CME 213 - Parallel Computing
Final project

Leon H. Kloker
Institute for Computational and Mathematical Engineering
CME 213, Stanford University
`leonkl@stanford.edu`

1 GEMM

General Matrix-Matrix multiplication (GEMM) operations of the form `C = alpha * A * B + beta * C` are fundamental in high performance computing, and they are commonly performed on GPUs using CUDA kernels for superior speed and efficiency. Moreover, they are ubiquitous when neural networks are trained, as both forward and backward pass especially for dense networks consist of a chain of such operations. Hence, writing an efficient GEMM kernel is the first step in implementing a dense neural network in CUDA.

A naive implementation of GEMM in CUDA (Algorithm 1) assigns each thread to calculate one entry in the output matrix. However, this approach suffers from performance issues due to uncoalesced memory access. In this method, each thread needs to load elements from non-contiguous memory locations, causing memory access latency as GPU memory access is optimal when threads within a warp (a collection of 32 threads) read from adjacent memory locations. Moreover, this approach doesn't take advantage of the high-speed shared memory available on the CUDA device, which can be a significant bottleneck.

To optimize GEMM, a blocking strategy (Algorithm 2) can be used. This strategy breaks the matrices into sub-matrices that can fit into the GPU's shared memory, and uses a CUDA thread block to calculate each sub-matrix of the output matrix. Typically, a block of 32x32 threads is used to compute a 32x32 output sub-matrix.

In this method, blocks from matrices A and B are loaded into shared memory, with each thread reading one element of each sub-matrix. Once the data is loaded, each thread calculates and updates its corresponding entry in the output sub-matrix. A loop is used to traverse and multiply all the required entries in the A and B sub-matrices.

The advantage of this blocking strategy is that it uses the high-speed shared memory efficiently, reducing uncoalesced memory access by loading contiguous data blocks. Moreover, it encourages data reuse, as each thread in a block processes multiple elements from the loaded sub-matrices, leading to higher computational throughput.

The difference in performance can be seen in figure 1 as there is essentially one order of magnitude between each of the performances of Algorithm 1, Algorithm 2 and NVIDIA's cuBLAS implementation.

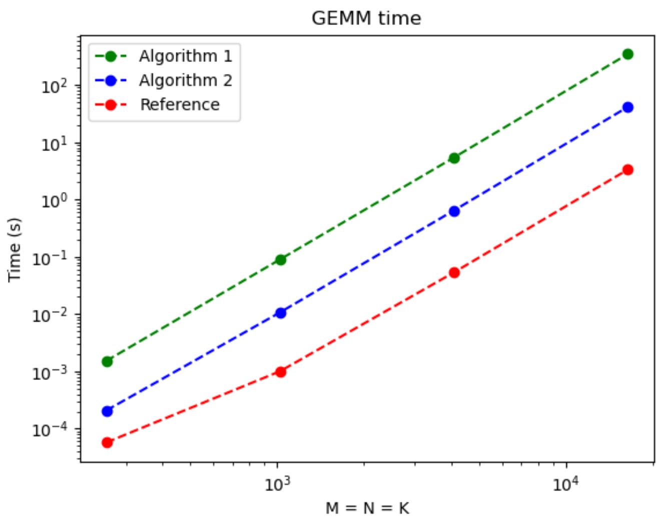


Figure 1: The plot shows the time required to run a GEMM over the problem size where the dimensions M, N and K of the matrices are all equal. The reference implementation is from NVIDIA's cuBLAS library.

2 Forward pass

The `feedforward_gpu` function implements the forward pass of a two-layer dense neural network on a GPU. The two layers use different activation functions: a sigmoid activation for the first layer and a softmax activation for the second layer.

The function operates as follows:

1. It first performs a forward propagation through the first dense layer of the neural network using the `GEMM_forward` function, which combines General Matrix-Matrix Multiplication (GEMM) with a broadcasted bias addition.
2. Next, it applies the sigmoid activation function to the output of the first layer using the `sigmoid_gpu` function, which assigns a thread to apply sigmoid to each entry in the matrix.
3. The function then propagates forward through the second dense layer using the `GEMM_forward` function again.
4. Finally, the softmax activation function is applied to the output of the second layer using the `softmax_gpu` function. Here, one thread is assigned to each column in order to calculate softmax along the first dimension.

Before the training loop starts, all variables are copied to the global GPU memory for efficient computation.

3 Backward pass

The `backprop_gpu` function implements the backpropagation for the two-layer neural network on the GPU.

The function consists of several kernel invocations, notably including a tiled transpose kernel, `transpose_gpu`, which utilizes shared memory of size 32×33 to perform the transpose while avoiding bank conflicts. Moreover, a `softmax_der_gpu` kernel was specifically written to calculate the elementwise matrix-matrix multiplications appearing in the calculation of the softmax derivative. At the end of the backwards pass, all gradients are multiplied by a weight parameter, which is equal to the minibatch size of the current MPI node divided by the overall batchsize. Thus, when the gradients are summed using `MPI_Allreduce`, no further normalization needs to be done.

The gradient updates are also done in parallel using a `addmat_gpu` kernel, which again assigns one matrix entry to each thread.

4 MPI

MPI is used to divide each batch during training into minibatches that are handled simultaneously by different processors. The first approach (Implementation 2) was to divide the current batch within the training loop into equal pieces and then distribute them to all nodes using `MPI_Scatterv`. Then, only after scattering, the received minibatch of data is copied into GPU memory and run through the forward and backward pass in order to calculate gradient updates. As alluded to earlier, the global gradient update is then calculated via `MPI_Allreduce`.

We will see, however, in section 5, most of the time is actually not spent on computations such as GEMM but on memory allocation and the GPU and MPI communication. Hence, for the final submission another strategy (implementation 1) was used to optimize the performance of the code. Here, the entire dataset is split into subsets containing all the minibatches an MPI node will look at during training. Then, the main process sends the training data to each process before the training loop is entered. This way, the only MPI communication except for exchanging gradients is done before the loop and only once. Moreover, the entire dataset each process operates on is copied to GPU memory only once before the training loop.

Figures 2, 3, 4 contain plots of the performance of implementation 1 vs. 2. It is generally visible that implementation 2 is faster when only one process is used. Implementation 1, however, performs increasingly better in comparison to implementation 2 the more processes are used. This makes sense, as the increased number of processes leads to more `MPI_Scatterv` commands in implementation 2, whereas the additional communication to split the dataset for implementation 1 only increases marginally.

5 Profiling

NVIDIA Nsight Systems and Nsight Compute are tools that provide profiling and debugging capabilities for optimizing CUDA applications and leveraging the full power of NVIDIA GPUs. Nsight Systems provides a system-wide performance analysis, enabling developers to understand and optimize the interaction between their application and the system. It generates a timeline that reveals how the application utilizes the GPU and CPU as well as memory resources. This is particularly useful for identifying bottlenecks and unoptimized parts in the code, such as unnecessary synchronizations, improper memory access patterns, or imbalanced usage of hardware resources.

In our case, we can use NVIDIA Nsight Systems in order to detect the operations whose performance has the biggest impact on overall performance. Table 6 contains the distribution of runtime between

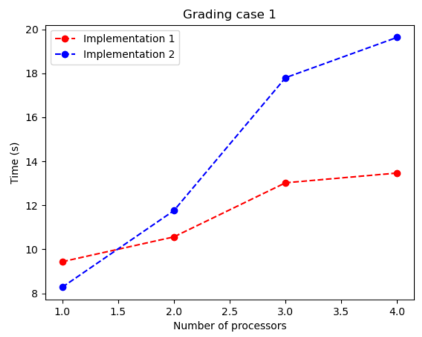


Figure 2: Performance of the different MPI implementations for 1, 2, 3 and 4 processes in grading case 1.

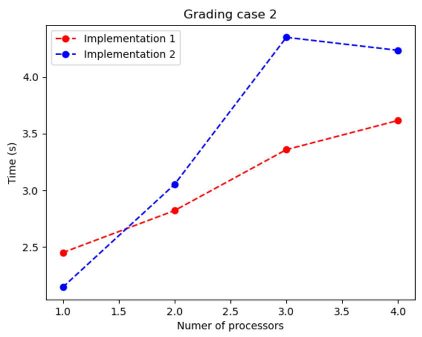


Figure 3: Performance of the different MPI implementations for 1, 2, 3 and 4 processes in grading case 2.

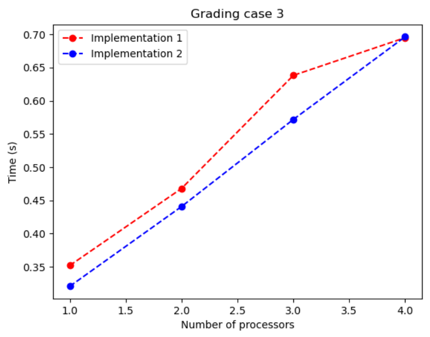


Figure 4: Performance of the different MPI implementations for 1, 2, 3 and 4 processes in grading case 3.

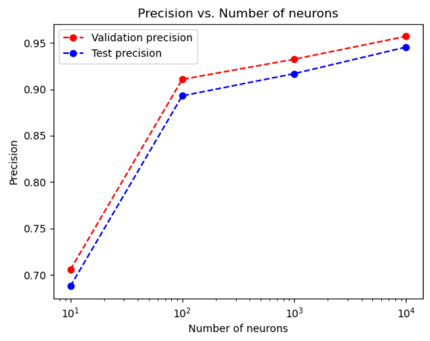


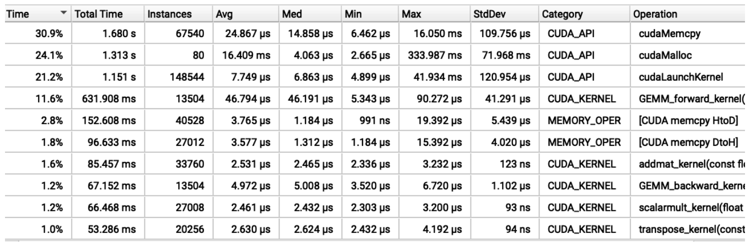
Figure 5: The plot shows the precision of several 2-layer dense networks after 100 epochs of training with a learning rate of $5 \cdot 10^{-4}$ for a varying number of neurons in the hidden layer.

different CUDA operations. As mentioned earlier, it is visible that copying data to the GPU and allocating GPU memory are the most time-consuming operations.

Nsight Compute, on the other hand, is a kernel profiling tool that provides detailed metrics and statistics for individual CUDA kernels. It helps to optimize kernel code by giving insights into the GPU's behavior, such as the utilization of the memory hierarchy, instruction throughput, or warp execution efficiency. Figure 7 shows a simple statistic of the average utilization of the streaming multiprocessors of the GPU. We can see that algorithm 1 for the GEMM utilizes the compute resources far less than algorithm 2 on the right. The profiler recognizes this and tells us that algorithm 1 suffers from latency issues, which is due to the access to noncontiguous global memory.

6 Results

As the goal of the neural network is to classify handwritten digits (MNIST), we ran 4 different experiments. All models were trained for 100 epochs with a learning rate of $5 \cdot 10^{-4}$, batchsize of 800 and a regularizer of 10^{-4}. Figure 5 shows the validation and test accuracy at the end of training. As expected, the bigger models are able to capture more nuanced features in the data in order to give a more informed prediction of the class label.



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
30.9%	1.680 s	67540	24.867 μs	14.858 μs	6.462 μs	16.050 ms	109.756 μs	CUDA_API	cudaMemcpy
24.1%	1.313 s	80	16.409 ms	4.063 μs	2.665 μs	333.987 ms	71.968 ms	CUDA_API	cudaMalloc
21.2%	1.151 s	148544	7.749 μs	6.863 μs	4.899 μs	41.934 ms	120.954 μs	CUDA_API	cudaLaunchKernel
11.6%	631.908 ms	13504	46.794 μs	46.191 μs	5.343 μs	90.272 μs	41.291 μs	CUDA_KERNEL	GEMM_forward_kernel(
2.8%	152.608 ms	40528	3.765 μs	1.184 μs	991 ns	19.392 μs	5.439 μs	MEMORY_OPER	[CUDA memcpy HtoD]
1.8%	96.633 ms	27012	3.577 μs	1.312 μs	1.184 μs	15.392 μs	4.020 μs	MEMORY_OPER	[CUDA memcpy DtoH]
1.6%	85.457 ms	33760	2.531 μs	2.465 μs	2.336 μs	3.232 μs	123 ns	CUDA_KERNEL	addmat_kernel(const fl
1.2%	67.152 ms	13504	4.972 μs	5.008 μs	3.520 μs	6.720 μs	1.102 μs	CUDA_KERNEL	GEMM_backward_kern
1.2%	66.468 ms	27008	2.461 μs	2.432 μs	2.303 μs	3.200 μs	93 ns	CUDA_KERNEL	scalarmult_kernel(float
1.0%	53.286 ms	20256	2.630 μs	2.624 μs	2.432 μs	4.192 μs	94 ns	CUDA_KERNEL	transpose_kernel(const

Figure 6: Table of the distribution of runtime among different CUDA functions as provided by Nsystems.

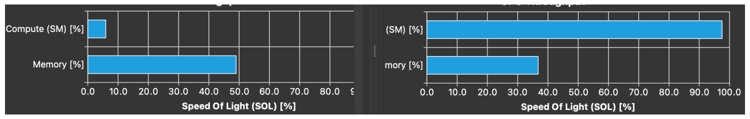


Figure 7: Utilization of theoretically available compute resources and memory for GEMM algorithm 1 on the left and algorithm 2 on the right. Provided by Ncompute.