unmodifiable components of the driver code. Given our similar focus on partitioning, we hope that in the future our own work can be combined with work like Jain *et al.*'s to produce a "complete" GPU-management system that offers not only memory and compute-unit partitioning but also uses a fully open-source software stack to remove some remaining uncertainties about inter- or intra-partition interference.

To our knowledge, the most recent real-time paper focused on GPU scheduling internals comes from Olmedo et al (2020), who seek to update work such as our own group's 2017 paper (Amert et al, 2017). Olmedo *et al.* use more specific terminology and, more importantly, provide new details regarding the assignment of thread blocks to streaming multiprocessors (computational units in NVIDIA GPUs). Olmedo *et al.* specifically refute some simplistic round-robin scheduling models assumed in a handful of prior papers, demonstrating the importance of accurate information when developing GPU scheduling models. Despite our use of a different hardware platform, in this paper we attempt to build an in-depth, confident model more similar to that of Olmedo *et al.* than that of our group's 2017 effort for the Jetson TX2. Naturally, the comparison is not entirely direct due to the fundamental architectural differences between AMD and NVIDIA GPUs, but the goal remains the same: provide a foundation for future models.

3 Motivating Experiments

With foreknowledge of AMD GPU behavior, we can craft workloads that intentionally trigger highly destructive interference between competing tasks. Doing so serves a dual purpose: first, the challenge of explaining degenerate cases gives structure to our subsequent explanation of scheduling behavior. Second, the experiments concretely illustrate the magnitude of the impact scheduling behavior can have on response times. In this case, we intentionally apply "worst practices" in contrived scenarios, but the lack of documentation discussed in Section 1 means that there is almost no relevant guidance from AMD on how to properly design GPU-sharing workloads. In other words, a naïve developer, not knowing to avoid these practices, may feasibly stumble into the same mistakes.

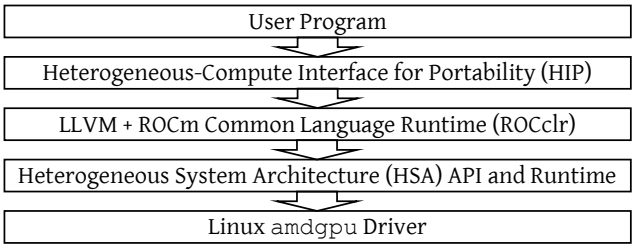


Fig. 2: Components of the ROCm software stack.

3.1 Hardware and Software Platform

Our test platform was a desktop PC containing an Intel Xeon E5-2630 CPU, 16 GB of DRAM, and running version 5.14.0-rc3 of the Linux kernel. We conducted our experiments using an AMD Radeon VII GPU, which at the time of writing is generally considered a higher-end model featuring 16 GB of memory and 60 compute units.

Compute units. Compute units (CUs) contain the bulk of the vector-processing logic responsible for carrying out parallel computations in AMD GPUs. For practical purposes, up to 2,048 GPU threads can be assigned to a CU at once (Otterness and Anderson, 2020), but other factors, such as register availability, may impose further restrictions (Aaltonen, 2017). This should sound familiar to readers who have worked with CUDA, as the role of CUs is analogous to that of NVIDIA's streaming multiprocessors (SMs). Unlike with CUDA, however, the HIP API allows users of AMD GPUs to specify a *compute unit mask* when creating streams—defining a set of CUs on which kernels issued to the stream are required to execute. We discuss this further in Section 4.

Software platform. For these experiments, we used version 4.2 of the ROCm (Radeon Open Compute) software stack. On NVIDIA GPUs, the term "CUDA" often monolithically refers to the GPU-programming compiler, API, and runtime libraries, but ROCm is less monolithic, and is typically described in terms of its components. Figure 2 shows the primary stack of components involved in ROCm. The top user-facing component of ROCm is typically the HIP API (Heterogeneous-Compute Interface for Portability), which is nearly identical to CUDA, with the main practical difference only being the names of the API functions. GPU kernels in HIP programs are compiled using the LLVM compiler's AMDGPU backend, and run using the ROCclr (ROCm Common Language Runtime) runtime library. Below ROCclr is a lower-level userspace library implementing the HSA (Heterogeneous System Architecture) API, which creates and manages the memory-mapped queues and commands that interface with the driver and hardware.

3.2 Experimental Setup

All of the experiments in this paper measure response times of a matrix-multiply task. All tasks multiply two 1,024x1,024 square matrices, writing the output into a third matrix of the same size. Each GPU thread is responsible for computing one element in the results matrix. All elements of each matrix are 32-bit floating-point numbers, randomly initialized to values between 0 and 1. All of our source code and data is available online.[3]

We chose to use a matrix-multiply workload for several reasons. First, each matrix multiplication requires a constant amount of computation and memory accesses, ideally resulting in low variability between kernel invocations. Second, matrix multiply is a relevant and common operation in many AI and graphics applications. Third, it is easy to configure matrix multiplication kernels to use differing thread block sizes without affecting the total amount of GPU computation required.

Choice of competing tasks. Even though all of our tasks carry out multiplication of 1,024x1,024 matricies, we use the flexibility with respect to block size to define two different tasks:

- `MM1024`: Uses blocks of 1,024 threads (specifically, 2D blocks with dimensions of 32x32 threads). Since the matrix contains 1,024x1,024 elements, this task launches exactly 1,024 blocks.
- `MM256`: Uses blocks of 256 threads (in this case, 16x16 2D blocks). Covering the complete matrix therefore requires 4,096 blocks.

Once again, we stress that both `MM1024` and `MM256` carry out an identical number of floating-point computations, just under slightly different configurations. Most of our experiments consist of running a *measured task* and a single *competitor* at the same time on the GPU. We measure the response times of the measured task while it contends with the competitor for GPU resources.

We took several additional steps when launching experiments. We disabled graphics on the host system, to prevent graphics processing from affecting our measurements. In order to amplify contention for compute resources (as opposed to memory), we configured our tests to copy the input matrices to the GPU only once, at initialization time. Tasks using CU masking created their streams using HIP's `hipExtStreamCreateWithCUMask` function. At runtime, we configured competing tasks to run as many multiplication iterations as possible within 60 seconds, as opposed to using a fixed number of iterations. (This is why the number of samples in Table 1 differs between tasks.) A fixed number of samples would require estimating a number of iterations for each competing task, risking outliers if the competitor ends too early.

[3]Our test framework, including code for GPU kernels, is available at https://github.com/yalue/hip_plugin_framework. Scripts and data specific to this paper are available at https://github.com/yalue/rtns2021_figures.

Scenario	Partitioning	# Samples	Min	Max	Median	Arith. Mean	Std. Dev.
Isolated MM1024	N/A	18258	3.094	3.631	3.203	3.201	0.018
Isolated MM256	N/A	11712	4.499	5.588	5.034	5.034	0.100
MM1024 (vs. MM1024)	Full GPU Sharing	9179	5.915	7.292	6.421	6.447	0.144
	Even Partitioning	8482	6.774	8.254	6.973	6.988	0.082
	Uneven Semi-Partitioning	789	61.986	98.380	73.402	76.011	5.888
MM1024 (vs. MM256)	Full GPU Sharing	3811	**12.214**	**19.578**	**15.503**	**15.652**	1.114
	Even Partitioning	8477	6.784	7.630	6.944	6.991	0.133
	Uneven Semi-Partitioning	711	64.873	101.531	84.047	84.362	4.381
MM256 (vs. MM256)	Full GPU Sharing	10383	5.115	7.932	5.552	5.687	0.367
	Even Partitioning	9269	6.126	7.031	6.316	6.386	0.182
	Uneven Semi-Partitioning	1064	55.535	56.928	56.311	56.310	0.171
MM256 (vs. MM1024)	Full GPU Sharing	15539	3.240	6.923	3.564	3.770	0.557
	Even Partitioning	9361	6.085	7.610	6.256	6.318	0.167
	Uneven Semi-Partitioning	1079	55.137	56.028	55.549	55.550	0.114

Table 1: Table of experimental results. All times are in milliseconds.

3.3 "Anomalous" Results

Table 1 contains the results of all of our experiments, showing the response times for each possible measured task against each possible competitor. Table 1 also includes three partitioning configurations for each combination of measured task and competitor:

- *Full GPU Sharing*: Both the measured task and the competitor have unrestricted access to all CUs on the GPU.
- *Even Partitioning*: Both the measured task and the competitor were restricted to separate non-overlapping partitions containing half of the GPU's CUs.
- *Uneven Semi-Partitioning*: CU partitions were identical to the "Even Partitioning" case, but the measured task was allowed access to one additional CU in the competitor's partition.

Without an understanding of AMD scheduling internals, these results likely contain several surprises.

Observation 1. *Competing against MM256 adversely affects MM1024's performance more than any other configuration.*

Observation 1 is illustrated by comparing the "MM1024 (vs. MM256)" section of Table 1 with the corresponding values in any other section. In particular, the "Full GPU Sharing" results, shown in bold, are the slowest under this configuration, with average-case response times more than double those against an identical MM1024 competitor, and five times that of MM1024 in isolation. We can check that this has almost no impact from MM256's perspective by checking the "MM256 (vs. MM1024)" portion of the table. Not only does MM1024 not harm MM256's performance, it slightly *improves* MM256's response times over MM256 in isolation.[4]

[4]The material in Section 4 does not entirely explain this particular anomaly, but the improvement likely is due to a competitor's presence improving the performance of block-dispatching hardware. Two factors support this assumption. MM256 launches four times the number of blocks as MM1024, meaning that speeding up block launches provides a stronger benefit to MM256. For example, the presence of the MM1024 competitor may help keep some hardware components active,

Observation 2. *Even partitioning protects `MM1024` against `MM256`, but otherwise moderately increases response times.*

Observation 2 is made evident by comparing the "Even Partitioning" lines in Table 1 against the "Full GPU Sharing" lines. In all cases except for `MM1024` vs. `MM256` (and the worst-case time of `MM256` vs. `MM256`), the response times when the full GPU is shared are at least one millisecond faster than the times when the competitors are partitioned. This is not particularly surprising for the common case; it makes sense that it will be faster to allow a kernel to occupy any CU as it becomes available across the entire GPU. However, partitioning's ability to protect a workload against an "evil" competitor is obvious when observing `MM1024`'s partitioned performance against `MM256`, where the improvement in the observed worst case is nearly 12 milliseconds. So, Observation 2 is not particularly surprising, and a classic example of a "real-time" tradeoff between overhead and predictability.

Observation 3. *Poor partitioning causes abysmal performance.*

The most surprising feature of Table 1 is undoubtedly the extreme increase in response times of "Uneven Semi-Partitioning," regardless of competitor choice (though `MM1024` vs. `MM256` is still the worst, especially in the average cases). Under this "partitioning" approach, the measured task still maintains sole ownership over all of the CUs it was allowed under "Even Partitioning," but is additionally granted one CU that is shared with the competitor's partition. In other words, one *additional* CU leads to response times around ten times slower than under full GPU sharing or with equal, non-overlapping partitions.[5]

Remarks on these results. We clearly demonstrated that a naïve application of CU masking is *dangerous*. With such extreme performance degradation, even someone only passingly familiar with GPU management should suspect that our uneven semi-partitioned setup is a "worst practice." Nonetheless, partitioning is also *essential*, as different combinations of GPU-sharing tasks (*i.e.*, `MM1024` vs. `MM256`) reveal that asymmetrically destructive interference is a real possibility. The interesting issue is, of course, not the results themselves, but the underlying causes. Why does simply reducing a kernel's block dimensions make it such a fierce competitor? Why can *adding* a compute unit, even a shared one, lead to a dramatic, nearly 14-fold increase in worst-case response time? Fortunately, answers to these questions become apparent with an understanding of AMD GPU scheduling internals.

---

but, as we shall see in Section 4, it will cause minimal additional contention for resources against `MM256`. Second, even though `MM256`'s times are faster in this case than in isolation, it still is not as fast as `MM1024` in isolation. This indicates that `MM1024` still has some advantage arising from its block configuration, as it is otherwise identical to `MM256`.

[5] The competitor's response times are barely impacted by sharing one CU with the measured task. For brevity, we chose to exclude these measurements from Table 1.

4 Scheduling Compute Kernels on AMD GPUs

The effects from Section 3 turn out to mostly arise from hardware, but we also must explain how a kernel arrives at the hardware to begin with. This section covers this entire path, beginning with a description of the queuing structure used to issue kernel-launch commands to AMD GPUs.

4.1 Queue Handling in Userspace

Readers familiar with prior work on NVIDIA GPUs (Amert et al, 2017; Olmedo et al, 2020; Capodieci et al, 2018) should not be surprised to learn that a kernel-launch request proceeds through a hierarchy of queues before reaching AMD GPU hardware. Figure 3 depicts the paths this request may take. To help reduce the complexity of our later explanations (and to provide an easier introduction than Figure 3), we begin with a high-level outline of the steps involved:

1. A user program calls the `hipLaunchKernelGGL` API function to launch a kernel.
2. The HIP runtime inserts a kernel-launch command into a software queue managed by the ROCclr runtime library.
3. ROCclr converts the kernel-launch command into an AQL (architected queuing language) packet.
4. ROCclr inserts the AQL packet into an HSA (heterogeneous system architecture) queue.
5. In hardware, an asynchronous compute engine (ACE) processes HSA queues, assigning kernels to compute hardware.

A kernel's journey to the GPU's computational hardware begins with the `hipLaunchKernelGGL` API call, which is shown at the top of Figure 3 and responsible for enqueuing a kernel-launch request. A programmer's typical point of contact with the queuing structure is through HIP's "stream" interface introduced in Section 2.1. Briefly restated, a HIP stream is one of several arguments a programmer may specify when calling `hipLaunchKernelGGL`. Each HIP stream is backed by a software queue managed by ROCclr,[6] the backend runtime library used by HIP (see Figure 2). ROCclr stores the arguments to `hipLaunchKernelGGL` in a C++ object, then inserts this object into the software queue.

HSA queues. Once a kernel-launch C++ object reaches the head of its software queue, ROCclr converts it into an kernel-dispatch AQL (Architected Queueing Language) packet. AQL packets are used to request single GPU operations, such as kernel launches or memory transfers.[7] In order to send the AQL packet to the GPU, ROCclr copies the AQL packet into an HSA

[6] This is largely defined in `platform/commandqueue.hpp` in ROCclr's source code.

[7] We do not cover memory-transfer requests further in this paper, but they follow the same queuing structure as kernel launches. Ultimately, memory transfers are dispatched to hardware "DMA engines" (Bauman et al, 2019) rather than asynchronous compute engines.

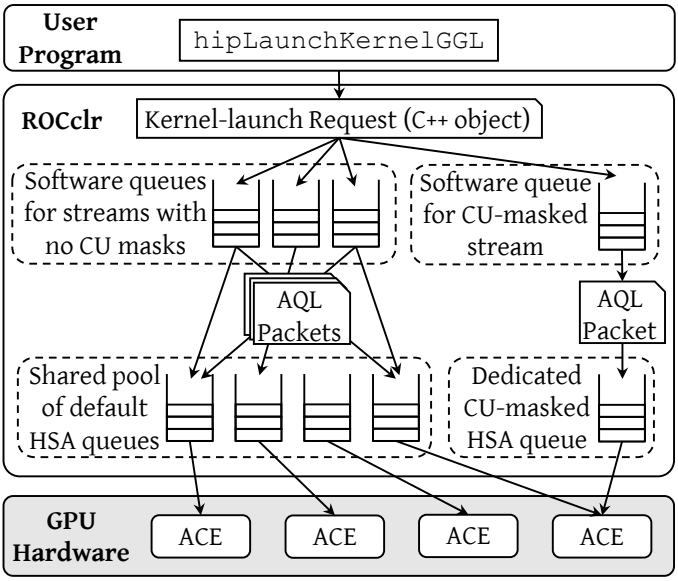


Fig. 3: Paths through ROCm's queuing structure.

(Heterogeneous System Architecture) queue. HSA queues are ring buffers of AQL packets, and are directly shared between the GPU and userspace memory. This direct memory sharing allows user programs to issue GPU commands without system calls (HSA Foundation, 2018b, Sec. 2.5).

It may seem like an intuitive choice to back each ROCclr queue with a dedicated HSA queue, but Figure 3 likely already revealed that ROCclr's behavior is more complicated. ROCclr's software queues internally share a pool of HSA queues: one software queue may submit work to multiple different HSA queues, and each HSA queue may contain work from multiple software queues. Even though sharing HSA queues may occasionally prevent concurrent kernel launches, it will not break any of the ordering guarantees behind the top-level "stream" abstraction. ROCm employs a combination of hardware (*i.e.*, "barrier" AQL packets) and software mechanisms (*i.e.*, ROCclr's software queues) to enforce the in-order completion of commands from a single stream.

Figure 3 depicts fewer ROCclr software queues than HSA queues, but this is just to save space in the figure. In practice, using a shared pool of HSA queues is intended to reduce the total number of HSA queues created by an application; even if there are dozens of HIP streams, ROCclr will still use the same small pool of HSA queues. In the default configuration of ROCm 4.2, the pool is limited to four HSA queues.[8] Understanding the reason for this limitation, however, requires traveling farther down the scheduling hierarchy.

[8]This, and related behavior can be observed by examining ROCclr's source code. For example, the `acquireQueue` function in https://github.com/ROCm-Developer-Tools/ROCclr/blob/master/device/rocm/rocdevice.cpp implements the functionality for selecting a single HSA queue from the pool of available queues.

4.2 Assigning Queues to GPU Hardware

As mentioned previously, the contents of HSA queues (*i.e.* kernel-launch packets) can be directly shared between user applications and GPU hardware, so driver code is not necessary required when launching any single kernel (this is why the driver is not shown in Figure 3). Even so, Linux's `amdgpu`[9] driver is required when initializing HSA queues and notifying the GPU of their existence. As such, driver code still maintains control over critical aspects of AMD GPU performance and reveals useful details about GPU scheduling internals, such as CU-masking specifics, discussed later in Section 5.1.

For now, though, we are concerned primarily with the functionality required to launch kernels. In the driver, this begins with the initialization of an HSA queue. Even here, a large portion of queue-creation logic is handled in ROCm's userspace code: the HSA API layer shown in Figure 2 is actually responsible for reserving the ring buffer for the HSA queue, setting up OS signals, *etc.*, via `mmap` and other standard Linux system calls.[10] Nonetheless, the driver must remain responsible for communicating this information to the hardware. Internally, it does so by populating a data structure called a *memory queue descriptor* (MQD), which includes the virtual address of the HSA queue's buffer, along with other metadata. MQDs are so named because they are allocated from GPU-accessible regions of CPU memory. In order for the GPU to actually start running work from the queues, however, MQDs must be assigned to *hardware queue descriptors* (HQDs) on the GPU itself.

In the default configuration for our test system, the `amdgpu` driver notifies the GPU about new queues by sending a *runlist* to the GPU—a buffer containing a list of all the MQDs on the system.[11] Interestingly, the act of "sending the runlist" itself requires writin the runlist to a special queue of GPU commands, known in driver code as the *HIQ* (HSA interface queue). The driver creates one HIQ for each GPU in the system, and, unlike HSA queues created in userspace, this queue of commands is mapped into kernelspace memory and is manually assigned to GPU hardware, allowing it to be initialized without needing to be part of the runlist itself.

Queues' arrival in hardware. Figure 4 gives a rough representation of the GPU hardware involved in compute workloads. As shown in Figure 3, *Asynchronous Compute Engines* (ACEs) are the hardware units responsible for processing the queues of AQL packets. Given its focus on kernels rather than queues, Figure 3 does not include the process by which MQDs are assigned to HQDs in the first place. Much of this process has little influence over the specific results shown in Section 3, but it still bears some explanation, as

[9] When ROCm was first introduced, compute-specific code for AMD GPUs was instead in a separate `amdkfd` driver. It was merged into the `amdgpu` driver in of version 4.20 of the Linux kernel. While not particlularly relevant to this paper's content, this distinction may be useful when consulting some of the older reference material we cite.

[10] For example, the `AqlQueue` constructor and related functions in the `ROCR-Runtime` library's `core/runtime/amd_aql_queue.cpp` source file are responsible for much of this low-level logic.

[11] As of Linux 5.14.0-rc3, source code for runlist construction is mostly contained in `drivers/gpu/drm/amd/amdkfd/kfd_packet_manager.c` in the Linux source tree.

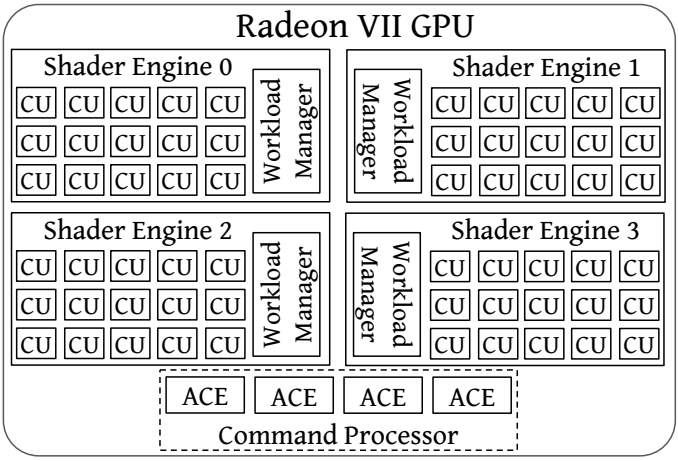


Fig. 4: The Radeon VII's compute-related components.

it ultimately illuminates the reason for ROCm's aforementioned attempt to reduce the number of active HSA queues.

After receiving a runlist from the driver, firmware in the GPU's top-level *command processor* uses *hardware scheduling* (HWS) to assign MQDs to HQD "slots" in the ACEs.[12] The command processor contains four ACEs, and each ACE can support up to eight queues.[13] Knowing this, we can finally explain why ROCm attempts to limit the number of HSA queues used by any single application: the GPU hardware only supports up to 32 concurrent queues—eight queues on each of the four ACEs. Creating queues in excess of this limit leads to the GPU entering "oversubscription" behavior, which time-slices between the queues mapped to the 32 available slots. During oversubscription, queues are swapped without any sort of prioritization, meaning that queues with active work may even be swapped out of a HQD slot in favor of an empty queue.

Naturally, this can lead to profoundly poor performance. To forestall any premature conclusions, though, oversubscription is not actually one of the "worst practices" affecting the data in Table 1. Puthoor *et al.*, AMD researchers, have already thoroughly investigated oversubscription in a prior publication (Puthoor et al, 2018), to which we refer readers interested in more details on the topic. Unfortunately, even Puthoor *et al.* resorted to a simulator when implementing alternative, more "intelligent," approaches for multiplexing queues among the 32 HQD slots, leaving little chance that researchers unaffiliated with AMD could effectively conduct similar research

[12]This is described in a comment in `drivers/gpu/drm/amd/include/kgd_kfd_interface.h` in the Linux 5.14.0-rc3 source tree.

[13]This can be confirmed in Linux 5.14 sources by observing where `num_pipe_per_mec` and `num_queues_per_pipe` are set in `drivers/gpu/drm/amd/amdgpu/gfx_v9_0.c`. Note that ACEs are typically called "pipes" in AMD's source code (bridgman, 2016).

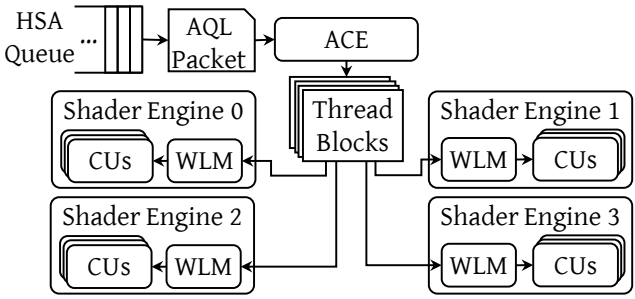


Fig. 5: Hardware involved in dispatching blocks to CUs. (This figure abbreviates "Workload Manager" as WLM.)

on real hardware. Instead, in a real-time setting, we would recommend ensuring oversubscription is not a problem by disabling it: the `amdgpu` driver can easily be configured to enforce such a policy,[14] which will ultimately return errors to userspace if too many HSA queues are created.

How CU masks affect the queuing hierarchy. Internally, the GPU hardware associates a single CU mask with each HSA queue. If no mask is explicitly set, HSA queues default to allowing work to execute on any CU. Recall from before that HIP streams generally are backed by ROCclr software queues, which in turn submit work to a shared pool of HSA queues. The HSA queues in this shared pool are all created in the default configuration, and are therefore not suitable for use by any HIP stream that requires a non-default CU mask. In order to support CU masking, ROCclr follows a slightly different code path when handling a HIP stream with a CU mask, shown on the right side of Figure 3: it creates a separate HSA queue with the requested mask,[15] and uses the new HSA queue exclusively on behalf of the single stream.

4.3 Scheduling Thread Blocks

We now describe how a kernel at the head of an HSA queue gets assigned to computing hardware. Recall that thread blocks are the basic schedulable entity for GPU computations, so when kernel-dispatch AQL packets reach the heads of their queues, the question becomes how the GPU decides which blocks to run, and where to run them. Figure 5 essentially continues the kernel-launch process after the end of Figure 3, from the perspective of an ACE handling a single HSA queue. To simplify Figure 5, we only included a single HSA queue and a single ACE. If multiple HSA queues are assigned to the same ACE, the ACE alternates between dispatching packets from the head of each queue in a time-sliced round-robin fashion.

[14] This is configured by the `sched_policy` parameter to the `amdgpu` driver, defined in `drivers/gpu/drm/amd/amdgpu/amdgpu_drv.c`.

[15] This behavior can be observed in the `acquireQueue` function defined in ROCclr's `device/rocm/rocdevice.cpp` source file.

Dispatching blocks to shader engines. One of the more prominent features in Figures 4 and 5 is the division of the GPU's compute resources into four *shader engines* (SEs). As illustrated in Figure 5, the primary role of an ACE is to dispatch blocks from the kernel at the head of an HSA queue to the SEs. However, without a prior explanation of the reasons underlying certain design decisions, the ACE's behavior when dispatching blocks to SEs may seem bizarre. To forestall such confusion, we first describe a thread-ordering guarantee made by the HSA specification, which AMD implements in their GPU-compute architecture (HSA Foundation, 2018a, Sec. 2.13).

The HSA specification states that it must be safe for GPU threads to *wait* for the completion of any GPU threads with a lower block index, *i.e.*, the value provided by `blockIdx.x` in Figure 1. Technically, block indices are three-dimensional tuples, so the HSA specification actually states its guarantee in terms of a block's *flattened ID*, which takes into account the fact that the special `blockIdx` variable is three-dimensional. For example, it must be safe for threads in block 1 (specifically, the block with a flattened ID of 1) to wait for threads in block 0 to complete, but it may be unsafe for a threads in block 0 to wait for block 1's completion—block 0 could occupy resources needed by block 1, preventing block 1 from ever starting to execute. A block-ordering guarantee has a practical application: prior work formally proves that it enables producer-consumer relationships between blocks in a single kernel (Sorensen et al, 2018). Even though the experiments in this paper do not use such complicated kernel logic, the block-ordering guarantee plays an important role in scheduling, with significant performance ramifications.

AMD hardware enforces the block ordering when assigning blocks to SEs. The method is simple: ACEs *must* assign blocks to SEs in sequential order. For example, an ACE cannot assign block 1 to SE 1 until after it has assigned block 0 to SE 0.[16] Figure 6 illustrates this concept as the ACE dispatches four consecutive blocks to SEs. The cycle depicted in Figure 6 continues with block 5 being assigned to SE 0, and only ends after all blocks in the kernel have been dispatched. To use a metaphor: the block-dispatching behavior can be likened to a card game where a dealer is dealing cards to four players. As in most real-life card games, even one slow player may force the dealer to wait, slowing down the entire game! Nonetheless, it would ruin the game for the dealer to skip the slow player. When applying this to AMD GPUs, the "dealer" is the ACE, the four "players" correspond to the four SEs, and the "cards" are the blocks of a kernel. But what can cause an SE, the metaphorical "player," to be abnormally slow? The answer is intertwined both with CU masking, and with the behavior of the workload managers.

The role of workload managers. In order for an ACE to assign a block to an SE, the block must be assigned to a specific CU on that SE. As shown in Figures 4 and 5, assigning blocks to CUs is the job of a piece of per-SE hardware

[16] Our only source for this claim remains private correspondence, which indicated that hardware enforces this rule using a "baton-passing" mechanism between the SEs. Despite the lack of additional external support for this claim, it is certainly well-supported by our experiments, *i.e.*, Table 1 or Figure 9.

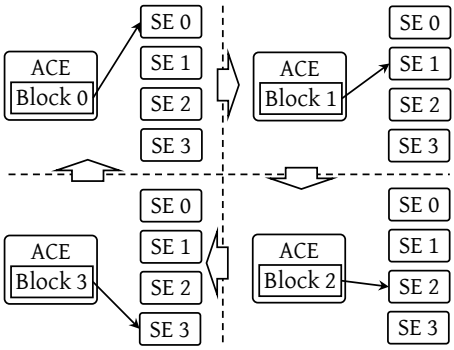


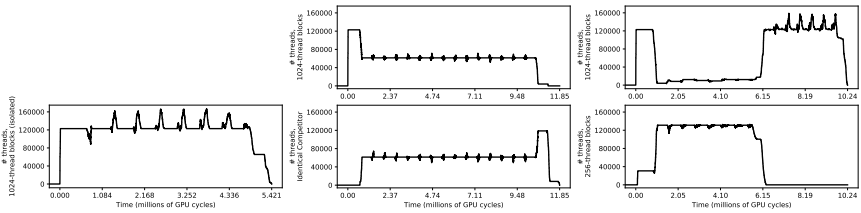
Fig. 6: A simplified diagram of an ACE's behavior when dispatching consecutive blocks to SEs.

called the *workload manager* (Bauman et al, 2019). Each workload manager has four dedicated "slots" for staging incoming blocks: one slot dedicated to each of the GPU's four ACEs. This design means that activity from one ACE cannot prevent another ACE from accessing the workload manager. Ideally, workload managers will assign blocks from each of these four slots to CUs in a round-robin manner.[17] This behavior changes, however, if no CU has sufficient available resources (*e.g.*, registers or threads (Aaltonen, 2017)) for a block from a particular slot. In this situation, the workload manager will allow blocks with smaller resource requirements to *cut ahead* if possible. This finally allows us to explain why, in Section 3, `MM256` was always more destructive to `MM1024`'s performance than any other configuration: each CU only supports a limited number of threads, and the completion of a 256-thread `MM256` block does not release enough resources to allow a block of `MM1024` to run. Instead, it merely allows another `MM256` block to cut ahead again, causing the problem to continue until no more `MM256` blocks remain!

Figure 7 illustrates this behavior on real hardware. The timelines correspond to the same workloads from the "Full GPU Sharing" configurations from the "`MM1024` (vs. `MM1024`)" and "`MM1024` (vs. `MM256`)" portions of Table 1. In order to generate the timelines, we instrumented our kernel code to use the `clock64()` function to obtain the start and end GPU clock cycle for every block of threads. After a kernel completes, we copy the block start and end times to the CPU and use them to compute the number of active threads at each GPU clock cycle. The timelines in Figures 7a, 7b, and 7c all cover a single kernel's execution for each respective task.

As expected, the kernel running in Figure 7a uses the GPU at near-full capacity for its entire duration. In prior work (Otterness and Anderson, 2020), we note that CUs appear to only run a maximum of 2,048 threads, despite AMD's documentation listing 2,560 as a theoretical limit. Figure 7a reaffirms

[17] Unfortunately, we also learned this from private conversation and were not able to find corroborating published material. Nonetheless, this claim is supported by the observations in Figure 7b.



(a) An isolated `MM1024` kernel.

(b) Two overlapping `MM1024` kernels.

(c) Overlapping `MM1024` and `MM256`.

Fig. 7: Comparison between timelines of matrix-multiply thread blocks in different configurations.

this observation, with an obvious plateau at 122,880 active threads corresponding to 2,048 threads running on each of the GPU's 60 CUs. The occasional spikes likely coincide with groups of blocks nearing the end of their execution, as, unlike on NVIDIA (Amert et al, 2017), our Radeon VII allows new blocks to start as soon as resources start to become available, even if the entire preceding block has not yet completed. To avoid the high overhead for tracking the start and end times of each individual thread, we only record the start of the first thread and end of the last thread in each block. However, we plot timelines as if all of a block's threads remain active so long as *any* thread in the block is active, leading to spikes where active thread or block counts are inaccurate. Fortunately, the behavior shown in Figure 7 is distinctive enough to be obvious even without perfectly tracking the active-thread count.

The cutting-ahead behavior is apparent when comparing Figures 7b and 7c. When two identical `MM1024` instances contend for GPU resources, Figure 7b shows that GPU computing capacity is divided evenly for the entire time that the two kernels overlap, corresponding to the workload managers dispatching blocks evenly from separate ACEs. Additionally, when the kernels do not overlap, the sole running kernel uses the full capacity. The contrast provided by Figure 7c is striking, where `MM1024` competes against an `MM256` kernel. Shortly after `MM256`'s kernel begins, it has taken sole control of virtually all GPU resources, with `MM1024` making practically no progress in the meantime.

4.4 Explanation of the Worst Practices in Section 3

"Cutting ahead" explains the destructive interference that `MM256` causes against `MM1024`, but the terrible performance of "Uneven Semi-Partitioning" in Section 3 is, perhaps unsurprisingly, better attributed to a poor choice of CU partitions.

With the prior explanation of block-SE distribution, this behavior is now easier to explain. Recall that the ACE will always distribute blocks to SEs in sequential order, and never skip an SE. There is one exception to this rule: the ACE will skip an SE only if a CU mask disables *all* CUs on that SE. In other words, blocks are evenly distributed among the set of SEs for which *any* CUs

are enabled. This approach to block-ordering enforcement can lead to extreme performance pitfalls: If only one CU is enabled on an SE, it is far more difficult for an ACE to assign a block to that particular SE, as it must wait for the single CU to be available. On top of this, the ACE is also prevented from "skipping" the slow SE and assigning blocks to the other SEs in the meantime! We exploited this behavior for our Section 3 experiments, by designing a CU mask that enables only one CU on an SE. Naturally, performance becomes even worse when the single CU is shared with a kernel where blocks can cut ahead: this is precisely what happens in Table 1 under "Uneven Semi-Partitioning" when `MM1024` competes against `MM256`.

5 Practical Applications

In this section we give some guidance for using CU masking in practice, remark on some obstacles that real-time research may encounter when using AMD GPUs, and discuss future work.

5.1 Usage of the CU-Masking API

The primary practical method for using CU masking on AMD GPUs is through the HIP API, so the relevant HIP API details bear some more explanation. In order to specify a CU mask for a HIP stream, programmers must use the `hipExtStreamCreateWithCUMask` function (HIP offers no function to set a CU mask for an existing stream, due to the HSA-queue multiplexing discussed in Section 4.1). In HIP, CU masks are specified as bit vectors using 32-bit integers, where set bits indicate enabled CUs and clear bits indicate forbidden CUs.

The hardware, however, does not use the same "flat" CU mask that a HIP programmer specifies, and instead requires a separate CU mask for each SE. In fact, the `amdgpu` driver is responsible for transforming the single user-provided CU mask into the per-SE masks. By examining the driver code,[18] we can discover the specific mapping. Figure 8 shows how bits in a HIP CU mask relate to shader engines and CUs in the GPU. The pattern in Figure 8 is simple: every fourth bit maps to a different CU in the same SE. In the example mask in Figure 8, the first of every group of four bits is set, defining a partition consisting only of CUs on SE 0.

Knowing the mapping between HIP's CU masks and SEs in hardware allows partitioning tasks to specific SEs, but it is not clear if doing so has benefits over an approach that distributes CUs evenly across SEs. In order to evaluate the possible benefits and drawbacks of limiting partitions to specific SEs, we contrast two basic partitioning approaches:

- *SE-packed*: Pack as many of the partition's CUs as possible into each single SE before starting to occupy CUs on an additional SE. This uses as few SEs as possible.

[18] In the source tree for Linux 5.14.0, this is found in the `mqd_symmetrically_map_cu_mask` function in `drivers/gpu/drm/amd/amdkfd/kfd_mqd_manager.c`.

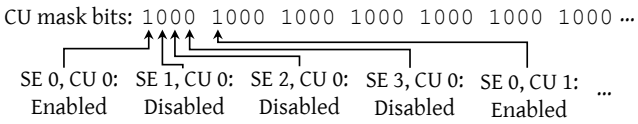


Fig. 8: The mapping of CU mask bits to SEs.

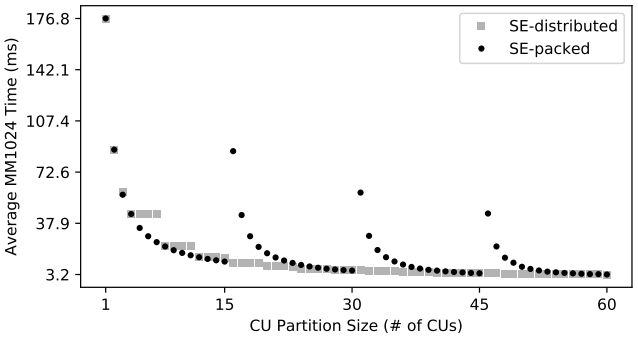


Fig. 9: Performance of CU-masking strategies for varying partition sizes.

- *SE-distributed*: Distribute the partition's CUs across all SEs as evenly as possible. This implies that any partition containing over four CUs will use all SEs.

Figure 9 shows the behavior of these two partitioning approaches when applied to an instance of `MM1024` running in isolation. For the most part, Figure 9 seems to show that there is little benefit to SE-packed partitioning: SE-packed visibly outperforms SE-distributed only for some partition sizes smaller than 15 CUs. Unsurprisingly, SE-packed performs far worse than SE-distributed when only one CU is enabled on an SE. The jumps in the SE-packed response times occur where expected: the Radeon VII has 15 CUs per SE, so SE-packed partitions with sizes of 16, 31, or 46 will end up occupying only a single CU on one of the SEs. Overall, SE-packed partitioning may look inferior in Figure 9, but Figure 9 is based on measurements taken without an important factor: contention.

CU partitioning in the presence of a competitor. We carried out a final experiment using an `MM1024` measured task facing an `MM256` competitor (as this results in the most destructive interference in the absence of partitioning). This experiment partially reuses data from Section 3, which used SE-packed partitioning. The main contribution from the new experiment is the inclusion of SE-distributed partitioning for contrast.

Table 2 shows the results of this experiment. For further illustration, Table 2 also includes the CU mask used by both the `MM1024` measured task and `MM256` competitor. For example, `MM1024`'s SE-packed CU mask sets every even-numbered bit, starting with bit 0, meaning that `MM1024` will occupy every CU on SEs 0 and 2, whereas `MM256`'s SE-packed CU mask causes it to occupy

Description	MM1024 CU Mask	MM256 CU Mask	Min	Max	Median	Arith. Mean	Std. Dev.
Full GPU Sharing (Unpartitioned)	1111...1111	1111...1111	12.214	19.578	15.503	15.652	1.114
SE-packed, Equal Partitions	1010...1010	0101...0101	6.784	7.630	6.944	6.991	0.133
SE-packed, Unequal Partitions	1010...1011	0101...0101	64.873	101.531	84.047	84.362	4.381
SE-distributed, Equal Partitions	1111...0000	0000...1111	6.391	9.113	7.250	7.269	0.074
SE-distributed, Unequal Partitions	1111...0001	0000...1111	7.109	7.906	7.288	7.324	0.122

Table 2: MM1024's response times in the presence of an MM256 competitor. All times are in milliseconds.

every CU on SEs 1 and 3. As we did in Section 3, we also include "Unequal Partitions" cases in Table 2, produced by adding a single CU to MM1024's "Equal Partitions" CU masks.

Observation 4. *For some partition sizes, SE-packed partitioning is slightly better than SE-distributed partitioning.*

Observation 4 is seen when comparing the two "Equal Partitions" rows from Table 2. Both partitioning approaches ensure that MM256 does not share CUs with MM1024, and therefore sufficiently prevent the poor unpartitioned performance due to MM256 cutting ahead (shown for convenience in Table 2's first row). However, SE-packed partitioning exhibits slightly better response times than SE-distributed, likely due to two factors. First, SE-distributed partitioning is actually unable to assign an equal number of CUs to all SEs, as a 30-CU partition is not evenly divisible among four SEs. Second, SE-packed CU masks likely prevent a small amount of contention for SE-wide resources such as workload managers.

Observation 5. *SE-distributed partitioning is vastly superior when a partition's size prevents it from occupying all CUs in an SE.*

Observation 5 is supported by comparing the "Unequal Partitions" lines in Table 2, where it should be apparent that it is highly undesirable to use an SE-packed partition containing 31 CUs. While it is hard to imagine a practical application where a task requires a partition containing exactly 31 CUs, one can certainly envision applications where greater flexibility in partition sizing could be useful. One such situation would be a task system requiring prioritization. For example, high-priority work may need a larger partition than low-priority work, but designating a full additional SE as "high-priority" could cause unacceptable adverse effects to low-priority performance. Instead, it could be better to allocate 40 CUs to high-priority work, while low-priority tasks use the remaining 20 CUs. With these partition sizes, Figure 9 indicates that SE-distributed partitioning would be better than SE-packed partitioning for both high- and low-priority tasks, with low-priority work seeing a particular benefit given the gap between SE-distributed and SE-packed performance at 20 CUs.

5.2 Other AMD-Specific Performance Concerns

While CU masking ended up being the largest cause of the poor performance in Section 3, it is certainly not the only pitfall with broader implications. For example, it is a common practice to directly port CUDA code to HIP Otterness and Anderson (2020). CUDA code cannot use CU masking, but the performance of multi-stream CUDA code may depend on implicit assumptions about other scheduling details. For example, on NVIDIA GPUs, smaller thread blocks will not cut ahead of larger blocks Amert et al (2017), but this is clearly not the case for AMD GPUs. Therefore, in order for research results to be portable between NVIDIA and AMD GPUs, the cutting-ahead behavior must be addressed and accounted for. In some cases it may be possible to modify source code and adjust the block dimensions. In other cases, however, source-code complexity or algorithmic details may make this impractical, meaning that CU partitioning or other scheduling methods must be applied to prevent cutting ahead, and performance assumptions must be re-tested.

Additionally, the multiplexing between HIP streams and HSA queues may be a concern for some task systems. The topic of HSA queue oversubscription is covered in prior work Puthoor et al (2018), but other tools can reduce the possibility for queue contention in the first place. For example, it is possible to use environment variables to adjust the number of shared HSA queues that ROCclr will create.[19] The ability to modify ROCm's source code enables still more possibilities, such as applying a process-wide CU mask to the shared HSA queues, or requiring a separate HSA queue for every HIP stream.

5.3 Future Work

In the future, we hope to continue investigating other features of AMD GPUs that we did not cover for this paper, such as the ability to preempt compute workloads. Our broader goal, however, focuses on solving the problems described in Section 1: we hope to produce a platform that can serve as a model for implementing and testing a variety of real-time GPU-management techniques. AMD GPUs' open-source software brings this goal closer to fruition, especially as we continue to learn new information such as what we present in this paper.

6 Conclusion

This work provides several clear demonstrations of the practical implications of knowing or failing to know key aspects of GPU-internal behavior. In doing so, we provide information about AMD GPUs using a combination of public (but poorly advertised) information, first-party source code, and experimental evidence. Additionally, we give the first published guidance on how to properly apply the CU-masking feature of AMD GPUs.

[19] Specifically, the `GPU_MAX_HW_QUEUES` variable.

Without knowing this information, the risks of disastrous performance pitfalls makes developing a reliable real-time system virtually impossible. But, from another view, *with* knowledge of this information, developers can expect predictable performance from their systems with a level of confidence that will not be present with a closed-source platform.

Declarations

- Work was supported by NSF grants CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

References

Aaltonen S (2017) Optimizing GPU occupancy and resource usage with large thread groups. Online at https://gpuopen.com/learn/optimizing-gpu-occupancy-resource-usage-large-thread-groups/

Amert T, Otterness N, Anderson JH, et al (2017) GPU scheduling on the NVIDIA TX2: Hidden details revealed. In: IEEE Real-Time Systems Symposium (RTSS)

Bauman P, Chalmers N, Curtis N, et al (2019) Introduction to AMD GPU programming with HIP. Presentation at Oak Ridge National Laboratory. Online at: https://www.olcf.ornl.gov/calendar/intro-to-amd-gpu-programming-with-hip/

bridgman (2016) amdgpu questions. Phoronix Forums. Online at https://www.phoronix.com/forums/forum/linux-graphics-x-org-drivers/open-source-amd-linux/856534-amdgpu-questions?p=857850#post857850. Accessed in 2020

Capodieci N, Cavicchioli R, Bertogna M, et al (2018) Deadline-based scheduling for GPU with preemption support. In: IEEE Real-Time Systems Symposium (RTSS)

Fujii Y, Azumi T, Nishio N, et al (2013) Exploring microcontrollers in GPUs. In: Asia-Pacific Workshop on Systems (APSys)

HSA Foundation (2018a) HSA programmer's reference manual: HSAIL virtual ISA and programming model, compiler writer, and object format (BRIG), version 1.2. Online at http://hsa.glossner.org/wp-content/uploads/2021/02/HSA-PRM-1.2.pdf

HSA Foundation (2018b) HSA runtime programmer's reference manual, version 1.2. Online at http://hsa.glossner.org/wp-content/uploads/2021/02/HSA-Runtime-1.2.pdf

Jain S, Baek I, Wang S, et al (2019) Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)

Jia Z, Maggioni M, Staiger B, et al (2018) Dissecting the NVIDIA volta GPU architecture via microbenchmarking. CoRR abs/1804.06826. URL http://arxiv.org/abs/1804.06826, https://arxiv.org/abs/arXiv:1804.06826

Jia Z, Maggioni M, Smith J, et al (2019) Dissecting the NVidia turing T4 GPU via microbenchmarking. CoRR abs/1903.07486. URL http://arxiv.org/abs/1903.07486, https://arxiv.org/abs/arXiv:1903.07486

Kato S, Lakshmanan K, Rajkumar R, et al (2011) TimeGraph: GPU scheduling for real-time multi-tasking environments. In: USENIX ATC

Larabel M (2020) The AMD Radeon graphics driver makes up roughly 10.5% of the Linux kernel. Phoronixcom Online at https://www.phoronix.com/scan.php?page=news_item&px=Linux-5.9-AMDGPU-Stats

Mei X, Chu X (2016) Dissecting GPU memory hierarchy through microbenchmarking. IEEE Transactions on Parallel and Distributed Systems (TPDS)

NVIDIA Corporation (2020) NVIDIA multi-instance GPU user guide. Online at https://docs.nvidia.com/datacenter/tesla/mig-user-guide/

Olmedo IS, Capodieci N, Martínez JL, et al (2020) Dissecting the CUDA scheduling hierarchy: A performance and predictability perspective. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)

Otterness N, Anderson JH (2020) AMD GPUs as an alternative to NVIDIA for supporting real-time workloads. In: Euromicro Conference on Real-Time Systems (ECRTS)

Peres M (2013) Reverse engineering power management on NVIDIA GPUs-anatomy of an autonomic-ready system. In: Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)

Puthoor S, Tang X, Gross J, et al (2018) Oversubscribed command queues in GPUs. In: ACM Workshop on General Purpose GPUs (GPGPU)

Sorensen T, Evrard H, Donaldson AF (2018) GPU schedulers: How fair is fair enough? In: International Conference on Concurrency Theory (CONCUR), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik

Yang M, Amert T, Yang K, et al (2018) Making OpenVX really 'real time'. In: IEEE Real-Time Systems Symposium (RTSS)

Appendix A: Detailed Kernel-Launch Behavior

Section 4.1 attempts to describe ROCm's kernel-launch behavior in human-readable terms, but researchers attempting to build modifications on top of existing ROCm code may benefit from a more detailed description, including more specific source-code references. This appendix assumes that the reader is already familiar with the content discussed in Section 4. While this is not necessary to understand the main body of our paper, we have elected to include it as a reference for researchers hoping to modify or work with AMD's ROCm software stack.

Figure 10 summarizes the kernel-launch process in greater detail. Each component of the flowchart in Figure 10 contains four pieces of information: the name of a function in ROCm's source code, a (brief) comment describing the purpose of the function, the ROCm component in which the function is defined, and the specific source-code file within the component containing the function's definition.

As shown in Figure 2, there are several ROCm components, all of which are open source. The source code for each relevant component is available online: HIP,[20] ROCclr,[21] the HSA runtime[22] and low-level driver interface (ROCT-Thunk-Interface).[23] This paper was based on ROCm version 4.2, but continues to apply to ROCm 4.3 (current at the time of submission), and has remained relatively stable since ROCm version 3.7.

Figure 10 follows the process outlined in Figure 3, but with a greater level of detail. As discussed in Section 4.1, kernel-launch requests are first enqueued in a userspace C++ `HostQueue` object, and eventually converted into an AQL packet and inserted into an HSA queue. One new detail shown in Figure 10 is that the conversion from a `HostQueue` entry into an AQL packet is carried out by an asynchronous thread, *i.e.*, a thread other than the one that called `hipLaunchKernelGGL`. While unsurprising, given the asynchronous behavior expected when launching kernels, this may be an important detail when designing real-time systems, as such a thread may block any other thread waiting for kernels to complete.

Curious readers may notice that Figure 10 does not cover queue creation in detail. Instead, we provide a deeper explanation of queue creation, including the code responsible for assigning queues to GPU hardware, in Appendix B. Note that the `Stream::Create` block occurs in both flowcharts, giving an indication of where, if necessary, queue creation will take place in the overall kernel-launch process.

[20] https://github.com/ROCm-Developer-Tools/HIP/tree/rocm-4.2.0
[21] https://github.com/ROCm-Developer-Tools/ROCclr/tree/rocm-4.2.0
[22] https://github.com/RadeonOpenCompute/ROCR-Runtime/tree/rocm-4.2.0
[23] https://github.com/RadeonOpenCompute/ROCT-Thunk-Interface/tree/roc-4.2.x

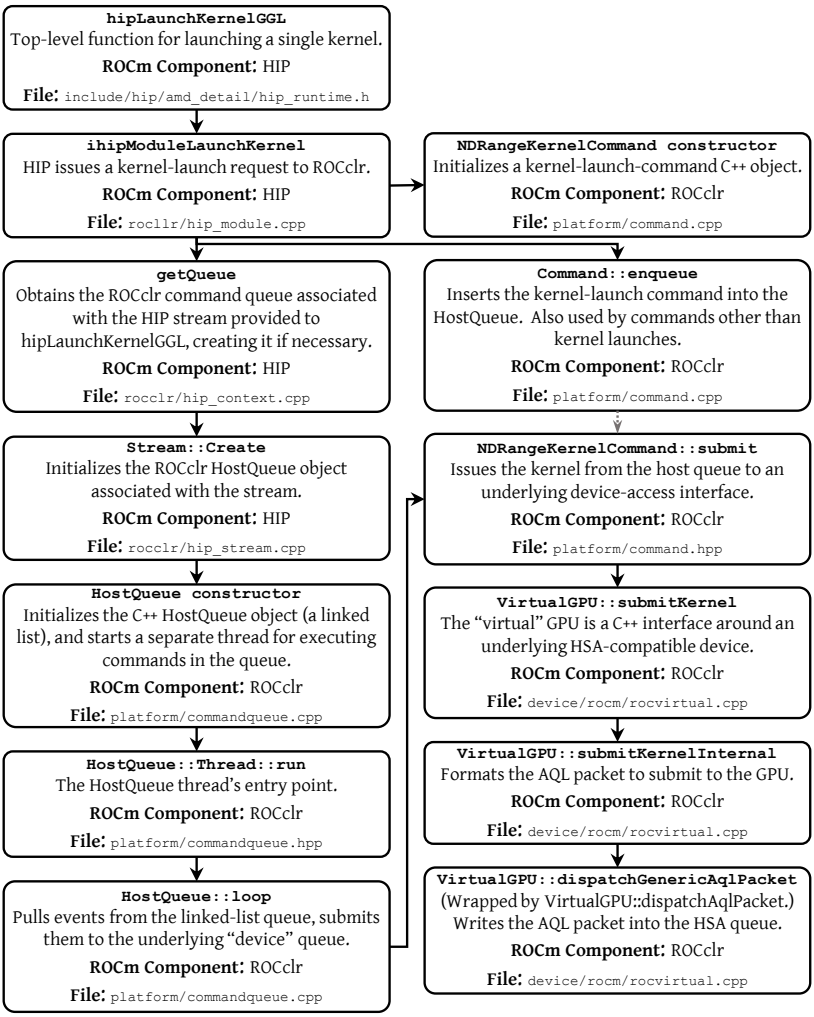


Fig. 10: Overview of ROCm source code involved in HIP kernel launches.

Appendix B: Detailed Queue-Handling Behavior

As with Appendix A, this appendix is intended for readers already familiar with the material in Sections 4.1 and 4.2. Figure 11 presents a more detailed view of the ROCm source code required to create queues and assign them to GPU hardware. Once again, this information is intended for researchers hoping to work with AMD's code, especially those hoping to modify AMD's GPU queue management at either the user level or within the driver.

As discussed in the main body of the paper, kernel launches on AMD GPUs do not require driver intervention, so Figure 10, in Appendix A, did not include any driver code. This is unfortunately not the case with the more complicated logic behind creating queues in the first place, meaning that Figure 11 also must include portions of AMD's driver code, located within the Linux kernel. In addition to labeling every individual component in Figure 11 with their respective ROCm components, driver portions of the flowchart are also enclosed in the dashed rectangle. Unfortunately, the full "File" path for the driver components of the flowchart is too long to cleanly fit in the flowchart, so we instead note here that all of the paths given in the "`amdgpu` Driver" boxes are located under the `drivers/gpu/drm/amd` directory in the Linux 5.14 source tree.

When casually observing ROCm's code, it may initially be difficult to discern where HSA queues are created. The key point is the `createVirtualDevice` function, called when ROCm starts a new thread to process a HIP stream's kernel launches. The "virtual device" is, in reality, a C++ interface granting access to a GPU; many virtual devices may be associated with a single underlying GPU.

An interesting characteristic of Figure 11 is the presence of the function names with the `_cpsch` postfix in the driver code. The "cpsch" postfix stands for **C**ommand **P**rocessor **Sch**eduling, referring to the use of HWS (discussed in Section 4.2). Alternative versions of these functions (postfixed `_nocpsch`) can also be found in kernel code, and are used when HWS is disabled. Internally, the driver is able to alternate between different versions of such functions by calling them indirectly, using a list of function pointers.

Similarly to how some functions in Figure 11 depend on whether or not HWS is enabled, some functions may change depending on GPU architecture. These functions are likewise called indirectly, via lists of function pointers. We only include one such architecture-specific function in Figure 11 (the `init_mqd` function), but several more are used for still-lower-level details, such as populating the contents of the runlist packet or the unmap-queues request.

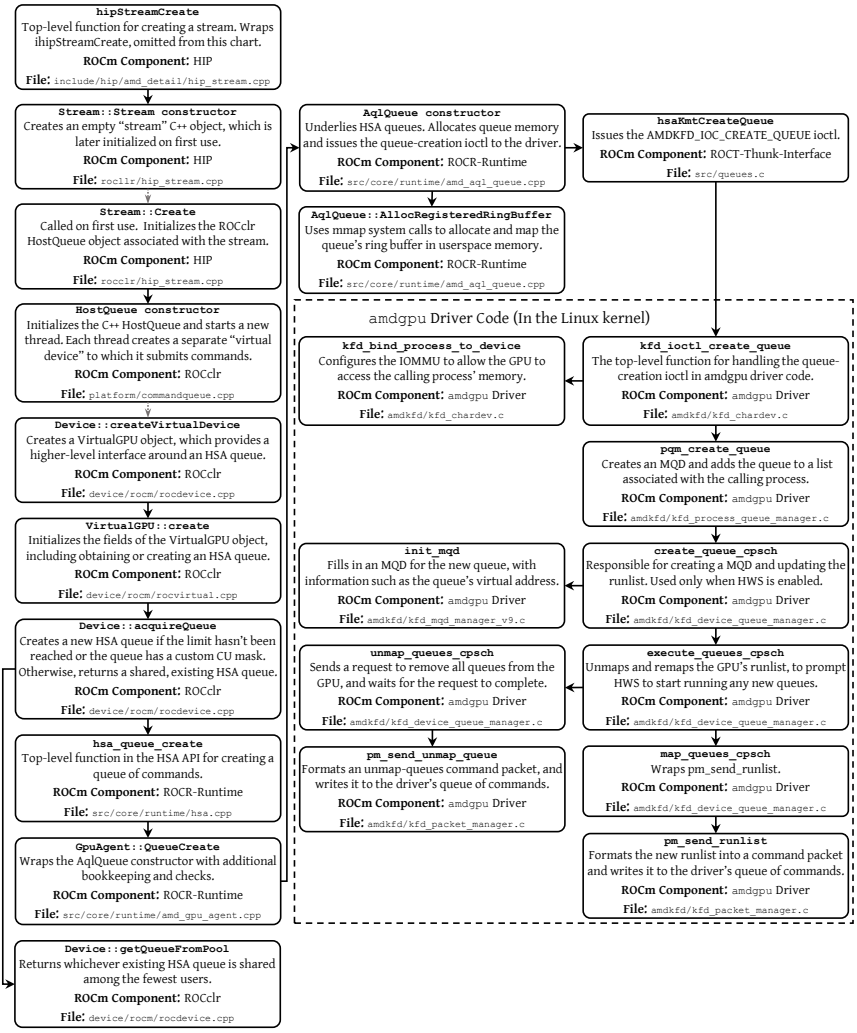


Fig. 11: Overview of ROCm source code involved in creating GPU queues.