CUDA Project Report

Overview

We were assigned to write an implementation of the triangular matrix solve operation on a GPU using CUDA, which solves a system of matrices such as the following, where *A*, and *B* are the given matrices, *X* is unknown, and α is a scalar:

Equation 1: System of Matrices

$$A \cdot X = \alpha B$$

We assume *A* to be a lower-triangular unit matrix. We also implemented a sequential version of our algorithm and used the `cublasStrsm()` function to solve the same problem, and we compared the results.

Methodology

If one were solving this system by hand, the *A* matrix would need to be inverted:

Equation 2

$$X = A^{-1} \cdot \alpha B$$

However our algorithm performed the inversion and multiplication in one step. To illustrate, consider solving for the first three elements in the first column of *B*. Given our assumptions that *A* is unit triangular, the equations are fairly simple:

Equation 3: Multiplication of a Unit Triangular Matrix

$$\begin{bmatrix} x_{00} \\ a_{10} \cdot x_{00} + x_{10} \\ a_{20} \cdot x_{00} + a_{21} x_{10} + x_{20} \end{bmatrix} = \begin{bmatrix} b_{00} \\ b_{10} \\ b_{20} \end{bmatrix}$$

Now if we solve the system above for *X*, specifically the highest-column element of *X* in each individual equation, we have the following system:

Equation 4: Multiplication of a Unit Triangular Matrix Solved for X

$$\begin{bmatrix} x_{00} \\ x_{10} \\ x_{20} \end{bmatrix} = \begin{bmatrix} b_{00} \\ b_{10} - a_{10} \cdot x_{00} \\ b_{20} - a_{20} \cdot x_{00} - a_{21} \cdot x_{10} \end{bmatrix}$$

Notice that any given element in *X* is only dependent on the corresponding element in *B*, the previous element in the current column of *X*, and the corresponding row of *A*. This allows us to solve the system without the step of inverting the *A* matrix.

Work Mapping Scheme

Using this algorithm also isolates the dependencies to columns in *X* (as shown in Equation 4), so in other words, each column can be calculated without any need for synchronization between them. So, naturally, we leveraged this fact by assigning each column a single thread. This made it possible to implement the kernel without synchronizing between threads except for cases where we used shared memory.

Optimizations and Best Practices

We tried several optimizations to improve performance, however not all of them made it into the final project. The CUDA best practices guide was an invaluable resource when determining the list of optimizations to explore. Here is a list of optimizations (in order of relative importance to our implementation) we either explored or implemented:

Coalescing Memory Accesses - The CUDA best practices guide states “perhaps the most important performance consideration in programming for the CUDA architecture is coalescing global memory accesses”. So, we structured our who GPU implementation around the fact that we want all memory accesses coalesced. There are conditions that need to be met in order for coalescing to occur. Basically, if a half-warp (16 threads) access memory in a 64 byte aligned and incremental fashion at the same time, the architecture will combine these loads/stores into one big transaction which is a lot faster than multiple single transactions. This is detailed in section 3.2.1 of the NVIDIA Best Practices document. In order to accomplish this, we went so far as to pad our A matrix with 0’s so that we could copy 16 floats in constant alignment and incrementally to shared memory. We structured our B/X matrices so this coalescing happened naturally. Another key piece of information is that the CUDA memory copy function ensures device memory alignment. We printed out pointer values to confirm this.

Caching Portions of A in Shared Memory - Since the calculation for each value of *X* reads values in the corresponding row of *A* numerous times (up to *n-1* times for an *n* x *n* matrix), we found we could reduce memory access time by copying the row into shared memory. Shared memory accesses are much cheaper than global memory accesses, and the memory is visible across all threads in a thread block. However, the amount of memory is limited. Copying an entire row of *A* to shared memory reduced our execution time by almost half, but it was limited to matrices of *n* <= 512, since 512 is the maximum number of threads for our GPU cards' compute capability. Since we needed something that could scale up to much larger matrices, we implemented a similar concept, but we only pull in a portion (16 floats at a time) of the row of *A*. 16 is an ideal number because it satisfies one requirement of coalesced accesses. You can have one transaction bring in all 16 values from global memory to shared memory.

Loop Unrolling - Although not mentioned in the CUDA best practices guide, we found loop unrolling to have a fairly large impact on performance. Basically, since we pull in a fixed-size block (`16*sizeof(float)`) to shared memory and we pad the A matrix with 0's to ensure `16*sizeof(float)` alignment, we are able to easily hand-unroll 16 inner loops without any extra branches needed. This gives us a performance boost since it avoids checking a loop termination condition each iteration. So loop unrolling also goes along with our goal to reduce branching and divergence which we talk about later.

Thread and Block Heuristics - We were surprised by how much performance changes as you modify the threads per block and blocks per grid settings. Basically, in software, you define how many threads you want per block. So, these threads have to share resources such as shared memory. If the threads do to much and you have a lot of threads defined per block, the GPU control logic will start serializing the warps because all the resources get used up. By running tests using different settings for threads per block and number of grids, we found our ideal threads per block is 128. It is important that this thread per block number is at a 32 thread (hardware warp) boundary.

Register Usage - The amount of instructions, branches, and, in general, code you have in a kernel effects the amount of registers you use. Since there are only a limited amount of registers per thread, you have to be careful with how much code and what type of code you put in the kernel. We had to find the right balance between unrolling loops and number of threads per block to make sure our register usage wasn't an issue.

Shared Memory and Bank Conflicts - We ensure that all accesses by threads to memory within a half-warp (16 threads) access different banks. A shared memory bank is defined as `Bank = Address % 16`. Since bank conflicts only occur within half-warp boundaries, you just have to make sure 16 threads that access shared memory at the same time access different banks. We manage this using the thread index.

Branching and Divergence - The CUDA best practices guide says, as a high priority, to "avoid different execution paths within the same warp". Given our need to use shared memory and due to some irregularity in looping, it's impossible not to have control flow instructions inside the kernel. However, using techniques like loop rolling and padding matrix A with 0's, we are able to minimize the amount of branches and therefore the amount of divergence. We notice as added more branches, the performance went down quickly even though we used the branch to read from faster shared memory at certain times. This example is explained more in the attempt to cache values of X optimization below.

Compacting the Triangular Matrix A - We only sent the non-zero and non-diagonal elements of the *A* matrix to the card since we could assume the value of *A* for those elements. However, as mentioned a lot above, we did pad rows of A with 0's to give us certain advantages we would have had to add branches in the kernel for. We felt this was a "no-brainer" tradeoff (extra time sending A vs. branch divergence) decision. Compacting the triangular matrix A as much as we could gave us a performance increase because it reduced our working set, which reduces the time needed to copy the matrix to the device and results in less movement of memory in and out of caches.

Using CUDA Memory Allocation Calls - We used `cudaMallocHost()` instead of `malloc()`. According to the documentation in cuda_runtime.h, `cudaMallocHost()` helps improve performance by ensuring that memory is aligned on blocks that are easily accessed by the GPU device. We saw modest performance benefits from these calls of under 5%.

Calculating the Results Matrix In-Place - We were able to reduce the memory footprint of our code by storing the results in the *B* matrix. If our algorithm is calculating a value *Xij*, then it only needs to read the *Bij* value once at the beginning of the calculation. This makes it easy to just store the results back into the memory for the *B* matrix. We didn't see any noticeable performance improvements, but we do save quite a bit of space in global memory.

Caching Most Reused Values of X in Shared Memory - One thing we thought would really give us a performance boost is caching values of X that we solve. Namely, caching the first 4-7 values of X calculated by each thread since they are reused the most. It turns out the branching, conditionals, and extra code needed to implement this outweigh the speedup due to accessing shared memory for just 4-7 values of X per thread. There is an example in the lib/mat_mul_gpu.cu code, however it isn't used because it gave us worse performance.

Intrinsic Functions - We tried using intrinsic functions, such as `__fadd_rn(x,y)` instead of `x + y`, but we did not see any performance improvement. We speculate that this was because the compiler was already performing these translations for us. Additionally, using the -use_fast_math compile option doesn't do anything for performance.

Texture Memory - We considered using texture memory for the B matrix since it is cached and therefore faster than global memory, however we decided not to do so since it is read-only, and we were already bringing the *A* matrix into shared memory.

Performance

While we were unable to reach the performance of the highly optimized `cublasStrsm()` implementation, we were able to realize dramatic improvements over the sequential calculation (Table 1). While the sequential calculation was at best 200 times *slower* than CUBLAS, our implementation approached only 5-7 times slower (Table 2).

	Test Input 1	Test Input 2	Test Input 3
Implementation	500	1000	5000
Sequential	0.532784	4.076346	762.339263
GPU	0.018986	0.073700	3.528778
CUBLAS	0.002591	0.014472	0.747662

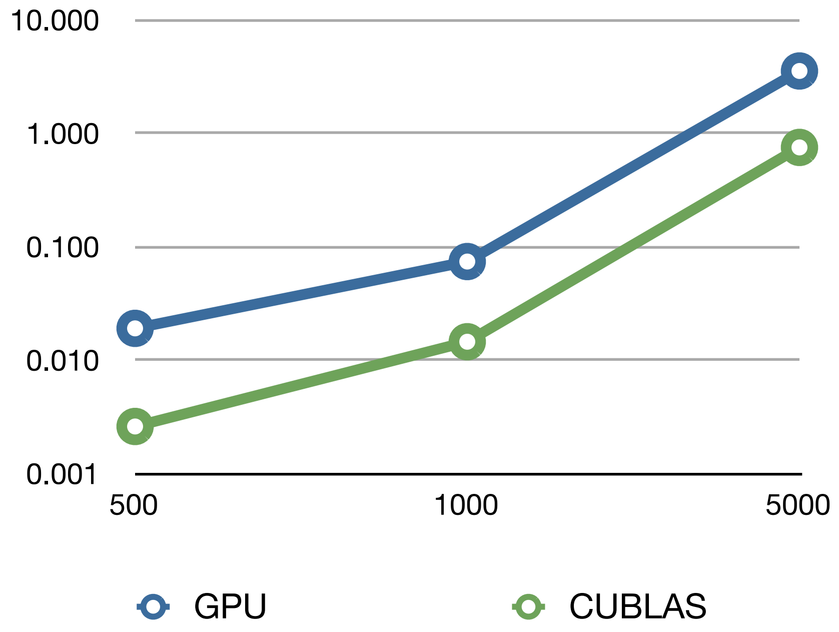
Table 1: Performance Results

Implementation	Test Input 1	Test Input 2	Test Input 3
Sequential	205.6	281.7	1019.6
GPU	7.3	5.1	4.7
CUBLAS	1.0	1.0	1.0

Table 2: Speedup Relative to CUBLAS

Figure 1 illustrates the nearly O(n) running time of both the GPU and CUBLAS implementations. Note that while the dependent axis is on a logarithmic scale, the times are plotted versus matrix width, not matrix size.

Figure 1: Execution Time vs. Input Matrix Width

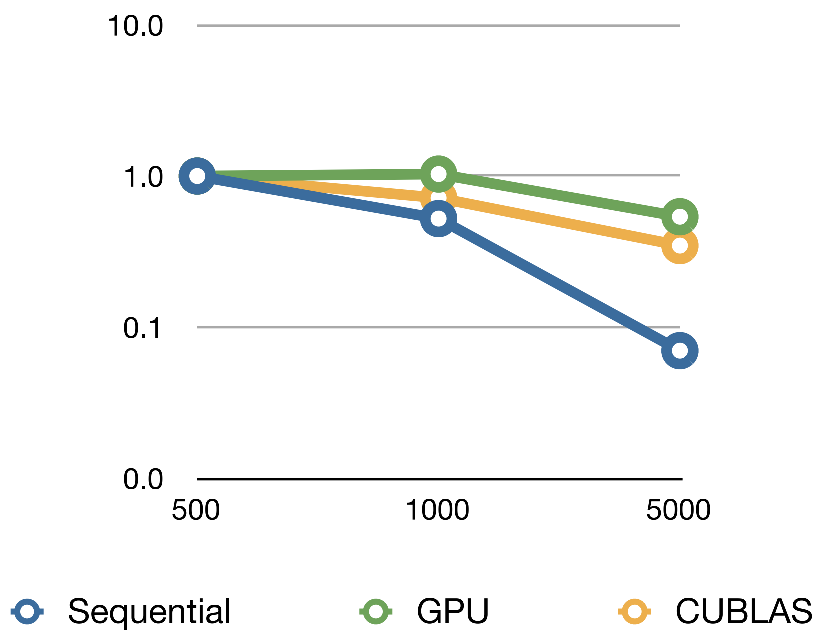


Another interesting performance characteristic was the rate at which elements were calculated. All of the implementations slowed their rate of calculation as the problem size grew, but the sequential implementations rate of calculation dropped off faster than the GPU and CUBLAS.

	Test Input 1	Test Input 2	Test Input 3
Implementation	**500**	**1000**	**5000**
Sequential	1.0	0.5	0.1
GPU	1.0	1.0	0.5
CUBLAS	1.0	0.7	0.3

Table 3: Slowdown with Respect to Problem Size

Figure 2: Slowdown



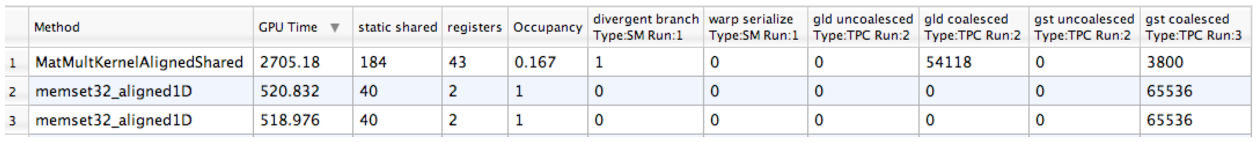
Overall there is no doubt that using the GPU for general purpose, dense calculations such as the triangular matrix solve operation is a clear win.

Profiling

In order to verify we succeeded with some of our optimizations, we ran the `cudaprof` program to analyze certain performance metrics. In order to do this, we created simple reduced matrices (100x100) based off the test_input_1 sample inputs. We ran this profiler on our Mac computers with NVIDIA GPUs. The profiler results can be found in the report directory of the submitted project (`csv` file format). We were able to get five important pieces of information from this profiling:

- Coalesced vs. uncoalesced accesses
- Branch divergence
- Register usage
- Occupancy
- Shared memory usage

The last two are important in general but they are skewed with this example because it is small compared to other inputs. Table 4 below shows the results of profiling the `MatMultKernelAlignedShared` kernel using 100x100 sized matrices (found in the `lib/mat_mult_gpu.cu` file in our project).



	Method	GPU Time ▼	static shared	registers	Occupancy	divergent branch Type:SM Run:1	warp serialize Type:SM Run:1	gld uncoalesced Type:TPC Run:2	gld coalesced Type:TPC Run:2	gst uncoalesced Type:TPC Run:2	gst coalesced Type:TPC Run:3
1	MatMultKernelAlignedShared	2705.18	184	43	0.167	1	0	0	54118	0	3800
2	memset32_aligned1D	520.832	40	2	1	0	0	0	0	0	65536
3	memset32_aligned1D	518.976	40	2	1	0	0	0	0	0	65536

Table 4: Screen Shot of Profiler Results

Based on the results from profiling, we were able to tell that we have no load (`gld uncoalesced`) or store (`gst uncoalesced`) accesses that weren't coalesced. This means that our efforts to optimize memory bandwidth by coalescing were successful. The second most important piece of information is the number of divergent branches. The CUDA best practices manual states your kernel should minimize the number of divergent branches. The profiler data shows the kernel has only one divergent branch. The third most important piece of information is register usage per thread. In our experience, if this is too high, it leads to the kernel not operating as expected. It is important to manage the code in the kernel such that you're not using too many registers. Finally, the occupancy (% usage of the GPU) and shared memory usage would be good to know for the largest tests. However, since larger tests wouldn't run on the relatively weak GPUs on our Macs, we couldn't get accurate numbers for these.

So, in conclusion, we were able to verify that our efforts to fully coalesce memory and to minimize divergent branches were successful. We also were able to pick up some useful information on the resources that our kernel used. We used this information mainly for debugging purposes.

Accuracy and Correctness

We want to note here that we've had a conversation over email about our accuracy and correctness with Akanksha. This section is just to reiterate the point made in those emails.

We designed a script (found in `bin/diff_coo_matrices.py`) to point out when we are more +/- .005 between two output files. Since CUBLAS is tried and tested, we always compared our outputs to CUBLAS. For the GPU output, we are seeing about 8 discrepancies against CUBLAS for input 1, and about 56 discrepancies for input 2. Input 3 has no differences. Likewise, the sequential version also fails to match perfectly with CUBLAS for inputs 1 and 2, but not for 3.

We believe the reason for this phenomenon may be found in Chapter 7 of the CUDA Best Practices Guide:

> 7.2.4 IEEE 754 Compliance
>
> All CUDA compute devices follow the IEEE 754 standard for binary floating-point representation, with some small exceptions. These exceptions, which are

detailed in Section G.2 of the CUDA C Programming Guide, can lead to results that differ from IEEE 754 values computed on the host system.

One of the key differences is the fused multiply-add (FMAD) instruction, which combines multiply-add operations into a single instruction execution and truncates the intermediate result of the multiplication. Its result will differ at times from results obtained by doing the two operations separately.

So, for example, this for loop:

```
for (int i = 0; i < N; i++) {
     S -= A[i] * X[i];
}
```

Would look something like this in our code:

```
S -= A[i] * X[i] + A[i+1] * X[i+1] + .... A[i+N-1] * X[i+N-1];
```

The point is, this code goes from having no fused multiply-adds to having numerous. We believe this may be the source of discrepancies between our GPU and CUBLAS implementations.

There is another section in the CUDA Best Practices Guide that talks about accuracy of floating point math in general:

7.2.2 Floating-Point Math Is Not Associative

Each floating-point arithmetic operation involves a certain amount of rounding. Consequently, the order in which arithmetic operations are performed is important. If A, B, and C are floating-point values, (A+B)+C is not guaranteed to equal A+(B+C) as it is in symbolic math. When you parallelize computations, you potentially change the order of operations and therefore the parallel results might not match sequential results. This limitation is not specific to CUDA, but an inherent part of parallel computation on floating-point values.

Since our implementation is row major and CUBLAS is column major in its calculations, we believe our calculations are done in a different order therefore causing them to be off somewhat.

Our understanding of the main goal of the project was that we needed to focus on optimizing the GPU and understanding the intricacies involved in obtaining good performance, and communications with the TA confirmed this position. As a result we chose not to focus on the very small percentage of mismatches we were seeing in the results. Overall we believe we achieved our goals of delving into GPU programming, and this report demonstrates our results and findings.

Works Cited

NVIDIA. (2010, January 1). CUDA C Best Practices Guide. Retrieved May 10, 2011, from NVIDIA Developer Site: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_BestPracticesGuide_3.1.pdf