CME213: Parallel Computing using MPI, OpenMP and CUDA
Final Project (Part II) **Chi Zhang**
Stanford University SUID: 06116342
Spring 2018 Date: June 10, 2018

1 Correctness Analysis

1.1 GEMM Correctness

According to the output file, my implementation passes the two default GEMM tests. Although it takes more time with respect to cuBLAS reference in the first test case, my implementation yields zero relative difference.

1.2 Overall Correctness

The differences between GPU and CPU code in three grading test cases are listed in the table below:

	$W^{(1)}$	$W^{(2)}$	$b^{(1)}$	$b^{(2)}$
Case 1	1.24881×10^{-15}	7.7821×10^{-16}	3.53874×10^{-15}	9.19953×10^{-16}
Case 2	4.53111×10^{-9}	5.07357×10^{-11}	6.10204×10^{-9}	2.28154×10^{-10}
Case 3	3.64473×10^{-11}	7.47798×10^{-13}	9.35092×10^{-11}	2.45754×10^{-12}

Table 1: Largest norm of difference between sequential and parallel training

It is shown that the largest norm of difference between CPU and GPU results appears on $b^{(1)}$ in the scale of 10^{-9}, which is significantly smaller than the threshold 10^{-7}. Also, the accuracy on the validation set computed by GPU is the same as that by CPU in all three test cases. Therefore, the GPU implementation is correct.

1.3 Effects of Parameters

The difference between CPU and GPU results depends on training hyperparameters, among which very important ones are number of epochs, batch size, learning rate and number of neurons in hidden layer. By running the program with different parameter settings, it can be found that:

- Number of epochs does not affect difference much. However, the precision on validation set decreases a lot if using a relatively small number of epochs.
- Batch size is, in most cases, decided by the computational capability of the device. If the device (CPU or GPU) could allocate enough memory for the batch of data, then using larger or smaller batch size does not change the result much.
- The training process is definitely sensitive to learning rate. Using a 10 times larger learning rate results in differences in the scale of 10^{-3}, which is not negligible.
- The number of hidden neurons have very little effect on differences. It seems differences become a little bit smaller (but remain at the same level of significant digits) probably due to

less arithmetic operations involved. However, it does affect the precision of prediction. After all, more neurons, more powerful the network is.

2 Performance Analysis

2.1 Overall Profiling

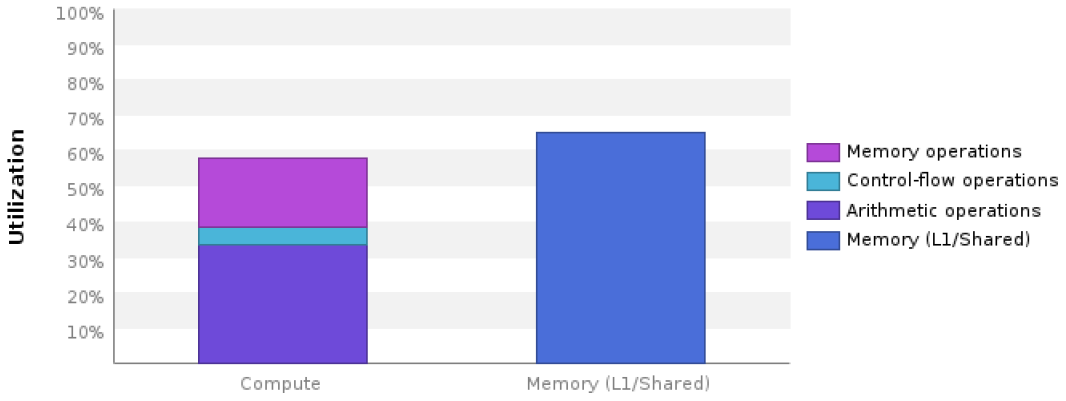
The NVIDIA Visual Profiler (NVVP) shows the major problems of my implementation lies in two aspects, low concurrency and low memcpy throughput:

- Technically, there is zero concurrency in my program since all kernels are launched sequentially. Due to time limit, concurrent approaches using `cudaStream` are not adopted. This constraint leads to low compute/memcpy efficiency and low compute/memcpy overlap.
- It is reported by NVVP that the bandwidth is 1.678GB/s in average, which is much lower than the theoretical peak bandwidth, 8GB/s (PCI 2.0 $\times$ 16 lanes) and practical peak bandwidth, 6 GB/s. This is caused by the fact that the memory allocated by `malloc()` is in virtual memory space (pageable non-pinned memory). When `cudaMemcpy()` is called, the CUDA driver has to first memcpy the data from the non-pinned pointer to an internal pinned memory pointer, and then the "host to device" DMA can be invoked. The low bandwidth problem could be alleviated by using `cudaMallocHost()` and initializing data directly in pinned memory. However, this was totally beyond my understanding when I wrote my parallel code.

It is also shown that most of computational cost comes from `gpu_GEMM` and `gpu_GEMMSigmoid` kernels. `gpu_GEMM` is the routine to perform general matrix multiplication while `gpu_GEMMSigmoid` takes advantage of the former kernel to apply sigmoid activation on the result matrix. The two kernels take a significant portion of computation (around 90%) on each of the four processors while all the other kernels only take less than 5%.

2.2 GEMM Kernel Profiling

Since it has been found that `gpu_GEMM` kernel overtakes most of computation in my implementation, I profiled it separately.



As mentioned above, the kernel's compute utilization is significantly lower than its memory utilization. Thus, the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of L1/Shared memory. After careful examination, it is found that the total bandwidth of L1/Shared memory reaches a medium level, 640.145 GB/s. It is suggested by NVVP to use 64-bit accesses to shared memory and 8-byte bank mode to achieve double throughput. The `myGEMM()` function was updated by adding the line

```
cudaDeviceSetSharedMemConfig(cudaSharedMemBankSizeEightByte)
```

before launching `gpu_GEMM` kernel.

2.3 Effects of Parameters

The effects of number of neurons and batch size are investigated. Based on statistics in the table below, varying the number of neurons in hidden layer seems to only affect precision but not runtime performance.

	$T_{\text{CPU}}(s)$	$T_{\text{GPU}}(s)$	$T_{\text{CPU}}/T_{\text{GPU}}$
10	15.5662	2.15743	7.22
20	22.102	2.5382	8.71
50	35.417	3.78921	9.35
100	60.9044	6.40634	9.51
500	236.861	24.2656	9.76

Table 2: Speedup with different numbers of neurons in hidden layer

	$T_{\text{GPU}}(s)$
100	282.643
200	150.082
400	83.1684
800	48.8993

Table 3: Performance with different batch size

As shown in Table 3, however, changing batch size does affect performance. Generally speaking, overhead induced by memory copies between host and device becomes dominant when batch size is relatively small.

3 Optimizations

3.1 General Matrix-Matrix Multiplication

In this project, the computational cost heavily depends on general matrix-matrix multiplication which is expressed as

$$D = \alpha AB + \beta C$$

Therefore, a considerable amount of effort is made to optimize the matrix-matrix multiplication process. The first version of GEMM kernel uses a naive approach where each thread inside 2D blocks is responsible for computing one element in D. It only uses global memory and does not consider memory coalescing at all. The current version of GEMM (Algorithm 2) uses 2D blocks with shared memory. In this method, each thread block (32×32 threads) compute a 32×32 sub-matrix of the output matrix D. It loops over rows and columns of matrices A and B respectively. At each iteration, corresponding 32×32 sub-matrices of A and B are loaded into shared memory.

3.2 Memory Coalescing

In the current implementation, memory accesses are organized according to the optimal memory access patterns to reduce number of cache lines used to load data.

3.3 Other Kernels

All arithmetic operations during the training process are implemented in CUDA kernels. It is worth pointing out that GEMM kernels can be combined with other operations to boost overall performance. For example, sigmoid activation is always applied after matrix-matrix multiplication, so it is a good choice to perform non-linearity computation inside GEMM kernel.

3.4 Memory Management

It is inevitable that the implementation involves a huge number of bytes of memory on both host and device. All memory should be managed elegantly to ensure the minimum memory allocations and copies, and no memory leaks occur. Compared with the initial version, which allocates and copies memory at will whenever in need, the current implementation takes care of memory management with three major optimizations:

- Allocate once and use repeatedly. Since the architecture of the two-layer neural network is fixed, $W^{(i)}$, $b^{(i)}$ and their derivatives $dW^{(i)}$, $db^{(i)} (i = 1, 2)$ in each batch are fix-sized matrices and vectors once the batch size is calculated. It is efficient if associated memory is allocated once and used repeatedly in different batches and epochs.

- Avoid redundant allocations. Taking advantage of the fact that `myGEMM()` performs matrix-matrix multiplication in-place, there is no need to allocate extra memory to store the result. For example, in the first layer, both $a^{(1)}$ and b^1 pointers could point to the same memory chunk after the multiplication $a^{(1)} = W^{(1)}X + b^{(1)}$.

- Avoid unnecessary data transfers. Intermediate results in feedforward and backpropagation do not need to be transferred back to host so the associated data transfer cost can be saved.

3.5 MPI Broadcasting and Scattering

MPI broadcasting brings overhead. It is found that the only variable that has to be broadcast is the data size N. In the initial implementation, MPI scatters data to each of the processors in every batch, meaning `MPI_scatterv()` is called `(2 * num_epochs * num_batches)` times in total. This is obviously computationally inefficient in that data for each batch can be determined before training. Therefore, the current implementation splits data into batches and memcopy it to device

in advance before scattering batched data to processors. In this way, `MPI_scatterv()` is only called `(2 * num_batches)` times in total. During the training process, processors can visit data for each batch via pre-stored pointers.

3.6 Pointer Aliasing

This is a bonus given by CUDA 8.0 and later versions of CUDA compliers. It is noticed that input matrices of most kernels are generally read-only data which are never written for the lifetime of the kernel. However, due to potential aliasing, the compiler cant be sure a pointer references read-only data. To avoid extra assembly instructions, the pointers referencing input matrices are marked with `const` and `__restrict__` keyword.