CME213 Final Project Report

Man Long, Wong

1. Implemented Strategies

1.1. General Matrix-Matrix Multiplication

In this project, the computational cost heavily depends on matrix-matrix multiplication. Therefore, a considerable amount of effort is made to optimize the matrix-matrix multiplication process. Four versions of General Matrix-Matrix Multiplication (GEMM) are implemented. GEMM operation can be expressed as $\mathrm{D} = \alpha * \mathrm{A} * \mathrm{B} + \beta * \mathrm{C}$. The first version of GEMM (algorithm 0) is a very naïve approach where 1D blocks are used and every thread of the blocks is responsible for the computation of one element in matrix D. The second version (algorithm 1) is still a naïve approach but 2D blocks are used instead of 1D blocks. The third version of GEMM (algorithm 2) uses 2D blocks with shared memory. In this method, each thread block (32x32 threads) compute a sub-matrix (32x32) of the output matrix D. A loop is used to iterate along rows and columns of matrices A and B respectively. At each iteration step, corresponding sub-matrices (32x32) of A and B are loaded into the shared memory. In the last version (algorithm 3), each block has 16x4 of threads. This block computes a sub-matrix of size 64x16 in D. All 64 threads loop over the corresponding rows in A and columns of B simultaneously. Within each iteration, only a 16x16 sub-matrix of B is loaded into the shared memory but not the sub-matrix of A. Each thread has a local array c[16] to store the sums of the 16 elements it is responsible for. A unit test was done to compare the Gflop/s of different GEMM's. The results are shown in figure 1.

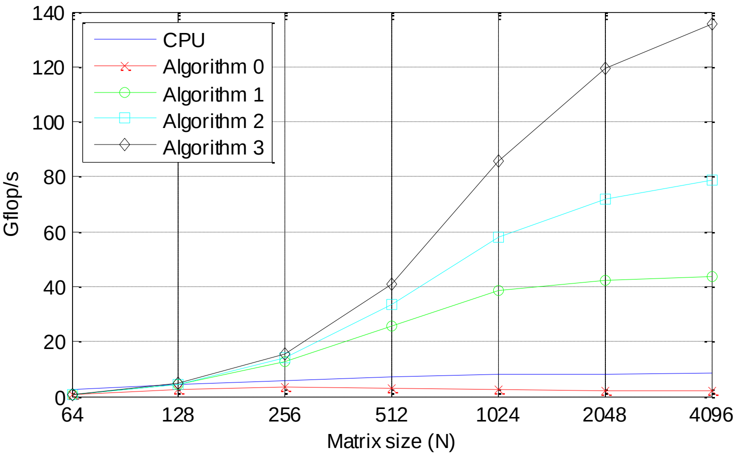

Figure 1 Comparison of Gflop/s of different algorithms

As we can see from the figure 1, algorithm 0 is very slow and is even slower than using CPU for computation. This is due to the fact that the matrices are in column-major order so the reads in matrix A are not coalesced. If the matrix size is large, excessive memory loaded into the cache will be erased before reuse. Algorithm 1 used 2D blocks so there is better-reuse of memory loaded into cache. As seen from the plot, performance of algorithm 1 increases significantly compared to algorithm 0. Algorithm 2 is even faster than algorithm 0 because data is accessed from shared memory instead of global memory. Shared memory is a faster on-chip memory compared with the global memory. However, it should be noted that shared memory is still slower than the on-chip cache/register file. In algorithm 3, each thread loops along the corresponding rows of matrix A. In each iteration, a thread block loads a column of A of size 64 into the cache which is beneficial from fetching and caching. The partial sums from each iteration is also stored in the cache. Only data of sub-matrix of B is loaded into shared memory. With the use of faster on-chip memory, algorithm 3 is faster than algorithm 2. From the plot, we can also see than GPU acceleration is only efficient when matrix size is larger because if the matrix is small, the cost of transferring data over the PCI express bus will become dominant compared to actual computation.

1.2. Parallelization optimization

Three versions of parallelized code were implemented. The comparison of profiles of the three codes for two processes is shown in figure 2. Profiling is done by using MPI_Wtime().

In the first version of code, during the iterative gradient descent process, rank 0 node is responsible for splitting inputs into batches according to the batch size. For each batch, rank 0 further splits the batch into sub-batches and send the sub-batches to different nodes by using MPI_Scatter(). Then each node performs the feedforward and backpropagation to find its $\Delta W^{(1)}, \Delta W^{(2)}, \Delta b^{(1)}, \Delta b^{(2)}$ from the sub-batch of inputs it receives. Then, each process uses MPI_Allreduce() to combine the gradients from different nodes. After that, each node update its own neural networks by using the reduced gradients it receives. In this version of code, data is sent to GPU only when GEMM's are required in the feedforward and backpropagation operations. Data has to be sent back to CPU to do other operations such as applying non-linear functions to outputs or element-wise multiplication. The naïve 2D GEMM (algorithm 1) is used.

In figure 2, it can be seen that for version 1 of the code, a large portion of time (over 80% of total time) is spent in performing feedforward and backpropagation. Therefore, to optimize the code, it is most effective if we can optimize the feedforward and backpropagation. From nvvp, it is noticed that a large amount of time is spent in the GEMM computation in GPU. To optimize this part, the algorithm 2 of GEMM, which makes use of shared memory is used in the second version of code. Besides, the data transferring cost between GPU and CPU over the PCI express bus is minimized by doing the feedforward and backpropagation entirely on GPU. It can be seen that total cost of version 2 is almost half of that of version 1 because lots of time is saved in computation and data transfer in the feedforward and backpropagation processes.

In the final version of code, the cost of feedforward and backpropagation is further minimized by using the final GEMM algorithm (algorithm 3). Besides, it is noticed some data such as $\mathrm{z}^{(1)}, a^{(1)}, z^{(2)}$ are not needed to be transferred back to CPU so the corresponding data transferring cost can be cut. It is also found the cost of scattering can be largely reduced by performing it outside the iteration loop since each process actually continuously works on the same portion of inputs. It can be seen from figure 2 that the corresponding portions of final version further speed up.

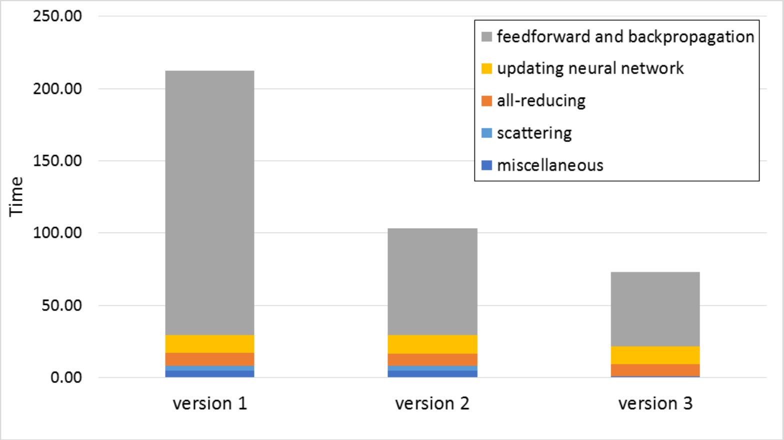

Figure 2 Profile of different versions of parallelized code. Two nodes are used

There were originally more ideas to optimize the code but not all of them succeed. One idea is to use streaming so data of next GEMM can be loaded to GPU by using cudaMemcpyAsync() while GPU is still computing the previous GEMM. However, to do this, host memory needs to be pinned and unpinned before data transfer to GPU and after data transfer from GPU. By using nvvp, it is noticed that these processes are quite time-consuming so the final computation time of using streams increases. Another idea is to minimize the cost of all-reducing by using MPI_Isend and MPI_Irecv. However, through testing, it is noticed this method does not decrease the reducing cost because the cost in all-reducing is actually mainly due to summation of gradients but not data communication. Therefore, these ideas were not implemented in the final version. It can be concluded that there is always some amount of unavoidable cost (bottleneck) in transferring data over PCI express bus and in reduction of gradients from different nodes.

2. Debugging and Verification

To verify that the code is correctly implemented, lots of unit tests were implemented. The unit tests confirm that the GPU version of the functions are correctly implemented by comparing the GPU results with those obtained by Armadillo library. Speed of the CPU and GPU functions is also compared. The results are shown below:

	transpose	elementwise substraction	elementwise multiplication	column reducting	sigmoid	softmax
CPU time (s)	0.0086	0.0215	0.0339	0.0032	0.0879	0.0726
GPU time (s)	0.0073	0.0095	0.0095	0.0032	0.0070	0.0079
Speed-up	1.18x	2.27x	3.55x	1.00x	12.47x	9.12x

Table 1 Speed compaison of CPU and GPU functions. Matrix size of 1000x1000 is used in all tests

The speed results of GEMM's are not shown here as it is already discussed in previous section. All GEMM's passes the test with combinations of matrix sizes of (80x70, 70x90) and (250x400, 400x150). Except GEMM's, other GPU kernels are not optimized because these functions only account for very small fraction of the computational time but It is confirmed that the GPU kernels are at least as fast as the CPU versions.

To test the whole code, results between sequential and parallel code are compared. The round-off errors are avoided by selecting a very small learning rate. In the test, the following parameters are chosen: num_neuron = 1000, reg = 0.0001, learning_rate = 0.01, num_epochs=20, batch_size = 800 and num_procs = 4. Both the sequential code and the parallel code has the same precision (0.9696666598) on dev set at the end of iteration. The L2 errors of difference of ΔW's and Δb's between the sequential and parallel iterations at each step are also checked to be very insignificantly small.

3. Benchmark and Performance Analysis

The first performance analysis is done on the final version of code with the following settings: num_neuron = 1000, reg = 0.0001, learning_rate = 0.05, num_eposchs = 20, batch_size = 800. The profiling results of different number of processes are shown in figure 3.

In figure 3, it can be seen that the parallelized code with GPU computing is much faster than the sequential code with only CPU computing. However, increasing the number of nodes from 2 to 6, the total computation time does not decrease. As reflected in the stacked columns, this is because the computation time in feedforward and backpropagation increases with number of nodes. When we increase the number of processes, the size of input matrix each process that is responsible for decreases. As discussed in previous version, GPU GEMM's are less efficient if matrix size is smaller as more time is spent in data transfer. This explains why the feedforward and backpropagation take more time for more total number of processes.

To have better scalability, we have to make sure at least the size of matrix (sub-batch) each node work on to be the same if we increase the number of processes. In the next performance analysis, the batch size increases in a way such that each node will work on input matrices with a constant number of rows (400). The batch sizes for different processes are shown in table 3.

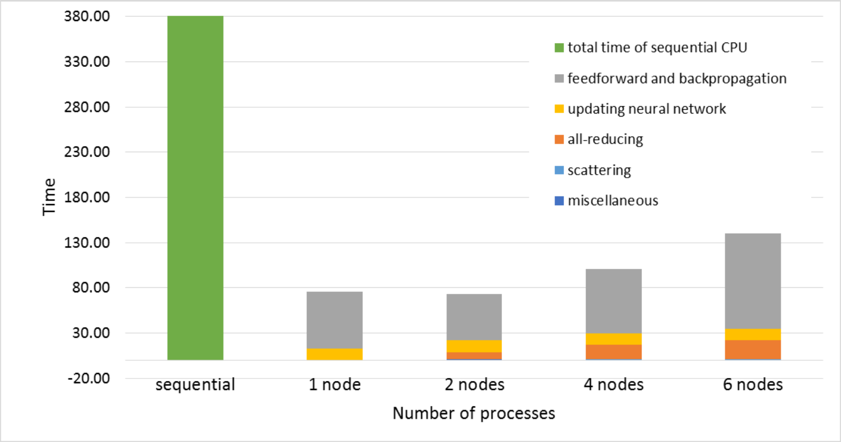

Figure 3 Profiles of different number of processes of first performance analysis. batch_size = 800 is used

Number of processes	1	2	4	6
Precision on dev set	0.975	0.976	0.976	0.972

Table 2 Precision of different number of processes for the first performance

The settings of the iterations are as same as the previous case except that batch size varies with the total number of processes. The total computation time and efficiency of different number of processes is shown in figure 4.

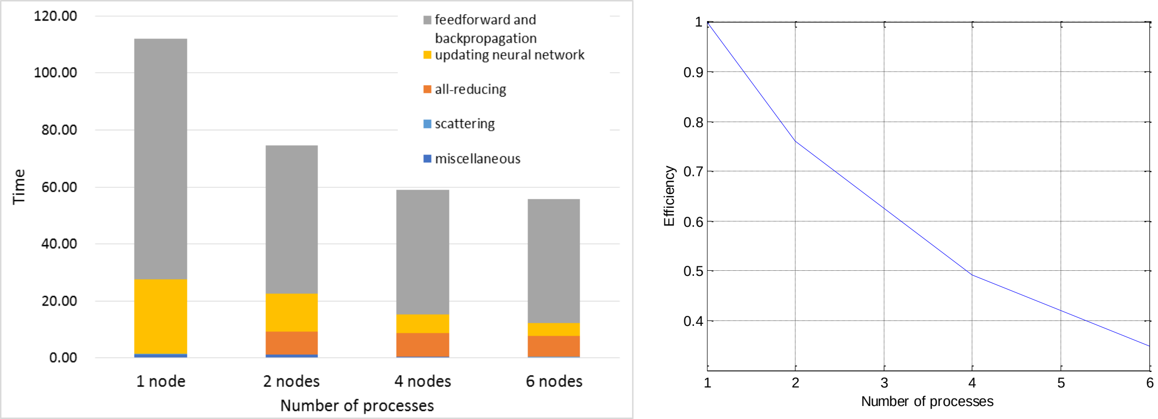

Figure 4 Profiles and efficiencies of different number of processes of second performance analysis

It can be seen from the plots that by fixing the sub-batch size for each node, the problem becomes more scalable but the efficiency still decreases as the total number of processes increases. Besides, the precision on dev set becomes lower with increasing number of processes, as shown in table 3. The bottleneck of performance is due to communication cost in all-reducing among different processes and also communication cost between CPU and GPU over PCI express bus.

Number of processes	1	2	4	6
Batch size	400	800	1600	2400
Precision on dev set	0.978	0.975	0.973	0.967

Table 3 Batch sizes and precision of different number of processes for the second performance analysis

The fastest time to achieve precision above 0.97 is with the following settings: num_neuron=1000, reg=0.0001, learning_rate=0.045, num_epochs=8, batch_size=800, num_procs=2. The total time including reading data is 35.0s and precision is 0.971.