Optimization of CNN Training in Parallel on GPUs

Taylor Geisler

1 Performance

1.1 GEMM Performance

	GEMM1	GEMM2
beginner_GEMM	0.5718 s	0.0103521 s
naive_advanced_GEMM	1.0723 s	0.00900078 s
advanced_GEMM	0.197803 s	0.00804114 s
CUBLAS	0.046957 S	0.0044651 s

I tested the three different cuda GEMM implementations shown in the previous table. The beginner_GEMM kernel assigns one thread to each element of the resulting matrix. To compute the row-column dot product, each thread reads all required elements from the global memory as needed inside of a for loop. Because information transfer is slow from global memory, and data reuse is not efficient here (different threads may read the same value at different times) this implementation is inefficient.

I next implemented a naive_advanced_GEMM kernel using shared memory. Each thread is again assigned to calculate one element in the resulting matrix. Blocks are 32 x 32 threads. I partition A and B into 32 x 32 blocks, load these blocks into the shared memory one at a time, and each thread performs a 32 element dot product. I stored A_local and B_local as 1D arrays. This code performed worse than my beginner_GEMM because the 1D array shared memory storage was inefficient in arranging memory required for the dot product subsequently in memory.

I fixed this problem in advanced_GEMM by storing the shard memory blocks as 2D arrays. This implementation is nearly 3x and 5x faster than the beginner kernel and naive kernel respectively due to fewer total memory reads (efficient memory ordering) and faster shared memory transfer speeds.

By running with the cuda-memcheck option enabled, I was able to further optimize my implementation. This option reported that I was performing memory reads from illegal memory (outside the range of A or B) that were never used in calculating the result and increased the speed of my GEMM by about 5%.

1.2 Overall NN Performance

	Serial	MPI (4 CPU)	MPI CUDA (4 CPU, 4 GPU)
Case 1	116.96 s	41.343 s	106.929 s
Case 2	26.0598 s	10.543 s	26.4188 s
Case 3	25.2335 s	9.834 s	2.89346 s

My code uses 4 MPI processes and 4 GPUs to parallelize the training of this neural network. By recognizing that each batch of images used in the training process can be divided into smaller independent operations I parallelized the training process with MPI. Each batch of images is read in to the master process and is MPI_scattered evenly among 4 processes. The corrections to the NN parameters are reduced to the master process and the updated NN is broadcast to all processes.

I performed a training case using only the MPI parallelism and the serial training functions and observed a 2-3x increase in training speed.

The feedforward and backprop funtions each perform multiple GEMM operations, and my idea was to improve performance by doing these operations on the GPU. Each MPI process is assigned a single GPU. I then wrote parallel versions of the required functions. I increased speed over the serial implementations by reducing the number of matrices that are initialized and copied.

Inside of parallel_feedforward and parallel_backprop I perform each GEMM operation by copying the required memory to the GPU, performing the GEMM, and copying the memory back to the CPU process.

The runtimes of this MPI CUDA process are no better than the serial case overall. I will show why in the following profiling section, and suggest ways to improve this.

2 Profiling

2.1 GEMM Profiling

Table: Grading case 4 profile

	beginner_GEMM	advanced_GEMM
cudaMalloc	149.5 ms	120.9 ms
cudaFree	210.4 ms	212.6 ms
kernel	636.4 ms	214.9 ms

The NVIDIA visual profiling tool results for total computation times per operation are shown above. Both GEMM implementations require the same time for memory allocation and freeing. The improvement is seen in the kernel execution times. There is a ~3x speedup of each kernel call due to reduced amount of total memory access and increased transfer bandwidth from shared memory.

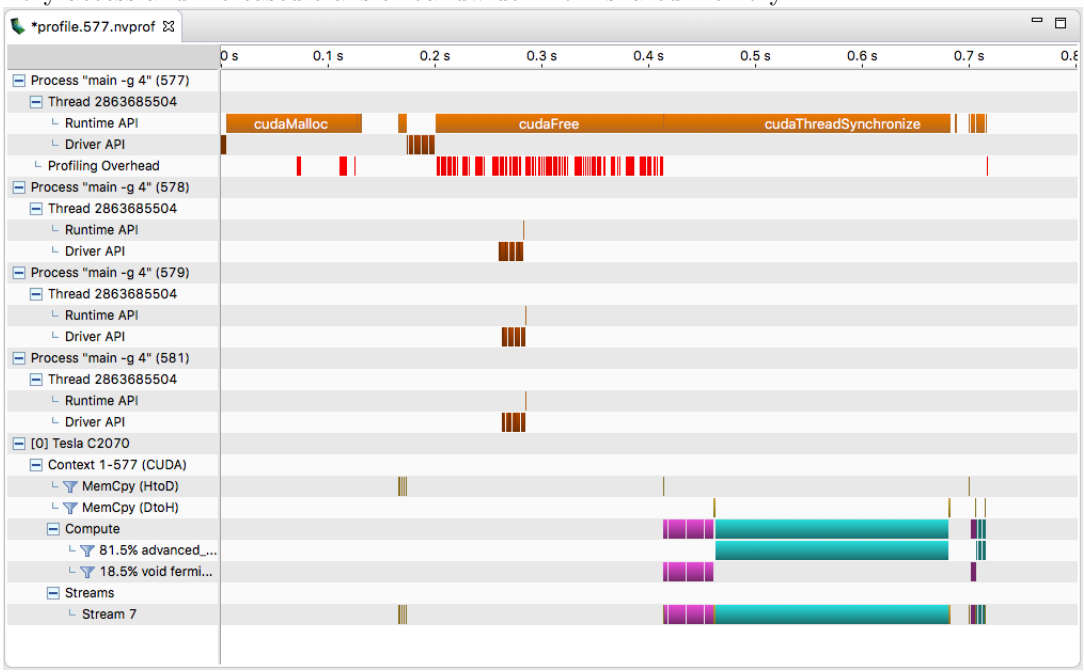

Figure 1: nvcc profile of advanced_GEMM showing faster kernel times and the high cost of cudaMalloc and cudaFree. This suggests that overhead to memory allocation to GPU can be more costly than matrix operation.

2.2 Overall NN Profiling

In my results section I showed that my MPI GPU implementation isn't much faster than the serial implementation. I didn't have time to completely rewrite my implementation to improve this, but here I will suggest reasons for this slowdown and ways to fix them.

Figure 1 shows that the GEMM operations speed can be dominated by memory management on the GPU. This suggests that the code runtime could be drastically improved by not allocating and freeing memory at each GEMM call. To fix this problem, I propose allocating a large block of memory on the GPU at the beginning of training, and reusing this memory with each matrix operation without cudaFree ing the memory until the training finishes. For a training with a significant number of GEMM operations, this would effectively eliminate cudaMalloc and cudaFree operations from overall runtime. Based on the above table I would expect over 2x speedup by doing this.

Table: Grading case 1 profile with advanced_GEMM.

Process	Current Implementation	Original Implementation
MemCpy (HtoD)	2.166 s	2.223 s
MemCpy (DtoH)	0.469 s	0.448 s
kernel	4.25 s	13.61 s
CPU Operations	219.85 s	269.83 s
total	226.8 s	286.1 s

I show here two stages of my code. The original implementation was correct, but used the beginner_GEMM kernel. It also used the feedforward and backprop functions taken directly from the serial implementation, but with GEMM operations replaced by a GPU GEMM.

From the previous table, it is again obvious that the bottleneck of my current run isn't the GEMM kernel. 97% of the simulation time is spent in CPU operations, and only 3% on GPU operations. For GPU operations, memory copies take less time than the kernel implementations.

The question is then to find out what is taking so long in the CPU operations. The profiling application shows that immense amounts of time are spent allocating and freeing memory. This is largely responsible for the huge CPU operation time.

My code performs three memory allocations memory frees every single time the GEMM function is called. Again, I plan to fix this by allocating the required GPU memory and leaving it allocated throughout the training. This could reduce the number of cudaMalloc and cudaFree calls by orders of magnitudes.

Another possibility would be to transfer the matrix information to the GPU at the beginning of a training step, perform all operations for training directly on the GPU, and then pass back to the CPU only at the end of the training step. This would reduce both the CPU operations time as well as the MemCpy time.

To summarize:

- Memory allocation and freeing are slowing the code considerably and should be reduced.
- Data transfer to/from GPU should only happen once per training step. Write all these functions directly on device.
- GEMM kernel can be improved further.
- MPI synchronization slows the process by requiring faster processes to wait for slower ones. Find way to have multiple streams to circumvent this requirement.