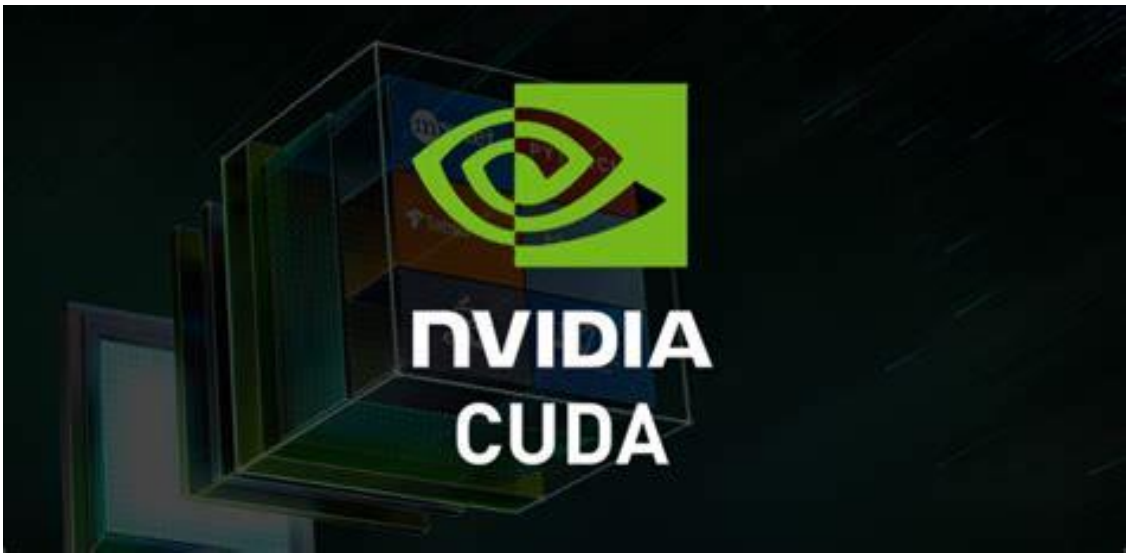


Performance Evaluation: CUDA GPU vs. OpenMP CPU

Garrett A. DiPalma

Nicolas Guerra

Jacob Johnson

Spring 2021

Dr. Apan Qasem

Table of Contents

Intro:

Our project was based on a performance evaluation between a GPU running CUDA architecture and an OpenMP CPU. We thought this comparison would be interesting, due to the nature of CUDA's parallel computing prowess and due to the fact that in general GPUs have far more cores - and thus more threads - than do standard CPUs. Because of this, our hypothesis was that the GPU would outperform the CPU on every program until the parameters exceeded the GPUs memory capabilities. The performance evaluations were gathered on one machine to reduce any variability in the results, and to keep consistency in our project. The GPU was an NVIDIA Turing [GeForce GTX 1660 Ti] with 1536 CUDA cores, each core having 1024 threads. The CPU was an AMD Ryzen 5 2600 Six-Core Processor with 12 threads.

To accurately compare the two processors, we collected benchmark data using the commands *perf stat* and *nvprof* on programs such as knapsack, matrixMul, vectorAdd, and gradeChecker. However, before we could gather performance data, not all of our chosen programs were able to run on CUDA or C architecture; it was necessary to rewrite them to work with the opposite architecture they were originally designed for. When the programs were re-written, performance code was also added to all versions of the programs to obtain the execution time and clock cycles. This re-coding allowed us to finally gather all the performance data we would need to make a proper comparison between our subjects. With this data, we assembled key statistics such as: clock rate and cycles, CPI, Execution times, cache performance, the overall performance of the OpenMP CPU and CUDA GPUs, and some cost/performance analysis. We analyzed and graphed these statistics to aid in comparing the processors.

The results of the performance evaluations were mostly in-line with our initial hypothesis. The CUDA GPU had about a 57% performance increase over the OpenMP CPU, which we will go into in more depth in the results section.

Team:

Garrett DiPalma:

Garrett was the ‘team leader’ and driving force behind the project idea and proposal. His main duty was to re-write the programs we used in the performance evaluation to work with the CUDA architecture. He also gathered all the performance data used in analysis; due to the fact his personal machine held both the GPU and CPU we compared. Garrett also contributed to the project report and presentation.

Nicolas Guerra:

Nicolas took all the performance data Garrett gathered and applied it to our performance analysis by graphing the principle metrics to obtain our results. Nicolas further helped in the report and presentation.

Jacob Johnson:

Jacob’s duties focused on the report and presentation. He was the ‘report lead’ and also created the visuals and main talking points for the presentation.

GPU and CPU Specs:

The GPU and CPU used in our performance comparison resided in Garrett's personal machine.

	CPU	GPU
Name	**AMD Ryzen 5 2600**	**Nvidia GTX 1660 Ti**
Cores	**6**	**1536**
Threads	**12**	**1,572,864**
Clock Speed (MHz)	**3900**	**1785**
Cache (KB)	**L1: 576 L2: 3000 L3: 16000**	**L1: 64 L2:1536**
Memory	**RAM: 16 GB**	**GRAM: 6 GB**
Parallelism	**OpenMP**	**CUDA**
Architecture	**Zen**	**Turing**
Power	**65W**	**120W**
Price	**$195**	**$300**

Tools:

Perf stat: Linux tool to show performance metrics

gcc: C compiler, used to compile .c files

nvcc: CUDA compiler, used to compile .cu files

Nsight Compute: Similar to perf, used to collect performance metrics

nvprof: Similar to perf, used to collect, percentage wise, GPU and CPU API calls and elapsed program time

Background:

CUDA:

CUDA architecture is a parallel computing platform developed in 2003 by Nvidia for general-purpose computing on Nvidia's own GPU's. The main goal of its creation was to aid developers by speeding up demanding applications for the parallel parts of computing. It was a major success in this regard. There are other competitive GPUs from companies such as AMD, but Nvidia's CUDA based GPUs are the industry benchmark. They dominate many fields that require high amounts of computing power: computational finance, climate sciences (modeling weather, climate, and ocean patterns), government defense and intelligence, medical services. In the realm of computer science, they are also the premier behind deep learning, machine learning, and are used in some of the fastest computers in the world.

OpenMP:

OpenMP is a multi-platform shared-memory for parallel programming in Fortran, C, and C++ created in 1997. It was designed to be standardized, easy to use, and portable; it's likely due to these features that it became so popular. It is supported on many platforms, including Windows, Mac, and Linux. It is managed and run by a non-profit: The OpenMP Architecture Review Board. This board consists of some of the leading members in the field: Intel, IBM, Texas Instruments, AMD, Nvidia, and Oracle are just a few of its members.

Methodology:

Rewriting programs to work with CUDA architecture:

The project required downloading the CUDA SDK and Nvidia Nsight Compute programs. We had 3 parallel programs each in C and CUDA. We used openMP for the CPU parallelism and the global kernel operations for the GPU parallelism. The coding process started out by writing the C program - usually unparalleled - and debugging programs for compiler errors. Each program had a main function and another function (i.e. vectorAdd or knapsack) usually named after the program. This allowed easy rewriting from the C code to CUDA code. The CUDA code has two sets of variables. First variables with data are initialized in the CPU then the data is then referred to pointer variables which are passed as arguments in the program function. The CUDA program also required memory management, meaning allocating GPU memory for the variables, copying data from the CPU variables to the GPU variables, copying the result back from the GPU to the CPU, and deleting the allocation before the program ends. After the CUDA function passed in the correct data and received the correct output, we began parallelizing the programs.

On the 3 programs we had a data set of 10000. The C program was easier to parallelize, mostly due to the fact you could set the number of threads to 10000. On CUDA, the max threads for 1 block was about 1024. Parallelizing in CUDA for the programs meant subdividing the data set you have in an X Y grid. We did this by setting each block to have a max threads of 1024 or 32x32 threads. The general rule for GPU parallelism is to define the X Y grid with a dim3 data type. This grid will determine how many blocks/cores the GPU will allocate. So, in general each program was a 100x100 grid and each block having 32x32 threads, meaning in total having 10,240,000 threads allocated. However, not all the threads will be acting on the data, this led to a parallelism problem that the O levels did not fix, or even made worse. The program function in both C and CUDA remained relatively the same. In future for GPU programming in parallel, limiting the thread count will improve performance. A lot of iterations in the program would move in a diagonal way, meaning Block 0: thread(0,0) would access data, then Block 0: thread(1,1) would access data then Block 0: thread(2,2) would access data. Leaving threads (0,1), (1,2), (2,1), (0,2) allocated but unused. Another way for better performance is with memory allocation. In C programs, you want the program to use L1, then L2; for GPU, you want the program to skip L1 and go straight to access L2. L1 on GPUs are reserved mainly for graphics/TEX.

Gathering Performance Data:

After rewriting the program, the programs used to collect metrics of execution time, Cache, cycles, and instruction count were Nsight compute and perf stat. Perf stat was used multiple times (usually 10 times) to get an average on all metrics on regular, O1, O2, and O3 levels of optimizations. The other metrics were calculated based on the raw data. For example, CPI was calculated by the cycles/instructions. The main metrics are from perf stat. To get relevant data to compare C files and the CUDA files, Nsight

was used. Nsight gave the total cache size and the hit rate for L1 and L2. The data took and averaged the hit rates; the cache-hits, cache-misses, and miss rate were calculated using that data.

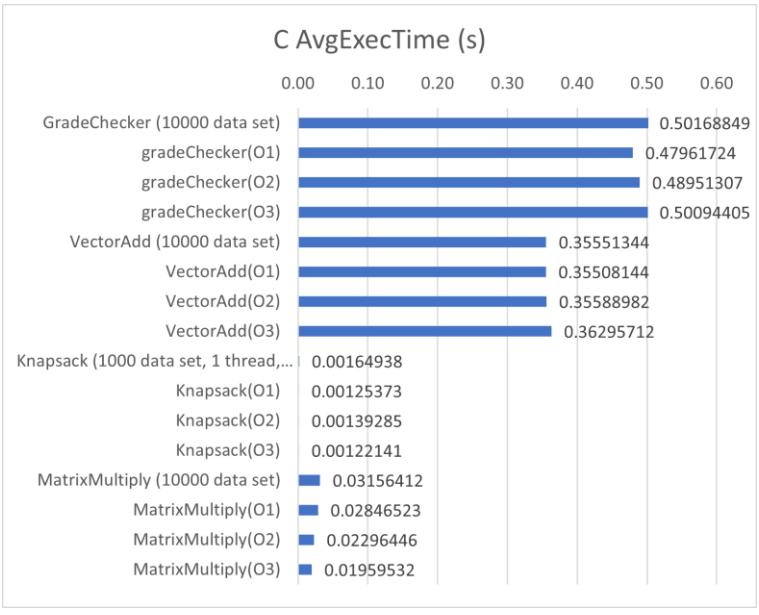
Analyzing and Graphing Performance Data:

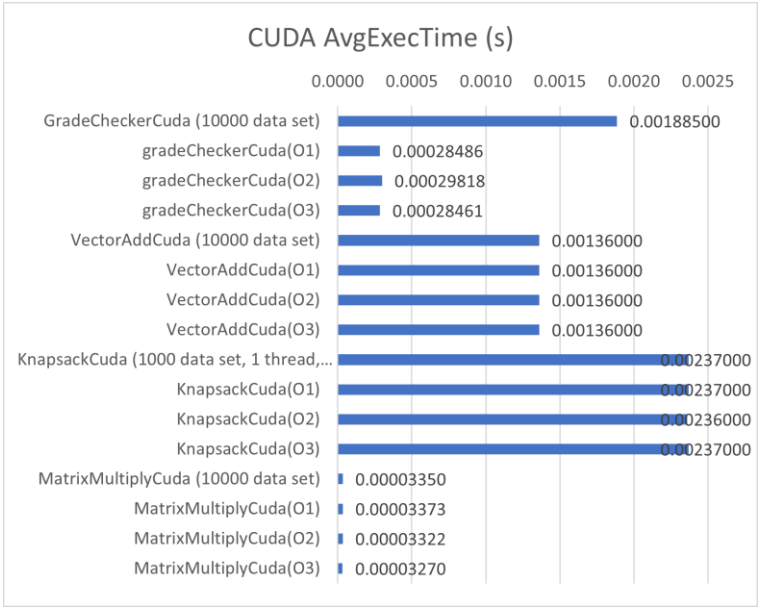
The different metrics from each program and their different optimization levels were recorded to an Excel spreadsheet to better organize the results. After which, the charts were created to better visualize the data. Determining the types and number of charts was decided based around the amount of content the chart would contain. The Excel spreadsheet contained many categories and entries for each metric, so it was decided to first split the charts between the C programs and the CUDA programs. The different metrics represented wildly different or unrelated data to one another, so the charts that were created were based around each individual metric.

With all the data organized visually into charts, analyzing the data at face value became much easier. While the CUDA versions of the programs could see vast increases in performance and specific metrics against the C programs. Seeing the different programs represented on charts and in raw data, a comprehension became clear that, for programs or metrics which involved memory usage, the CUDA versions would fall behind the C versions, the big tell for this being the Knapsack program.

Results:

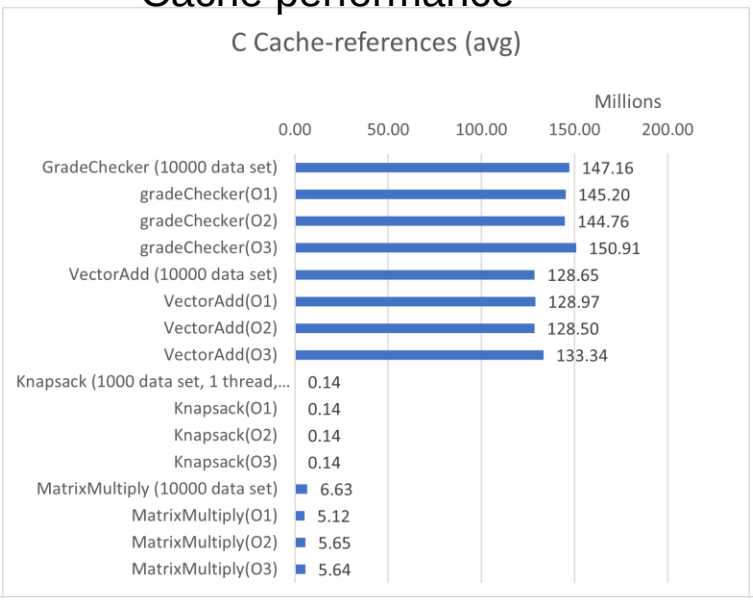
Average Execution Times

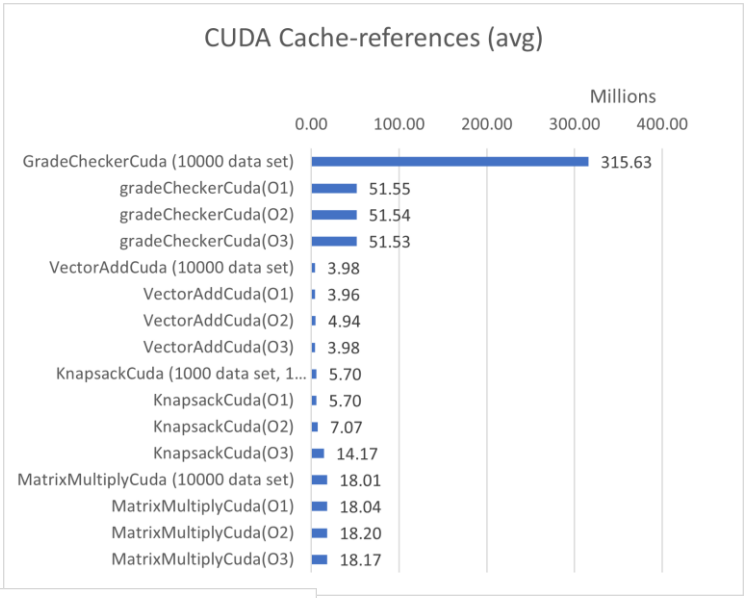


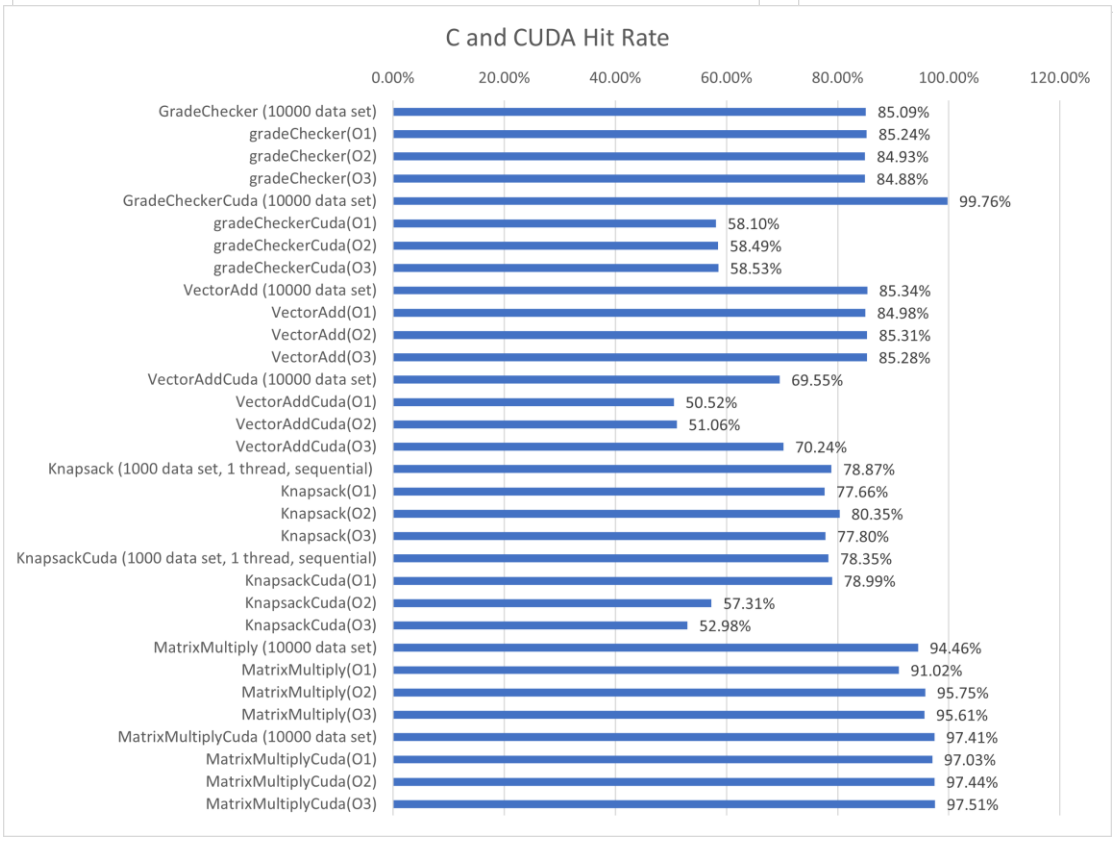


In the comparison of C and CUDA, the CUDA programs have an average execution time 100x to 1000x faster than the CPU programs.

Cache performance

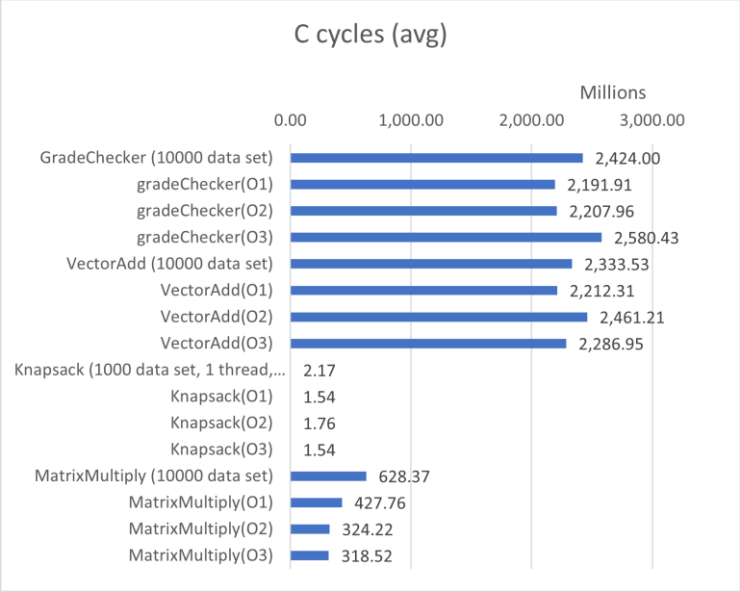


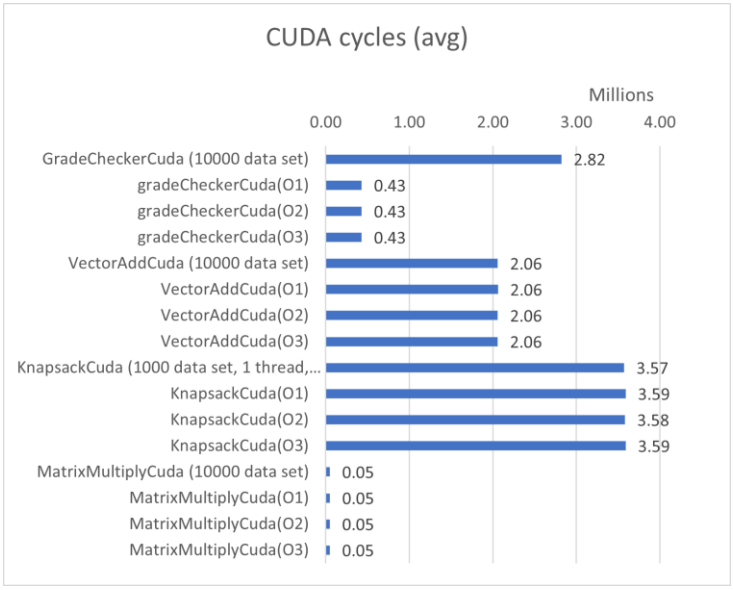


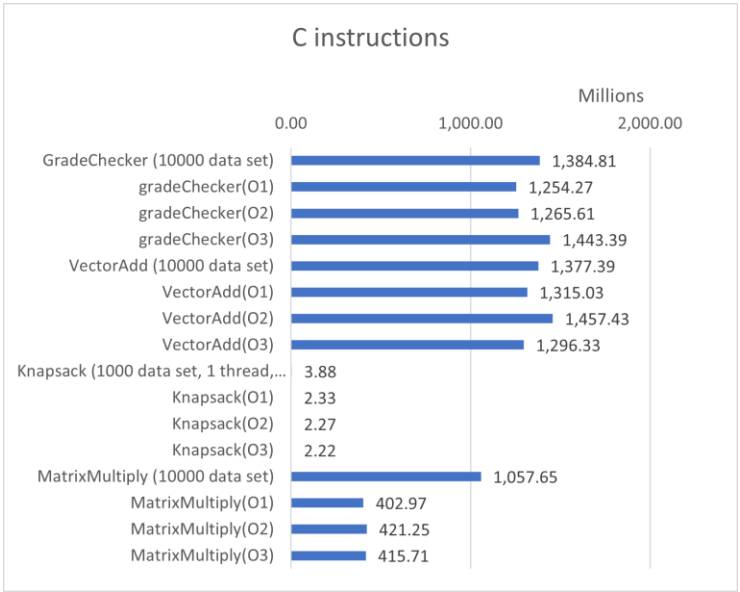


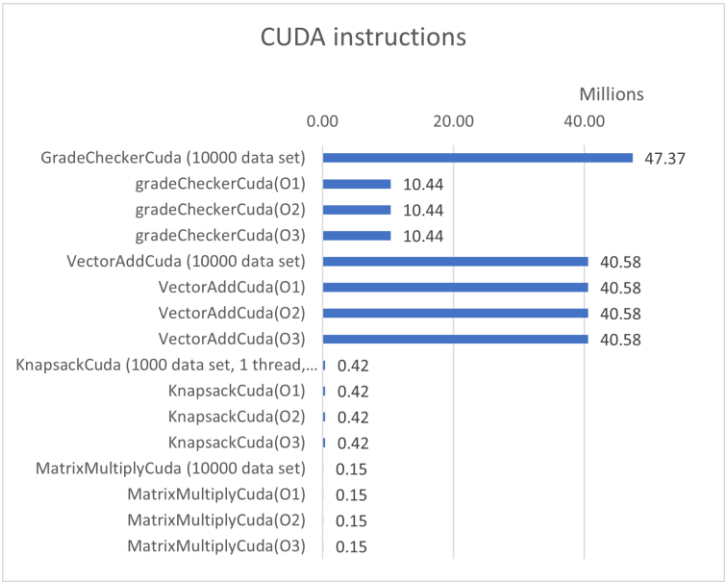
On the cache performance, the GPU does have a higher or the same cache-miss rates, and according to specs, has a smaller cache than the CPU. The GPU makes about half the cache accesses than the CPU. The Hit/Miss rates is a combination of the GPU having a smaller cache and poor parallel programming which is talked in the conclusion.

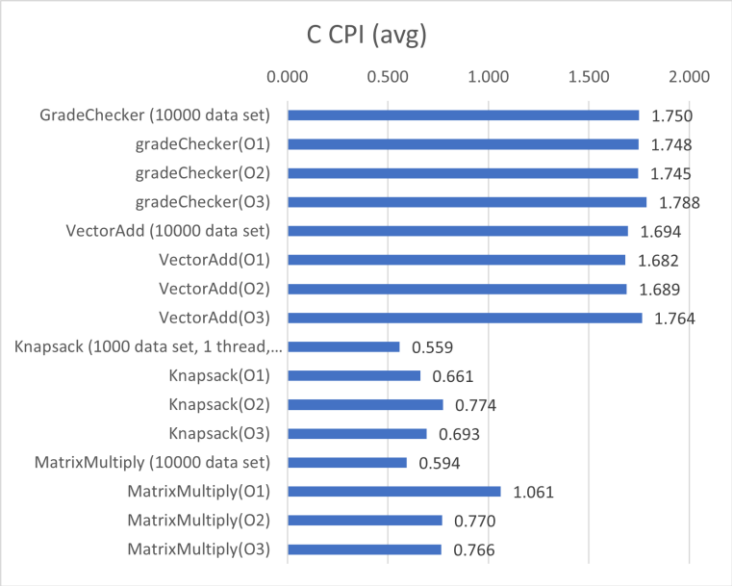
CPI

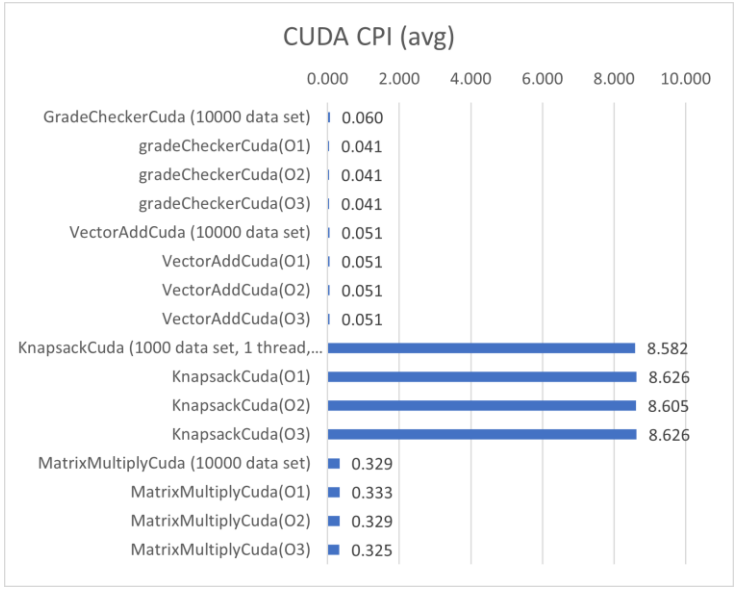

C cycles (avg)
Millions
0.00
1,000.00
2,000.00
3,000.00
GradeChecker (10000 data set) 2,424.00
gradeChecker(O1) 2,191.91
gradeChecker(O2) 2,207.96
gradeChecker(O3) 2,580.43
VectorAdd (10000 data set) 2,333.53
VectorAdd(O1) 2,212.31
VectorAdd(O2) 2,461.21
VectorAdd(O3) 2,286.95
Knapsack (1000 data set, 1 thread,... 2.17
Knapsack(O1) 1.54
Knapsack(O2) 1.76
Knapsack(O3) 1.54
MatrixMultiply (10000 data set) 628.37
MatrixMultiply(O1) 427.76
MatrixMultiply(O2) 324.22
MatrixMultiply(O3) 318.52


CUDA cycles (avg)
Millions
0.00
1.00
2.00
3.00
4.00
GradeCheckerCuda (10000 data set) 2.82
gradeCheckerCuda(O1) 0.43
gradeCheckerCuda(O2) 0.43
gradeCheckerCuda(O3) 0.43
VectorAddCuda (10000 data set) 2.06
VectorAddCuda(O1) 2.06
VectorAddCuda(O2) 2.06
VectorAddCuda(O3) 2.06
KnapsackCuda (1000 data set, 1 thread,... 3.57
KnapsackCuda(O1) 3.59
KnapsackCuda(O2) 3.58
KnapsackCuda(O3) 3.59
MatrixMultiplyCuda (10000 data set) 0.05
MatrixMultiplyCuda(O1) 0.05
MatrixMultiplyCuda(O2) 0.05
MatrixMultiplyCuda(O3) 0.05


C instructions
Millions
0.00
1,000.00
2,000.00
GradeChecker (10000 data set) 1,384.81
gradeChecker(O1) 1,254.27
gradeChecker(O2) 1,265.61
gradeChecker(O3) 1,443.39
VectorAdd (10000 data set) 1,377.39
VectorAdd(O1) 1,315.03
VectorAdd(O2) 1,457.43
VectorAdd(O3) 1,296.33
Knapsack (1000 data set, 1 thread,... 3.88
Knapsack(O1) 2.33
Knapsack(O2) 2.27
Knapsack(O3) 2.22
MatrixMultiply (10000 data set) 1,057.65
MatrixMultiply(O1) 402.97
MatrixMultiply(O2) 421.25
MatrixMultiply(O3) 415.71


CUDA instructions
Millions
0.00
20.00
40.00
GradeCheckerCuda (10000 data set) 47.37
gradeCheckerCuda(O1) 10.44
gradeCheckerCuda(O2) 10.44
gradeCheckerCuda(O3) 10.44
VectorAddCuda (10000 data set) 40.58
VectorAddCuda(O1) 40.58
VectorAddCuda(O2) 40.58
VectorAddCuda(O3) 40.58
KnapsackCuda (1000 data set, 1 thread,... 0.42
KnapsackCuda(O1) 0.42
KnapsackCuda(O2) 0.42
KnapsackCuda(O3) 0.42
MatrixMultiplyCuda (10000 data set) 0.15
MatrixMultiplyCuda(O1) 0.15
MatrixMultiplyCuda(O2) 0.15
MatrixMultiplyCuda(O3) 0.15



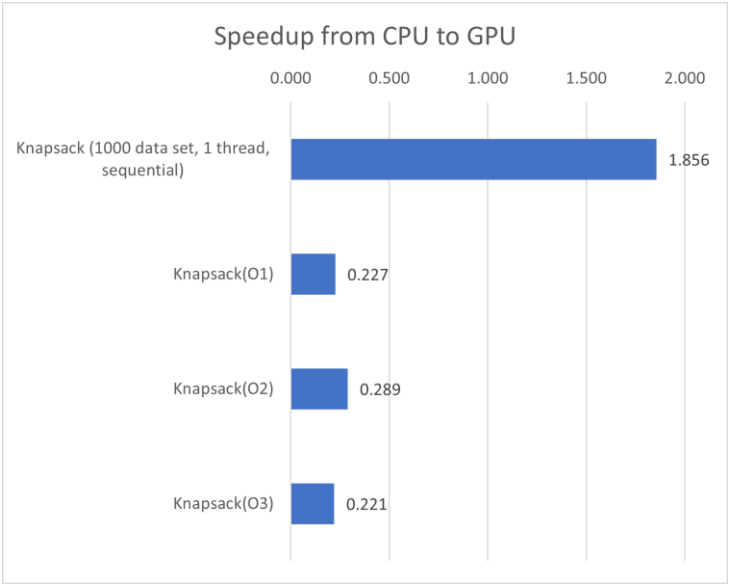


The nvcc compiler has fewer instructions and a lower CPI, meaning that it takes more clock 'ticks' for a CPU program to execute than it does for the GPU. A smaller CPI means the GPU is performing better than the CPU executing similar programs.

Knapsack anomaly

Knapsack, both C and CUDA programs were not parallel. Knapsack is a purely sequential program that our team used to compare the performance of a single thread. The CUDA program took almost twice as long to execute than the C code. On Cache performance, CUDA had a higher miss rate but lower cache accesses. KnapsackCuda still had lower instructions, however, the sequential program took almost twice as many cycles to execute those instructions even though the C code had 10 times more instructions. This result led us to conclude that while the GPU is outperforming in CUDA parallelism, having a program execute on a single thread, either on the CPU or GPU, will have better performance on the CPU rather than the GPU. This is explained in our conclusion and interpretation on why that is the case.

.



Conclusion:

Across our main 10 metrics, on average, CUDA performs 57% better than C. CUDA falls on cache-hits, cache-misses, hit rate, and miss rate. It does better on cycles, clock rate, execution time, CPI, and instructions. Other times CUDA performs at or worse than C code.

In comparing CPU vs GPU, the GPU performs better on parallel programs. The trade-offs, for a program correctly parallelized, are cache hit-rate and cache-miss rates. There will be lower hit rates and higher miss rates, however, there will be less cache-references, lower clock-rates (which should affect speed on single thread but being parallel means lower clock rates and lower cost per performance), and faster Execution time. Furthermore, if the CUDA programs were coded with more optimization for GPU parallelism, that 57% should be higher.

Knapsack is an anomaly, it shouldn't be compared to the other CUDA programs because, in both the C code and CUDA code, it compares the performance of a single thread in a sequential operation. Which is an important observation, because it led us to realize (insert observation here).

The clock rate, on the CPU, is applied to every single thread, meaning each thread runs at 3.4GHz to 3.9 GHz. While the GPU only applies to every core, and the GPU has a slower clock rate than the CPU. So, in knapsack, one CPU thread will have a rate of 3.5 GHz, while the GPU thread is at 1.7 GHz. The GPU was able to perform better in parallelism because of the amount of cores. So, each CUDA core runs slower than the CPU core, however there are 1500 cores. For Example, in the matrixMulti program, there are 16 CUDA cores allocated. This means, if you divide the program out, those 16 cores will execute the program faster than the 12 threads on the CPU. This is also because of the CPI. A comparison would be like a rally race; The CPU can run faster than the GPU, but each thread has to run 150 km. In the GPU, each core runs slower, each thread runs even slower, but the instruction count means there is less to run. Each core will only have to run 0.5 km and each thread (Ex.32 x 32 threads per block) will only have to run 0.5 m. By pure quantity, the GPU will finish faster than the CPU, even though a CPU thread can run faster than a GPU thread.

This explains why certain programs will run faster on the GPU in parallel. Video games use parallelism. If you try running something like doom on a CPU, and compare it running on a GPU, it will run faster because of that parallelism. This also explains why cryptocurrency mining is used in multi-GPU racks rather than the regular CPU server racks.

Let’s take an example of matrixMulti or similar program. The CPU executes it in 0.03 seconds, the GPU program executes it in 0.00003 seconds, meaning it does 1000x faster. The CPU, in a 24hr time stand, will compute 287,971,200 times. The GPU, in the same time stand, will compute 2.8E14 times. For Austin Energy, cost per kWh is

$0.028. The GPU will have 2.88 kWh costing $0.086 per day while the CPU will have 1.56 kWh costing $0.043 per day. But because the GPU can execute the program faster, the cost per performance for the CPU is $1.49E-10 per executed program, the GPU is $3.07E-16 per executed program. The GPU cost 485,342 times less to execute than the CPU.

In conclusion, our hypothesis was correct but only correct if the program is running in parallel. We were not correct to assume that every situation and every metric that the GPU will outperform the CPU. The CPU had better cache performance, and better thread performance.

Resources:

Advanced Micro Devices, Inc. (n.d.). *AMD Ryzen™ 5 2600 Processor*. AMD. https://www.amd.com/en/products/cpu/amd-ryzen-5-2600.

Cho, P. S. (n.d.). *Cuda Thread Basics*. *CSC 391/691: GPU Programming*. Wake Forest University, Winston-Salem , North Carolina , United States; Wake Forest University, Winston-Salem , North Carolina , United States. http://users.wfu.edu/choss/CUDA/docs/Lecture%205.pdf.

Cooper, K. (n.d.). *Using Cuda to Unwrap Loops*. *Department of Mathematics*. Washington State University; Washington State University. http://www.math.wsu.edu/math/kcooper/CUDA/12CUDAunwrap.pdf.

CUDA C++ Programming Guide. NVIDIA Developer Documentation. (n.d.). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

CUDA Installation Guide for Microsoft Windows. NVIDIA Developer Documentation. (n.d.). https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/.

The GeForce 16 Series Graphics Cards are Here. NVIDIA. (n.d.). https://www.nvidia.com/en-us/geforce/graphics-cards/gtx-1660-ti/.

GPU Accelerated Computing with C and C++. NVIDIA Developer. (2021, April 14). https://developer.nvidia.com/how-to-cuda-c-cpp.

Harris, M. (2020, November 12). *An Even Easier Introduction to CUDA*. NVIDIA Developer Blog. https://developer.nvidia.com/blog/even-easier-introduction-cuda/.

Harris, M. (2020, October 14). *An Easy Introduction to CUDA C and C++*. NVIDIA Developer Blog. https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/.

Johnson, J., Guerra, N., & DiPalma, G. (2021, March 31). *Performance Evaluation: CUDA GPU vs. OpenMP CPU*. https://github.com/jtj60/Computer-Architecture-Project./

NVIDIA CUDA Installation Guide for Linux. NVIDIA Developer Documentation. (n.d.). https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html.

Profiler User's Guide. NVIDIA Developer Documentation. (n.d.). https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

Qasem, A. (n.d.). *Evaluating Performance*. *CS 3339*. San Marcos,Texas; Texas State University.

Qasem, A. (n.d.). *Memory Hierarchy : Caches*. *CS 3339*. San Marcos, Texas; Texas State University.

Sanders, J., Kandrot, E., & Dongarra, J. J. (2015). *Cuda by example: an introduction to general-purpose Gpu programming*. Addison-Wesley/Pearson Education.

Venkataraman, V. (n.d.). *Cuda Debugging With Command Line Tools*. *GPU Technology Conference*. https://www.nvidia.com/en-us/gtc/.

Zeller, C. (n.d.). *Cuda C/C++ Basics*. *Supercomputing 2011 Tutorial*. https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf.

OpenMP. OpenMPorg RSS. (n.d.). https://web.archive.org/web/20130809153922/http://openmp.org/wp/about-openmp/.