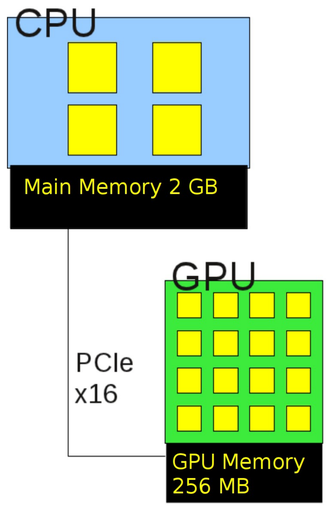
Algorithms for Large Matrix Multiplication on Graphics Processing Units

By Matthew Beine
Mentor: Prof. Rong Ge



Introduction to GPU Computing

Graphics Processing Units (GPUs) achieve high performance through parallel computing, and are equipped with many more processing cores than a traditional CPU. Lower-end GPUs, such as the Ge-Force 8400 which we used, have 16 cores, while newer, higher-end models can have well over 100 cores. However, all of this computing power comes with a significant drawback. GPUs have only small amounts of memory available to them (256 MB on the model we used), and data transfers between the CPU and GPU are rather slow. Therefore, in order to utilize the computing power of a GPU for general purpose computing, large problems must be efficiently partitioned.

CUSUMMA

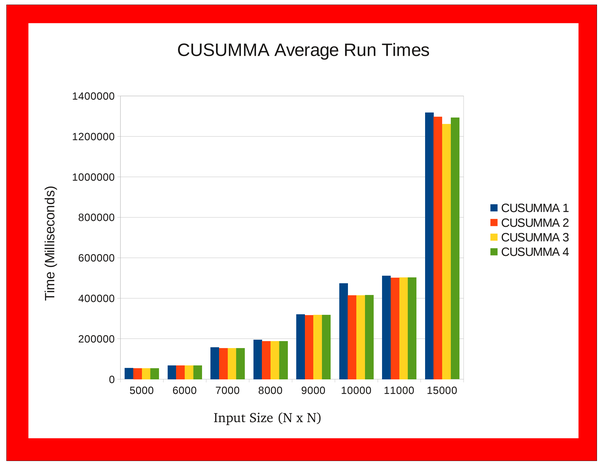
The Scalable Universal Matrix Multiplication Algorithm (SUMMA) is a parallel algorithm based on the outer product approach for partitioning input matrices. This approach partitions the matrices along the shared dimension, uses the partitions to calculate partial output values, and then sums the partial values to obtain the final results. CUSUMMA is a GPU implementation of SUMMA, built using NVIDIA's CUDA library. We have implemented a few variants of CUSUMMA, and have numbered them in the order in which we tested them.

CUSUMMA 1 we implemented from scratch. It is a fairly concise implementation of the algorithm. The CUBLAS library assumes matrices are stored in column-major format, so we used column-major indexing throughout this implementation.

The second version which we tested (referred to as CUSUMMA 2) was a previously programmed implementation provided for free by Byron Galbraith. His implementation uses row-major indexing. By changing the order of the parameters when calling cublasSgemm(), the correct results can still be achieved. Another key difference is that he used data packing when transferring the input to the GPU. CUSUMMA 2 outperformed CUSUMMA 1, typically by about one or two percentage points.

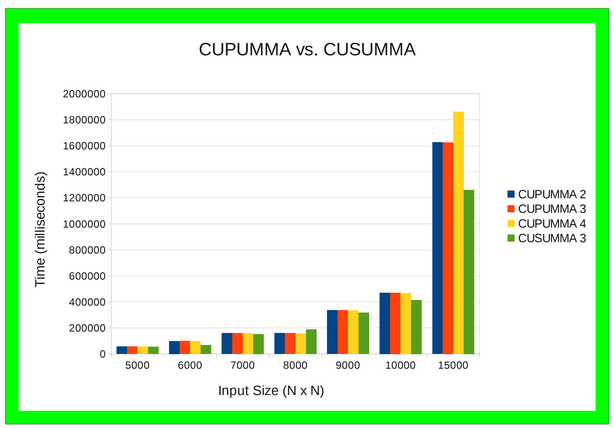
CUSUMMA 3 is a modified version of CUSUMMA 2, this time without data packing. For most input sizes, this had no effect on performance, but for the largest size we tested (n = 15000), we saw an improvement of 2.8%.

In CUSUMMA 4, we revisited our original implementation. We modified our original code to use row-major indexing, as we had noticed that it made data transfers more efficient. In theory, the performance of CUSUMMA 4 should be comparable to CUSUMMA 3. In practice, there was no difference between the two for most input sizes, but for the largest size, CUSUMMA 3 performed better by about 2.5%.



CUSUMMA vs. CUPUMMA

CUSUMMA, which aims to minimize data transfers, generally outperformed CUPUMMA. To compare the two algorithms, we used the data for CUSUMMA 3 (as it performed as good or better than the other versions, depending on input size), and CUPUMMA 2 (as there was essentially no difference between 2 and 3). The difference in execution time varied greatly for different input sizes. It was greatest for n = 6000, at which CUSUMMA ran in 31.5% less time. At n = 8000, CUPUMMA actually outperformed CUSUMMA, running in 17.7% less time.



CUPUMMA

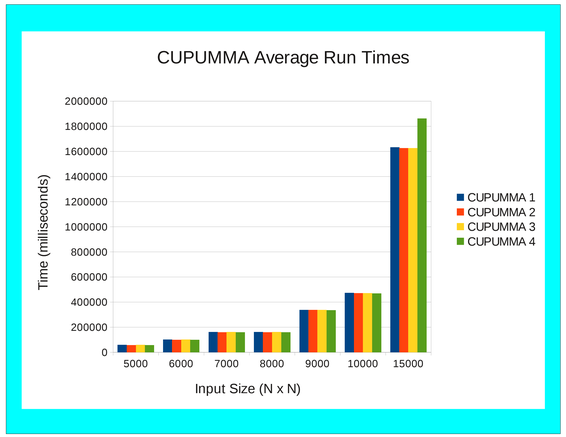
The Parallel Universal Matrix Multiplication Algorithms (PUMMA) is a parallel algorithm based on the inner product approach. The CUDA version of this algorithm, CUPUMMA, partitions input matrices by rows for matrix A and by columns for matrix B, allowing the final products to be computed right away for each piece of output matrix C. Again we have implemented a few variants of the algorithm.

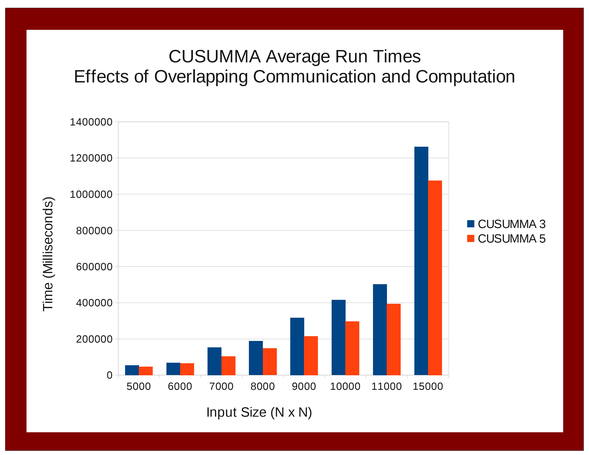
CUPUMMA 1 is a simple implementation of the algorithm. Pieces of A are copied to the GPU one by one, and for each piece, we iterate through the pieces of B.

CUPUMMA 2 is a minor improvement on CUPUMMA 1. When starting on the next piece of A, the last piece of B that had been used is reused right away, slightly reducing the number of data transfers. The gain in performance was about .5%

In CUPUMMA 3, we added data packing. This had no significant effect on performance.

In CUPUMMA 4, we cycled through pieces of B in the outer loop and pieces of A in the inner loop, since pieces of A should transfer more efficiently. This improved performance by about .5% (over CUPUMMA 2) for some input sizes, but for the largest size (n = 15000), CUPUMMA 4 took about 14% longer to execute.





Overlapping Communication and Computation

A final idea which we tested was to have data transfers and computations occur at the same time. To do this, we simply cut partition sizes in half, so that while the GPU was performing matrix multiplication on one set of pieces, the next parts can be transferred. On CUSUMMA, this greatly enhanced performance (by as much as 32% for n = 7000). On CUPUMMA, initial results showed no gain in performance. Future work will involve ensuring optimization, to see if performance can be increased any more.

Conclusions

We have tested two algorithms for large matrix multiplication on GPUs. Of the two, CUSUMMA performed better. This is due to the fact that it uses the outer product approach, which minimizes data transfers in this setting. We also found that overlapping communication and computation can yield even bigger performance gains. Future work involves ensuring that the best implementations are fully optimized, and trying to integrate the best approach into MAGMA (Matrix Algebra on GPU and Multi-core Architectures), a linear algebra package for heterogeneous architectures.

References

J. Choi, J. J. Dongarra, and D. W. Walker, “Pumma: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers,” Concurrency: Practice and Experience, Vol 6(7): 543-570, 1993.

R. A. V. D. Geijn, and J. Watts, “Summa: Scalable Universal Matrix Multiplication Algorithm,” Concurrency: Practice and Experience, 1995.

CUSUMMA code from Byron Galbraith available at: http://code.google.com/p/cusumma/