General-Purpose computation on Graphics Processing Unit

In Depth Look at GPGPU Speed Up, CUDA and OpenCL

Jorden Hodges
Computing Sciences
Coastal Carolina University
Myrtle Beach, South Carolina,

ABSTRACT

General-Purpose computation on Graphics Processing Unit (GPGPU) is growing in popularity since its introduction in 2007 with the release of Compute Unified Device Architecture (CUDA) a parallel computing framework developed by NVIDIA. The scientific community has taken special interest in GPGPU due to its ability to solve highly computational problems that can be run in parallel.

In this paper, an example program using exclusively the CPU to execute matrix-matrix multiplication of different sizes was compared against a GPGPU program executing the same sized matrix-matrix multiplication. The results of this experiment are analyzed and explained with a focus on the speed up between the CPU and GPGPU program.

In general, there are two GPGPU frameworks that can be used to create programs utilizing the GPU including CUDA and OpenCL. Both allow data to be computed on the graphics card which improves performance in larger datasets that are able to be run in parallel. There are small differences between CUDA and OpenCL which will be discussed in this paper.

KEYWORDS

GPGPU, CUDA, OpenCL, parallel computing, GPU computing

ACM Reference format:

FirstName Surname, FirstName Surname and FirstName Surname. 2018. General-Purpose computation on Graphics Processing Unit : In Depth Look at GPGPU Speed Up, CUDA, and OpenCL. In *Proceedings of ACM Woodstock conference (WOODSTOCK'18). ACM, New York, NY, USA, 2 pages.* https://doi.org/10.1145/1234567890

THE NEED FOR GPGPU

For years, the Central Processing Unit (CPU) and Graphics Processing Unit (GPU) lived in separation. In which, the GPU took in vectors from the CPU, performed calculations and outputted the result of those calculations. The GPU was only capable of one way data flow. In contrast, the CPU was tasked with handling all calculations not related to vectors since it was capable of allocating memory and transferring data back and forth between multiple levels of caches, RAM, and storage devices. At the same time, CPU and GPU performance was increasing according to Moore's Law. For a while, this was acceptable. However, in recent years the rate of performance increase for the CPU has started to decrease due to the limitations of the materials used to make CPUs, the power wall, heat concerns, and other limitations.

In addition, applications were growing more demanding of faster processing for larger datasets especially in fields such as physics, economics, engineering, etc. This combination led to processor companies like Intel to increase the number of cores in their flagship CPUs. [2] The problem is that increasing the number of cores in CPUs is expensive due to all of the additional circuits needed in each core including the ALU, additional registers, caches, control units, etc.

In 2003, Mark Harris, NVIDIAs Chief Technologists for GPU computing, acknowledged that GPUs could be used for general purpose calculations. [2] NVIDIA GPUs have many cores that are designed to run in parallel most commonly known as CUDA cores. For comparison sake, CPUs typically have anywhere from 2 to 16 cores for most consumer grade CPUs. The number of cores can grow larger but this is only taking into account consumer grade CPUs. GPUs on

the other hand are composed of hundreds to thousands of CUDA cores. This makes GPUs the likely candidate to be used to supplement the performance gaps of CPUs.

DIFFERENCE BETWEEN CPU AND GPU

In addition to the many processors that GPUs are composed of, the structure of a GPU is also much simpler than that of a CPU. This is because GPUs use much more of its architecture for data processing rather than other tasks such as caching, control information, and ALU computation. In fact, the inside of a GPU at its heart consists of transistors dedicated to mainly data manipulation. [2]

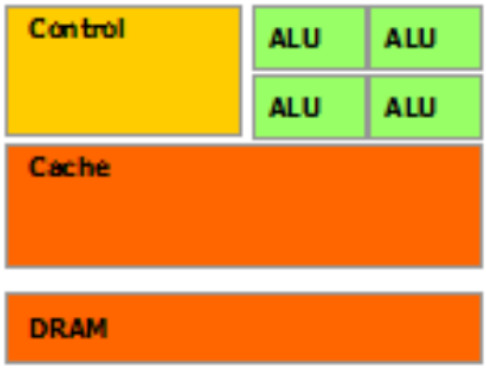

Figure 1: Overview of CPU architecture from NVIDIA [2]

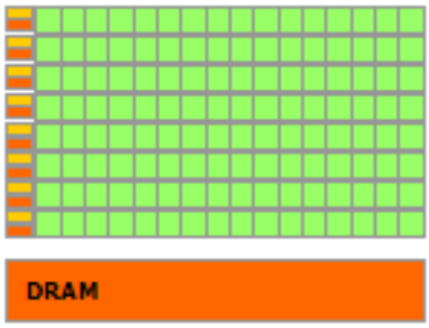

Figure 2: Overview of GPU architecture from NVIDIA [2]

Note that the CPU has a large percentage of its architecture devoted to the control unit which is used to handle instructions and it's cache which is used for storing temporary data as seen in figure 1. The leftover space is used for the ALU and the DRAM.

Note that the GPU does not have as much architecture dedicated to a control unit or to caching. Most of the GPU architecture is used for computation per figure 3. This allows for much better performance in data manipulation leading to widespread use of GPGPU especially related to floating point computation. [2] Furthermore, with multiple computational units, more operations are able to be performed at one time via parallelism.

EXAMPLE OF GPGPU SPEED UP

The purpose of the following example is to get an idea of how much GPGPU speeds up the processing of a simple matrix-matrix multiplication. For this example, the matrix example program provided by NVIDIA in the CUDA Developer Toolkit version 10.2 was used. This program was not necessarily made to show the performance of GPGPU but should be adequate with showing the basic performance increase offered.

This experiment was performed by starting with multiplying two 32 by 32 matrices together on CUDAs example matrix multiplication code and the CPU basic matrix multiplication code. The execution time of the CUDA example program and the basic matrix multiplication program was calculated by the running program and recorded in a table. The size of the matrix is incremented by 32 by 32 each time up until the size 448 by 448. The results of the experiment is seen in figure 3. The specifications of the machine that this test was run on were as follows: Intel core i7-8700k (no overclock), 32 GB DDR4 3200MHz ram, and a EVGA NVIDIA GeForce RTX 2080 (manufacture overclock only).

Matrices size	CPU Execution Time(ms)	GPGPU Execution Time(ms)	Speed Up (ms/ms)
32X32	0.079	0.1415	0.558303887
64X64	0.562	0.1745	3.220630372
96X96	1.9763	0.2093	9.442427138
128X128	5.1445	0.2443	21.05812526
160X160	9.3353	0.3086	30.25048607
192X192	15.9762	0.3391	47.11353583
224X224	25.1913	0.6721	37.48147597
256X256	37.5614	0.7057	53.22573331
288X288	55.3764	0.8025	69.00485981
320X320	78.221	1.2111	64.58673933
356X356	109.644	1.3656	80.28998243
384X384	135.569	1.8978	71.43481926
416X416	171.582	2.1495	79.82414515
448X448	226.769	2.6953	84.1349757

Figure 3: Resulting table of CPU and GPGPU Execution

There was a clear difference in the GPGPU and CPU execution times with the different matrix sizes. At the starting matrix size, the GPU execution time was actually worse than the CPU execution time. This is most likely due to the limited parallelism with a matrix as small as 32 by 32. In addition, pipelining in the CPU also helps to decrease the CPU execution time. Not to mention the GPU core frequency is about 1.6 GHz where the CPU core frequency is 4 GHz. Despite this one anomaly, throughout the rest of the experiment, the speed up between the GPU and CPU execution time increases. However, there are certain points where the speed up decreases briefly and then begins to increase again.

For example, in between the matrix size 192 by 192 and 224 by 224 the speed up decreased from a 47.1 times speed up to 37.5 times, but in the next iteration the speed up increases to 53.2 times speed up. This then happens again after matrix size 288 by 288 and 384 by 384. Presumably, this is because the matrix multiplications most likely reaching maximum amounts of parallelism causing the GPU to have to wait until previous calculations finish before moving to the next set of calculations. The fact that the decrease of speed up happens about every 3 iterations supports this assumption because the matrix grows by 96 by 96 every 3 iterations. Since this is a smaller dataset, growing by 96 by 96 does not increase the number of elements by that much. This makes sense when looking at the architecture behind the RTX 2080.

In GPUs, instructions are carried out in threads where each thread can carry out one instruction at a time. For square matrices of size N by N, the number of threads required to multiply two matrices of size N by N is equal to N^2. [3] This means for two matrices of size 96 by 96, 9,216 threads are needed to compute each resulting element in the result matrix for example. The NVIDIA GeForce RTX 2080 is based off of the Turing architecture from NVIDA. According to the guide for Turing architecture from NVIDIA's website, the maximum number of concurrent warps on a single Streaming Processor (SM) is 32, where each warp contains 32 threads. This would mean that each Streaming Processor can actively have 1024 threads (32 Warps x 32 threads) being used at one time. [4] In addition, the RTX 2080 has a total of 46 Streaming Processors. Thus, the RTX 2080 has a total of:

$$\boldsymbol{Total\ Threads = 46(32\ Warps * 32\ Threads)}$$

$$\boldsymbol{Total\ Threads = 47,104\ Threads}$$

So this means that when multiplying two square matrices together on the RTX 2080 the amount of threads required to calculate the resulting matrix needs to be less than or equal to 47,104 threads in order to be completed in one batch. With this information, it is easy to make a table about the batches required for each matrix multiplication as seen in figure 4 where the number of batches is equal to:

$$\#\ of\ batches = \frac{Threads\ required}{47,104}$$

The number of batches required to finish the matrix multiplication grows exponentially as the resulting matrix size grows exponentially as well. This is why the beginning small sized matrices have such small execution times on the GPU. Thus, the speed up does not decrease until the 224 by 224 matrix multiplication when the threads required crosses the 47,104 threshold for the RTX 2080, requiring over 50,000 threads to be executed in parallel. In other words, all computational structures are occupied and there is no threads available to execute the next batch. Thus, the remaining batches will have to wait until the batches before them are finished.

Matrices size	# of threads Required	# of batches
32X32	1,024	0.02173913
64X64	4,096	0.086956522
96X96	9,216	0.195652174
128X128	16,384	0.347826087
160X160	25,600	0.543478261
192X192	36,864	0.782608696
224X224	50,176	1.065217391
256X256	65,536	1.391304348
288X288	82,944	1.760869565
320X320	102,400	2.173913043
356X356	126,736	2.690557065
384X384	147,456	3.130434783
416X416	173,056	3.673913043
448X448	200,704	4.260869565

Figure 4: Table displaying the number of batches required for each matrix multiplication.

As shown in figure 5, the number of batches required and the speed up difference is directly correlated where the speed up difference is the difference of the speed up between the CPU and GPU execution time in the current matrix multiplication and the CPU and GPU execution time in the previous matrix multiplication. More specifically, when the number of batches increases the speed up decreases.

# of batches	Speed Up Difference
0.02173913	0
0.086956522	2.662326486
0.195652174	6.221796766
0.347826087	11.61569812
0.543478261	9.19236081
0.782608696	16.86304976
1.065217391	-9.632059859
1.391304348	15.74425734
1.760869565	15.7791265
2.173913043	-4.418120485
2.690557065	15.7032431
3.130434783	-8.855163161
3.673913043	8.389325886
4.260869565	4.310830548

Figure 5: Table displaying the number of batches required for each matrix multiplication and whether speed up increased or decreased.

As it turns out, the speed up increases between the CPU and GPGPU execution time exponentially until the first decrease in speed up as a result of requiring more than one batch to be completed as seen in figure 6. This makes sense when looking at a graph of the GPGPU execution time vs a graph of the CPU execution time as shown in figure 7. After the first speed up decrease, the speed up appears to increase linearly apart from the brief decreases in speed up about every 3 iterations. More than likely, the speed up is still increasing exponentially after the brief decreases in speed up, but there is probably not enough data in this small example to display an exponential growth in speed up. Surely if the amount at which the matrices sizes were incremented was smaller, an exponential growth of speed up would have been more prevalent. For example if the matrices sizes were increased by 8 by 8 instead of 32 by 32, an exponential growth would have probably been more prevalent.

This exponential growth in speed up is because the CPU execution time increases exponentially while the GPGPU execution time increases mostly linearly as seen in figure 7. This can be shown algebraically

since the speed up (S) is equal to the CPU execution time (CPU_T) over the GPGPU execution time (GPU_T).

$$S = \frac{CPU_T}{GPU_T}$$

The CPU execution time can be thought of being determined by the number elements needed to be calculated. As stated earlier, the number of elements needing to be calculated when multiplying a square matrix of size N by N by another square matrix of the same size is equal to N^2. This means that the CPU time is proportional to the number of elements needing to be computed. On the other hand in the case of the GPU, the number of elements needing to be computed are being divided amongst the many different threads to be computed. Because of this, the GPU time could be thought of being proportional to the logarithmic value of the number of elements squared.

$$S\alpha \frac{N^2}{log_{47,104}(N^2)}$$

Therefore as the dataset gets larger the CPU time will keep growing exponentially while the GPGPU time will keep growing slowly. As a result the speed up theoretically should grow exponential. Of course, since there is not an infinite amount of threads this is the not the case as the data needs to be computed in batches. With larger datasets there should be more evidence of exponential growth in speed up.

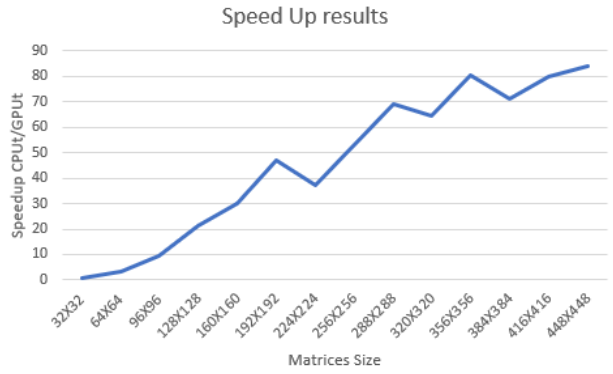

Figure 6: Graph displaying GPGPU speed up compared to CPU.

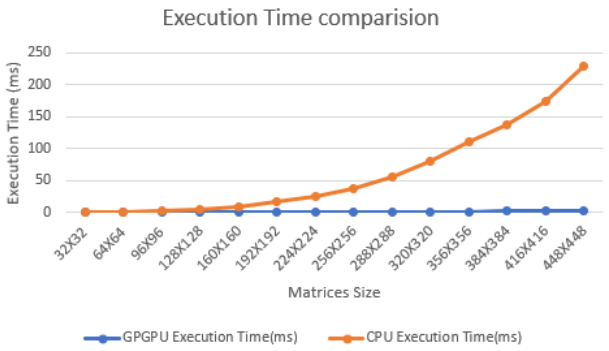

Figure 7: Graph displaying GPGPU execution time compared to CPU execution time.

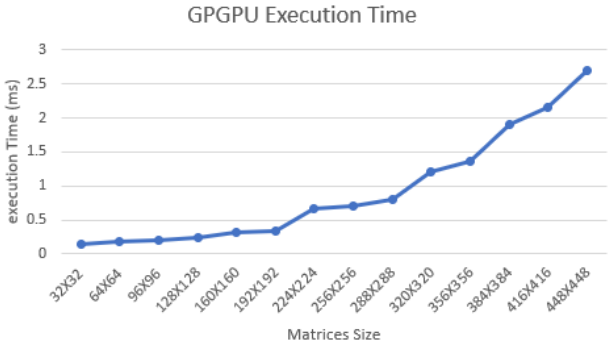

Figure 8: Graph displaying GPGPU execution time.

OPENCL VS. CUDA

When it comes to GPGPU programming, there are a few different options for frameworks. The two most common GPGPU frameworks are OpenCL developed by the Kronos Group and CUDA developed by the NVIDIA Corporation. CUDA and OpenCL are largely similar in their structure, but they do have small differences in their execution and availability. CUDA is a proprietary framework developed by NVIDIA to develop parallel programs that can run on NVIDIA GPUs. [6] It was first released in 2007. [1] It has since then been widely adopted and used on compatible NVIDIA hardware. OpenCL on the other hand is an open source GPGPU framework developed by the Kronos Group in partnership with other companies such as Apple that can be run on any GPU that has appropriate hardware implementation for GPGPU. [6] This means that OpenCL has the ability to be run on many more devices than CUDA as

OpenCL can be run on NVIDIA cards, AMD cards, and the new Intel cards.

Kernel compilation is one difference between OpenCL and CUDA. When compiling a CUDA kernel, it is compiled to PTX code which is a pseudo assembler language. This means that compiled CUDA programs are already in a low level language. Thus, the decoding of PTX code to machine code provides very little overhead in terms of the programs execution time. By contrast, OpenCL compiles all GPU kernels at runtime from languages similar to C. This is a necessary evil due to the fact that OpenCL needs to be able to be run on every compatible GPU. Unfortunately, this creates an overhead when compiling OpenCL programs leading to performance degradation. It is worth noting that the performance degradation is much more noticeable in smaller problems than in larger problems. Since GPGPU is mainly used for bigger problems, this should not prove to be too much of an issue in the real world. That being said, there is still degradation in problems of large size, nonetheless. [6]

As stated earlier, GPUs computational units are called threads and are arranged in what are called warp schedules in CUDA programming. When programming using GPGPU there needs to be a way to control which instruction goes to which thread to be executed. In addition, it was mentioned earlier that GPUs do not have the same control interface that modern CPUs have. This means that the user needs to carefully assign each instruction to a specific thread to be computed. Luckily, in both CUDA and OpenCL there is built in variables and methods to aid the programmer in scheduling these instructions through abstraction. In both frameworks, there is a variable dedicated to the thread number which can be identified in a one-dimensional, two-dimensional, and three-dimensional space. [6] In CUDA this variable is named threadIdx. Each thread is tagged to a thread block which also can be identified in a 1, 2, or 3-dimensional space as well. These variables help the programmer and compiler assign tasks to threads and keep track of which threads are handling which computation. In OpenCL, the variables effectively behave the same but are named differently. Instead of threads, OpenCL has work items which are organized in work groups which is effectively the same as NVIDIA's thread blocks. [6]

The CUDA toolkit comes with many CUDA libraries to aid the programmer in writing GPGPU code. OpenCL, on the other hand does not come with many pre-installed libraries but rather relies on open source projects to provide libraries. With that in mind, both GPGPU frameworks have a variety of 3rd party libraries available for install so this should not be much of an issue. [7]

When CUDA released in 2007, it only supported the C programming language. Today, CUDA supports multiple programming languages such as FORTRAN, Java, C++, C, Python, etc. Making it easier for multiple different developers of different languages to start programming in CUDA. [6] OpenCL, on the other hand, only supports C and C++. However, there are language extensions available to extend support to other programming languages like Java, Python, and FORTRAN from third party downloads. [7]

With the difference between OpenCL and CUDA, the question arises: "Which is better?" Well, like most other questions of which is better, it depends. CUDA is typically considered to be faster in most cases especially in the smaller problem sets. However, CUDA is only an option if you have an NVIDIA GPU. With that said, if you are working with NVIDIA GPUs, you should use the CUDA developer kit. CUDA is more optimized for NVIDIA GPUs and is considered to be faster and easier to write programs with due the variety of language support and libraries. Of course, if you do not have an NVIDIA GPU your only choice would be OpenCL.

APPLICATIONS OF GPGPU

Since the release of CUDA, GPGPU has been adopted in many fields to speed up the processing of data. One field that benefitted very early on from GPGPU, was the medical field, more specifically medical imaging. Images such as CT scans and Ultrasounds, would take several hours to process on CPUs alone. However, with the speedup of GPGPU, the processing of these scans takes only 10-15 minutes on a GPU. [6] There are many other areas that advantage greatly from the speedup of a GPU. These will not be talked about in detail in this paper. This list is provided by an Academic Journal written by multiple authors found in the resources section of this paper. [6] Linear algebra and large scale numerical simulations,

molecular dynamics, protein folding, finance modeling, signal processing, raytracing, physics simulation, speech and image recognition databases, sorting and searching algorithms, astrophysics, lattice QCD, and theoretical physics have benefited form GPGPU.

CONCLUSION

GPGPU has evolved since the release of CUDA in 2007. Today, GPGPU is used in many areas to speed up the computation of large datasets. The many threads in GPUs give the ability to process data many times faster than CPUs could have before. Looking at the experiment conducted in this paper, it is apparent that GPUs can offer valuable performance increases in certain areas. GPGPU is also growing and improving rapidly as the performance improvements for CPUs is beginning to slow and Moore's Law is being pushed back.

In the context of OpenCL and CUDA, both frameworks work well for creating GPGPU programs. The use of one over the other is going to largely depend on what your program is going to run on the most whether it be computers with NVIDIA, AMD, or Intel graphics cards.

For years to come, hopefully GPGPU programming will continue to be improved. Hope further benefits of GPGPU will be discovered to advance our modern world.

REFERENCES

[1] Martin Heller. 2018. What is CUDA? Parallel Programming for GPUs. (August 2018). Retrieved December 9, 2019 from https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html

[2] Bogdan Oancea, Tudorel Andrei, and Raluca Mariana Dragoescu. 2012. GPGPU Computing. DOI: https://arxiv.org/ftp/arxiv/papers/1408/1408.6923.pdf

[3] GPU Programming: CUDA Threading Basics. 2011. (Fall 2011). http://users.wfu.edu/choss/CUDA/docs/Lecture%205.pdf

[4] NVIDIA Corp. Turing Tuning Guide. 2019. (November 2019). Retrieved December 11, 2019 from https://docs.nvidia.com/cuda/turing-tuning-guide/index.html

[5] Peter Zunitch. 2018. CUDA vs. OpenCL vs. OpenGL. (March 2018). Retrieved December 11, 2019 from https://docs.nvidia.com/cuda/turing-tuning-guide/index.html

[6] Bogdan Oancea, Tudorel Andrei, and Raluca Mariana Dragoescu. 2013. GPGPU COMPUTING. Challenges of Knowledge Society, IT. Nicolae Titulescu University, Romania.

[7] Denis Demidov, Karsten Ahnert, Karl Rupp, and Peter Gottschling. 2013. PROGRAMMING CUDA AND OPENCL: A CASE STUDY USING MODERN C++ LIBRARIES. SIAM, 35, 5. DOI: 10.1137.