Accelerating GPGPU Architecture Simulation

Zhibin Yu[*], Lieven Eeckhout[+], Nilanjan Goswami[#], Tao Li[#],
Lizy K. John[^], Hai Jin[&], Chengzhong Xu[*%]

* Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, China
\+ ELIS Department, Ghent University, Belgium
\# Intelligent Design of Efficient Architectures Lab, University of Florida, Gainesville, FL, USA
^ Department of Electrical and Computer Engineering, University of Texas at Austin, TX, USA
& Service Computing Technologies and System Lab/Cluster and Grid Computing Lab, HUST, Wuhan, China
% Department of Electrical and Computer Engineering, Wayne State University, MI, USA

zb.yu@siat.ac.cn, leeckhou@elis.UGent.be, nil@ufl.edu, taoli@ece.ufl.edu,
ljohn@ece.utexas.edu, jinhust@gmail.com, czxu@wayne.edu

ABSTRACT

Recently, graphics processing units (GPUs) have opened up new opportunities for speeding up general-purpose parallel applications due to their massive computational power and up to hundreds of thousands of threads enabled by programming models such as CUDA. However, due to the serial nature of existing micro-architecture simulators, these massively parallel architectures and workloads need to be simulated sequentially. As a result, simulating GPGPU architectures with typical benchmarks and input data sets is extremely time-consuming.

This paper addresses the GPGPU architecture simulation challenge by generating miniature, yet representative GPGPU kernels. We first summarize the static characteristics of an existing GPGPU kernel in a profile, and analyze its dynamic behavior using the novel concept of the divergence flow statistics graph (DFSG). We subsequently use a GPGPU kernel synthesizing framework to generate a miniature proxy of the original kernel, which can reduce simulation time significantly. The key idea is to reduce the number of simulated instructions by decreasing per-thread iteration counts of loops. Our experimental results show that our approach can accelerate GPGPU architecture simulation by a factor of 88X on average and up to 589X with an average IPC relative error of 5.6%.

Categories and Subject Descriptors

B.8 [**Hardware**]: Performance and Reliability — Simulation; C.1 [**Processor Architectures**] Single-instruction-stream, multiple-data-stream processors; C.4 [**Computer Systems Organization**] Performance of Systems — Simulation

Keywords: General Purpose Graphics Processing Unit (GPGPU), Performance, Micro-architecture Simulation

1. INTRODUCTION

In recent years, interest has grown substantially towards harnessing the explosive growth in computational power of graphics hardware to perform general-purpose tasks – referred to as GPGPU computing. GPGPU computing achieves high throughput by concurrently running massive numbers of threads enabled by general-purpose GPU programming models such as CUDA [1]. Unfortunately, existing GPGPU architecture simulators are sequential – a mismatch with the massive number of threads in the workloads and the number of parallel units in the graphics hardware – which indicates that GPGPU architecture simulation with typical benchmarks and input data sets is extremely time-consuming.

Table 1 shows the execution time of several CUDA benchmarks on a GPU device (NVIDIA GeForce 295) versus simulation time on a GPGPU performance simulator (GPGPU-Sim [2]). These measurements show that GPGPU performance simulation is approximately 9 orders of magnitude slower compared to real hardware. Given how computer architects heavily rely on simulators for exploration purposes at various stages of the design, accelerating GPGPU architectural simulation is imperative.

By characterizing existing GPGPU workloads, we find that existing CPU and GPU architectural simulation acceleration solutions cannot be readily applied to GPGPU simulation. First, the number of basic blocks and the instruction count per thread of GPGPU workloads are relatively small compared to typical CPU workloads. This implies that the prerequisites of CPU architectural simulation acceleration techniques such as sampling [3][4] do not apply. Second, the large number of branch instructions in GPGPU workloads prohibits the use of spreadsheet-based modeling techniques typically used for pure-graphics workloads based GPU performance evaluation.

We therefore propose a synthetic GPGPU workload framework that generates miniature proxies of workloads to address the GPGPU architectural simulation challenge. Experimental results show that our approach can speed up GPGPU architecture simulation by a factor of 88X on average and up to 589X, with an average IPC error of 5.6% across a broad set of GPGPU benchmarks.

2. WORKLOAD CHARACTERIZATION

To gain insight regarding how to accelerate GPGPU architectural simulation, it is important to characterize existing GPGPU workloads. We therefore developed a tool based on GPGPU-sim [2] to extract the features of GPGPU workloads at the instruction, basic block, and thread levels. We obtain three key findings. (i) The number of static basic blocks for most workloads ranges between 10 to 25 with an average of 23.4. This is relatively small compared to CPU benchmarks such as SPEC CPU (265.5 on average) and MediaBench (584.2 on average). (ii) The dynamic instruction count per thread varies from dozens to tens of

SIGMETRICS'13, June 17–21, 2013, Pittsburgh, PA, USA.
ACM 978-1-4503-1900-3/13/06.

Table 1. Execution time comparison of CUDA programs on real GPU device and GPGPU architectural simulator.

benchmark	grid	CTA	GPU time(ms)	Simulation time (s)
RPES	(65535,1,1)	(64,1,1)	0.524	333641 >3.8 days
TPACF	(201,1,1)	(256,1,1)	0.054	1484125>17days
BLK	(480,1,1)	(128,1,1)	0.918	1038931>12 days
PNS	(256,1,1)	(256,1,1)	0.699	1019169>11 days

thousands, which is extremely small compared to SPEC CPU and PARSEC benchmarks. (iii) GPGPU workloads contain more irregular code compared to pure graphics programs because of higher percentages of branch instructions in GPGPU workloads.

The first two findings break the prerequisites of CPU architectural simulation acceleration techniques such as sampling and statistical simulation. Sampling techniques select snapshots from a dynamic instruction stream, implying a large number of instructions per thread. Due to the small instruction count per thread for GPGPU workloads, sampling techniques cannot be used. One possible way to use sampling is to sample a small number of threads from the large number of threads in a typical GPGPU workload to be simulated in detail but this is highly likely to alter the inter-thread interactions. Statistical simulation generates a synthetic workload with similar characteristics as the original workload while dropping basic blocks that are seldom executed. Likewise, this approach cannot be applied to GPGPU architectural simulation because of the small number of basic blocks in GPGPU workloads.

The third finding prohibits the use of spreadsheet-based modeling techniques for pure GPU performance evaluation in the GPGPU case. If the number of threads or input data sets of a GPGPU workload is reduced, the simulation results such as IPC (Instructions Per Cycle) is highly possible to be altered. Therefore, the existing architectural simulation acceleration solutions for CPUs and pure GPUs cannot be readily applied to GPGPUs.

3. GPGPU BENCHMARK SYNTHESIS

We propose GPGPU workload synthesis to accelerate GPGPU architectural simulation. As shown in Figure 1, our approach consists of three steps. In the first step, a profile is collected by capturing the threads' inherent execution characteristics by executing the GPGPU workload with a given input. Subsequently, the profile is used as input to a code generator to generate a synthetic miniature GPGPU benchmark. In the final step, the synthetic benchmark is simulated on an execution-driven architectural simulator such as GPGPU-Sim.

There are two key differences between our GPGPU synthesis and previous CPU synthesis approach. (i) We use the control flow graph of the original workload as the code skeleton for its synthetic version. (ii) We employ the same number of threads and thread layout for the original and synthetic version. Both of these features are important towards keeping similar performance between the original and synthetic code versions in terms of warp divergence, shared memory bank conflict, and memory coalescing behavior. The key idea to reduce simulation time is to decrease the iteration counts of loops. As a result, the number of simulated instructions is reduced, and faster simulation times are achieved.

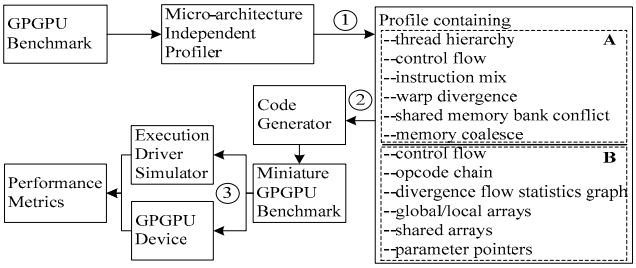

Figure 1. GPGPU benchmark synthesis framework.

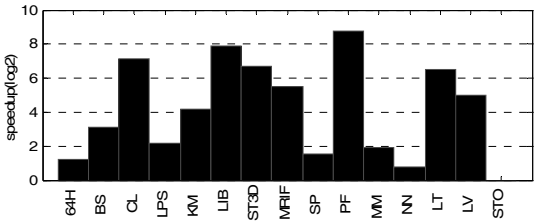

Figure 2. The speedup of micro-architecture simulation obtained by using synthesized GPGPU benchmarks

4. EVALUATION

We employ 15 benchmarks to evaluate the efficacy of our approach on the GPGPU-sim simulator. The benchmarks are taken from CUDA SDK, Rodinia, and Parboil, which are popular GPGPU benchmark suites. The GPGPU simulator is configured as a 56 SM (Streaming Multi-processor) GPGPU. Figure 2 shows the speedup of our approach. The speedup achieves 88X on average and up to 589X. The achieved speedups depend on the benchmarks' loop characteristics. For example, the iteration count of PF is large and that of STO is very small. The accuracy of our approach achieves 5.6% on average.

5. ACKNOWLEDGEMENTS

This work was supported by NSF China under grants 60973036 and 61272132.

6. REFERENCES

[1] NVIDIA CORPORATION, *CUDA Programming Guide Version 3.0*, 2010.

[2] Bakhoda. A, Yuan. G. L, Fung. W. L, Wong. H, and Aamodt. T. M. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 163-174, April 2009.

[3] Wunderlich. R. E, Wenisch. T. F, Fasafi. B, and Hoe. J. C. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA),* pp. 84-95, June 2003.

[4] Sherwood. T, Perelman. E, Hamerly. G, and Calder. B. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 45-57, Oct 2002.