Deep Networks on the GPU

Michael Harris, Jae Hyun Park, and Jeffrey Wang

December 11, 2009

1 Project Goals

The overall goal of this project was to implement various components of the Stanford Deep Network Library on graphics processing units (GPUs). Training neural networks is usually a long and tedious task, especially when there are many layers of weights; on a single CPU, training a convolutional neural network to get state-of-the-art results for a task such as MNIST can take hours or even days. Fortunately, most of the computations involved in training can be easily parallelized, which make GPUs the perfect candidate to speed them up. Recent research has shown that the use of GPUs can immensely reduce the overhead of training a large neural network [1], [2]. We are using GPUs to accelerate the matrix operations used in both forward and back propagation so that it is possible to train deep networks much more efficiently. The implementation uses NVIDIA's Compute Unified Device Architecture (CUDA), which supports fast parallel execution on GPU processors, and CUBLAS, a library written over CUDA that implements many matrix computations [3], [4]. We implemented this in a way so as to hide as much of the complexities of using the GPU as possible, e.g. by providing a clean interface that can seamlessly go from running on CPU to GPU as appropriate. Finally, we tested and benchmarked our results on image datasets like MNIST, and saw significant (up to 15x) speedup in training time. Recently, we integrated our work into the SDN library.

2 Features

We implemented a wrapper around CUDA and CUBLAS that provides a clean interface and integrates directly into the SDN library, a class called CudaMatrix (and the corresponding CudaVector). SDN currently uses the Eigen linear algebra package's matrix implementation. In order to get our code to compile, link, and work well with the SDN library, we had to implement many of Eigen's matrix features as well as some others. Here is a list of major features implemented in CudaMatrix:

- All matrix-matrix and matrix-scalar arithmetic operations
- Component-wise square, exponentiation, absolute value, hyperbolic tangent, and inverse functions
- Component-wise arithmetic operations and row-wise and column-wise sum
- Static initializers for matrices of all zeros, all ones, or all of a certain scalar value, and random matrices
- Block (submatrix) abstraction for operating on rectangular blocks of a matrix without operating on the whole matrix. Blocks support all of the operations that matrices as a whole support except for row-wise and column-wise reductions.
- Ability to initialize a matrix via a comma-separated list of values via the << operator
- GPU memory manager for improved allocation/deallocation speed and reduced fragmentation
- Lazy transpose and lazy transfer between CPU and GPU
- Padding of matrix dimensions to multiples of 64, which significantly improves performance
- Custom kernel for memset, as built-in memset is not parallelized

Most of the arithmetic operators use either a CUBLAS CUDA kernel, or a CUDA kernel of our own design derived from CUBLAS kernel code. The end result is an interface that is almost the same as Eigen's matrix interface, which is very Matlab-like, while still getting massive performance gains. This is great because our code plugs directly into the SDN library with literally a single change of a #define. It will be very convenient to continue writing code for the library using CudaMatrix in the current style and have the freedom to go back to Eigen if desired.

We discuss in more detail several of the more complex and essential features of the class.

2.1 Block Operations

Eigen implements an abstraction for operating on blocks (submatrices) of a matrix. Since this feature is used quite extensively in the SDN library, we also needed to implement this using CUDA kernels on the GPU. An important aspect of block operations is that the blocks are "virtual," that is they are never explicitly constructed until necessary (their representation is four integers specifying the block within a matrix). This makes operations such as copying from one block to another or adding blocks bypass matrix construction, instead just calling a kernel.

2.2 Lazy Transfer

Since GPU memory is not directly accessible from the CPU, it is necessary to maintain copies of the matrix both on the GPU and in CPU memory. Keeping both copies always synchronized, however, would incur a huge overhead as memory transfers from GPU to CPU are extremely expensive. Thus we use lazy transfers, i.e. only synchronizing when the copy being used is known to be dirty. While training a neural network, the computations are all done on the GPU while the copy of the matrix on the CPU is never updated; only when the weights need to be read (e.g. to evaluate the network) does the synchronization occur. The latter occurs rarely during the training process, only to produce some intermediate results.

2.3 GPU Memory Manager

We found that memory management in CUDA is somewhat of a mess. cudaMalloc (the GPU equivalent of malloc) returns pointers drawn primarily from a pool of 64KB pages, which it can only allocate discretely. We observed that if we allocate a large number of 4KB arrays, the memory used will be approximately 16 times as much as we expect, as 64KB instead of 4KB are being used for each allocation. This is a big problem because the training data consists primarily of small matrices so this wasteful allocation often results in running out of GPU memory, which is a very limited resource.

To resolve this issue, we wrote our own memory manager for GPU memory. All of our allocations pass through the memory manager, which implements two major pieces of functionality. Firstly, it caches allocated arrays; because our matrix implementation generates a temporary result for essentially every operation, arrays are allocated and freed very often. This is expensive, but since the arrays are almost always the same size, caching them works very well. Secondly, the memory manager treats small allocations specially. It uses cudaMalloc to allocate 64KB pages, but returns pointers within those pages to represent smaller arrays, so that in a single 64KB page we can hold sixteen 4KB arrays instead of just one. The memory manager greatly improved the memory performance of the matrix class.

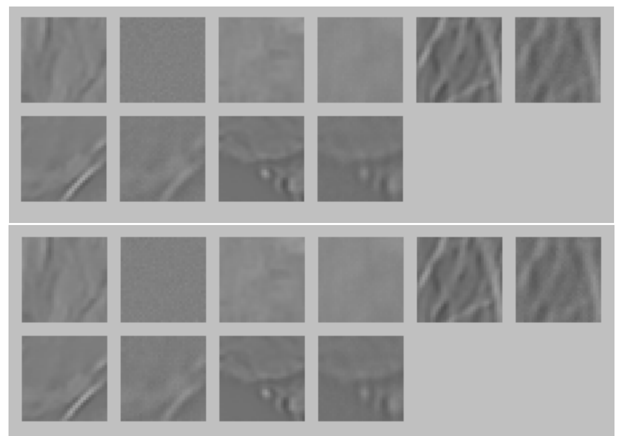

Figure 1: CudaMatrix reconstructions (top) and Eigen reconstructions (bottom)

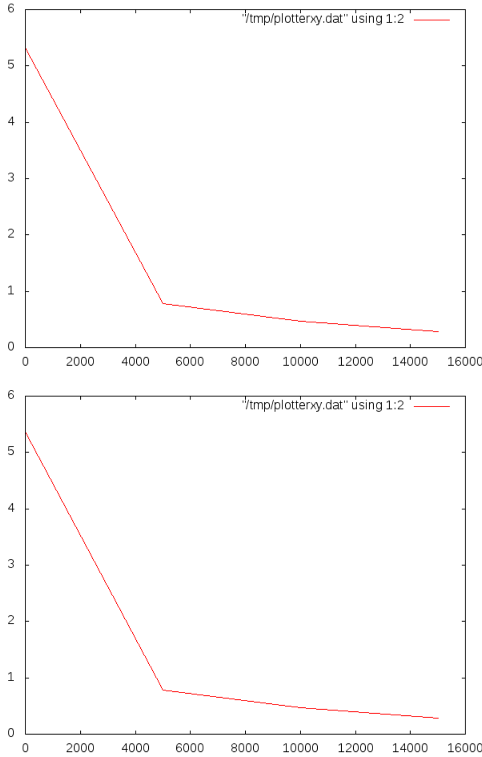

Figure 2: CudaMatrix training error (top) and Eigen training error (bottom)

3 Experiments

We ran three different sets of experiments using the SDN library in order to test both the correctness and performance of our matrix class. In each of these we output both reconstructions and training error every 5000 iterations. The goal is to see how CudaMatrix performs against Eigen in a variety of different settings.

- Training a 1-layer autoencoder on mnist (28x28) data using 20000 iterations with a variable number of hidden units
- Training a 2-layer autoencoder on mnist (28x28) data using 20000 iterations for each layer with a variable number of hidden units
- Training a 1-layer autoencoder on ng_video data using 20000 iterations, 4000 hidden units, and variable input size

3.1 Correctness

In each of our experiments, we found the reconstructions and training error to be almost identical every 5000 iterations. Figure 1 shows the reconstructions of a 1-layer autoencoder on mnist data with 1000 hidden units for both CudaMatrix and Eigen. Figure 2 shows the training error for the same experiment. We verified that the results are the same for CudaMatrix and Eigen in each of the other experiments as well.

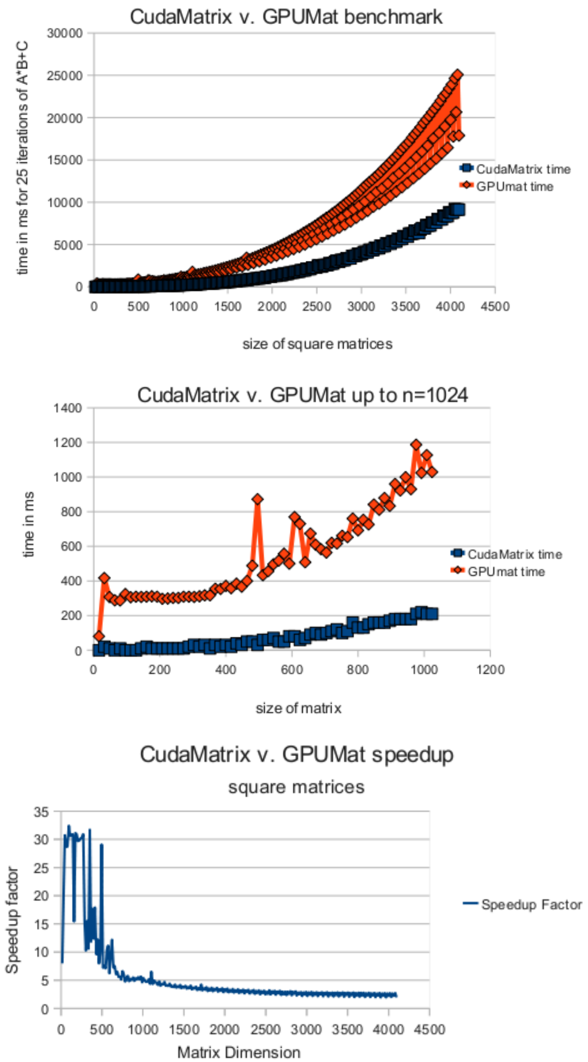

Figure 3: Computation times for CudaMatrix and GPUmat up to 4096x4096 matrices (top) and up to 1024x1024 matrices (middle); and speedup factor (bottom)

3.2 Benchmarks/Results

All of our benchmarks were performed on Michael's desktop, which has a quad-core 2.27GHz Intel Xeon E5520 CPU and an Nvidia GTX 275 GPU. GPU performance should be much higher still when moving to the Stanford lab computers, which have dual Nvidia GTX 285 GPUs.

Our first step was to benchmark CudaMatrix against GPUmat, which is a very nice CUDA library for MATLAB that mostly provides the interface of our CudaMatrix. We found that CudaMatrix outperforms GPUmat in doing matrix multiplies and adds by a factor of ≈4x for large matrices and much larger factors for matrices smaller than 1024x1024. Graphs of this performance increase are shown in Figure 3. The graphs show some noise, but the trends are clear.

We also noticed while benchmarking the performance of our CudaMatrix class that matrix operations are much faster (≈2x) when matrices are padded to dimensions that are multiples of 64 floats. The overhead of computations, especially memory transfer, is too large for this to be effective for very small matrices, but we do pad all matrix dimensions of size at least 65 to the next multiple of 64, and we have noticed significant performance improvements (again, ≈2x) from doing so. This has complicated the internals of CudaMatrix significantly, but the performance gain is worth it, and the complications are hidden by our interface.

Figure 4 shows the speedup of CudaMatrix over Eigen for the 1-layer and 2-layer autoencoders using mnist data as a function of the number of hidden units. The speedup improves dramatically for larger networks, reaching about 15x for networks with 4000 hidden units. Figure 5 shows the speedup for the ng_video data as a function of input size, showing speedups similar to the mnist data. Figure 6 shows the raw speedup of CudaMatrix arithmetic operations over Eigen arithmetic operations. The speedup again improves as a function of the matrix dimensions, with a speedup of over 100x for 4000x4000 matrices.

Clearly this is a huge win for the library, as training of large networks that used to take days will now only take a few hours.

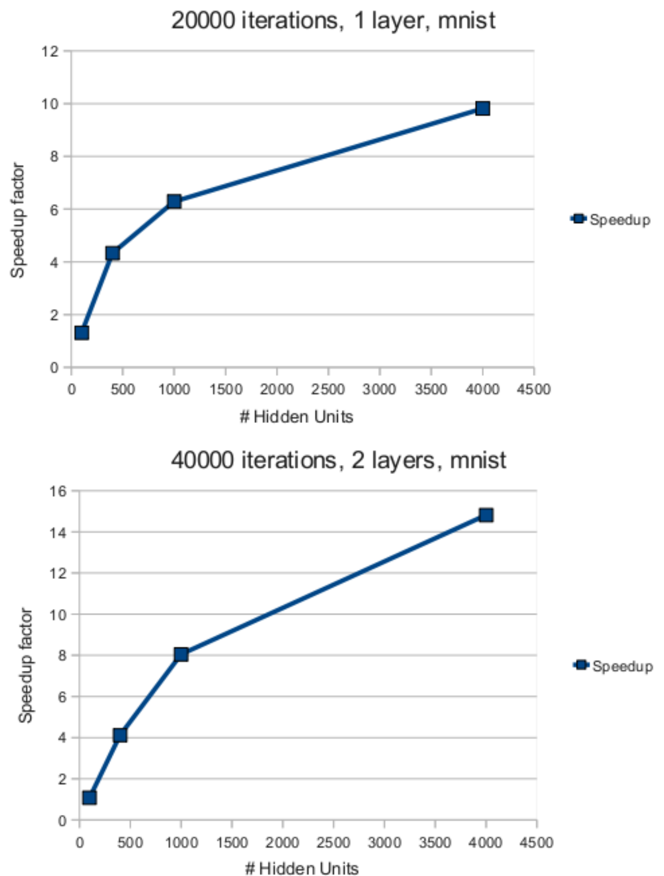

Figure 4: Training autoencoders on mnist

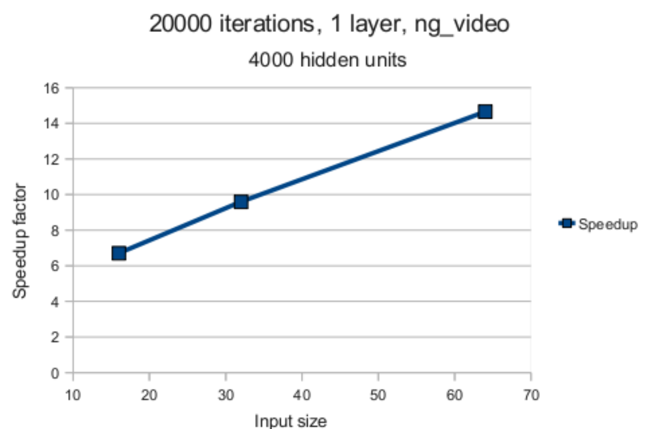

Figure 5: Training autoencoders on ng_video

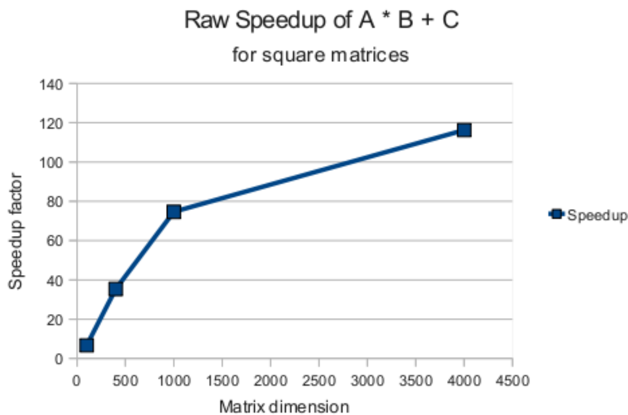

Figure 6: Raw speedup of CudaMatrix over Eigen

4 Future Work and Conclusions

As we integrated CudaMatrix into the SDN library, we moved some of the computations done in the library into CudaMatrix. For example, in one part of the library, tanh is called on each element of a matrix in turn in a tight loop, which is extremely slow because it is not parallelized. This could be implemented very efficiently in a CUDA kernel, however, so we moved that code into CudaMatrix with a component-wise tanh function. This helped improve the speed of training significantly, but there is certainly more of this kind of work to be done. With a raw speedup over Eigen of 100x, we would expect to get much better speedup while training a network. Currently, the SDN codebase uses a lot of Eigen functions and is not optimized for CudaMatrix. If we spent some more time on this, we expect that we could improve performance significantly over the current speedup.

We found in this project that, as research has indicated, moving matrix computations from the CPU to the GPU greatly speeds up the training and testing of deep networks. Our CudaMatrix class gets up to a 15x performance improvement in training networks over Eigen and still has room for more improvements. It should also be noted that CudaMatrix is a general purpose matrix class that could be used for many other applications both in machine learning and elsewhere. Any application that makes heavy use of matrix computations and requires a clean matrix abstraction could benefit from using CudaMatrix.

5 Acknowledgements

Thanks to Ian and Quoc for advising us on the project and to Paul for providing us with a great profiler for CUDA kernels that allowed us to very quickly identify performance issues in our code. Thanks also to Professor Ng for teaching CS 221 and CS 229 and for the opportunity to work on such an awesome project.

References

[1] D. L. Ly, V. Paprotski, and D. Yen. Neural Networks on GPUs: Restricted Boltzmann Machines. Department of Electrical and Computer Engineering, University of Toronto, 2009.

[2] R. Raina, A. Madhavan, and A. Y. Ng. Large-scale Deep Unsupervised Learning using Graphics Processors. *Proceedings of the 26th International Conference on Machine Learning*, 2009.

[3] *CUBLAS Library.* NVIDIA Corporation. Available at http://www.nvidia.com/object/cuda_develop.html.

[4] *CUDA Programming Guide.* NVIDIA Corporation. Available at http://www.nvidia.com/object/cuda_develop.html