Concurrency Final Project: Machine Learning With CUDA

Kevin Black

May 6th, 2021

1 Overview

For my final project, I built a rudimentary GPU-accelerated machine learning library à la PyTorch or TensorFlow. It supports matrix multiplication, vector addition, ReLU, arg max, and cross-entropy loss: enough to implement a simple feedforward neural network. It also includes an automatic backpropagation engine that computes gradients. The final goal was to be able to train a small neural network on a toy dataset and achieve good performance. This would then allow me to compare train-time throughput and inference-time latency between my sequential implementation, my parallel implementation, and an existing production-grade machine learning library such as TensorFlow or PyTorch.

The entire project is written in CUDA C++ for NVIDIA GPUs and uses no external libraries (aside from the C++ standard library).

2 High-Level Design

One of the biggest challenges of this project was designing the high-level structure in which to place the various operations. I had to keep in mind from the get-go that it needed to support both GPU parallelization and backpropagation. Unlike a numerical computing library (e.g. NumPy) that just has to support forward propagation, a machine learning library has to automatically keep track of a "computational graph" in order to do backpropagation. Each operation needs to keep track of the operations that produced its operands, as well as cache the value of its result to be later used in the backward pass. This leads to a lot of difficult design decisions. How is the graph built and traversed? How does data enter the graph? When and how is memory allocated (and how does it know to be allocated on the CPU or GPU)? When are the operations actually executed?

For inspiration, I looked to the two most popular existing machine learning libraries: TensorFlow and PyTorch. They each have very different approaches. TensorFlow generally (pre-2.0) requires users to explicitly think about the computational graph. The graph is fully constructed and then executed in two separate steps, with data being fed through "placeholder" nodes at execution time. PyTorch, on the other hand, builds the graph on the fly while eagerly executing all of the operations. The graph is not explicitly re-used (except for optimizations under the hood) and is usually re-built on every forward pass.

The final design I landed on is sort of a hybrid of both approaches. It uses the "static graph" paradigm from TensorFlow with "placeholder" nodes; the graph is built once and then reused multiple times. However, there is no explicit conception of a "graph" exposed to the user. The graph is defined implicitly by the operations: the user invokes the execution of the graph using methods on individual operations, which then fetch their dependencies in a simple depth-first manner. Memory allocation is also done lazily: the space for outputs and gradients are allocated the first time they are needed, at which point the user has already specified if they want computation to run on the CPU or GPU.

This design decision was made primarily for ease of implementation. The dynamic graph-building of PyTorch makes efficient memory allocation very hard since a naive implementation without complex optimizations will waste tons of time on allocation every time it builds a graph. However, managing entire graphs explicitly like TensorFlow requires a bunch more code that knows how to store, allocate, and traverse a graph. With my design, each operation just needs to implement two public methods: `forward()` and `backward()`. The `forward()` method recursively fetches the operation's dependencies by calling `forward()`

on its operands, and then performs the operation. Similarly, the `backward()` method computes gradients for that operation, and then recursively calls `backward()` on its operands. Execution stops when it reaches a `Placeholder` node, for feeding in data, or a `Parameter` node, which stores network parameters.

3 Noteworthy Implementation Details

3.1 The Matrix Abstraction

Rather than implementing generalized multi-dimensional tensors, I decided to represent all data as a matrix, since that's the most I would need for my simple feedforward network. A `Matrix` instance stores `float` data in row-major order, has the ability to read and write data to/from the disk, and has the ability to move its data between the CPU and GPU. However, it does not actually allocate space for its data until it is asked to.

3.2 Persistent Data Storage

In order to validate the correctness of my implementation, I needed a way to transport data (training data and network parameters) to/from the disk. Rather than try and parse a generalized file format (like NumPy's `.npy`), I use my own very simple binary file format that stores a height and width followed by `float` data in row-major order.

3.3 Matrix Multiplication

My sequential implementation for matrix multiplication is done naively using the schoolbook algorithm. It is likely that using a hyper-optimized BLAS implementation would be much faster. Therefore, I am technically not following the principle of comparing to the fastest possible sequential algorithm. (Although my understanding of BLAS implementations is that they utilize parallel vector instructions to get at least some of their performance.)

My CUDA matrix multiplication implementation is also naive. Each thread block handles an $n \times n$ tile of the output matrix, where each thread handles a single element. Each thread performs the appropriate dot product in a loop.

Backpropagating gradients for matrix multiplication just involves more matrix multiplications on various transposed versions of the input matrices and the input gradient. Therefore, they use very similar kernels, the only difference being the indexing.

3.4 Other Operations

The remaining operations are: addition, element-wise ReLU, cross-entropy loss, and argmax. Cross-entropy loss is used only during training, and argmax is used only during evaluation (to make predictions and calculate accuracy). The common theme of their implementations is that they are not very general. For ease of implementation, I made them very specific to the exact use-case I had in mind, which is multi-class classification using a simple feedforward neural network. There are no generalized broadcasting rules: the only broadcast performed is that of the bias vector across a batch of input vectors, which is hard-coded. Thus, there is s strict requirement for the order of the operands and which dimension is the batch dimension. Gradients are automatically accumulated across the batch dimension.

Each operation has different levels of parallelization. Addition and ReLU are fully parallelized across both dimensions. Cross-entropy loss is only parallelized across the batch dimension since it must accumulate sums within each batch. Same with argmax since it must keep track of a maximum within each batch.

Parallelization differs in subtle ways when calculating gradients. Addition is no longer parallelized across the batch dimension, since it must accumulate a sum for each batch. Argmax has no gradient. Cross-entropy loss is fully parallelized when calculating the gradient, since I cache the accumulated sums during the forward pass. Parameter updates are also parallelized when GPU computation is enabled.

Like with matrix multiplication, a single thread is spawned for every element of every dimension which is parallelized. Each thread block handles an $n \times n$ tile (if two dimensions are parallelized) or an $n \times 1$ segment (if only one dimension is parallelized).

3.5 Placeholder Filling

Placeholder nodes are where new input data is fed to the graph before each invocation. I copy data into the placeholder nodes one batch at a time. When GPU computation is enabled, this copies data directly into GPU memory. With the datasets I use, the entire dataset could fit into GPU memory at once. However, I opted not to do this as it is not a realistic usage pattern.

4 Evaluation

All of my experiments are run on `pedagogical-2@cs.utexas.edu` which has a Quadro RTX 6000 GPU, an Intel Xeon Gold 6226R CPU, and is running Ubuntu 18.04. Timing data is not averaged over multiple runs of the program, since in all cases, important timings are inherently accumulated/averaged over many invocations of the neural network.

4.1 Machine Learning Details

For evaluation, I use classification on the Fashion-MNIST dataset as my reference problem. The Fashion-MNIST dataset consists of 28×28 grayscale images that each belong to one of 10 classes. The images are flattened to vectors of length 784 before being fed into the network. The network architecture I use is a simple 2-layer fully-connected neural network. Its sequence of operations is: multiplication by a 784×256 weight matrix, addition of a 256×1 bias vector, a ReLU nonlinearity, multiplication by a 256×10 weight matrix, and addition of a final 10×1 bias vector. Such a network easily reaches $\sim$87% accuracy on the Fashion-MNIST dataset. For time-saving purposes, all of my experiments use only the 10,000 image test set rather than the 60,000 image training set.

4.2 Correctness Evaluation

I used PyTorch to evaluate the correctness of my implementation. I first trained an identical network in PyTorch for 50 epochs and then exported the weights and biases to my specialized file format. I then verified that my implementation produced the same accuracy and loss, with both sequential and GPU computations, with varying batch sizes. Finally, I performed 10 epochs of stochastic gradient descent with both PyTorch and my implementation, starting from those same parameters, and made sure that they produced the same losses after every epoch.

4.3 Initial Timing Breakdown

First things first, a breakdown of the timings for both sequential and parallel execution in order to identify the primary bottlenecks. These timings were collected by running in train mode for 10 epochs with a batch size of 100, which captures many forward and backward passes. The tile size for all operations was 32 (i.e. $32 \times 32 = 1024$ threads per block for 2D kernels, 32 threads per block for 1D kernels).

Section	Sequential	Parallel
CUDA Runtime Init	-	1.59
Data Load Time	0.015	0.016
Memory Allocation Time	0.000	0.000
Memcpy Time	0.034	0.108
Matmul Forward Time	132	0.522
Add Forward Time	0.073	0.014
ReLU Forward Time	0.107	0.007
Cross-Entropy Forward Time	0.013	0.010
Matmul Backward Time	184	0.105
Add Backward Time	0.069	0.021
ReLU Backward Time	0.208	0.007
Cross-Entropy Backward Time	0.015	0.007
Parameter Step Time	0.705	0.029

Table 1: Timing breakdown. All times are in seconds, and refer to the total time spent (not per iteration) across all 10 epochs. Memory Allocation Time refers to both CPU and GPU memory allocation for the parallel case. Memcpy Time refers to time spent copying data into placeholder nodes, plus all time spent transferring data to/from the GPU in the parallel case.

As I discovered in our previous CUDA lab, simply initializing the CUDA runtime is quite expensive. Even across 10 epochs, it still makes up the majority of the end-to-end time. However, as I explained in the previous lab, this overhead will be ignored for analysis purposes. In any real latency-sensitive application, the process would be kept running so that the runtime remained initialized. In a real training workload, the process would run for hours or days, and any overhead on the order of seconds would become insignificant.

Aside from the CUDA initialization, we see that unsurprisingly, the matrix multiplications completely dominate both times. Right out of the gate, GPU parallelization provides a 504x speedup on the combined matrix multiplication sections. What is much more surprising is the relative timing of the forward and backward pass. The backward pass for matrix multiplication requires two full matrix multiplications, while the forward pass only requires one. In fact, the backward pass requires exactly twice as many floating point operations. And yet, neither the sequential nor parallel timings reflect this.

The sequential implementation does take 1.39x as much time for the backward passes. This number falling short of 2x likely comes from sort of cache optimally, as the two backward pass matrix multiplications are done one after another and reuse one of the matrices involved. However, the parallel implementation takes 4.97x *less* time for the backward pass. This was baffling to me, until I thought more carefully about parallel matrix multiplication.

The matrix multiplication kernel assigns one thread per element of the output matrix. Assuming perfect parallelization, the time complexity is thus determined by the dot product performed in each thread, which is done on the dimension that is equal between the two input matrices. The backward pass is helped even more in that it performs its two matrix multiplications in parallel, so again assuming perfect parallelization, its time complexity is determined by the maximum of the dot product dimension of the two matrix multiplications. For the first layer, the dot product dimension for the forward pass is 784, whereas the dot product dimensions for the two backward pass matrix multiplications are 256 and 100. For the second layer, it is 256 for the forward pass, and 100 and 10 for the backward pass. Thus, assuming perfect parallelization, we get a theoretical difference factor of $(784 + 256)/(256 + 100) = 2.92$. This *still* falls short of the 4.97x difference we see. However, it at least explains why we expect the backward pass to be significantly faster than the forward pass despite requiring more total operations. As in the sequential case, the rest of the difference could possibly be explained by cache optimally or differing memory bank conflicts caused by differing memory access strides.

4.4 Forward Inference Latency

Forward inference latency is the first timing scenario I will explore for machine learning models. This scenario corresponds to deploying machine learning in production: for example, on a self-driving car, or in a search engine. These scenarios are clearly latency-focused: how fast can the application return a single result? This also means that this scenario exhibits strong scaling, which is why we usually don't see super-powerful GPUs (or often GPUs at all): the latency is governed by Amdahl's law, and more parallelism yields severely diminishing returns.

To test forward inference, I perform evaluation of the pre-trained network with a batch size of 1. Since data is copied into the graph (and onto the GPU) one batch at a time, this simulates the network receiving one image at a time. This time ignores overhead such as loading weights, memory allocation, or CUDA runtime initialization, since in a real inference scenario these would all be done at application startup.

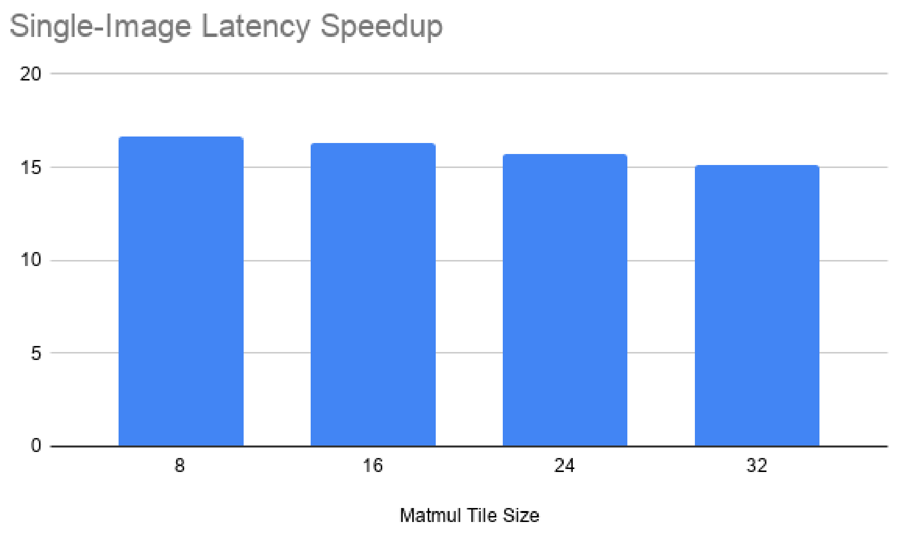

Figure 1: Speedup over sequential baseline for single-image latency. Matrix multiplication tile size is the only parameter varied, since matrix multiplication was found to be the primary bottleneck in section 4.3. The tile size is the side length of each tile in the output matrix that one thread block is responsible for (so the number of threads per block is the tile size squared). The sequential baseline took 1.31ms/image.

The GPU is able to provide a fairly respectable 15-16x speedup, even with the limited parallelism exposed by a single image. It seems that in this single-image case, smaller tile sizes perform slightly better, although it does not make a very big difference.

4.5 Training Throughput

Training throughput is the second timing scenario I will explore. This scenario is clearly throughput-focused: how many images per second can we train with? This scenario exhibits weak scaling, since we can increase the problem size (batch size) in an unbounded fashion. This is why we see increasingly more powerful GPUs, TPUs, and all sorts of massive clusters of parallel computing being used in industry.

To test training throughput, I start with arbitrary parameters and train for 10 epochs. The sequential baseline is run using a batch size of 10,000 (the whole dataset). This is done to simulate the fact that in reality, CPU-only training can support much larger batch sizes due to the abundance of memory. This way, we are comparing against the "best possible" sequential implementation. Again, I ignore constant initialization overhead, since in reality training goes on for many hours or days so this overhead becomes insignificant.

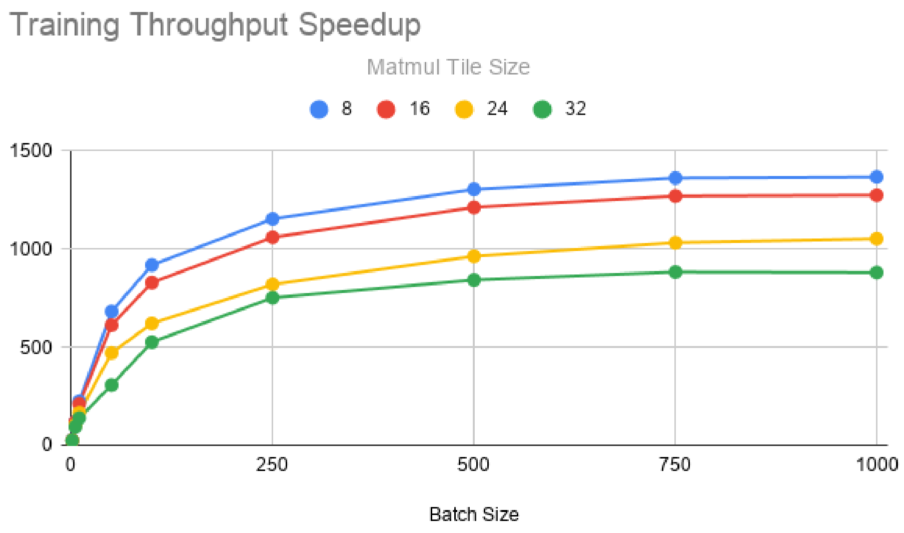

Figure 2: Speedup over sequential baseline for training throughput. The sequential baseline achieved 228 images per second, meaning the fastest parallel result (8 tile size, 1000 batch size) achieved 311,000 images per second.

The parallel implementation achieves a peak speedup of 1366x. Tile size makes a fairly significant difference compared to single-image latency, which makes sense. It follows the same trend as the single-image latency, where larger tile sizes are significantly worse, which is slightly surprising to me. I thought using the maximum number of threads per block ($32 \times 32 = 1024$) would be at least as efficient if not more efficient than using fewer threads per block. This was supported by my research: several instances of PyTorch CUDA code that I looked at used either the maximum possible number of threads per block or 1024 threads per block[1]. Some combination of the hardware parameters for the Quadro RTX 6000 must just cause smaller thread blocks to achieve better occupancy. This result illustrates the importance of benchmarking different launch parameters on your particular hardware.

The trend with respect to batch size is interesting. At smaller batch sizes (below 100), the growth is approximately linear as predicted by Gustafson's law. However, I am surprised at how quickly this growth caps out and starts to give very diminishing returns. I conjecture that the large size of matrix multiplications quickly maxes out the CUDA parallelism available on the Quadro RTX 6000. One of the matrix multiplication gradient computations for the first layer uses $784 \times 256 = 200,704$ threads alone, independent of the batch size. The other gradient computation grows as 784 times the batch size. I can see these numbers of threads in the hundreds of thousands giving diminishing returns in terms of parallelism.

4.6 Comparison To PyTorch

Finally, I compare my implementation to PyTorch in both single-image latency as well as training throughput. I run an identical training loop to my implementation, calling `backward()` and updating parameters manually. The data is stored in CPU memory and copied one batch at a time using `.cuda()`.

Firstly, for single-image latency, PyTorch achieves a speedup of 10.2x. A graph or table is omitted since this is only one number. This performance is worse than any of my configurations from section 4.4, in which a peak speedup of 16.6x was achieved. This result is not surprising; in fact, PyTorch does better than I

[1] An example from the AveragePool2D operation

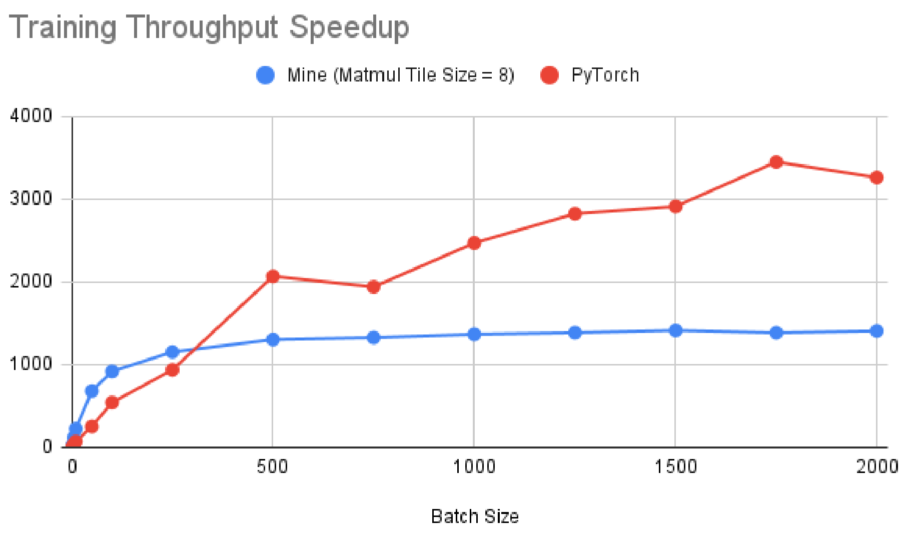

Figure 3: Speedup over my sequential baseline for my best parallel configuration (matmul tile size of 8) and GPU-enabled PyTorch. PyTorch achieves a peak speedup of 3448x, which is 785,500 images per second.

expected. For such small matrix multiplications, the implementation does not matter as much, and a real library like PyTorch has a lot more overhead than my bare-bones implementation. PyTorch in particular has never been known for (or really designed for) fast inference in production.

The same effect occurs for training throughput at lower batch sizes. Below a batch size of 250, the overheads outweigh the benefits and PyTorch performs worse than my implementation. However, it is right at a batch size of 250 that my performance pretty much levels out, while PyTorch keeps scaling up with a roughly linear trend. I am fairly certain that this performance gap comes down to the matrix multiplication and fundamental differences in the implementation (and even the hardware) used. Any real library like PyTorch will not write their own CUDA kernels for an operation as fundamental as matrix multiplication: they will call into cuBLAS, NVIDIA's own highly optimized implementation of basic linear algebra subroutines. Furthermore, cuBLAS will not use regular CUDA cores, but leverage the Quadro RTX 6000's 576 Tensor Cores: hardware specialized specifically for matrix multiplication. This is why PyTorch is able to keep up weak scaling for much longer, and my naive CUDA implementation really had no chance.

5 Limitations/Improvements

Obviously, my library has a lot of severe limitations compared to a general-purpose machine learning library. However, imitating a real machine learning library was never my goal. Here are a couple of directions that I think could be worthwhile from an educational standpoint.

5.1 Better Matmul Implementation

As I've established, my matrix multiplication implementation is fairly naive. The matrix multiplication algorithm is a perfect candidate for shared memory speedup, as well as other optimizations like minimizing bank conflicts. Based on my research, using shared memory to cache certain rows/columns wouldn't be too difficult and could provide at least an order of magnitude of speedup. Of course, as I discussed in section 4.6, using a bare CUDA kernel is a significant hardware limitation. I don't think calling directly into cuBLAS would be too interesting, since then I would be basically imitating what PyTorch does. However, learning how to use the CUDA runtime to directly utilize Tensor Cores would probably be very educational, and would give my implementation a fighting chance against real machine learning libraries.

5.2 Accumulation Operations

Another big inefficiency in my kernel implementations has to do with accumulation operations. Throughout section 3.4, I mention several times where I reduce parallelism to one dimension due to the need to accumulate sums (or a maximum) in the other dimension. In the cross-entropy loss forward pass, I actually need to accumulate sums across both dimensions, so I achieve accumulation in the batch dimension using `atomicAdd`. None of these approaches are optimal. Instead, parallelizing across both dimensions and then using a second pass with a parallel prefix sum algorithm would probably be more efficient. Even though these operations are not the bottleneck, I think this would cause a pretty big speedup in the addition backward pass, which may be enough to improve the overall runtime by a small but significant amount.

6 Conclusions

From a concurrency perspective, I think this project is very interesting in the way it perfectly demonstrates both strong scaling and weak scaling scenarios within a single paradigm. In the single-image latency strong scaling scenario, parallelization is still useful: a 16x speedup is nothing to scoff at. However, this project has given me a much deeper appreciation of just how powerful GPUs are as throughput-maximizing devices, and the reason they are such a key requirement to the deep learning revolution we have had in the past decade. Even my naive CUDA-based implementation of the numerical operations involved in neural network optimization provides an orders-of-magnitude speedup.

In particular, I now see how backpropagation is even more uniquely suited to GPU-level parallelization than I previously realized. The bottleneck on parallel matrix multiplication is fundamentally different than in the sequential case, which causes gradient computation to actually be faster than forward inference (in spite of requiring more total floating point operations) for many common neural network architectures.

Finally, I want to observe a funny bit of irony in the way that Tensor Cores are used to boost machine learning computations even further. GPUs started out as specialized pieces of hardware intended for one specific workload: graphics. Over time, people realized their usefulness for any highly parallel computations, and they became increasingly utilized for general-purpose processing. Now, they've become so ubiquitous for one particular workload – machine learning – that they're coming full circle, and once again becoming specialized pieces of hardware specifically for that workload.