Profiling CUDA Benchmarks for Performance Analysis on Modern GPUs

2021-03-06

Jakob Evans, Andrew Eljumaily, Matyas Krizek, and Shubbi Taneja

Sonoma State University

Profiling CUDA Benchmarks for Performance Analysis on Modern GPUs

Jakob Evans, Andrew Eljumaily, Matyas Krizek, Shubbhi Taneja

Sonoma State University, Rohnert Park, California, USA

Abstract

High-Performance Computing (HPC) systems are becoming highly heterogeneous as we enter the exascale era. Performance as well as power analysis of applications on hardware components such as Graphics Processing Units (GPUs) is thus of high interest and importance to the computing industry. In this paper, we delve into evaluating the performance characteristics of two state-of-the-art NVIDIA GPU architectures: Volta and Turing. We select four CUDA applications, namely, Gaussian Elimination, Lower-Upper Decomposition, Stream Cluster, and Jacobi for profiling. We aim to study how these applications make effective use of the hardware by collecting and analyzing the profiling data available through six performance counters, laying the framework for future analysis under power and energy constraints. Such analysis is a precursor to identifying performance and scalability bottlenecks, improving GPU occupancy and utilization, and reducing the power print of HPC applications.

1. Introduction

While the performance of multicore processors is at an all time high, there is a need for added parallelism and scalability in engineering, financial, and scientific applications. As a result of this, accelerator-based systems have seen tremendous growth in the recent years for use in computation-heavy applications as they significantly reduce the computation time of High Performance Computing (HPC) applications. Graphics Processing Unit (GPU) based accelerators have become common for general-purpose computing beyond just 3D rendering. However, this comes at the cost of high power usage.

In the past, parallelization on popular architectures would be implemented using CPUs. These employ a few cache memory levels and a large number of computing cores while using a small number of software threads.

Today's GPUs consist of thousands of computing cores which have the ability to execute multiple threads concurrently. The GPU's ability to run several thousands of threads can speed up applications, sometimes even by 100x, when compared to the applications being run solely on a CPU.

Previous research [1][2] suggests that performance counters are a primary source for gaining insight into hardware activities. Performance counters give users access to low-level hardware activities. With such insights, it becomes possible to establish strong correlations between energy efficiency and performance.

Modern GPUs are capable of providing data from between 100 and 200 hardware counters when executing a CUDA application or more specifically, a CUDA *kernel* (parallel portions of an application are executed on the device as kernels). These counters are also referred to as *events*. On the other hand, another term called *metric* is used to represent a kernel's characteristic, which is calculated based upon one or multiple events [3]. Since metrics are statistics built from performance counters that best explain hardware events [1], in this paper, we use the six metrics that best represent the system performance that we are interested in.

The rest of this paper is organized into the following sections. Section 2 describes related work while Sec. 3 discusses experimental methodology as well as the chosen applications and metrics. Section 4 describes the nature of our chosen CUDA application and metrics. Next, Sec. 5 presents experimental results and discussions. Finally, Sec. 6 discusses our conclusions.

2. Related Work

In this section, we will delve into some of the existing work that has inspired our research.

In [1] [2] [3], authors combine hardware performance counter data with machine learning and advanced analytics to accurately model power and performance efficiency for GPU systems which is difficult to achieve in the absence of any standard performance and power models. [2] also presents a survey on GPU profiling, modeling and simulation methods. In [3], authors use memory bandwidth, instruction and memory latency to build their model for predicting power.

In their research, [4] present an integrated GPU Power and Performance model. Using the runtime and power consumption of applications, their dynamic profiling model is able to eliminate the overhead of runtime profilers.

3. Experimental Methodology

In this section, we will describe the experimental methodology by focusing on the two tested systems – *Lassen* and *Monolith* (see Table 1). Following previous research, we select kernels that stress the GPU in different ways. the detailed description of four chosen CUDA applications, namely, Jacobi, LU Decomposition, Gaussian, and Stream Cluster; the latter three belong to a popular benchmark suite called Rodinia [5].

Data collection: Analysis of the application requires us to not only understand the characteristics of our application but how effectively it runs on the GPU. We use a performance analysis tool called NSight for the initial profiling of the chosen applications. Collection of performance metrics is the key feature of NVIDIA Nsight Compute [7]. On the host system, the user launches Nsight Compute CLI which in turn starts the actual application as a new process on the target system (device). When a kernel launch is detected, the libraries can collect the requested performance metrics from the GPU. The results are then transferred back to the user [7].

By default, NSight collects a relatively small number of metrics. Since there is a huge list of metrics available, we use the *--metrics* options on the CLI to specify the six metrics that we are interested in. We are aware of the runtime overhead incurred by NSight when collecting the profiling data.

4. CUDA Benchmarks and Metrics

In the following subsections, we will describe each of the four chosen CUDA applications and six performance counters chosen for the performance analysis. A CUDA event corresponds to a single hardware counter value which is collected during kernel execution and a metric is a characteristic of an application that is calculated from one or more event values [7].

4.1 Benchmarks

The execution time for each benchmark when executed with multiple block sizes (4x4, 8x8, 16x16, 32x32) on two systems, *Lassen* and *Monolith*, is available in Tables 2 and 3.

Table 1. Differences between the hardware specifications of the tested systems - *Lassen* and *Monolith*.

Criteria	Lassen	Monolith
GPU Architecture	Volta	Turing
Nvidia GPU Model	Tesla V100 SXM2 16 GB	RTX 2080 Ti
Compute Capability	7.0	7.5
Number of SMs	84	68
Memory/SM	96 KB	64 KB
Memory Bandwidth	900 GB/sec	616 GB/sec
L2 Cache Size	6144 KB	5632 KB
Cuda Version	10.1.243	10.0.130
Nsight Version	2020.1.0	2020.1.2

4.1.1 Jacobi

Jacobi [6] is an application which solves dense linear algebra problems spending a lot of time computing an approximate solution for a system of linear equations in the form of Ax = b. It achieves this by estimating the initial value for x and iteratively updating its value until it converges. If the Matrix A is diagonally dominant, the Jacobi method will always converge and due to this, Jacobi is a CPU-bound benchmark.

4.1.2 Stream Cluster

Stream Cluster (SC) is a data mining algorithm that finds a predetermined number of medians so that each point is assigned to its nearest center, given a stream of input points. Through initial profiling, we found this benchmark to be more memory-bound than CPU-bound.

4.1.3 Gaussian Elimination

Gaussian Elimination solves for all variables in a linear system row by row. The algorithm must synchronize before each iteration which allows the values to be computed in parallel. Gaussian elimination is an essential step in LU Decomposition. This benchmark belongs to a dense linear algebra domain and is found to be memory-bound.

Table 2. Benchmark run times for the *Lassen*. All timings are in *seconds*.

Block Size / Benchmark	Jacobi	Gaussian Elimin.	SC	LUD
4x4	173.48	101.50	1311.29	322.87
8x8	53.84	101.82	1314.34	323.34
16x16	53.78	101.87	1287.77	325.97
32x32	53.65	101.43	1296.86	323.37

4.1.4 LU Decomposition

LU Decomposition is a memory bound algorithm which finds the solutions for a set of linear equations given matrix A. LU is within the dense linear algebra domain. Using row reduction, it finds the upper and lower triangular matrices. Once lower (L) and upper (U) triangular matrices are found then you have a decomposition of original matrix A.

Table 3. Benchmark run times for *Monolith*. All timings are in *seconds*.

Block Size / Benchmark	Jacobi	Gaussian Elimin.	SC	LUD
4x4	424.98	98.99	1340.10	57.44
8x8	72.75	99.53	1347.57	28.47
16x16	73.02	98.03	1341.05	14.23
32x32	72.63	98.68	1342.86	7.04

4.2. Metrics

We provide a brief description of the six CUDA metrics chosen for experimentation in Table 4.

5. Results

Next, we will present the results of the initial profiling of the chosen CUDA benchmarks. More specifically, we will focus on comparing the global memory throughput with the instructions executed on the device (GPU). Profiling is an integral part of parallel program development. Profiling can help us pinpoint the parts of the program that contribute the most to the execution time. In the case of a GPU program, profiling can help us understand the factors limiting or preventing the GPU from achieving its peak performance. For instance, collecting cache misses (second and third metric in Table 4) allows us to track power-hungry memory hierarchy activities for the next steps of this work. Monitoring active warps gives us information on the occupancy as the theoretical occupancy acts as upper limit to active warps and consequently also eligible warps per SM.

Table 4: Selected CUDA metrics and their description.

Name	Description
Dram__bytes.sum.per_second	Measures read/write throughput to device memory in bytes per second, and is a good indicator of memory utilization by an application.
L1tex__t_bytes.sum.per_second	Measures the number of bytes that are read/written to the L1 cache per second.
Lts__t_bytes.sum.per_second	This metric records L2 cache byte hit rate per second.
Sm__inst_executed.sum.per_second	This helps us record L1 cache byte hit rate per second. This metric was chosen to show the amount of computation the entire GPU is doing at the sample time.
Sm_warps_active.avg.pct_of_peak_sustained_active	Measures occupancy of running kernel, or the ratio of the active warps per multiprocessor and the maximum number of possible active warps.
Smsp_cycles_active.avg.pct_of_peak_sustained_active	This metric was chosen because it shows the fraction of cycles where the SM has useful work to do.

As per NVIDIA's CUDA documentation [8], GPU's Streaming Multiprocessors (SMs) are that part of the GPU that run CUDA kernels; each SM contains thousands of registers, several levels of caches, warp schedulers, and execution cores. The SM also has a maximum number of blocks that can be active at once. A higher occupancy (preferably over 95%) is desired and it can be increased by increasing block size. We vary the block sizes for all our benchmarks across the two systems to observe the change in the overall performance with increasing block sizes.

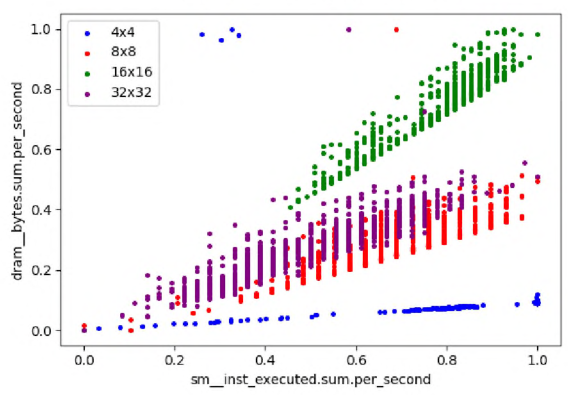


Figure 1: DRAM Bytes Vs Instructions Executed For Varying Block Sizes
Benchmark: **Jacobi** System: **Lassen**

In Fig. 1, we plot the memory throughput (dram_bytes.sum.per_second) against the instructions executed (sm_inst_executed.sum.per_second) for Jacobi on the *Lassen* system. We execute a CUDA implementation of Jacobi with four block sizes - *4x4, 8x8, 16x16, and 32x32.* Due to the wide range in data between block sizes, we normalized all values with min-max normalization; this assists us with visualizing data across benchmarks easily. With smaller block sizes, the cache size on Lassen is significantly larger to fit all the data required and thus, we observe fewer memory I/O. As the block sizes increase, the number of memory accesses increase linearly as lesser data is now able to fit in the caches on the GPUs.

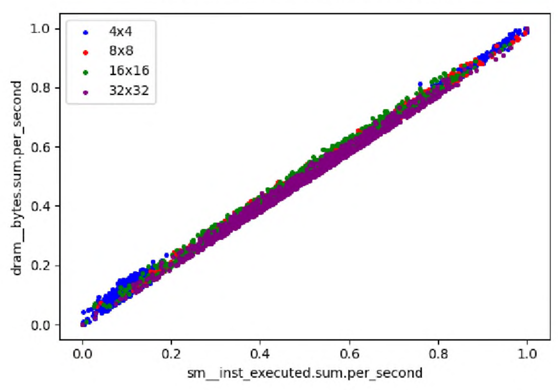


Figure 2: DRAM Bytes Vs Instructions Executed For Varying Block Sizes
Benchmark: **Jacobi** System: **Monolith**

In Fig. 2, we plot with the same metrics as Fig. 1 for the *Monolith* system. As Jacobi is a computation heavy application with comparatively fewer memory calls, it spends the majority of its execution time performing computations. As the block size goes up, we see similar trends like on *Lassen* (Fig.1) but since *Lassen's* caches are much larger in size, we believe that Jacobi has more cache misses on Monolith when compared to Lassen.

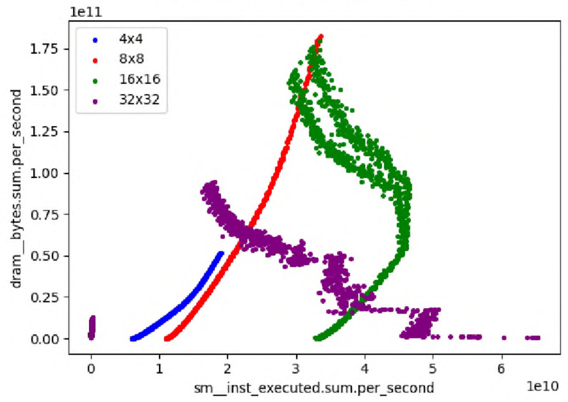


Figure 3: DRAM Bytes Vs Instructions Executed For Varying Block Sizes
Benchmark: **Gaussian** System: **Lassen**

To observe the performance of a more memory intensive application, we run a CUDA implementation of another benchmark called Gaussian Elimination on both the *Lassen* (Fig. 3) and *Monolith* (Fig. 4). There is a noticeable similarity between *Lassen* and *Monolith* when observing memory throughput for a given number of instructions executed. We observe that for smaller block sizes (i.e., 4x4 & 8x8), the number of memory accesses increase linearly with the number of instructions executed.

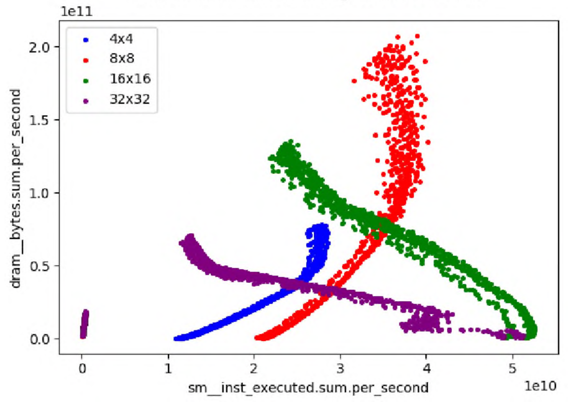
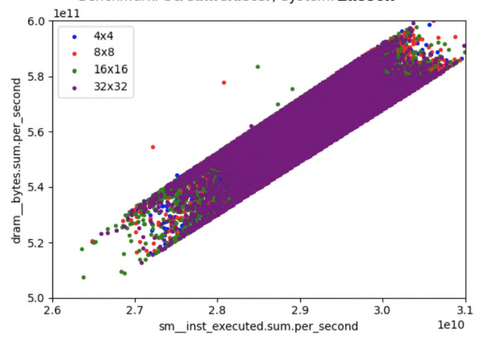


Figure 4: DRAM Bytes VS Instructions Executed For Varying Block Sizes
Benchmark: **Gaussian**, System: **Monolith**

Further, there is a decline in memory accesses as the block size increases (see block sizes 16x16 and 32x32 in Fig. 3 and 4) and we believe the reason for this that later stages in Gaussian Elimination reuses a lot of data in the cache memory for computations leading to fewer memory requests as the application progresses.

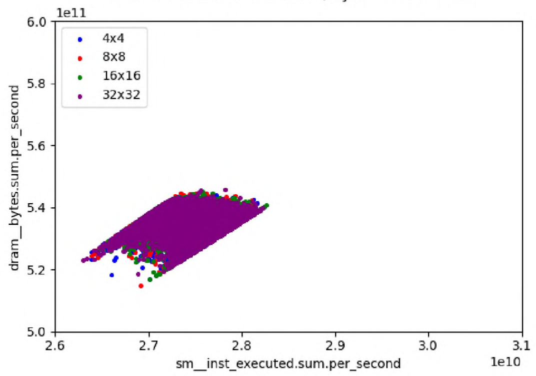
It is noteworthy that Volta GPU on *Lassen* has many more Streaming Multiprocessors (84) when compared with RTX 2080 on *Monolith* (68) and each of the SMs on *Lassen* also has much more memory (see Table 1). Further, Volta GPUs have a newly revamped L1 data cache so caches are more efficiently used on *Lassen* than on *Monolith*.

Figure 5: DRAM Bytes VS Instructions Executed For Varying Block Sizes
Benchmark: **StreamCluster**, System: **Lassen**



We show the effect of instructions executed on memory throughput (DRAM bytes) for SC benchmark in Figs. 5 and 6. Due to limited space, we are not presenting results

Figure 6: DRAM Bytes VS Instructions Executed For Varying Block Sizes
Benchmark: **StreamCluster**, System: **Monolith**



for LUD in this paper.

6. Conclusion

In this study, we collected six performance metrics available on two modern GPUs to analyze the profiling data available laying the framework for future analysis under power and energy constraints. We hope to use this initial analysis to identify performance and scalability bottlenecks, improving GPU occupancy and utilization, and reducing the power print of HPC applications. We picked four benchmarks, mostly from Rodinia benchmark suite, as representatives of popular domains like data mining, streaming applications and so on. In the future, we plan to look into each application in more depth to understand the location of its data on the GPU.

Acknowledgments

This work is supported by Koret Scholars Grant. We thank the system administrators at Sonoma State University for their guidance with the experimental setups.

References

[1] Song, S., Su, C., Rountree, B., & Cameron, K. W. (2013, May). A simplified and accurate model of power-performance efficiency on emergent GPU architectures. In 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (pp. 673-686). IEEE.

[2] Bridges, R. A., Imam, N., & Mintz, T. M. (2016). Understanding GPU power: A survey of profiling, modeling, and simulation methods. ACM Computing Surveys (CSUR), 49(3), 1-27.

[3] Zigon, B., & Song, F. (2020, June). Utilizing GPU Performance Counters to Characterize GPU Kernels via Machine Learning. In International Conference on Computational Science (pp. 88-101). Springer, Cham.

[4] Hong, S., & Kim, H. (2010, June). An integrated GPU power and performance model. In Proceedings of the 37th annual international symposium on Computer architecture (pp. 280-289).

[5] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S. H., & Skadron, K. (2009, October). Rodinia: A benchmark suite for heterogeneous computing. In 2009 IEEE international symposium on workload characterization (IISWC) (pp. 44-54). IEEE.

[6] Michel, M. Michel/CudaJacobi. (2015, March 3), Github repository, Retrieved from https://github.com/MMichel/CudaJacobi

[7] Nvidia Compute. Nsight Compute CLI :: Nsight Compute Documentation

[8] CUDA Architecture Overview, http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf