# Power Efficient GPU Programming

Alankrith Krishnan, Alisha Sawant
Department of Computer Science
New York University
{ak7380, ans698}@nyu.edu

***Abstract*— GPUs, though traditionally used for graphics purposes, are increasingly being marketed and used for non-graphical applications like running deep learning models, doing scientific simulations, etc. As more and more intensive applications are run on GPUs, the power consumed by them, and thus the energy dissipated by them, in the form of heat, has increased. In order to cool down the GPU, more energy needs to be supplied to increase the fan speed, or more money needs to be used to install liquid cooling systems. Thus, there is a need for GPUs to use as little power as possible for doing the same computations. In this project, we studied the effects of various optimizations on the power consumption (GFLOPs/Watt) of certain benchmarks when run on GPUs. From our experiments, we can conclude that memory accesses play a big role in increasing the power consumption and reducing that is key to increasing power efficiency.**

***Index Terms*—GPU, benchmark, power efficiency, parallel computing**

## I. Introduction

GPUs were initially primarily used for graphics purposes like rendering, shading, antialiasing, etc. However, the last several years has seen a tremendous rise in the use of GPUs for non – graphical purposes. With the advent of deep learning, there is a huge demand in the market for laptop and desktop computers to be manufactured with powerful GPUs. They have also seen a rise in the field of scientific computing, where they can be used to perform complicated simulations with high precision.

While GPUs offer a lot of computation power, they come with the caveat that they require a lot of power to operate. In addition to this, the GPU wastes power during its execution in the form of generated heat which needs to be dissipated. In most laptops and desktops, this means even more energy needs to be supplied to increase the fan speeds to cool down the GPU else we risk damaging the chip. In some sophisticated systems, liquid cooling systems are used but these are very expensive. Hence, it becomes very important to efficiently make use of the power consumed by the GPU by increasing the number of floating-point operations performed per watt of power. Thus, we look to improve the power efficiency of programs run on a GPU measured in GFLOPs/Watt.

## II. Related Work

Power efficiency is a hot topic of discussion, and with GPU's consuming a lot of power, a few papers have managed to survey the idea of reducing power consumption. The paper by Qasaimeh et. al[1] talks about using comparing the energy efficiency across CPU's, GPU's and FPGA's using Computer Vision Kernels. Their findings show that FPGA's are most energy efficient, while GPU's and CPU's perform badly in comparison. Since GPU's are more widely used as accelerators, energy efficiency is going to be an important point of contention when it comes to writing GPU code.

The paper by Huang et. al[2] talks about using scientific computing kernels as benchmarks to test GPU efficiency, they profile the amount of power used by multiple implementations of the GEM kernel (serial, multithreaded and GPU). They realize that while the GPU version consumes a lot more power, it finishes much quicker, and the power efficiency of the GPU version is much higher than the serial and multithreaded versions. In their findings, they also list that the power consumption does not vary much irrespective of the optimizations being made, which seems to indicate that the best way to achieve power efficiency might be directly linked to performance optimizations. They also note that they must explore the memory layout for potentially reducing power, which we build on in this paper.

## III. Proposed Idea

In this section we discuss our approach to testing the power efficiency of GPU programs and subsequently improving them. We started by creating a benchmark suite based on the Rodinia 3.1 Benchmark Suite[3]. It was designed for heterogenous computing infrastructures supporting CUDA, OpenMP, and OpenCL implementations. Rodinia comprises several benchmark applications across various domains, representing different Berkeley Dwarves[4]. We then used the functionalities of nvprof metrics to measure performance and power consumption. We followed this by analyzing sections of each benchmark program to find areas where the most power was being wasted. Based on these findings, we modified the benchmark codes to be more power efficient based on a list of possible optimizations that we compiled.

## IV. Experimental Setup

Although the Rodinia Benchmark Suite was pre-existing, there were several benchmarks within it that were written with very old CUDA code (the libraries used no longer existed). We removed these benchmarks from consideration. Further, after execution, some benchmarks returned floating point operation count equal to 0. These benchmarks, too, were removed.

We then wrote a bash script to automatically install the CUDA samples to the local file system for the benchmarks to use during execution. We wrote a second bash script which takes inputs of which benchmark to run, runs the original and our optimized CUDA code, and finally profiles both codes with nvprof to retrieve time, power and FLOP count. This script also links to a python script that parses the output of nvprof to calculate GFLOPs/Watt and output to a text file.

We made use of the following benchmarks to test our hypotheses:

1. Backprop - Train neural networks by propagating error and updating weights
2. CFD – Computational Fluid Dynamics
3. Gaussian - The Gaussian Elimination application solves systems of equations using the gaussian elimination method.
4. Heartwall - The Heart Wall application tracks the movement of a mouse heart over a sequence of 104 609x590 ultrasound images to record response to the stimulus.
5. LavaMD - The code calculates particle potential and relocation due to mutual forces between particles within a large 3D space.
6. NN – Calculates nearest neighbors.
7. Particle Filter – Particle collision detection and weight distribution.
8. Stream Cluster - Online clustering with a predetermined number of medians

We used NYU Courant's CUDA clusters – cuda1, cuda3, and cuda4 for our experiments. All experiments were run using Cuda Toolkit 10.1.

Since changing the algorithm is usually dependent on the specific program at hand, we looked at more general points in GPU programming code to make them more power efficient.

The scripts only run on GPU devices with a compute capability less than 7.5, since nvprof does not have any profiling options on GPU's with a compute capability greater than 7.5. While using nsight was an option, it does not provide metrics to find flop counts directly, and nsight has a small list of supported GPUs. nvprof seemed like a bigger and simpler option, and hence we chose to make it work on all GPU's up till Turing.

Some of the profiling metrics might overflow on devices with smaller memory, and hence we recommend running it only on GPU's which have a high register count (Titan Z overflowed on metrics, which is why we did not use cuda5).

## V. Results and Discussion

We compiled a list of optimizations that could potentially increase the power efficiency and edited each benchmark to incorporate these optimizations. We then compared the results to the original to see if the GFLOPs/Watt increased. The optimizations we tried were:

1. Use nvcc compile flag --use_fast_math

Setting this flag makes use of the fast math library. Thus, all mathematical operations are done by calling the corresponding function from the fast math library. For divisions and square roots, the fast approximation mode is enabled. This however did not always lead to better efficiency, we noticed that it lowered efficiency when the FLOP count is low and when it does not have a lot of divisions or square roots to make good use of it, probably due to the processing overhead involved in switching functions.

2. Using Unified Memory rather than traditional memory allocation

Unified memory is a single memory address space that can be accessed from any processor in a system. We changed the cudaMalloc() calls to cudaMallocManaged(). With this, the memory pages accessed by a processor are automatically and transparently migrated to the memory of that processor. We also changed cudaMemcpyAsync() to cudaMemPrefetchAsync() to prefetch data from host to device or device to host. This did not give us similar efficiency results, and it most cases using Unified Memory gave us marginally lower efficiency, hence we stuck to traditional memory allocation and copy.

3. Using Pinned Pages

Using cudaHostAlloc() and cudaHostFree() to create pinned pages on the host. As these pages are not swapped out by the OS, they facilitate higher data transfer speeds thus more floating-point operations can be carried out in the same time, using the same amount of power. This gave us a good performance boost in most cases where the size of the data is not small.

4. Using Streams

We made use of cudaMemcpyAsync() with streams instead of cudaMemcpy() if the program requires multiple transfers at the same time. This worked well when the number of transfers was greater than 3, using 2 streams alone for asynchronous data transfer did not make any difference to efficiency when done with blocking and non-blocking methods.

5. Using shared memory

Using shared memory more effectively to reduce the number of trips to the global memory. As the shared memory is physically closer, the power used to read from / write to it is less. It is also significantly faster for read/write operations than the global memory. Most of the benchmarks already utilized shared memory well whenever necessary, so there was not much scope for improvement in this regard.

6. Converting double precision to single precision

In cases where the applications do not require double precision floating points, we changed them to single precision floating points. Single precision floating point operations are both faster and consume less power than their double precision counterparts. Conversion to a lower precision gave us slightly better efficiency, and it only increases with the size of the problem and the dataset.

7. Staged concurrent copy

If the application allows for it, and the GPU supports it, the kernel can be split into multiple kernels and the chunk of the data required for each kernel is transferred concurrently with the execution of the previous kernel. Most of the benchmarks had no direct way to recall the same kernel with different data chunks, and hence this step was ignored.

8. Zero copy

Zero copy is a feature that allows threads on the GPU to directly access the host memory. However, this is only a performance gain when using integrated GPUs. Since we only have discrete GPU's on the cluster, this method was not tested.

After implementing different combinations of optimizations on our benchmark based on multiple factors, here are the optimized results of power consumption:

| | Power consumption (in GFLOPs/Watt) | |
|---|---|---|
| Benchmark | Original CUDA code | Optimized CUDA code |
| Backprop | 0.044661 | 0.350616 |
| Single precision cfd | 0.002562 | 0.002675 |
| Double precision cfd | 0.003456 | 0.003810 |
| Pre single precision cfd | 0.001039 | 0.001157 |
| Pre double precision cfd | 0.001475 | 0.001604 |
| Gaussian | 4.273872e-05 | 4.414154e-05 |
| Heartwall | 0.366035 | 0.386241 |
| LavaMD | 26.472007 | 27.417955 |
| NN | 0.213490 | 0.220319 |
| Particlefilter | 0.014127 | 0.014738 |
| Streamcluster | 0.001048 | 0.001128 |

**Table 1: Comparison of Power Consumption between base benchmark and optimized benchmarks**

## VI. Conclusions

Out of the methods that we experimented with, here are the results with their recommended use:

- Using the fast math library (--use_fast_math) is useful when pinpoint accuracy isn't necessary, and the application performs a lot of floating-point operations.
- Traditional Memory Management is slightly better than Unified Memory Management.
- Pinned Pages are useful but overusing them might lead to performance losses as well. This has to be done in moderation.
- Streams and non-blocking transfers are more efficient, as long as the number of simultaneous transfers is greater than 3.
- Single Precision Floating Points are more efficient if the loss of precision is acceptable.

## Acknowledgment

We would like to express our gratitude to our Professor Mohamed Zahran for pointing us in the right direction and getting us started off on this project. We would also like to thank NYU Courant for hosting the CUDA clusters that helped us test our hypotheses. A final word of thanks to the makers of the Rodinia Benchmark Suite, as the benchmarks served as a great starting point for our testing.

## References

[1] Qasaimeh, Murad, et al. "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels." (2019)

[2] Huang, Song, Shucai Xiao, and Wu-chun Feng. "On the energy efficiency of graphics processing units for scientific computing." 2009 IEEE International Symposium on Parallel & Distributed Processing. IEEE, 2009. H. Poor, *An Introduction to Signal Detection and Estimation*. New York: Springer-Verlag, 1985, ch. 4.

[3] Rodinia 3.1 Benchmark Suite: https://rodinia.cs.virginia.edu/doku.php

[4] Asanovic, Krste, et al. The landscape of parallel computing research: A view from berkeley. Vol. 2. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006..

[5] Bakhoda, Ali, et al. "Analyzing CUDA workloads using a detailed GPU simulator." 2009 IEEE International Symposium on Performance Analysis of Systems and Software. IEEE, 2009.

[6] Lew, Jonathan, et al. "Analyzing machine learning workloads using a detailed GPU simulator." 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2019.

[7] Hong, Sunpyo, and Hyesoon Kim. "An integrated GPU power and performance model." ACM SIGARCH Computer Architecture News. Vol. 38. No. 3. ACM, 2010.