Capturing & Modifying GPU Binaries in User-mode

By

Charles Anthony Rozhon

Thesis

Submitted in partial satisfaction of the requirements for the degree of

Master of Science

in

Computer Science

in the

Office of Graduate Studies

of the

University of California

Davis

Approved:

John D. Owens, Chair

Jason Lowe-Power

Venkatesh Akella

Committee in Charge

2023

Dedicated to my parents

Contents

List of Figures

List of Tables

Abstract

Capturing & Modifying GPU Binaries in User-mode

Binary instrumentation refers to adding additional instructions to an executable in order to measure or instrument some aspect of the program, such as adding instructions that count the number of times a certain instruction is executed. This thesis argues that GPU binary instrumentation and rewriting can be accomplished through user-mode tools without driver support by presenting a design and implementation of a binary instrumentation tool for AMD's HIP runtime. We compare our design to similar tools such as PIN for CPUs and NVBIT for NVIDIA GPUs. We provide example applications for instrumenting GPU code and give an analysis of our design decisions. This thesis evaluates the quality of the instrumentation applications and discusses challenges that arose during their implementation as well as evaluating the costs of the system itself through measurements of rewrite and instrumentation times. Finally, we give recommendations for designing future binary instrumentation systems without driver or kernel support.

Acknowledgments

First, thanks to Professor John Owens for his guidance and support over many weekly and individual meetings. His patience, experience, and skill are all noteworthy.

Thank you to Timour Paltashev and AMD for graciously funding this research, meeting with me often, and providing valuable connections to people across the company whenever I had questions or issues. Thanks as well to those from AMD who attended our meetings, such as Trinayan Baruah, Brian Sumner, and others.

Thank you to my fellow labmates, Teja Aluru, Daniel Loran, Cameron Shinn, Muhammad Osama, Toluwa Odemuyiwa, Agnieszka Łupińska, Jonathan Wapman, Yuxin Chen, Afton Geil, Jason Mak, Vehbi Eşref Bayraktar, Collin McCarthy, Ahmed Mahmoud, Zhongyi Lin, Radoyeh Shojaei, Mythreya Kuricheti, Mythreya K., Annie Robinson, Chenfei Yao, Muhammad Awad, and Kerry Seitz. I am especially grateful to our wonderful postdocs Serban Porumbescu and Matthew Drescher.

Thank you to everyone else at UC Davis, especially my committee members, Professor Jason Lowe-Power and Professor Venkatesh Akella, and the wonderful administrative staff, especially Jessica Stoller and Alyssa Bates. I would also like to thank my teachers here at UC Davis, especially Professor Nelson Max and Professor François Gygi, whose courses were memorable and inspiring.

And thank you to my wonderful aunts, uncles and grandparents who raised and supported me, especially Gayle Wright and Michele Welsh. I would also like to thank my friend Surtai Han, who has always been able to offer invaluable technical advice and feedback.

Chapter 1

Introduction

1.1 Background and Motivation

Today's GPUs are essential for applications in machine learning, scientific computing, and are increasingly used for variety of general purpose computing tasks. As programmers use GPUs more often and encounter performance problems that are unique to these systems, it will be useful to implement profiling tools that can find these specific problems.

Programs compiled for modern GPU accelerators use common APIs such as NVIDIA's CUDA or AMD's HIP to launch compute kernels, manage memory, and interact with the driver. Almost all GPU-accelerated programs use these common APIs since some amount of driver support is required for these tasks. This provides an opportunity—the behavior of almost any program can be captured without additional support from the driver. Many methods exist for "hooking" C library calls, and these mechanisms can be used to capture and modify the launched GPU programs. And this is possible without access to the original source code and without additional driver support. One application of capturing and modifying binaries in this way is binary instrumentation. Binary instrumentation refers to rewriting an executed binary with additional instructions whose purpose is the instrumentation of some part of the program. This can be used to collect additional information about the program's behavior or performance, much like a profiler. But unlike a profiler, instrumentation can give information about "in-the-moment" performance. A profiler may tell you that that 1% of threads experienced divergence—an instrumentation

tool can tell you which threads those were and give you tools to find them. Some other examples of this include:

- Adding timers to between individual lines of executed code. This allows a programmer to measure the time a specific operation takes without necessarily having to recompile the original program.
- Identifying memory access behavior and layouts. By adding instructions that log memory addresses before they are accessed, the programmer can look for certain access patterns they want to avoid or get a sense of how objects are laid out in memory.
- Injecting specific types of behavior or errors. For example a floating point error could be introduced to determine how well the program handles the error when it propagates to results.

Binary instrumentation offers a great deal of flexibility in the types of profiling tools that can be created. But this comes with many tradeoffs. It typically requires greater knowledge of the system being profiled since modifications are being made to compiled code, and the performance impact of profiling is typically much greater. But since instrumentation can be added dynamically, there's greater flexibility over the granularity of profiling. If the instrumentation is only needed in some rare scenarios or only for a subset of the system, its impact won't be noticed as much.

This thesis argues that a binary instrumentation tool can be built with tools available to any user due to how GPU programs are executed on modern systems. Hardware manufacturers could also implement this functionality by adding it to their driver implementation, but we will show that this isn't strictly necessary. Thus anyone can create a tool to add instrumentation to their binaries without depending on additional support from the driver.

The driver is a program that executes user-compiled programs and communicates with the device on behalf of the programmer. Rather than compiling a GPU binary and

configuring the device for its execution, users typically write GPU programs in "single-source" programs that consist of both CPU and GPU code. These programs are compiled so that the binary code for both programs is contained in a single file. The driver is then responsible for untangling these two programs and launching the GPU program when needed. This design is very powerful and easy for newcomers to grasp, but it makes observing and interacting with GPU binaries more difficult. Command line tools exist to untangle the binaries yourself, but it's often not clear which parts of those binaries are being used or how a separated binary can be used.

We designed a system that can capture these GPU binaries using standard user-space functionality on the Linux operating system. We could then modify the binaries before they are launched, changing the behavior of a GPU program. We utilized this rewriting functionality to implement a system that instruments the GPU binaries. We argue that rich instrumentation and profiling techniques are available by simply capturing the contents of GPU API calls—no additional driver or operating system support was required to implement new profiling tools.

We designed this system using AMD's HIP API for GPU computing. In contrast to NVIDIA's driver, the source for AMD's driver and compiler is available online. While we didn't have to modify any of this code, we needed it to determine the format of some structures and behavior of certain API calls. Thus while we did not need driver support, information about the driver was essential to modify the binaries. Still, we claim that with a minimal amount of information (which we will highlight when relevant) GPU programs can be captured and modified in a relatively simple fashion.

We built a proof-of-concept instrumentation tool and several different instrumentation functions for AMD's CDNA architecture using our modification system. These instrumentation functions allow a programmer to do things like determine the location of bank conflicts or uncoalesced accesses, and detect how often certain instructions were executed—things that are helpful for obtaining ideal GPU performance. We found that an instrumentation tool has several unique advantages over profilers and that when used in combination with a profiler, the programmer can "zero in" on locations of interest for

performance. In GPU programs which launch millions of threads, this is incredibly valuable. If a single workgroup is slowing down the system, that workgroup can be easily identified. When improving the performance of a often-running machine learning kernel, these small optimizations can quickly add up to major cost savings in training.

1.2 Contributions

We present the following contributions:

1. a system for capturing, replaying, and modifying AMD GPU binaries at runtime and without source code.

2. a method for performing binary instrumentation using our capture and modification system and proof of concept tools for instruction counting and memory tracing.

3. a comparison of our design with other instrumentation tools that require additional driver support or compiler engineering

4. a discussion of future work and uses for a GPU binary instrumentation system.

1.3 Thesis Organization

NVIDIA has also released a tool that can instrument arbitrary SASS Assembly binaries. In Chapter 2, we'll discuss this tool and others built by Intel and ARM for the same purpose. We'll also discuss the various applications of binary instrumentation tools and consider the potential for future tools. In Chapters 3, 4, and 5 we describe the implementation of our system. We'll also discuss the differences between our design and one that relies on driver support, such as NVIDIA's. We'll use this as an opportunity to compare some in-depth details of the hardware and show how these decisions influence the differences in our designs.

Chapter 3 focuses on our method for capturing the binaries and modifying the files. Chapter 4 dives into how a binary can be rewritten using this modification, accounting for things like instruction formats and an increase in the number of registers needed.

And Chapter 5 discusses the implementation of the individual instrumentation tools, explaining how instruction counting or a memory trace are actually implemented on AMD's architecture. We also present results of our instrumentation tools in action and compare them with the equivalent NVIDIA instrumentation tool along with measurements of rewriting costs and the impact of instrumentation on the final GPU code.

In Chapter 6, we'll evaluate our overall system, comparing it with the NVIDIA equivalent and CPU instrumentation tools when possible. We discuss the tradeoffs of the technical and design decisions we made, and illustrate their impact on the ease of use and quality of our system. We discuss where additional engineering effort can mitigate issues with our design and discuss what steps need to be taken to address any flaws. And finally, in Chapter 7, we offer lessons learned from our implementation and provide directions for future work to take. We hope that these insights, in addition to the tools that we have built, will provide others with the information necessary to create an instrumentation tool for their architecture. Or potentially, to improve our instrumentation tools for AMD's GPU architecture.

Chapter 2

Related Works

To discuss our contributions, we compare ourselves to several other tools for alternative architectures that accomplish the same goals. Both CPU and GPU tools exist for modifying and rewriting binaries for instrumentation. The CPU-based tools have a longer history than GPU-based tools, but there is a large amount of literature on the GPU side as well. We also discuss applications of instrumentation, such as fault tolerance or program analysis.

2.1 CPU Instrumentation Tools

2.1.1 Intel

The most notable binary instrumentation tool is Intel's PIN [8]. PIN allows a programmer to iterate over a series of instructions in a target binary and register callbacks at different levels of granularity such as every instruction or every function call. Notably, this process is architecture agnostic—it doesn't require the programmer to know anything about the actual instruction set. A program that uses this API is known as a *Pintool*. Pintools are inserted into the original executable using just-in-time (JIT) compilation. Specifically, PIN generates a sequence of instructions that includes the instrumentation, then "compiles" this sequence to a more efficient representation. Anytime a branch is encountered, control is given to PIN first before the executable continues execution. In this way, PIN is able to add the additional instructions required for a Pintool into the original stream of instructions.

This JIT compilation approach is very powerful. It allows optimization to be performed, and when extra registers are required, registers can be reallocated or rearranged. This means that PIN does a lot of book-keeping by keeping track of jump targets and relative offsets. They use a virtual machine to track these targets and have several heuristics that can skip using the VM and save performance. The system generates traces of instructions and then uses the JIT compiler to optimize this trace or sequence. This idea comes from HP's Dynamo [1], a system for optimizing native binaries with a runtime JIT compilation system. It was extended into DynamicRIO [2], which uses this technique to profile and instrument code. It is worth noting that this JIT compilation technique is complicated to map to a GPU execution system. A GPU driver expects to know how many registers a kernel will require before it ever executes, and in the SIMD execution model, it is inefficient to have one thread responsible for generating and compiling instructions while other threads do nothing. Our work does not use a VM to represent the state of the GPU and instead executes instructions directly. This is simpler, but it gives less power and flexibility for modifying and optimizing the generated code.

Both PIN and DynamicRIO contain a whole suite of profiling and debugging tools. Thread sanitizers, memory analysis, and trace generation are just some of the applications available. These tools have been used to simulate cache behavior, detect memory errors or multi-threading issues, and for various security applications. We do not have as rich of a programming interface as PIN, so we don't support writing nearly as many applications. In many cases we could improve our API to support some features, however. We discuss our programming model and its limitations in Chapter 6.

2.1.2 ARM

Similar tools have been written for embedded architectures [4, 5]. They use a similar JIT compilation system, modifying every branch so that the VM remains in control. In ARM the PC can be used as a source or destination register in any instruction, so extra care has to be taken for each branch. The applications of such a tool are similar to PIN, but with the additional mention of compatibility—an instruction can be rewritten for a new version of the architecture, allowing the original binary to work on the new system

without recompiling.

2.2 GPU Instrumentation Tools

2.2.1 NVIDIA

NVBIT [12] is, like us, a tool for instrumenting arbitrary GPU binaries. Like PIN, NVBIT includes several tools like instruction counting and memory tracing. These tools were mostly designed for hardware simulators and architecture designers. NVIDIA's focus was on providing a system that allowed the programmer to write their own instrumentation functions, so a lot of care was taken to implement a simple interface for writing your own instrumentation. Our interface requires the programmer to manually insert instructions for instrumenting the target binary, but they simplify the process through a simpler API and more thorough integration with the build and compilation systems. NVBIT allows you to compile an instrumentation function separately and just provide the compiled version, whereas we have to still hand-modify this assembly before use.

NVBIT also does not utilize the JIT compilation or tracing approach of PIN and other instrumentation tools. Instead, the GPU binary is modified before it is launched on the device. Instrumentation points are overwritten with a branch instruction to a trampoline before then executing the instrumentation function. Thus there's no tracking of branch targets or other additional bookkeeping required. NVBIT is integrated with the driver, which is responsible for creating a new binary from the original code and instrumentation functions. Since they are part of the driver, they can utilize additional compilation functionality and obtain better performance for the resulting code. It also allows them to pick and choose which kernels to rewrite at runtime, rather than rewriting everything ahead of time.

Many authors went on to publish their own tools that built off the NVBIT base. For example, NVBITFi [11] and GPU-FPX [7] are tools that use NVBIT to detect floating point exceptions and the instructions that cause NaNs to percolate through the system. These tools are a very effective of use instrumentation because there's no need to be very fast or performant.

2.2.2 Intel

Intel takes a very different approach to perform binary instrumentation on GPUs [6, 10]. Rather than add custom GPU instructions while the GPU is executing, they only add enough instrumentation to generate an instruction trace. They then emulate that instruction trace on the CPU, adding any additional profiling and instrumentation on top of the emulation process. This greatly simplifies designing a system for instrumentation because they don't have to care about inserting arbitrary snippets of code anywhere in the GPU binaries. Instead, they only need to consider how to generate their instruction traces and once they are emulating these instruction traces on the CPU, it is trivial to add additional instrumentation. The added instrumentation works at the level of the emulator, not the underlying architecture. This makes it considerably simpler to add new tools at the cost of an even larger increase to instrumentation time. It also requires an existing implementation of the GPU emulator, which is not a trivial engineering task.

Chapter 3

Binary Capture & Replay

This thesis began with the observation that every GPU program accesses a driver API in order to allocate memory and launch kernels. Since this driver API is implemented as a Linux shared library, it can be "hooked", or redirected, to an alternative implementation using the `quick` environment variable. Intended for use by profilers, debuggers, and other system utilities, `LD_PRELOAD` is often used for these kinds of tasks. Through hooking the API, we can capture, modify, or replay the API calls executed by a GPU binary. In this chapter, we discuss how we accomplished this capture process, how the data we capture can be modified before being used, and how we can use this to replay GPU binaries without the original executable. We also discuss the challenges of our design and its limitations and compare our design with alternative methods for capturing the same data.

3.1 Design

To capture and replay a GPU binary we

1. overload shared library calls.
2. write the program binary to a file when `__hipRegisterFatBinary` is called.
3. record kernel name and arguments for every call to `hipLaunchKernel`.
4. record all allocations and frees (`hipMalloc` and `hipFree`).

With this, we can reconstruct the same GPU behavior without access to the original program.

Every HIP program consists of a series of calls to library functions implemented in a shared object. We use `LD_PRELOAD` to overload a function's definition with our own implementation. This implementation records some information about the API call (like the arguments) and then calls the actual implementation in the HIP library. HIP programs consist of functions that inspect the state of the device, perform memory allocations and copies, and launch kernels on the GPU. These kernels can be compiled separately or compiled as part of the same source file as the host code.

At the start of execution of every HIP program, a function called `__hipRegisterFatBinary` is called with a pointer to a GPU binary as a the first argument. This GPU binary is the one obtained by compiling the GPU portion of the source file. We examined the LLVM source code to determine the format of this structure that is being pointed to, and found it is what LLVM calls a "bundled" binary—both the host program executable code and GPU executable code are stored alongside each other. Once we knew the structure of this object, we could access it through the void pointer argument provided to `__hipRegisterFatBinary`. This structure contains a size and a buffer containing the ELF executable.

When a user wants to launch a kernel, they specify a kernel name using one of several different syntaxes. Regardless, these different methods compile down to the `hipLaunchKernel` API, which takes a function pointer as the first argument. This pointer corresponds to the kernel located in the binary provided to `__hipRegisterFatBinary`. The ELF file also contains a section that has additional information about each kernel, like how many registers it requires. With this information, the GPU can send the command to launch the appropriate kernel, specifying its initial arguments and ensuring that the GPU is setup properly.

To monitor the behavior of these two functions, we wrote our own custom implementation of both `__hipRegisterFatBinary` and `hipLaunchKernel`. In `__hipRegisterFatBinary` we record the value of the pointer, access the structure using the information we know

about the format of the binary, and then record the contents of the GPU binary. This file can be used later on by other HIP API functions that use GPU binaries. For example, the `hipModuleLoad` function takes a filename and loads the binary for the user. This file is the same ELF file we obtain from separating the original bundled binary.

In our implementation of `hipLaunchKernel`, we lookup which kernel is being launched in the code object we recorded. We examine the `.notes` section of the file, which is a MsgPack [9] encoded buffer, to find the size and type of the kernel's arguments. If the type is a primitive type, we can simply copy that argument that was passed to the launch invocation.

By hooking these two functions, we can obtain

1. an ELF executable containing the GPU code that will be executed.
2. a list of every kernel launched and the runtime parameters provided to that kernel.
3. the order that kernels are launched and the number of times that they are executed (as well as the different compute streams they may be issued on).

3.1.1 Capturing Data

When copies are performed to and from the GPU, we need to save these buffers in order to recreate the same behavior. These allocations can be very large, from several MBs to GBs in size. In order to effectively capture all of this data, we utilize a SQLite database. Being both lightweight and performant, SQLite allows us to quickly record the relatively small datatypes provided to the API calls (as well as identifiers for the API calls themselves). Using SQLite also helpfully abstracts away both the file format and asynchronously writing the data to a file. We don't necessarily have to block the execution of any API call while we finish writing the data to disk.

3.1.2 Replay

In Chapter 4 we describe how we use the capture of these two functions to rewrite binaries. This in turn can be used to implement binary instrumentation tools which we describe in Chapter 5. But for now let's consider also hooking the `hipMalloc` and `hipMemcpy`

functions and capturing their arguments. If we do so, we obtain a listing of every memory allocation performed and the requested size of the memory allocation. If we record the value of pointers returned by the driver, we can get a mapping of every buffer in use in GPU memory. This information can be recorded alongside the GPU binary and list of kernel launches.

From this data that we captured, we can completely recreate the GPU behavior of many programs without access to the original executable. We simply iterate through each call to the `hipMalloc` and `hipMemcpy` function calls and repeat them. We will obtain different pointer values from the driver when we execute these functions, but the sizes of the buffers will be identical since we use the same arguments. If we maintain a mapping from the original pointer values to the new pointer values that we obtain, we can then replace any occurrence of the old pointers with the new ones. If a kernel performs pointer arithmetic, this technique stops working, but we discuss a potential solution in 3.3.

Once the memory allocations and copies have been recreated, we load the GPU binary with the `hipModuleLoad` function we mentioned before, and execute the same kernel launches we recorded in our database. We examine the arguments to the kernel and check for any pointer types. If we encounter one, we check the mappings between allocations and replace the pointer with its current counterpart. In this way we are able to execute the exact same kernels with the same arguments and the same data without access to the original executable and using techniques that don't require modifying the driver.

3.2 Difficulties

In implementing this system, we found that determining the formats of binaries and recording kernel arguments were the most challenging problems to solve. While there is some documentation, it is not sufficient for working with the binary objects. The bundled binary is different than what is used by `hipModuleLoad`, for example. Similarly, kernel arguments have notes that describe what their purpose is, and there is some documentation, but it doesn't explain the meaning of different argument types. As a result, a lot of this information had to be determined through trial and error.

3.3 Limitations

Since we use `LD_PRELOAD` to redirect the function calls, we need to know the exact function signatures of anything we'd like to redirect. We also needed to investigate the format of the binary provided at program start, and we depended on this function existing and behaving the way it does. If there isn't an equivalent of the `__hipRegisterFatBinary` function, then some other way of accessing the GPU binary has to be found.

When we record the arguments provided to a launch kernel call, we use information in the `.notes` section of the binary to determine what these arguments were. Some information simply isn't provided, however. If an argument is a C structure or C++ object, we only know the size of the structure—nothing about its contents. Any pointers contained in structures are completely unknown to us when we capture in this way.

This problem can be addressed through another layer of indirection—a virtual memory system. CUDA implements a form of virtual memory that allows allocations to be placed at any address specified by the user of the library, forcing the allocation at that address if possible. On AMD this functionality exists within the driver but is not exposed to the user in any way. There are likely plans to release a virtual memory system similar to CUDA's, and this system would allow any binary to be replayed.

We also discuss a variety of downsides to our approach in Chapter 6 in comparison to tools like NVBIT.

Chapter 4

Binary Rewriting

In the previous chapter, we discussed how GPU binaries can be captured and replayed. In this section, we describe how a captured AMD GPU binary can be rewritten with different instructions, changing the behavior at runtime. We focus on rewriting binaries to execute additional instructions at certain points in the program. We could also rewrite binaries to update certain instructions for a new architecture, but we will focus on rewriting for the sake of instrumentation. The programmer should think of rewriting as editing a single binary instruction to potentially execute any number of instructions instead. This is valuable for various binary instrumentation tools. For example, we can replace a load instruction with code to save the loaded address somewhere, followed by the original load instruction. We could similarly replace any instruction with instructions that start a clock or increment a counter.

4.1 Design

To rewrite a section of a program binary, we need to open the executable code, find the instruction we'd like to rewrite, replace the rewritten instruction with a branch, and append new code to the end of the program. We choose to append new code rather than insert it in place since we can avoid adjusting branch and data offsets in the original code. Relative branches can be broken by inserting instructions in place since the new instructions can change the distance between branch and target. PIN keeps a record of branch targets and updates them when it rewrites code, but we'd like to avoid this

complexity. This branch causes our programs to be less performant since we introduce a branch to and from a completely different part of the executable.

First, let's consider opening the program executable. Once a binary is captured and saved to a file, it is relatively easy to work with. The file format is ELF, just like any normal executable on the system. In the .text section of the ELF file are the GPU instructions. Additional information about the kernel is kept in the `.notes` section of the binary. It is encoded as a MsgPack formatted buffer, a compact JSON-like encoding. The information stored here is the kernel names, their properties and the number of registers used for them. An additional section of the binary known as the kernel descriptor also stores information about the kernel, but this section is actually used by the driver to setup the hardware. If the number of registers used is modified in the `.notes` section, nothing changes—the modifications need to occur to the kernel descriptor. It appears that the `.notes` section is only a listing of additional information that is not actually used for any purpose, while the kernel descriptor is a part of the setup process.

Suppose we would like to rewrite part of this binary. We simply have to find the relevant part of the `.text` section, edit it, and save a new ELF file. We're also free to append new data to the `.text` section. When rewriting it is important to keep program offsets in mind. Shifting instructions or data can break the behavior of the GPU program because branches and data can be described with relative offsets, so when rewriting it's important to keep the edit to a single instruction.

In order to interpret the instruction data, we use the open-source library DynInst [3], which can map instruction encodings to the name of the relevant instruction and extract register identifiers. The LLVM command-line tools and the LLVM library itself can both be used to accomplish the same task, but they can be more difficult to use—the command-line tools require the programmer to parse their output and the LLVM library can be difficult to integrate into an existing project. However, AMD GPU support in DynInst is also limited, and it is also a complicated library that can be difficult to integrate into an existing project.

When a binary is rewritten, we need to have instruction data ready to insert into the

binary. There are many ways to get this inserted instruction data. One could write the assembly themselves, using the output of an assembler as the data to insert. One could use the compiler to write a kernel, compile it, and then modify the output by hand to insert the relevant instructions. Or one can directly use compiled binaries as the rewritten code. We use the first option, while NVBIT uses the last. It's much preferred to use NVBIT's method, but we were unable to integrate the compiler into our system as successfully. Instead, we wrote custom assembly whenever we needed to rewrite a binary. We describe this further in Chapter 6.

When we rewrite, it's possible that we need additional registers that the original program did not require. When this happens, we edit the kernel descriptor to change the number of registers that the kernel uses.

4.2 Limitations

If a kernel already uses the maximum amount of registers, then it is not possible to increase the number of registers. This is incredibly rare, however. Typically a kernel will be written in a way to use as few registers as possible since this increases the number of copies of that kernel that can be executed at any given time. Even if a kernel such as this is encountered, there are strategies to solve this problem. For example, a stack can be used if the kernel already uses one or some vector registers can be moved to scalar registers or vice versa. We could not find a kernel where the maximum number of registers was in use.

As mentioned earlier, we also simply append new instructions to the executable. We could affect performance greatly by introducing a branch. Since we simply append the code, we're not able to perform any optimizations that consider the original executable code.

Chapter 5

Instrumentation & Profiling

In this Chapter we discuss two tools that we implemented, evaluate them when used with a real program, and compare with NVBIT.

5.1 Implemented Tools

We used our rewriting functionality to implement two tools—a tool for counting instructions and a tool for obtaining a memory trace of a program. Currently, if we can write a simple assembly program, we can use it an instrumentation tool. We simply rewrite the instrumented instruction and append the instrumentation code to the executable. But this means that all changes have to be very explicit about the assembly code that they will add. It's not possible to just directly take the output of the compiler without additional work that we discuss in Chapter 6, where we also discuss methods for improving the usability of our system. Below, we describe our implementation, provide results from their use, and compare with NVBIT's methods for obtaining the same information.

5.1.1 Instruction Counting

To count the number of times an instruction is executed, we only need a single atomic counter. We rewrite the binary to increment this atomic before returning the original execution. We describe the rewrite process in Chapter 4, but to summarize, the instruction that we would like to count is replaced with an `s_branch` instruction. Then we append code to perform the counting operation at the end of the executable. The `s_branch`

branches to this new part of the executable, which increments an atomic counter, and then returns to the original program.

The NVBIT equivalent of this tool is very similar. Again a single counter is incremented every time the instruction is executed. NVBIT has also extended this functionality to an execution trace, where they keep separate counters for every instruction and increment them as execution proceeds through the program. Instrumentation does not need to be placed on every single instruction in order to compute an instruction trace. Counters need only be placed at the start of basic blocks since it's known that every instruction between them will be executed.

The program has to be further modified to allocate the atomic on the host side of execution. This is done when the binary is first rewritten, and then the address of the atomic is directly used in the rewrite operations—the memory address is placed directly in the assembly file. This allocation can also be done at the time the kernel is launched. Then when the program is finished executing, we copy the counter back from the device, and provide the value to the user.

5.1.1.1 Results

To evaluate our ability to count instructions, consider the simple HIP program in Figure 5.1 that adds two vectors and stores the result. To ensure that we don't access the array beyond its bounds, a conditional check is included that checks whether the thread index is outside the bounds of the arrays. If we examine the assembly code generated by the compiler (Figure 5.1.1.1) we can see that on line 13 a condition is evaluated and if all threads fail the condition, on line 14 we branch to the end of the program (if only some of the threads fail the condition, the failed threads are masked out and execution continues). We would thus expect that if launch this kernel with too many threads, some thread groups would take the branch and the instructions on line 15 and beyond would be executed less.

Let N be the size of the vectors we are adding. When we place an instrumentation point on line 13 in Figure 5.2a, and launch with N threads, we obtain the expected execution count, N. When we launch with more threads than the size of the array, we see

```
1  __global__ void vectoradd_float(float* a,
2                                  const float* b,
3                                  const float* c,
4                                  int width, int height)
5  {
6      int x = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
7      int y = hipBlockDim_y * hipBlockIdx_y + hipThreadIdx_y;
8
9      int i = y * width + x;
10     if (i < (width * height)) {
11         a[i] = b[i] + c[i];
12     }
13 }
```

Figure 5.1: A simple GPU program that contains a branch.

the count increase. Instrumentation placed after line 15, however, always has an execution count of N as expected.

Figures 5.2b and 5.3 show the assembly code after instrumentation is added. Note that the instruction on line 13 is changed to an `s_branch` instruction to beyond the end of the program at line 35. In Figure 5.3, the instrumentation code can be seen. Registers are setup in lines 36–47, the address of the atomic is stored in lines 50–54, and the atomic that stores the count is incremented in line 56. After restoring the original registers, the original instrumented instruction is executed on line 71 before returning to the original program.

Using NVBIT to compute the same information is considerably simpler. They include a pre-written instruction counting tool written as a GPU device function. The tool also increments an atomic corresponding to the instruction that was executed, but this data and program logic is entirely separate to the details of rewriting the binary. Our system is much more tightly coupled—changes to the tools require changes to the hooked API

```
1         s_load_dword s2, s[4:5], 0x2c
2         s_load_dwordx2 s[0:1], s[4:5], 0x18
3         s_waitcnt lgkmcnt(0)
4         s_lshr_b32 s3, s2, 16
5         s_mul_i32 s7, s7, s3
6         v_add_u32_e32 v1, s7, v1
7         v_mul_lo_u32 v1, v1, s0
8         s_and_b32 s2, s2, 0xffff
9         s_mul_i32 s6, s6, s2
10        s_mul_i32 s0, s1, s0
11        v_add3_u32 v0, s6, v0, v1
12        v_cmp_gt_i32_e32 vcc, s0, v0
13        s_and_saveexec_b64 s[0:1], vcc
14        s_cbranch_execz END
15        s_load_dwordx2 s[6:7], s[4:5], 0x10
16        s_load_dwordx4 s[0:3], s[4:5], 0x0
17        v_ashrrev_i32_e32 v1, 31, v0
18        v_lshlrev_b64 v[0:1], 2, v[0:1]
19        s_waitcnt lgkmcnt(0)
20        v_mov_b32_e32 v3, s7
21        v_add_co_u32_e32 v2, vcc, s6, v0
22        v_addc_co_u32_e32 v3, vcc, v3, v1, vcc
23        v_mov_b32_e32 v5, s3
24        v_add_co_u32_e32 v4, vcc, s2, v0
25        v_addc_co_u32_e32 v5, vcc, v5, v1, vcc
26        global_load_dword v6, v[4:5], off
27        global_load_dword v7, v[2:3], off
28        v_mov_b32_e32 v2, s1
29        v_add_co_u32_e32 v0, vcc, s0, v0
30        v_addc_co_u32_e32 v1, vcc, v2, v1, vcc
31        s_waitcnt vmcnt(0)
32        v_add_f32_e32 v2, v6, v7
33        global_store_dword v[0:1], v2, off
34  END   s_endpgm
```

(a) Before

```
          s_load_dword s2, s[4:5], 0x2c                          1
          s_load_dwordx2 s[0:1], s[4:5], 0x18                    2
          s_waitcnt lgkmcnt(0)                                   3
          s_lshr_b32 s3, s2, 16                                  4
          s_mul_i32 s7, s7, s3                                   5
          v_add_u32_e32 v1, s7, v1                               6
          v_mul_lo_u32 v1, v1, s0                                7
          s_and_b32 s2, s2, 0xffff                               8
          s_mul_i32 s6, s6, s2                                   9
          s_mul_i32 s0, s1, s0                                   10
          v_add3_u32 v0, s6, v0, v1                              11
          v_cmp_gt_i32_e32 vcc, s0, v0                           12
          s_branch INST                                          13
RET       s_cbranch_execz END                                    14
          s_load_dwordx2 s[6:7], s[4:5], 0x10                    15
          s_load_dwordx4 s[0:3], s[4:5], 0x0                     16
          v_ashrrev_i32_e32 v1, 31, v0                           17
          v_lshlrev_b64 v[0:1], 2, v[0:1]                        18
          s_waitcnt lgkmcnt(0)                                   19
          v_mov_b32_e32 v3, s7                                   20
          v_add_co_u32_e32 v2, vcc, s6, v0                       21
          v_addc_co_u32_e32 v3, vcc, v3, v1, vcc                 22
          v_mov_b32_e32 v5, s3                                   23
          v_add_co_u32_e32 v4, vcc, s2, v0                       24
          v_addc_co_u32_e32 v5, vcc, v5, v1, vcc                 25
          global_load_dword v6, v[4:5], off                      26
          global_load_dword v7, v[2:3], off                      27
          v_mov_b32_e32 v2, s1                                   28
          v_add_co_u32_e32 v0, vcc, s0, v0                       29
          v_addc_co_u32_e32 v1, vcc, v2, v1, vcc                 30
          s_waitcnt vmcnt(0)                                     31
          v_add_f32_e32 v2, v6, v7                               32
          global_store_dword v[0:1], v2, off                     33
END       s_endpgm                                               34
INST      s_waitcnt vmcnt(0) expcnt(0) lgkmcnt(0)                35
          v_mov_b32_e32 v8, v0     // Setup registers            36
          v_mov_b32_e32 v9, v1                                   37
          v_mov_b32_e32 v10, v2                                  38
```

(b) After

Figure 5.2: The corresponding assembly listing for the program in Figure 5.1 before and after instrumentation. We place an instrumentation point on line 13 before the branch occurs. With another instrumentation point on line 15, we can see how many threads took this branch. The branch jumps to the instrumentation at line 35, which saves off registers before returning. Figure 5.3 shows the remaining instrumentation code.

```
36          v_mov_b32_e32 v8, v0     // Setup registers
37          v_mov_b32_e32 v9, v1
38          v_mov_b32_e32 v10, v2
39          v_mov_b32_e32 v11, v3
40          v_mov_b32_e32 v12, v4
41          v_mov_b32_e32 v13, v5
42          v_writelane_b32 v14, s0, 0
43          v_writelane_b32 v15, s1, 0
44          v_writelane_b32 v16, s2, 0
45          v_writelane_b32 v17, s3, 0
46          v_writelane_b32 v18, s4, 0
47          v_writelane_b32 v19, s5, 0
48          s_waitcnt vmcnt(0) expcnt(0) lgkmcnt(0)
49          s_waitcnt vmcnt(0) expcnt(0) lgkmcnt(0)
50          s_mov_b32 s2, 0x9fc00000
51          s_mov_b32 s3, 0x7f6b
52          v_mov_b32_e32 v0, 1
53          v_mov_b32_e32 v1, 0
54          v_mov_b32_e32 v2, 0
55          s_waitcnt lgkmcnt(0)
56          global_atomic_add_x2 v[0:1], v2, v[0:1], s[2:3] glc
57          s_waitcnt vmcnt(0)
58          v_mov_b32_e32 v0, v8
59          v_mov_b32_e32 v1, v9
60          v_mov_b32_e32 v2, v10
61          v_mov_b32_e32 v3, v11
62          v_mov_b32_e32 v4, v12
63          v_mov_b32_e32 v5, v13
64          v_readlane_b32 s0, v14, 0
65          v_readlane_b32 s1, v15, 0
66          v_readlane_b32 s2, v16, 0
67          v_readlane_b32 s3, v17, 0
68          v_readlane_b32 s4, v18, 0
69          v_readlane_b32 s5, v19, 0
70          s_waitcnt vmcnt(0) expcnt(0) lgkmcnt(0)
71          s_and_saveexec_b64 s[0:1], vcc
72          s_branch RET
```

Figure 5.3: Continued listing from Figure 5.2b of the instrumented assembly program. The program finishes saving registers, increments an atomic stored in memory on line 56, and returns to the original program on line 72.

calls, whereas NVBIT abstracts all of this into a single source file.

5.1.2 Memory Tracing

A memory trace is a sequence of addresses accessed by a program. A memory trace can be used for cache simulators, architecture design, and for getting a sense of the structure of a program's memory accesses. To obtain a memory trace for a single instruction, we rewrite the binary to increment an atomic as before, but we now use this atomic as an index into a buffer which stores a memory address. The binary is examined to determine which register holds the address and that register's contents are stored in the buffer at the atomic index. The NVBIT version of this tool is very similar except that their implementation is written in CUDA C++ and is able to record additional information like the thread and block index that corresponds to every memory access.

5.1.2.1 Results

When we set an instrumentation point on the first store instruction and print addresses, we find that each memory address accessed by this instruction is accessed equally likely. This is expected for the program in Figure 5.1. Each thread only writes to a single element of the array.

When using NVBIT for a similar program, we obtained the same results as expected. In the NVBIT program, we were able to use any compiler features or additional data structures that we wrote. In the memory trace example, it's useful to have each thread write some additional information like the thread and block index.

Since we used a handwritten assembly program, we found it difficult to add additional information as easily. In our case, the thread index can be clobbered by a later operation, overwriting its value which meant our instrumentation functions could not access it. We could work around this by storing the thread index and other information as soon as the kernel starts, but this method is not as flexible as just accessing the values in source code. It's also possible that the thread index is kept in a special register or accessible by other means, but we were unable to determine any such capability on AMD's hardware.

5.2 Runtime Costs

In this section we summarize the costs of using our instrumentation tools in the two examples above. We measure both the time taken to capture and rewrite the binary (the cost on the host) and the time taken to execute the new binary (the cost on the device). When possible we compare with a similar example that uses NVBIT.

	APP	KERNEL TIME	% CHANGE	TOTAL TIME	% CHANGE
HIPTRACER	vectoradd	0.0113 ms	–	302 ms	–
	w/ INSTR_COUNT	0.197 ms	1643%	1014 ms	235.76%
	w/ MEMTRACE	0.208 ms	1740%	1024 ms	239.07%
NVBIT	vectoradd	0.0064 ms	–	182 ms	–
	w/ INSTR_COUNT	–	–	341 ms	87.36%
	w/ MEMTRACE	–	–	352 ms	93.41%

Table 5.1: Table of kernel execution times and total execution times when using instrumentation tools. NVIDIA profiling tools cannot be run with NVBIT, so it's not possible to see kernel execution times in those cases.

5.2.1 Rewriting Cost

In Table 5.2 we show the increase in total execution time for both our application and an application instrumented with NVBIT. We attempted to use as similar an application as possible. The runtime cost can be used to estimate the rewriting cost. Since it's not possible to isolate the rewriting portion of NVBIT's execution, this is our only way to measure their rewriting time. Note that we have a more significant increase in total execution time than NVBIT, increasing the total runtime by 235% for instruction counting instead of only 87%. We can see that the majority of this cost must be from rewriting since the new kernel execution time is only 0.2 ms. This is because we save rewritten binaries to disk and then have to load them again when they're launched whereas NVBIT performs all of its rewriting in memory.

5.2.2 Kernel Execution Cost

We were also able to measure the impact on kernel execution for our binaries. This wasn't possible for NVBIT since they prevent multiple profiling tools from working together. But on AMD's GPUs, we were still able to profile the instrumented kernel as well. We found that before instrumentation, the kernel ran in 0.0113 ms but after adding the instruction count tool it was 0.197 ms (and 0.208 ms with memory tracing). We believe this increase is due to the introduction of the `s_branch` and `s_wait` instructions. These operations will disrupt the optimizations performed by the compiler and will have a large impact on performance. We're aren't able to measure how NVBIT affects performance in this way, but we speculate that it is not as significant since as part of the driver, they're able to call the compiler if necessary which should produce more optimal code.

5.3 Future Tools

In this section we describe potential tools that can be built as extensions of our existing tools.

5.3.1 Measuring Coalescing of Loads & Stores

From a memory trace with thread and block index information, we can determine which loads and stores are from the same thread group. We can then examine these addresses to determine if the loads and stores are able to coalesce. Coalescing refers to memory accesses that can be combined because they lie in the same cache line sector as another access. Rather than issuing two loads for the data, the hardware can determine that they can be combined, reducing the bandwidth required. This tool can be implemented entirely as a script that scans the memory trace, or it can be implemented as an instrumentation tool itself, examining the addresses at runtime. The most challenging aspect of this is simply ensuring that the coalescing rules are correct.

When bank conflicts are reported, it could be helpful to add some additional debugging information to the results, so that a user can see which source lines in their program don't coalesce very well. This is achievable by interpreting the debug format of the GPU binaries. This is an involved engineering task since the format is DWARF, but it is

achievable with some additional effort.

5.3.2 Detecting Bank Conflicts

Bank conflicts can be detected in a manner similar to coalescing. A memory trace is collected and a script can run to inspect the memory addresses for bank conflicts. The most challenging aspect of writing such a tool is correctly detecting the conflicts. Each memory address is assigned one of 32 different memory banks. A bank conflict occurs when a thread group executes a memory access instruction, and two memory addresses correspond to the same bank. When this occurs, the memory loads or stores have to be serialized, increasing the cost of the memory instruction. We can again use a script to interpret the memory trace results and check for bank conflicts. It is again potentially useful to report these issues at runtime with additional debugging information. It can be difficult for a programmer to determine which of their memory instructions actually has bank conflicts, even if they are aware that the kernel has them frequently.

5.3.3 Cache Reuse Distance

By again interpreting the results of the memory trace, we can compute the average cache reuse distance or the average distance between usage of data in the same cache line. We simply need to count the number of accesses between any two accesses to the same cache line. This information can again be obtained by running a script over the memory trace. An entry is created for every cache line and a running average of the number of accesses can be updated as the script iterates over memory accesses in the memory trace.

Chapter 6

Insights & Evaluation

We now attempt to evaluate our system in comparison with other, similar systems, such as NVBIT. We discuss the shortcomings of our design and note what design decisions led to these shortcomings. We then make recommendations for fixing these issues in future work and note areas where our system can be improved through additional engineering effort. We will compare and evaluate overall capabilities, shortcomings, the programming model for inserting instrumentation, and the potential for future applications. We will also evaluate the potential performance implications of our design decisions.

6.1 Capabilities & Shortcomings

Both hiptracer and NVBIT have similar potential for instrumenting binaries. Any additional instructions that a programmer would like to be added can be. But in NVBIT and PIN's case, the programmer doesn't have to know anything about the underlying ISA in order to make an addition. They just write a CUDA or C function that will get "inserted" at a certain instruction. In our case, the programmer does have to know AMD's RDNA assembly language to capably use the system because our interface only allows adding the instructions directly. It's possible for us to reach the same level of abstraction as NVBIT and PIN, but it requires additional understanding of compiling "remote" (not inlined) device functions for AMD's architecture. We struggled to manage the complexity of all of the features that must be considered to make this workflow possible. It's not clear what 'setup' has to be done before a function can be called. NVIDIA has complete understand-

ing of what goes into one compiling these device functions, and since they're operating inside the driver, they have access to any information they might need to perform this setup. And in some cases, they would know what parts of setup aren't required to insert the function into the instrumented program. This is ultimately a major shortcoming of our approach. Since we can only obtain information that is output by compiler tools or able to be reverse-engineered, we don't have the same ability to easily insert a function anywhere. But this doesn't mean we can't insert them. It just requires a lot of effort to understand each feature, when it's needed, when it's not, and exactly how it works. This information has to be discovered through trial and error and through instrumenting a large variety of programs that utilize all of the hardware's features and expose any oddities of the instruction set.

Our system has a few capabilities that NVBIT lacks entirely. Since our system started as a way to capture and replay kernels, we have maintained that capability throughout the process of implementing binary instrumentation. This means that all the information needed to replay a kernel can be saved to disk. NVBIT and other tools don't look at the problem this way, so they don't provide this sort of capability. For CPU tools, it's unnecessary as the binary already exists as a single file. But for GPU programs it can be valuable to see exactly which kernels were launched and what their arguments were. Then this subset of the binary can be replayed exactly as it was originally. This can be very valuable for debugging, program analysis, and profiling because you can get a consistent run to judge a program's performance by. We think that a lot of value exists from looking at GPU programs this way, even without any sort of instrumentation added.

Another shortcoming of our work is its overall functionality at this point. We are unable to instrument any program outside of very simple examples due to bugs and misunderstandings of the assembly. We've attempted to resolve these issues the best that we can and have coordinated with AMD to better understand the ISA, but the level of complexity is high. We believe an issue is the modification of kernel descriptors to increase the number of registers being used in a kernel launch. Checking the kernel descriptor is not straightforward, and there's no way to confirm the resulting descriptor is still valid

when the kernel is launched. There is technical documentation available, but it's dense and difficult to understand. But overall, we believe that these issues can be resolved through more thorough reverse-engineering of the GPU assembly programs.

6.2 Programming Model

Our programming model is similar to tools like NVBIT and PIN, but we lack an abstraction over the architecture-specific details. We require the programmer to either write assembly or use the output of the compiler themselves, and we require them to know the size and contents of instructions. We offload some of this responsibility to DynInst, but the programmer is still expected to keep track of a great deal.

NVBIT avoids this problem by abstracting away their binary ISA, SASS, and by allowing users to directly use compiled device functions. In NVBIT's case, the device function is compiled with the CPU code for instrumentation. Then the driver can see the code in the same executable, and combine it with the program to be instrumented. We could support a similar programming model with additional information about how these device functions work on AMD's architecture, but it will never be quite as easy since we work external to the driver and don't have the same information available. For example, if a compiled instrumentation function requires a stack, then we'll need to modify the original program to use a stack. The driver can easily allocate this stack and know its memory address, but we would have to allocate something separately or depend upon it already being allocated (and then we would need to know the memory address).

6.3 Extendability

Since our program operates without driver support, it is easier to extend the capabilities. Everything is freely modifiable and external to the packages produced by others. NVBIT and PIN support writing additional tools using their API, but we also support the expansion of this API with additional features.

Since we use `LD_PRELOAD` to implement our system, any of the driver functions can be redirected with the same mechanism. If we wanted to enable instrumentation only for a very specific launch, or only if a certain condition is triggered, this is a lot easier to

implement in our system. NVBIT supports enabling instrumentation on specific launches by exposing a callback that is called before and after every launch, but they can't support this for every single GPU driver function like we can.

The most obvious way to extend functionality is the same for both systems—the creation of additional instrumentation tools. We discuss our tools in Chapter 5 and propose some tools that could be implemented in the future, such as detecting coalescence and bank conflicts—tools that would be very specific to GPU applications. Like NVBIT and PIN, we want to support the creation of additional tools like this. But as we discussed in Section 6.2, our programming model does not directly support compiled device functions. A lot of additional work has to be done by the programmer to manipulate a compiled program into a suitable form for being an instrumentation function.

6.4 Performance Implications

Our design also has different performance implications than PIN and NVBIT. In Chapter 5, we conducted some simple experiments on overall impact of rewriting on the program's execution time. We found that compared to NVBIT, we add significantly more time to execution. This is because we save a copy of the binary file to disk and have the driver load it from disk. Since we are not part of the driver, we have use the `hipModule` functions to load the binary even though we are able to make all of our modifications in memory. This implies that we will always have a higher one-time cost for modification than NVBIT or PIN. It's possible that an alternative design that does not use these functions and instead finds some other mechanism to load the binary is faster.

In terms of an instrumented kernel's performance, we also will have more of an impact on the original program than NVBIT. This is due to a quirk of AMD's ISA, the `s_wait` instructions. These instructions tell the system to wait for a certain number of memory operations to complete before resuming execution. The compiler is very aggressive about this form of optimization in AMD's kernels since it allows a lot of work to be pipelined while long-running memory operations finish completion. When we add instrumentation, we have to ensure that the previous vector memory instructions have completed before

we overwrite their registers. This means we always have to introduce an `s_wait` that has to wait for *all* outstanding memory operations to complete. NVBIT does not have to perform any such operation, so we always have to pay a higher price when instrumentation is performed. Also, if the programmer is trying to measure the impact of placing `s_wait` instructions in a certain location or modify the number of operations to wait, they won't see any change in an instrumented binary. Working around this issue would require implementing an algorithm for determining the ideal placement of an additional `s_wait` that waits for as few operations as possible.

Overall we feel that many of our shortcomings can be worked around, but there are fundamental issues to creating a tool like this outside of the driver, and we believe that the impact can be most seen in performance and in the increased effort in engineering when some information is unavailable.

Chapter 7

Conclusion and Future Work

Through the previous chapters we have shown that a system for capturing and modifying GPU activity can be completely implemented without driver support from GPU vendors. We have shown that these tools are functional, that the method of capturing and replaying binaries is sensible, and that rewriting and thus instrumenting these binaries is all feasible within these restrictions. We have compared the system we implemented with other systems that accomplish the same goals without the same restrictions and found that while our tools fall short in a myriad of ways, these shortcomings are mostly addressable. In some cases the problem needs to worked around, while in other cases small features which are useful for all users need to provided. In this section, we'll discuss the potential of future work, and talk about what needs to be fixed in order to make further progress. We'll then close with some final thoughts on our system.

7.1 Next Steps

Various shortcomings of our system are already discussed in Chapter 6. We believe the most helpful first step in addressing these shortcomings is some mechanism for interacting with kernel descriptors and determining whether they are being correctly modified for kernels with more registers than our simple examples. These more complex kernels will also employ additional hardware features that will be useful for stressing the system's functionality.

After a more thorough set of kernels is supported, the next step is simplifying the

programming model for adding an instrumentation function. As we mentioned, we require the programmer to specify the function as compiled assembly code, which does not make adding new functions easy. Writing assembly or modifying the assembly of a compiled program by hand is incredibly error-prone. To accomplish this, a better understanding of AMD's process for using remotely compiled device functions is required. Reverse-engineering compiled binaries to determine which features have to be set up and the proper process for setting them up is required. Many of these features are described in AMD's official documentation, but it's difficult to understand how they should be accounted for in an instrumentation system.

Once any kernel written in C can be used as an instrumentation function, the possibilities for further instrumentation tools become numerous. We mentioned detecting coalescence and bank conflicts—this information can be combined with debugging information to give the programmer specific line numbers of bank conflicts. It may also be useful to instrument other GPU-specific constructs like shared memory usage.

Visualizing memory access patterns or the usage of resources may also be valuable. Since a memory trace can produce a list of all memory accesses, this information could be visualized in a grid and used to determine if memory access patterns are ideal or whether resources are scattered across the memory space.

We believe that these tools can be combined in rich and interesting ways to answer specific questions about GPU programs. While a traditional profiler offers a coarse view of the entire system, these sorts of instrumentation tools can be much more fine-grained, allowing a programmer to measure only specific threads or thread groups and look for very specific patterns.

The capture functionality of our system can also be built upon. By using a virtual memory system, any kernel should be able to be captured and replayed. At some point this feature should be released by AMD, but it's also possible to implement a virtual memory system yourself using the lower level constructs available in AMD's HSA. We believe this has significant value for debugging and profiling tasks since it allows a perfect reproduction of some subset of GPU work. As systems become more heterogeneous, it

will be valuable to have a representation of 'just' the GPU work, so the whole program doesn't have to be profiled together.

7.2 Closing Thoughts

Regardless of whether others continue to build upon this system, we believe that it proves that capturing, modifying, replaying and instrumenting binaries is all possible at the user level. No kernel or driver support is needed. There may be additional challenges, and the effort required to implement the same functionality may be increased. But programmers have a lot more power to inspect the performance of a GPU system than they may think. They're less dependent on hardware manufacturers than they may believe. Powerful tools that measure all kinds of detailed information about the system can be created. It may require a lot of reverse-engineering and tinkering, but it is possible, and these tools can be immensely valuable. We hope to see others build off this work either directly or by creating their own instrumentation tools for *their* system in the future and hope that the lessons we learned will be valuable to their efforts.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, May 2000.

[2] D. Bruening. Efficient, transparent, and comprehensive runtime code manipulation, 2004. `https://api.semanticscholar.org/CorpusID:33909610`.

[3] Dyninst documentation and libraries, 2012. `https://github.com/dyninst/dyninst/`.

[4] C. Gorgovan, A. d'Antras, and M. Luján. MAMBO: A low-overhead dynamic binary modification tool for ARM. *ACM Trans. Archit. Code Optim.*, 13(1), Apr. 2016.

[5] K. Hazelwood and A. Klauser. A dynamic binary instrumentation engine for the arm architecture. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 261–270, New York, NY, USA, 2006. Association for Computing Machinery.

[6] K. Levit-Gurevich, A. Skaletsky, M. Berezalsky, Y. Kuznetcova, and H. Yakov. Profiling Intel graphics architecture with long instruction traces. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 1–11, 2022.

[7] X. Li, I. Laguna, B. Fang, K. Swirydowicz, A. Li, and G. Gopalakrishnan. Design and evaluation of GPU-FPX: A low-overhead tool for floating-point exception detection in NVIDIA GPUs. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '23, pages 59–71, New York, NY, USA, 2023. Association for Computing Machinery.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[9] Msgpack specification and libraries, 2013. `https://msgpack.org/`.

[10] A. Skaletsky, K. Levit-Gurevich, M. Berezalsky, Y. Kuznetcova, and H. Yakov. Flexible binary instrumentation framework to profile code running on Intel GPUs. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 109–120, 2022.

[11] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler. NVBitFI: Dynamic fault injection for GPUs. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 284–291, 2021.

[12] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler. NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO '52, pages 372–383, New York, NY, USA, 2019. Association for Computing Machinery.