Effective Profiling for Data-Intensive GPU Programs: Work-in-Progress

Hwiwon Kim
College of ICE
Sungkyunkwan University
Suwon, Korea
kim.hwiwon@skku.edu

Hyunjun Kim
College of ICE
Sungkyunkwan University
Suwon, Korea
hjunkim@skku.edu

Hwansoo Han
College of Computing
Sungkyunkwan University
Suwon, Korea
hhan@skku.edu

***Abstract*—GPU profilers have been successfully used to analyze bottlenecks and slowdowns of GPU programs. Several instrumentation tools for profiling GPU binaries are introduced, but these tools take little consideration into GPU architectures. In this paper, we investigate common factors of performance degradation on existing GPU profilers and provide design guide to improve the performance.**

***Index Terms*—GPU, CUDA, Profiler, Optimization**

I. Introduction

Recently, GPUs are widely used as accelerators on many application domains. Due to the parallel structure of GPU, where many threads execute the identical operations at the same time, programs with single instruction, multiple data (SIMD) workloads are efficient for GPUs. They actually perform the same operations on many data with multiple threads, which is called a single instruction, multiple threads (SIMT) execution model. In other words, to utilize the GPU as an accelerator efficiently, it is the key that maximizing thread-level parallelism (TLP) by simultaneously executing as many threads as possible.

In many programs which require lots of memory accesses, such as deep learning and big data analytics, memory system of GPU is often saturated due to the lack of GPU resources such as load/store units and caches [1]. In this case, each thread cannot perform compute until it gets all the requested operands from the memory, resulting in stalls for every compute operation. As such hindrance of the thread-level parallelism leads to low performance of GPU computing, there has been many researches to reduce the negative impact on data-intensive workloads after analyzing profiled memory access patterns.

Run-time GPU profilers are commonly used for this purpose, helping researchers understand the factors causing bottlenecks. However, the ways of recording and handling profiled data on existing profilers lack consideration for GPU architectures, which causes unnecessary overheads on profiling. This slowdown is significant especially in programs processing with a huge amount of data.

In this paper, we analyzed common overhead factors appeared in existing profiling tools. Based on our investigation, we propose several methodologies which are suitable for the GPU architecture. Overall overhead of profiling with our methodologies is 196×, and it is the fastest implementation compared to the existing profilers.

II. Existing GPU Profilers

SASSI [2], NVBit [3], CUDAAdvisor [4], and CUDA Memtrace [5] are representative instrumentation frameworks for profiling GPU programs. Note that the latest version of NVBit and CUDA Memtrace support instrumentation for the latest GPU architectures, while SASSI and CUDAAdvisor do not.

SASSI and NVBit are closed source tools for instrumenting user-defined profiling code before any and all *SASS* (native ISA of NVIDIA GPUs) instructions. The main difference between the two is that NVBit is a dynamic binary instrumentation tool and does not require recompilation after instrumentation, whereas SASSI requires the recompilation of the instrumented source code and can only instrument profiling code at compile time.

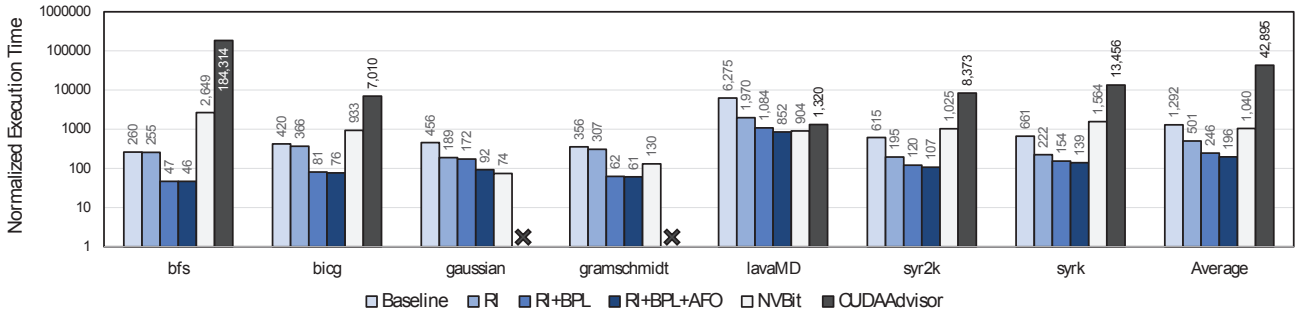
CUDAAdvisor and CUDA Memtrace are open source profiling framework. They both leverage the LLVM infrastructure to instrument LLVM IR-level profiling instructions at compile time. CUDAAdvisor provides several analysis on profiling such as memory divergence or reuse distance. However, it has several limitations on profiling data-intensive programs, as the size of the record buffer is fixed on device memory and the tool only flushes records on the buffer when each kernel exits.

A. Limitations on Existing Profilers

Every instrumentation framework listed in Section 2 has its own pros and cons depending on the design, but there are several common limitations. To design a cost-effective profiler, these performance overhead issues must be addressed.

1) Limited Memory Resource: GPU has hierarchical memory that takes the benefits of spatial and temporal locality. However, given that the GPU profiler produces large logs, great amount of profiling logs are stored in memories and transacted between them. As GPU has limited amount of resources such as bandwidth and memory capacity, GPU memory subsystem under heavy profiling suffers from bandwidth saturation or capacity shortage. This makes both profiling and kernel data on caches evicted frequently, which leads to underutilized memory system hierarchy and overall performance slowdown.

978-1-7281-9192-8/20/$31.00



Application	Benchmark Suite	Input Data	Application	Benchmark Suite	Input Data
bfs	Rodinia	graph4096.txt	lavaMD	Rodinia	-boxes1d 10
bicg	Polybench	1024×1024	syr2k	Polybench	128×128
gaussian	Rodinia	matrix1024.txt	syrk	Polybench	128×128
gramschmidt	Polybench	2048×2048			

Fig. 1. Execution times normalized to kernels w/o profiling (× in the graph means no result due to the profiler error)

2) Serialized Execution: Since there are thousands of threads executed simultaneously, a large amount of records are generated at the same time. To avoid data collision, all records must be stored in a record buffer orderly using an atomic operation. This makes dependencies across all threads in an application. As mentioned in Section 1, GPUs utilize the maximum TLP to reach their peak throughput. However, serialized execution caused by atomic operations reduces TLP, which often becomes significant overheads on profiling.

III. Effective GPU Profiling

As the base implementation of an open-source cost-effective profiling, we chose CUDA Memtrace. It has record producers on device-side and record consumers on host-side. They communicate through record buffers and record buffer pointers reside in host mapped-pinned memory. For the overhead factors taking a look, we changed the structure which was causing the overheads and analyzed the impacts to the overall profiling performance.

A. Memory Shortage Reduction

In order to handle the large profiling data, we focused on both reducing the number of transactions and the size of the profiling data itself to reduce the overhead. As all threads in a warp execute the same instruction, records generated by all threads can be represented as a single record. We applied *RI* (*Record integration*), which combines multiple records in a warp together.

The design of *RI* is as follows. When writing records, record producers first write metadata (*header*, which includes information of an instruction and thread block), and then write the actual data to be profiled repeatedly. It is reasonable to write both header and data in contiguous memory space in terms of a thread. However, if we consider it in terms of a warp, it is inefficient since a single warp writes in stride pattern on both header-write and data-write. To address the problem, we changed the structure of a warp record. We divided the warp record into two regions, header and data. The warp record now have both multiple headers and multiple data in two different regions. In fact, all headers of a warp have the same value. We merged the duplicated headers of a warp. This not only reduced the data size but also cache lines required by a warp.

B. Serialized Execution Overhead Reduction

In order to alleviate the overhead of atomic operations, it is essential to reduce the latency of each atomic operation. Atomic operations on each level of memory hierarchy have different latencies. So atomic calculations on higher hierarchy level is a key to reduce the overhead of serialized execution. In the CUDA Memtrace, atomic operations of device are done in host mapped-pinned memory, which has higher latency than device global memory from the perspective of GPU threads. Thus, we applied *BPL* (*Buffer pointer localization*), which moves the record buffer pointer from host memory to device memory. This lowers the latency of the whole atomic operation effectively.

Even though the latency of atomic operation is shortened, there is still room for improvement. When a record buffer is fully filled, all atomic allocations are hung until the buffer is flushed completely. To reduce such overheads, we applied *AFO* (*Atomic-flush overlapping*), which overlaps the latencies of both atomic operations and buffer flushing. This pre-allocates the record buffer pointers regardless of the buffer flushing. Pre-allocated warps, which are allocated but corresponding buffer is not flushed yet, keep track on the buffer to write data when the buffer is available.

IV. Experiment

A. Experiment Environment

Our experimental evaluation was performed on an Nvidia Tesla V100 GPU along with an Intel Xeon Silver 4110 CPU.

All profilers are tested with LLVM 10.0 and CUDA Toolkit 10.1 on Nvidia 440 driver, except for CUDAAdvisor (LLVM 4.0 and CUDA Toolkit 7.0) since it does not support recent development environments. To compare our profiler to other existing GPU profilers, we chose a collection of benchmarks from polybench/GPU [6] and rodinia [7], which are selected based on the set of benchmark used by CUDAAdvisor. As CUDAAdvisor has limitation on its input size, we used smaller input data on some benchmarks for comparison.

B. Profiling Overhead Analysis

The result of the experiment, which measured the sum of kernel execution times, is shown in Figure 1. *Baseline* in the figure is our profiler with no introduced methodologies is applied. It is CUDA Memtrace with some modifications, reducing several host-side overheads that are not relevant to the overheads of devices. Compared to the *baseline*, *RI, RI+BPL*, and *RI+BPL+AFO* improved by 61%, 81%, 85%, respectively. *RI+BPL+AFO* shows the slowdown by 196 times in average compared to the program without profiling overhead, which is a reasonable result compared to NVBit (1,040×) and CUDAAdvisor (42,895×, without failed benchmarks).

V. Conclusion

In order to design an effective GPU profiler, we first analyze the main overhead factors of the existing profilers. Then, we propose several methodologies to reduce these overheads. Our profiler with the proposed methodologies shows 81% reductions of profiling time compared to the state-of-the-art GPU profiler.

Acknowledgment

This research is based upon work supported by the National Research Foundation in Korea under PF Class Heterogeneous High Performance Computer Development (NRF2016M3C4A7952587).

References

[1] H. Kim, S. Hong, H. Lee, E. Seo, and H. Han, "Compiler-Assisted GPU Thread Throttling for Reduced Cache Contention," In Proc. *48th International Conference on Parallel Processing (ICPP)*, 2019, pp. 1–10.

[2] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of GPU architectures," In Proc. *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 185–197.

[3] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," In Proc. *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 372–383.

[4] D. Shen, S. L. Song, A. Li, and X. Liu, "CUDAAdvisor: LLVM-based runtime profiling for modern GPUs," In Proc. *2018 International Symposium on Code Generation and Optimization (CGO)*, 2018, 214–227.

[5] A. Matz and H. Fröning. "Quantifying the NUMA Behavior of Partitioned GPGPU Applications," In Proc. *12th Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2019, pp. 53–62.

[6] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," In Proc. *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing", In Proc. *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54.