Matrix Multiplication
As An Introduction To GPGPU
In The CUDA/C++ Programming Environment

Jeffrey Tamashiro
ICS 692 •Casanova •Fall 2015

***Abstract*—This project is meant to serve as an introduction to GPGPU programming using the CUDA with C++ programming language. This is achieved through creating a simple matrix multiplication application, experimenting with settings, scale and comparing it to similar CPU and GPU library based versions.**

I. Introduction

Graphical Processing Units (GPUs) have been traditionally used by PC gamers, video editors, and CAD designers. More recently they have also found a secondary purpose running of calculation-intensive, highly parallelizable code. General Purpose Graphical Processing Units (GPGPU) are now being touted as a comparable alternative to CPU computing in terms of performance and financial cost.

A GPU typically fits into a motherboard's PCI slot giving it a fast interface with the CPU. When compared to a CPU it has far less features and performs individual calculations slowly. However, a GPU shines in the sheer number of computing cores it can harness, roughly 100 times that of a CPU. Multiple GPUs can even be installed in a single system, doubling or tripling that number.

However, programming for the GPU brings its own set of unique challenges due to its extremely specialized architecture. A GPU is originally designed for rendering images on a screen so repurposing it to do something like crunching numbers is an inherently difficult task. Its intricate organization of thread hierarchy, complex variants of limited memory, and constrained CPU interface requires deep understanding and proprietary knowledge to implement code efficiently. Fortunately, GPU makers are supporting this new frontier with language support and toolkits.

The purpose of this project is to take the first steps into developing code specifically for the GPU. The goal is to create a basic matrix multiplication application, do some experimenting with it, and compare it to alternate implementations. Ultimately, these findings will hopefully reinforce concepts of concurrent programming and distributed computing.

II. Background

A. Hardware and Software

Perhaps the most confusing barrier to entry when beginning GPU programming is setting up the particulars of hardware and software. This must be done before any code can be compiled and run. For this project, the choice of hardware was a decently powered home PC and a 2 year old consumer level NVDIA GPU purchased for under $300.

The software used is NVDIA's proprietary CUDA programming language which has ample documentation and community support. It is specific to NVDIA's GPU architecture, but hopefully the learning experience and knowledge gained with this project will overlap with enough general GPU concepts to be beneficial.

Included is a file `my_gpu_profile.txt` produced by a helpful program included in the CUDA packages. It outlines the specifics of an installed GPU. It also served as a useful outline for the different GPU components and what metrics are associated with them.

B. GPU Architecture

At first glance, the GPU architecture is an unfriendly list of terms and values. Luckily, there is an intermediary layer of abstraction that connects the hardware circuits and the CUDA code. The basic idea describes the GPU launching a job as a grid. The grid is made up of blocks, and within each block there is a number of threads. Threads are what will run concurrent sections of code.

This grid-block-thread hierarchy may seem strange but it is tied to the GPU architecture. One core example is a limitation on the number of threads per block, in this case it is 1024. The number of blocks on the other hand can be much higher, like over 2 billion. Ideally, a user can run a large number of total threads with a set number of threads per block and then determine how many blocks are needed. This would be useful if two GPUs had different maximum threads per block. Writing code for the lower GPU's limit could still be used with the higher GPU with little sacrifice of performance.

The final, most complex piece of the architecture is GPU memory. The GPU actually has many different types of memory that all work differently. The most basic and easy to understand is global memory. Global memory is typically the most abundant and can be read and written to by all threads. This GPU has 2GB, and a $1000+ card comes with 12GB. Although not implemented for this project, the other memory types actually are one of the most interesting aspects of the

GPU, so there will be a small section discussing them later in this report.

C. CUDA Coding

Running CUDA code requires the installation of many CUDA packages and libraries, most notably a special compiler wrapper for gcc called nvcc. This is similar to mpicc.

The basic structure for a CUDA program is outlined here:

- `cudaMalloc(...)` Allocate space in the GPU's memory
- `cudaMemcpy(...)` Copy memory into GPU's memory
- Launch a kernel which is code run entirely in the GPU
- `cudaMalloc(...)` Copy memory back
- `cudaFree(...)` Clean up GPU memory

Another important aspect of CUDA programming is the dim3 data type which takes up three values which can be used as x,y,z vectors. These are used frequently in CUDA programming. A key instance is when invoking a kernel, they are passed to determine its launch configuration.

```
//setup launch configuration
  dim3 threads(32,32);
  dim3 blocks(3,3);

//kernel call
  my_gpu_code<<<blocks,threads>>>();
```

This snippet of code shows the dim3 data types defined as *threads* and *blocks*. Their arguments define their x and y vector quantities. Although no argument is passed for z, it is implicitly defined as 1.

threads and *blocks* are then passed to the kernel within the triple angle brackets along with any parameters like a typical function. It is important to note that the sizes are products of the three dimensions and must fall within the GPUs specifications. In this case 3 x 3 x 1 = 9 blocks with 32 x 32 x 1 = 1024 threads per block. That's a total of 9126 threads, meaning the function *my_gpu_code* would run 9126 concurrent instances.

Similar to MPI ranks and communicators, when inside each of theses threads there are several function calls to identify threads and distribute work among them. Here are a few:

- `blockIdx` index of the current block
- `threadIdx` index of the current thread
- `blockDim` dimensions for current block

Theses are all dim3 data types so they must be followed by .x .y or .z to get the relative value.

III. Testing

The setup was an Intel 8-core i7-4770K 3.5GHz CPU with 24GB RAM and an NVDIA GeForce GTX 660 running Ubunutu 14.04 for the OS. The GPU's video output was disabled from the BIOS with the display running off the motherboard's video. This would ensure the GPU wouldn't be sharing duties with anything else.

`matmul_gpu.cu` is the central piece of code for testing. It multiplys two 2D matrices of the same size filled with 4-byte floating point numbers and storing the result in a third matrix. Each cell of the product matrix is worked on by one thread reading and computing from the other matrices. 4-byte floats were used instead of integers to give each computation a little more time and give the overall computation a more sizable chunk of time.

There was one problem with a CUDA kernel time out when matrix sizes went above 6400x6400. The expectation was that the matrix sizes would max out as the GPU's RAM was filled but this was a different problem that had to do with a kernel operation taking too long and prompting a watchdog process to stop the kernel completely. There may have been a difficult work around for this but instead it was decided to just use 6400x6400 as the maximum matrix size for all tests. For comparison, two alternate matrix multiply implementations were created.

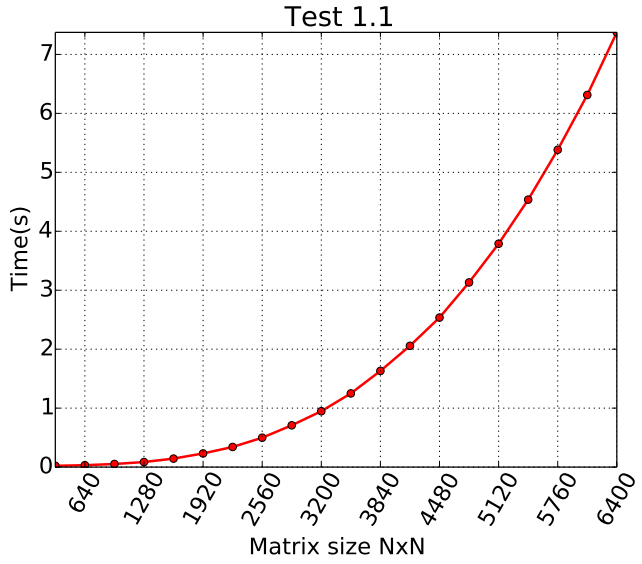
`matmul_cpu.cu` used the CPU, which used i-k-j loop updating with 8 OpenMP threads spawned over the outermost loop. It's important to note the CPU is running this process chain along side a running operating system which should be taken account when doing the comparison analysis.

`matmul_lib.cu` was created using a library for CUDA called Cublas. This is the CUDA version of the BLAS library which stands for Basic Linear Algebra Subroutines. It has a matrix multiply function which could be inserted into the code in place of the kernel launch with very minimal effort. Interestingly, the Cublas program did not have the same kernel time out problem with matrix sizes above 6400x6400 like the hand coded version but did max out on memory as expected.

IV. Results

A. Test 1.1

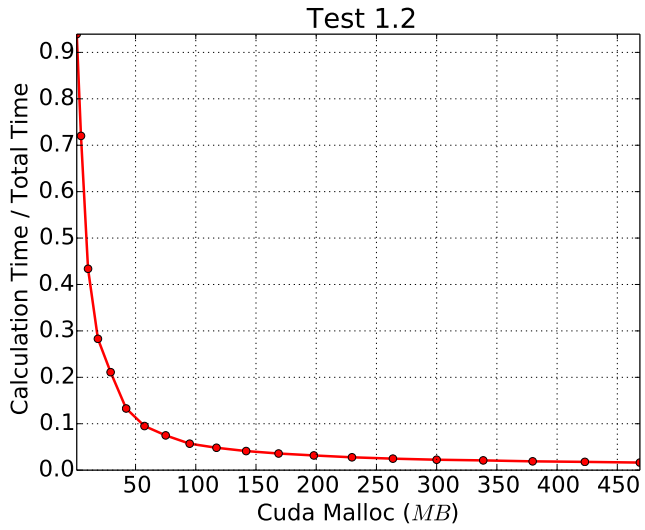
Test 1.1 was a straight running of the `matmul_gpu.cu` application with increasing matrix size up to 6400x6400 and 32x32 threads per block (1024) with number of blocks increasing as needed. The graph shows typical exponential growth as the matrix size grows exponentially. This will serve as comparison for alternate tests.



B. Test 1.2

Along with timing the entire matrix multiplication process, and second timer was also set around just the kernel call to the function actually doing the matrix multiply. This provided a basis for comparison of the time spent in calculation vs the total time of the process, which basically is calculation + memory copying.

This graph takes the ratio of calculation time over total time ranging from 0 to 1. It shows that when the matrix size is small, the copying process dominates the time and as the matrix size increases, the copying process becomes insignificant and the time spent in calculation becomes the dominant factor. This makes sense because although the number of bytes copied increases as the matrix size grows, the number of calculations increases by much more with each row and column added.

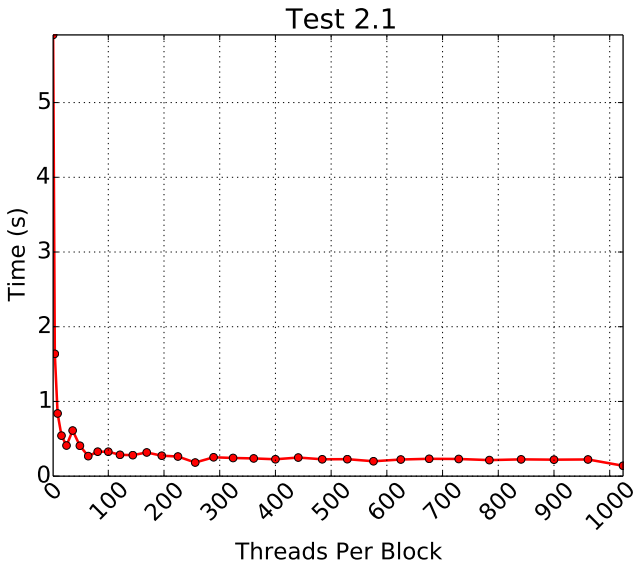


C. Test 2.1

This next set of tests kept the matrix size at a consistent 1600x1600 while varying the launch configuration settings for the GPU kernel. By starting with the total number of threads (each representing a matrix cell in this case) and the threads per block (which will be shifted in these tests) the number of blocks is determined by dividing these two figures. So the number of blocks will vary as the threads per blocks varies.

Test 2.1 only used one dimension of the dim3 data type. The graph ranges from 1 thread to 1024 threads per block which is the maximum. The number of blocks shrinks inversely starting at 1024 blocks and ending with one.

Early on the run time is slowed down considerably with threads in double digits. However, around 100 threads and beyond the time seemed to flatten out to around .2 seconds. This is a little strange but is probably due to the underlying architecture being more complicated than blocks and threads. An interesting follow up would be to repeat varying the matrix size and see if the flattening occurs at the same number of threads.



D. Test 2.2

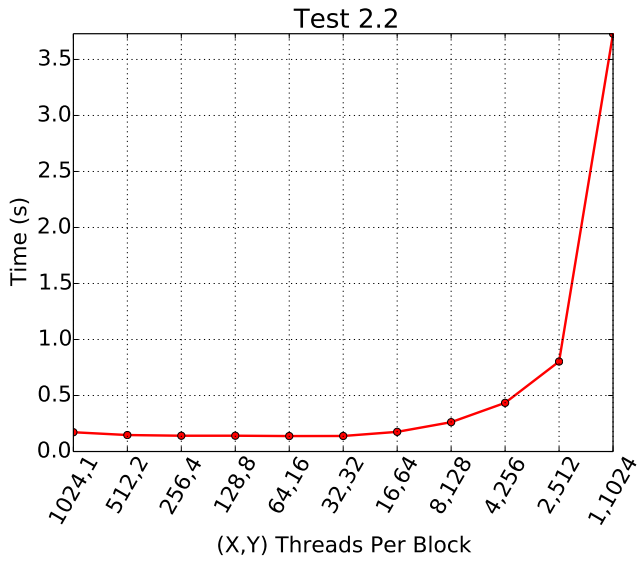
Test 2.2 was carried out similarly but using multiple dimensions for the launch configuration and determining if there was an effect on speed. The x and y coordinates were varied such that their product still equaled 1024.

$(dim3.x, dim3.y) :: [(1024, 1)...(32, 32)...(1, 1024)]$

For the first part of the graph there is not much difference. It's difficult to see but (32,32) does have the minimum time. The interesting part is on the right side where the graph gets closer to (1,1024) and the slow down is detrimental.

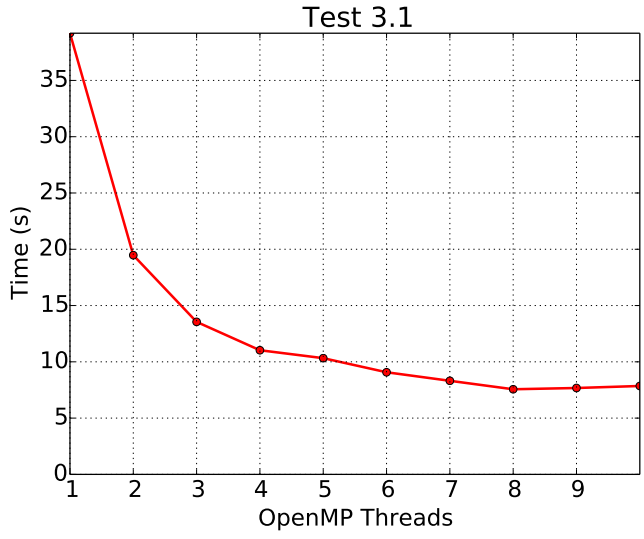
Apparently, the organization of threads by the dim3 vector is not just for the convenience of the programmer for

working with multiple dimension data structures. It looks like having a small x and large y is causing slowdown. In any event, when using obtuse dimensions it seems like larger ones should go first.



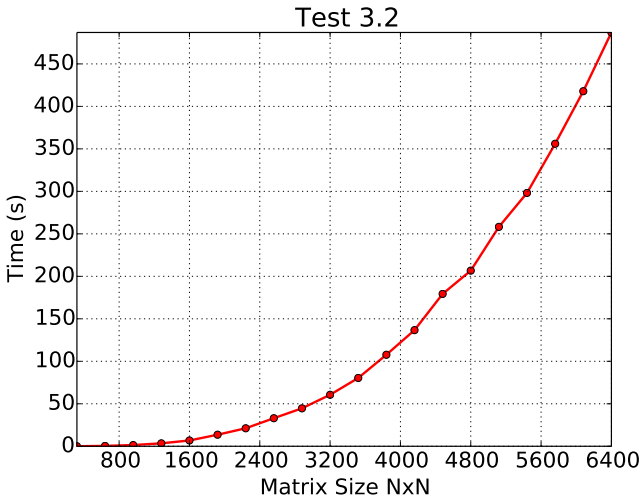
E. Test 3.1

This third section of tests was for the comparison using only the cpu `matmul_cpu.cu`. Test 3.1 was kind of a dummy test just intended to make sure there was a speedup along with the usage of increasing OpenMP threads and get an idea of what the optimal number of threads to use for the next test. It was done with matrix size of 1600x1600 and threads increasing from 1 to 10. The results showed that using 8 threads had the best times which made sense because the cpu had 8 cores.



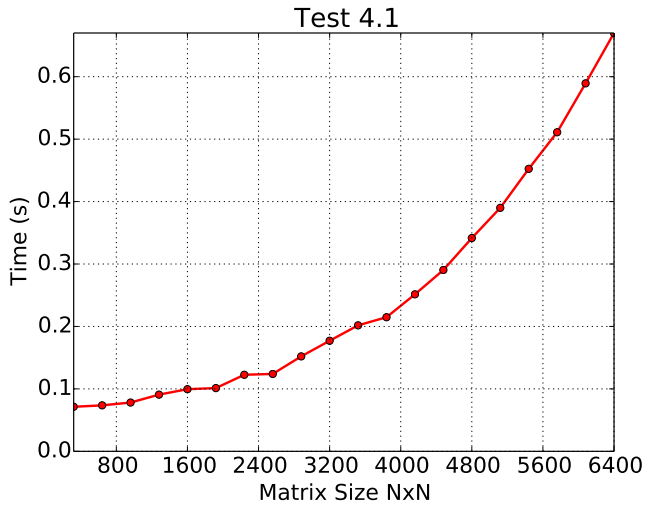
F. Test 3.2

Test 3.2 is meant to be a comparison for Test 1.1 GPU case with the same matrix size scale up to 6400x6400 and 8 OpenMP threads. As the results show the shape of the curve is similar indicating exponential time increase, however the CPU times are around the order of fifty times slower with the largest matrix taking over 400 seconds compared to the GPUs 7 seconds. So although this cannot be considered a straight comparison because the CPU is work switching with all the other OS processes, the difference is significant enough to draw some conclusions about performance comparisons.



G. Test 4.1

The last test was carried out in the same parameters for Tests 1.1 and 3.1 matrix size scale up to 6400x6400 except using the Cublas library for Matrix Multiplication which was a function called `cublasSgemm()`. It's important to note that using would have required barely any knowledge beyond standard C++. There was no special kernel launching or copying memory to the GPU. The results turned out to be about ten times faster than the hand-coded GPU version. This is pretty much expected but frustrating nonetheless.



V. Conclusion

The tests mostly gave results as expected but there are a few interesting points worth mentioning.

Firstly, as is the case with most things. The library version of matrix multiply was better than a naively coded version. It was not only faster but did not run into the same system level difficulties. These are clearly more optimized and have more going on under the hood. When doing a utility function like matrix multiplication it is worth the time and effort it takes to learn the library instead of trying to do it yourself.

Secondly, the memory copy process is not very intensive as number of calculations grow. Even when working with multiple kernel launches and allocating more of the GPUs memory, as long as it's done in a smart, minimal way, the memory copy process does not need to be considered as much when thinking about processing times.

Finally, using the global memory made visualization and implementation of the program very simple. The most difficult part of ICS632's course work this semester was the mapping of distributed memory structures and working with communicators to connect them. The GPU allows you to work with thousands of cores which can all access the same memory. Locking and synchronization were very easy to use. Although it was relatively easy to use doing this way may not be the best use of all the GPU has to offer as will be discussed in the next section.

VI. Further Study

The next big step in utilizing the GPU would be the use of Shared memory. Shared memory is a portion of memory that can be shared among all threads in a block. It is special because it is actually is physically the same as the L2 cache. This means it can read and write quickly, much more so than the global memory.

This creates an environment that is much more typical HPC over several nodes with distributed memory. It also is much more complicated because the shared memory is sectioned into banks which if accessed in a certain way can conflict and actually run slower than intended. Getting optimal performance would require some level of intimate hardware knowledge and experimentation in data structures and how they are accessed concurrently.

According to my GPU profile, I have 48K of shared memory per block which is pretty small when compared to the 2GB of global memory. This would limit the number and size of data structures but would be really good for small data that would need to be computed extensively and several times over.

There are actually several more different types of memory that are intended to work for the intricacies of graphical processing but can be manipulated by CUDA.

Although there is a steep learning curve that goes along with the NVDIA architecture and CUDA language as well as GPGPU programming in general, there also seems to be a large potential for creating powerful parallel computing machines that can act as a small scale or more economical version of larger super computer counterparts.

References

[1] N. Wilt, *CUDA Handbook: A Comprehensive Guide to GPU Programming, The*. Upper Saddle River, NJ: Addison-Wesley, 2013.

[2] M. Halls-Moore. (2014) Installing nvidia cuda on ubuntu 14.04 for linux gpu computing. [Online]. Available: https://www.quantstart.com/articles/Installing-Nvidia-cuda-on-Ubuntu-14-04-for-Linux-GPU-Computing

[3] Sol. (2012) Matrix multiplication on gpu using cuda with cublas, curand and thrust. [Online]. Available: https://solarianprogrammer.com/2012/05/31/matrix-multiplication-cuda-cublas-curand-thrust/

[4] WhatsACreel (youtube username). (2012) Cuda tutorials: Tutorial series on one of my favorite topics, programming nvidia gpu's with cuda. [Online]. Available: https://www.youtube.com/playlist?list=PLKK11Ligqititws0ZOoGk3SW-TZCar4dK

[5] A. Gray. (2013) Learn cuda in an afternoon. [Online]. Available: https://youtu.be/_41LCMFpsFs

[6] C. Zeller. (2011) Cuda c/c++ basics: Supercomputing 2011 tutorial. [Online]. Available: http://www.nvidia.com/docs/io/116711/sc11-cuda-c-basics.pdf

[7] Nvdia Corporation. (2015) Cuda toolkit documentation: cublas level-3 function reference. [Online]. Available: http://docs.nvidia.com/cuda/cublas/

[8] V. Natoli. (2011) Top 10 objections to gpu computing reconsidered. [Online]. Available: http://www.hpcwire.com/2011/06/09/top_10_objections_to_gpu_computing_reconsidered/