Horus: A modular GPU emulator framework

Amr S. Elhelw
Electrical and Computer Engineering Department
University of Rochester
Rochester, USA
aelhelw@ur.rochester.edu

Sreepathi Pai
Computer Science Department
University of Rochester
Rochester, USA
sree@cs.rochester.edu

***Abstract*—Graphics Processing Units (GPUs) are widely used to run general-purpose computing workloads. Three approaches currently exist to observe the dynamic behaviour of these workloads: real hardware, architectural simulators, and functional simulators (or emulators). However, the rapid evolution of GPU hardware and software stacks means that, in reality, using hardware is the only option to study current GPU workloads. Unfortunately, GPU toolchain support for advanced characterization capabilities is still far behind CPU toolchains like Pin.**

In this paper, we present an early glimpse of Horus, an emulator for NVIDIA GPUs. Although it is not the first such emulator, Horus is being engineered using a novel methodology to keep pace with the rapid changes in GPU hardware. Horus is highly modular, and is the first to utilize a specially designed DSL for specifying formal semantics for GPU instruction sets (NVIDIA PTX). A semantics compiler uses these semantics to generate the emulator. Horus also features a well-defined interface by which utility functions – instrumentation, new instruction support, analysis tools – can be coupled with the main emulator to increase reuse while studying the dynamic behaviour of GPU kernels. Horus is now mature enough to run all the Polybench and Rodinia benchmarks correctly.

***Index Terms*—functional simulator, formal semantics, GPU, instrumentation**

I. Introduction

GPUs now accelerate general-purpose applications in machines ranging from supercomputers to smartphones. The large number of cores and high memory bandwidth enable GPUs to have higher throughput over CPUs for data parallel applications [1]. NVIDIA's CUDA [7] and the Khronos Group's OpenCL [22] languages have enabled porting large number of general-purpose data-parallel applications to GPUs.

To study these new applications (GPU kernels), researchers have developed a number of models. Cycle-level simulators are used widely by computer architects. NVIDIA GPUs can be simulated by GPGPU-Sim [4], Multi2Sim [15] (Kepler GPUs), and GPUTejas [21]. AMD GPUs are supported by Multi2Sim, Gem5 [17], and MGPUSim [26].

Tools also exist to study GPU kernels on real GPUs. NVIDIA's SASSI [24] and NVBit [27] are used to instrument GPU kernels, similar to Pin [20] on CPUs. These were preceded by GPU Lynx [13] which provided similar capabilities. CUDAAdvisor [23] and CUDA Flux [5] are recent proposals to insert instrumentation during the compilation process. Notably, these are all NVIDIA-specific, and we are unaware of instrumentation support for non-NVIDIA GPUs.

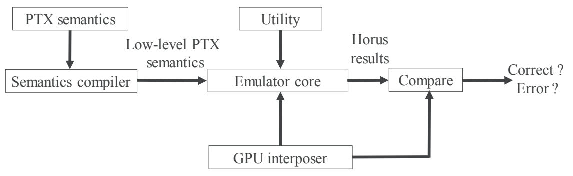

Fig. 1. Horus block diagram

CUDA kernels can be emulated on CPUs [25], an effort culminating in GPU Ocelot [12] that could emulate GPU kernels on CPUs and even instrument them. Recently, a full-system dynamic binary translator for ARM-based systems, including GPUs has been proposed [18].

Cycle-level simulators and emulators become obsolete rapidly as GPU hardware and their software stacks evolve. Researchers cannot continue to rely statistics from these tools. Another problem is that the design of many of these tools is a monolithic design, making them difficult to reuse or extend.

Our tool, Horus, is currently a functional simulator (i.e. emulator). Its modular design consists of five main components that are connected with well-defined interfaces. To keep up with the rapid pace of GPU evolution, we *generate* most of the emulator instead of writing it manually. This will also allow us to explore high-performance emulation techniques like JIT compilation in the future.

Horus is built on top of the first formal semantics for NVIDIA PTX instructions. This is used by a semantics compiler to generate the emulator for GPU applications. This reflects a CPU trend [3], [9] to define low-level semantics for ISAs in order to improve fidelity, reusability, and portability. Horus also allows generic "Utility" components that can plug in to the emulator to study, analyze and modify the behaviour of GPU kernels. Since GPUs have their own parallel "universe" of compilers, linkers, and loaders, the ability to reuse the emulator cleanly should significantly reduce effort for such components.

II. Horus Design and Current Implementation

Horus is a functional emulator for NVIDIA GPUs written in a dynamic language (Python) so we can take advantage of JIT compilation later [2], [19]. It consists of five main components:

DOI 10.1109/ISPASS48437.2020.00020

a GPU interposer, an ISA semantics (here, PTX semantics), a semantics compiler, an emulator core, and a utility component.

Figure 1 shows the Horus workflow and how its components interact with each other. We begin with an ISA-level semantics. Currently, we support the NVIDIA PTX virtual instruction set, whose semantics we encode in a custom DSL (a subset of Python). Our semantics compiler uses these high-level semantics (which only contain functional behaviour of the instruction and are *not* executable) to generate executable low-level semantics which also contain additional book-keeping such as maintaining the program counter, and so on. These low-level semantics fully specify the execution of a GPU instructions at a functional level and are executed by the Horus emulator core. The emulator core calls out to the Utility component, which can further modify the behaviour of these semantics at runtime. The distinction between high and low-level reduces duplication by creating generic templates that are then specialized to machine types.

To emulate an application, we intercept its interactions with the GPU using a library interposer [4], [27]. Our interposer is automatically generated from the CUDA Device API header files (instead of the CUDA Runtime API), and uses high-performance Linux tracing tools (LTTNG [11]) to record the trace offline. The trace records all API calls, their arguments, and all data moved from the CPU to GPU and back. After the user provides a GPU binary, Horus executes it while tracing the GPU API activity using the GPU interposer. We would like to actually intercept CPU–GPU activity at a much lower (and less frequently changing) level, as has been pioneered for ARM GPUs [18], notably the NoMaliGPU effort [10].

Our PTX semantics are based on the PTX documentation [8], though we introduce additional semantics for thread divergence [14], thread/warp/block communication, and synchronization that are not specified operationally in the documentation. Our handwritten high-level semantics (e.g. `dst = src1 + src2` for `add`) specifies a template for a PTX instruction using a Python-like syntax. From this, we generate executable lower-level semantics (Listing 1), which uses appropriate utility functions (`add` on line 13) to implement the actual semantics according to the argument types. The arguments capture notions of signedness (`sign`), rounding (`roundModifier`), etc. Warp-level instructions (e.g., barriers, shuffles) use a two-part semantics, one for thread and one for warp, but are not covered here for lack of space. Some instruction semantics are not fully equivalent (e.g., `cos`, `sin`) since they use undocumented approximation tables.

Horus can report statistics such as memory operations, and divergence stack depth. Timing-reliant statistics are not currently supported.

III. Evaluation

We evaluated the semantics for individual instructions by comparing to a real GPU. Then, we ran full applications, though the `fma` instruction's lack of associativity leads to differences with hardware. However, our cumulative average error is less than 5% for Rodinia [6] and Polybench [16].

```
1  def execute_add(sign, sat, cc, roundModifier,
       bitWidth, dst, src1, src2, number,
       single_precision_src1, double_precision_src1,
       pred_inst_reg, invert, regFile, cc_reg, varType,
        thread_pc_count):
2      if (pred_inst_reg != None):
3          pred = (regFile[pred_inst_reg] if (not
       invert) else (not regFile[pred_inst_reg]))
4          if (not pred):
5              return thread_pc_count
6      thread_pc_count = (thread_pc_count + 1)
7      if ((number == None) and (double_precision_src1
       == None)):
8          number = single_precision_src1
9      elif ((number == None) and (
       single_precision_src1 == None)):
10         number = double_precision_src1
11     else:
12         number = number
13     tmp_dst = add(set_sign_bitWidth(regFile[src1],
       sign, varType, bitWidth), (set_sign_bitWidth(
       regFile[src2], sign, varType, bitWidth) if (src2
        != None) else number), varType, bitWidth)
14     if cc:
15         cc_reg.cf = int((regFile[dst] > ((2 **
       bitWidth) - 1)))
16     tmp_dst_2 = set_round(tmp_dst, roundModifier)
17     regFile[dst] = set_sign_bitWidth(tmp_dst_2, sign
       , varType, bitWidth)
18     return thread_pc_count
```

Listing 1. Compiler-generated low-level semantics for PTX addition that includes predicate checking and thread PC increment

TABLE I
GPGPU-Sim implementation of *mad*

PTX instruction	PTX documentation	GPGPU-Sim

mad.hi.u16	t = a * b
	d = t[31..16] + c
		t = a * b + c
		d = t[31..16]

mad.hi.u32	t = a * b
	d = t[63..32] + c
		t = a * b + c
		d = t[63..32]

Our testing revealed that GPGPU-Sim incorrectly implements PTX semantics for some instructions such as `mad`, `fma` and `mul24`. The difference in `mad` semantics between PTX and GPGPU-Sim are shown in Table I where *t* is temporary, *d* is destination, *a*, *b* and *c* are inputs. Essentially, GPGPU-Sim computes a $a*b+c$ before extracting high-order bits, whereas the PTX semantics adds c to the high-order bits of $a*b$. We do note that Horus is currently 9x slower than GPGPU-Sim for small data sets of some Rodinia and Polybench benchmarks.

IV. Future Work

Currently, instrumentation must be built into the emulator, but we are exploring an additional component that would allow users to write instrumentation outside the emulator using well-defined interfaces similar to Pin [20] and NVBit [27]. A key difference from the latter is to be vendor neutral and safe – instrumentation should be unable to crash or block progress of the emulator. We also aim to complete our semantics of PTX.

References

[1] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. General-purpose graphics processor architectures. *Synthesis Lectures on Computer Architecture*, 13(2):1–140, 2018.

[2] D. Ancona, M. Ancona, A Cuni, and N. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *OOPSLA 2007 Proceedings and Companion, DLS'07: Proceedings of the 2007 Symposium on Dynamic Languages*, pages 53–64. ACM, 2007.

[3] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL):71:1–71:31, January 2019.

[4] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174. IEEE, 2009.

[5] Lorenz Braun and Holger Fröning. CUDA Flux: A Lightweight Instruction Profiler for CUDA Applications. page 9. IEEE, 2019.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[7] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*. NVIDIA Corporation, 4.2 edition, 2012.

[8] NVIDIA Corporation. *PTX: Parallel Thread Execution ISA*. NVIDIA Corporation, 2.2 edition, 2017.

[9] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S Adve, and Grigore Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1133–1148, 2019.

[10] René de Jong and Andreas Sandberg. NoMali: Simulating a realistic graphics driver stack using a stub GPU. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 255–262, April 2016. ISSN: null.

[11] Mathieu Desnoyers and Michel Dagenais. The LTTNG tracer : A low impact performance and behavior monitor for GNU/Linux. In *Proceedings of the Linux Symposium*, July 2006.

[12] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 353–364. IEEE, 2010.

[13] Naila Farooqui, Andrew Kerr, Greg Eisenhauer, Karsten Schwan, and Sudhakar Yalamanchili. Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 58–67. IEEE, 2012.

[14] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 407–420. IEEE, 2007.

[15] Xun Gong, Rafael Ubal, and David Kaeli. Multi2sim kepler: A detailed architectural GPU simulator. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 269–278. IEEE, 2017.

[16] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10. Ieee, 2012.

[17] Tony Gutierrez, Sooraj Puthoor, Brad Beckmann, and Tuan Ta. The AMD gem5 APU Simulator: Modeling GPUs using the machine ISA. In *Tutorial at ISCA 2018*, 2018.

[18] Kuba Kaszyk, Harry Wagstaff, Tom Spink, Björn Franke, Mike O'Boyle, Bruno Bodin, and Henrik Uhrenholt. Full-system simulation of mobile CPU/GPU platforms. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 68–78. IEEE, 2019.

[19] Derek Lockhart, Berkin Ilbeyi, and Christopher Batten. Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing jit compilers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267. IEEE, 2015.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[21] Geetika Malhotra, Seep Goel, and Smruti R. Sarangi. GpuTejas: A parallel simulator for GPU architectures. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, December 2014. ISSN: 1094-7256.

[22] Aaftab Munshi. *The OpenCL specifications*. KHRONOS Group, 2.0 edition, 2013.

[23] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. CUDAAdvisor: LLVM-based Runtime Profiling for Modern GPUs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 214–227, New York, NY, USA, 2018. ACM.

[24] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R Johnson, David Nellans, Mike O'Connor, and Stephen W Keckler. Flexible software profiling of gpu architectures. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 185–197. IEEE, 2015.

[25] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In José Nelson Amaral, editor, *Languages and Compilers for Parallel Computing*, pages 16–30, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[26] Yifan Sun, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, David Kaeli, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, and Carter McCardwell. MGPUSim: enabling multi-GPU performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture - ISCA '19*, pages 197–209, Phoenix, Arizona, 2019. ACM Press.

[27] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. Nvbit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 372–383, 2019.