Aaron Walsh

Parallel and Distributed Computing

5/9/17

Simple Matrix Multiplication:

CUDA GPGPU computation vs CPU computation.

Matrix multiplication is a common computer science problem that has applications ranging from computer graphics to physics calculations and simulations. Finding the product of two matrices involves the multiplicative sum of the values of a row of one matrix to the values of a column of another matrix. The simple algorithm I implemented for this project has a runtime complexity of $O(n^3)$, as seen in figure one. I

```
1 for M rows in A:
2   for N cols in B:
3     for K rows in B (or cols in A):
4       result[M][N] += A[M][K] * B[K][N]
```

Figure 1 - sequential pseudocode

I then implemented a CUDA version of this algorithm by breaking the resultant matrix into blocks of size 128 x 1 threads. I then created a kernel function in which one single index of the resultant matrix was computed by multiplying one row of A against one column of B. This kernel function would be called by the Nvidia GPU scheduler and assigned to many different CUDA cores.

I implemented this algorithm sequentially using C++ with the g++ compiler optimizations set to 4. I ran this on implementation on square matrices with sizes ranging from 250x250 to 2750x2750, using a computer powered by an Intel Core™ i7-6700K processor running at 4Ghz. At this point, the number of computations had become inordinate for the single CPU core to handle, taking upwards of two minutes to complete computation.

The GPU parallel implementation had access to up to 1,920 identical CUDA cores present in the Nvidia GTX™ 1070 graphics card. Running this simple algorithm on this GPU with blocks as described above yielded a speedup of between 50 and 130 times faster than the sequential CPU implementations. I ran this version on matrices of sizes 250x250 to 8000x8000. At this point, the number of computations had begun to exceed what the GPU was capable of and the computation took just over one minute.

This algorithm does not particularly favor memory locality speedups, as each of the values in each matrix are accessed sequentially. In the A matrix, rows can quickly be loaded into cache, but in the B matrix, columns are not fetched. This is particularly evident in the GPU, as the cache size is limited and the actual computational speed far exceeds that of even the fastest VRAM. GPUs are designed specifically to compute matrices, so hardware speedups can definitely be found. Nvidia's CUDA has the ability to create a pool of "shared memory" which is a block of reserved cache available to all processes. Speedups could be achieved by taking advantage of this built in cache. I was however unable to determine how large this cache is on my particular GPU die, and the use of this memory is out of the scope of this project. Memory advantages could be exploited on both CPU and GPU implementations.

Below I have plotted the GPU and CPU computation times vs the size of the matrices, seen here in Figure 2. You can clearly see that the CPU implementation far overshadows the GPU implementation by several orders of magnitude. I also plotted the relative speedup vs input size as seen in Figure 3. Speedup remains over 50x for all sizes, but varies slightly. This could possibly be explained as background tasks consuming resources, overhead time, GPU core availability, CPU/GPU scheduling, or memory latency.

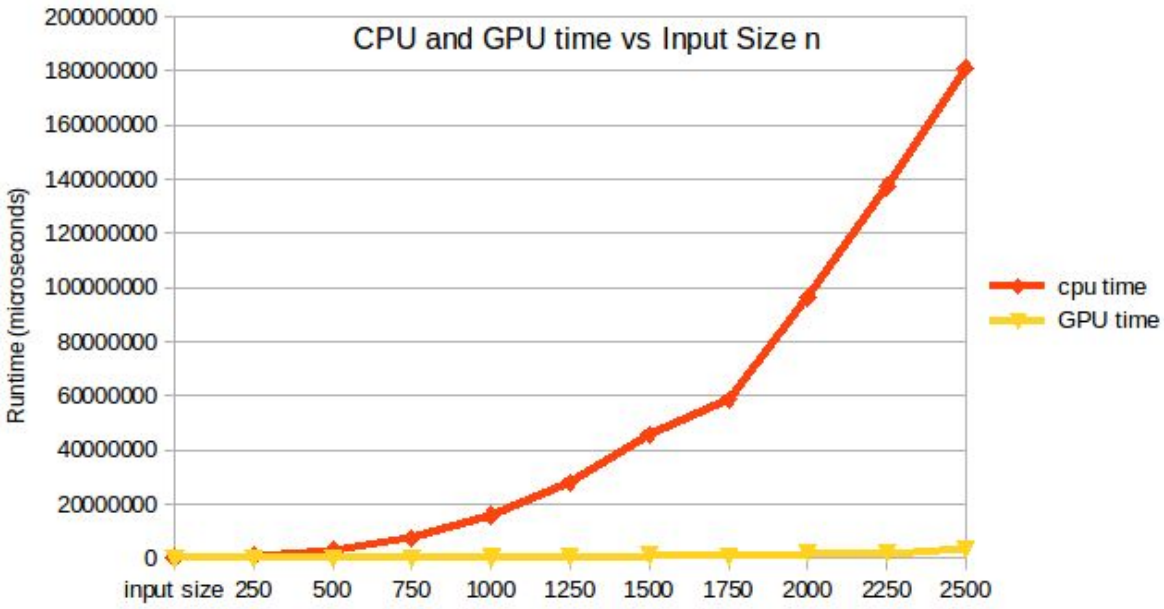

Figure 2: CPU and GPU time vs input size

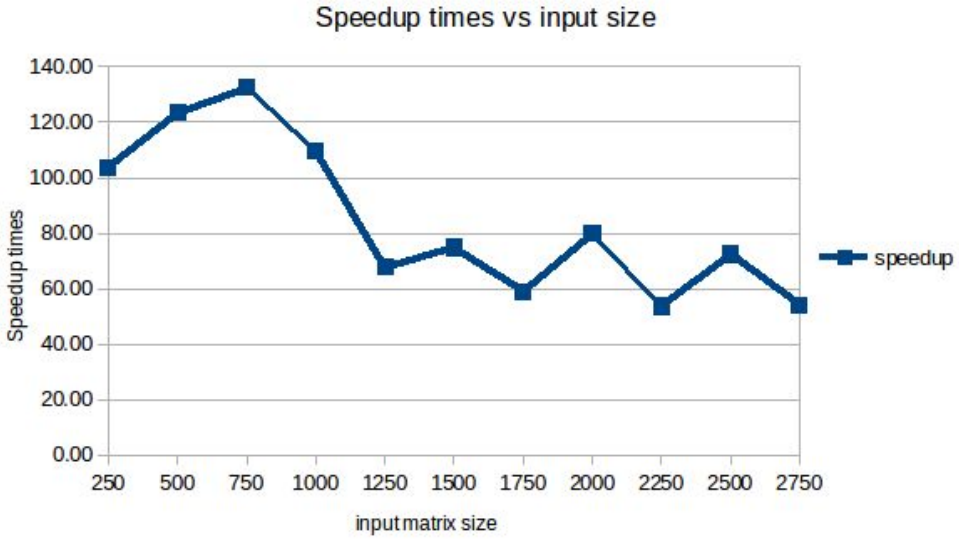

Figure 3- GPU Speedup times vs input size.