Neural Networks on GPUs: Restricted Boltzmann Machines

Daniel L. Ly (994068682), Volodymyr Paprotski (992912919), Danny Yen (992903453)
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, Canada M5S 3G4
lyd@eecg.toronto.edu, paprots@gmail.com, danny.yen@utoronto.ca

ABSTRACT

Despite the popularity and success of neural networks in research, the number of resulting commercial or industrial applications have been limited. A primary cause of this lack of adoption is due to the fact that neural networks are usually implemented as software running on general-purpose processors. In this paper, we investigate how GPUs can be used to take advantage of the inherent parallelism in neural networks to provide a better implementation in terms of performance. We will focus on the Restricted Boltzmann machine, a popular type of neural network. The algorithm is tested on a NVIDIA GTX280 GPU, resulting in a computational speed of 672 million connections-per-second and a speed-up of 66-fold over an optimized C++ program running on a 2.83GHz Intel processor.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Base Systems

General Terms

Design, Performance

Keywords

Restricted Boltzmann machines, GPU applications, CUDA, high-performance computing

1. INTRODUCTION

There is a growing interest for large, high-performance neural networks. The capabilities of a neural network are highly dependent on its size; this raises a computational barrier since the complexity of software implementations grows quadratically with respect to network size. As a result, training large networks for real-world applications often takes weeks on general-purpose processors. It should be noted that neural networks are composed of an interconnected network of independent processing elements, and thus, are intrinsically parallel. A GPU implementation can achieve superior performance by taking advantage of this parallelism.

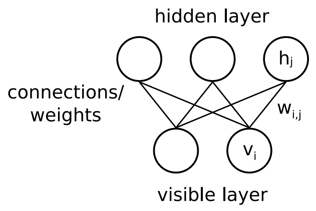

Figure 1: A schematic diagram of a Restricted Boltzmann Machine with labelled components.

Of the many neural network varieties, our work focuses on the Restricted Boltzmann Machine (RBM). It is a stochastic and generative network that is capable of capturing and reproducing the underlying statistical properties of a given data set. These unique properties have resulted in a variety of successful applications ranging from recognizing handwritten digits to reducing the dimensionality of data.

A RBM consists of two layers of processing elements, or *nodes*, the *visible layer* and the *hidden layer*. The visible layer is used for input/output access while the hidden layer acts as a latent representation of the data. The nodes have binary states. There are *connections* between every node in opposite layers and no connections between any nodes in the same layer. Every connection has an associated *weight*, which provides the learning parameters for the RBM.

The following notation system will be used: v_i and h_j are the binary states of the ith and jth node in the visible and hidden layer, respectively; $w_{i,j}[k]$ is the weight between the ith and jth node for the kth update. The terminology is summarized in a schematic representation in Fig. 1.

For brevity, matrix expressions are often used to represent the note states and weights, as shown in Eqs. 1-3.

$$\mathbf{v} = [v_0 \ldots v_{i-1}] \in \mathbb{B}^{1\times i} \tag{1}$$

$$\mathbf{h} = [h_0 \ldots h_{j-1}] \in \mathbb{B}^{1\times j} \tag{2}$$

$$\mathbf{W}[k] = \begin{bmatrix} w_{0,0}[k] & \cdots & w_{0,j-1}[k] \\ \vdots & \ddots & \vdots \\ w_{i-1,0}[k] & \cdots & w_{i-1,j-1}[k] \end{bmatrix} \in \mathbb{R}^{i\times j} \tag{3}$$

The RBM operates and learns via a method called Alternating Gibbs Sampling (AGS). AGS determines the node states of one layer given the other layer. The process starts with an initial data vector in the visible layer, and generates each layer in an alternating fashion. To differentiate between the successive AGS cycles, each state will be indexed with a superscript, x. To determine the node states, an intermediate value, called the *partial energy* for each layer must be calculated, $\mathbf{E_v}$ and $\mathbf{E_h}$ respectively. The AGS computations are summarized in Eqs. 4-8.

$$\mathbf{V}^{x+1} = \begin{cases} \mathbf{V}^0 & , \quad x = 0 \\ f(\mathbf{E}_\mathbf{v}^x) & , \quad x \text{ is odd} \\ \mathbf{V}^x & , \quad x \text{ is even} \end{cases} \tag{4}$$

$$\mathbf{H}^{x+1} = \begin{cases} f(\mathbf{E}_\mathbf{h}^x) & , \quad x \text{ is even} \\ \mathbf{H}^x & , \quad x \text{ is odd} \end{cases} \tag{5}$$

$$\mathbf{E}_\mathbf{v}^x = (\mathbf{H}^x)\mathbf{W}^\mathrm{T}, \in \mathbb{R}^{1\times i} \tag{6}$$

$$\mathbf{E}_\mathbf{h}^x = (\mathbf{V}^x)\mathbf{W}, \in \mathbb{R}^{1\times j} \tag{7}$$

$$\mathbf{W}[k+1] = \mathbf{W}[k] + \epsilon\left((\mathbf{V}^1)^\mathrm{T}\mathbf{H}^1 - (\mathbf{V}^X)^\mathrm{T}(\mathbf{H}^X)\right) \tag{8}$$

Where $x = \{0, \ldots, X\}$ and $f(\cdot)$ is a sampled sigmoid probability distribution applied to each element in the partial energy vector (Eqs. 9-10).

$$\mathrm{P}(v_i = 1) = \frac{1}{1 + e^{-E_i}} \tag{9}$$

$$\mathrm{P}(h_j = 1) = \frac{1}{1 + e^{-E_j}} \tag{10}$$

Additional details regarding the operation and learning of RBMs can be found in [1].

2. RELATED WORK

Due to the parallel nature of neural networks, CUDA programming is a very attractive method for performance gain. Due to the increasing popularity of RBMs, [2] investigates how to implement large RBMs using graphic processors. However, since these results have not yet been published, we cannot compare our implementations.

Next, there have numerous neural network implementations of different architectures than RBMs. Researchers from Soongsil University in Korea used a combination of CUDA and OpenMP in their attempt to speedup their feedforward neural network [3]. While they reported 20-fold speedup with CUDA and OpenMP over CPU-only program, they also claimed it to also have 5-fold speedup over a GPU-only implementation. This may be stem from the fact their neural network required image processing that does not exist on ours. They claimed that such computation is slower on the GPU. It would seem that CUDA may not be ideal for sophisticated processing, but more for massive simple calculations.

There has also been work on accelerating RBM via FPGAs. [4] reports a FPGA implementation for RBMs that can achieve a computational speed of 1.02 billion connection-updates-per-second and a speed-up of 35-fold over an optimized C program running on a 2.8GHz Intel Processor. However, their work is limited by the possible network size, and reported a maximum size of 128 × 128. Furthermore, there are architecture limitations that are not present in GPU implementations.

3. CUDA IMPLEMENTATION

After some analysis of the AGS algorithm, three major classifications of computational kernels were identified: matrix operations, random number generation and sigmoid transfer function. These three kernels were generated and tested individually and then integrated at a final stage.

3.1 Matrix Operations

As noted by Eqs. 4-8, the computation required to run an RBM is dominated by three types of matrix operations: matrix addition, matrix transpose, and matrix multiplication. Furthermore, since the type of operations is known and limited, several of the operations could be joined together to increase performance; for example, Eq. 6 requires both a matrix transpose and a matrix multiply, which can be combined into a single operation to increase the total computational throughput.

The library for these matrix operations were inspired by the NVIDIA CUDA SDK [5]. The SDK provided an optimized implementation that effectively used the shared memory and coalesced memory calls to maximize computation throughput. However, because the application was very specific and the implementation was designed for a designated GPU, the libraries were further optimized. For example, the original matrix multiply required block sizes to be square which did not utilize the maximum number of threads. This can be supplemented by extending the kernel such that the block size could be rectangular. The functionality of the entire matrix library was verified against a sequential implementation and extra care was taken to ensure that all the memory calls were coalesced and there were no bank conflicts.

A short description of each of the implementations for the matrix operations are provided:

Matrix Addition This operation was used in the calculation of weight updates (Eq. 8). As a result, it is not a matrix addition in the strict sense, but rather a matrix addition followed by a scalar multiplication. Each thread retrieved an element from each of the matrices, and added them together. Since this can be done element-wise, the implementation was straightforward and coalescing memory calls was trivial.

Matrix Transpose This operation was used in the calculation of the energies (Eq. 6) and the weight updates (Eq. 6). The implementation of the transpose was non-trivial since the structure of the matrix required clever use of the GPU architecture. Rather than attempting to transpose the entire matrix in one shot, the matrix is divided into submatrices according to the block size. As a result, each submatrix can be effectively read from global memory and copied into the shared memory, ensuring coalescing memory calls. The matrix can then be transposed and written to the appropriate memory location of the resulting matrix.

Matrix Multiply This operation was used in the calculation of the energies (Eqs. 6-7) and the weight updates (Eq. 6). The implementation of the multiplication was non-trivial since the structure of the matrix required clever use of the GPU architecture. Each thread was responsible from copying a single element from each matrix in the global memory to shared memory. Since both submatrices exists in shared memory, a for loop was used to do the multiply and accumulate required in matrix calculations. Each thread is responsible for keeping a running tally for the final matrix. The thread blocks are then moved and this is repeated until the final values are computed.

3.2 Random Number Generation

Random number are required to generate the node states (Eq. 9-10). For the random number generator (RNG), a method called Mersenne Twister (MT) was used. It is based on the example in the SDK. There are numerous advantages to using the MT:

- Relies on bitwise operations – each of the stream processors on the GPU has fast bitwise operations units in their ALU, allowing for an effective implementation
- Long period – the period of the MT is $(2^{19937} - 1)$ which provides a plenty of random numbers
- High dimensional equidistribution – MT provides high quality random numbers with little autocorrelation. This is necessary for the neural network to work since correlated random numbers would greatly affect the results
- Efficient memory usage – memory read and writes can be easily coalesced

However, MT is insecure: predictable after N outputs. This is not a big issue for RBM implementations since it is not being used for security-critical purposes. Moreover, like most RNG, it requires an iterative process, which makes it hard to parallelize. A simple and fast solution to parallelization is to launch many MT simultaneously, one per thread. Each MT is completely characterized by 11 parameters.

Unfortunately, even with completely different initial states, MTs with the same parameters cannot be guaranteed to generate random numbers that are uncorrelated. To rectify this issue, we use a custom library called *dcmt0.4*. This library, called upon initialization of RNG, tweaks a few parameters on a per-thread basis, using the thread ID as a generator. This initialization is done on the CPU and can be time consuming depending on the number of MT desired. To optimize our RNG, we need to adjust 3 values: number of MTs, blocks, and threads. Number of blocks and threads multiplies to the number of MTs. Oddly enough, it was not necessarily the more threads, the better performance.

3.3 Sigmoid Transfer Function

The RBM requires an efficient sigmoid function implementation (Eq. 9-10). The sigmoid calculation must be performed for each node in the particular layer being calculated, therefore needs to be as efficient as possible.

A sigmoid function, as described in (Eq. 9-10), is plotted in Fig. 2. Its range is (0,1) and quite obviously needs floating point numbers for the most precise implementation. The use of exponential function further complicates the implementation as most floating point units either does not exist or is expensive. As the implementation deals with floating point numbers, one is left with a design decision whether accept the possibly more expensive floating points or approximate with integer fixed point notation.

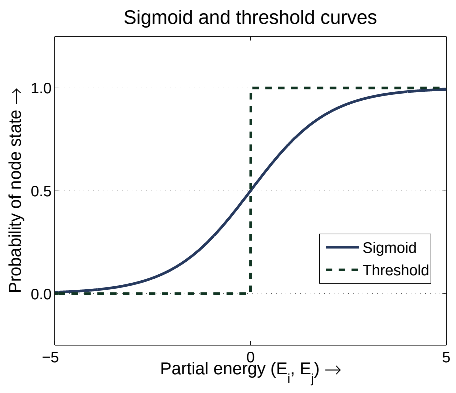

Figure 2: A plot of a sigmoid function.

The first option is relatively simple; implementation details will be discussed in the next section. The integer approximation on other hand can quickly devolve into a large problem with many parameters to choose from. However, such functions have already been extensively studied on platforms where floating point numbers are not available natively [6]. There are several algorithms to choose from, ranging in error and number of computations needed.

3.3.1 Sigmoid Functions implemented for this project

The first and simplest function, referred to as *native*, uses the CUDA provided exponential function call:

```
return 1/(1+__expf(-x));
```

Note the use of the '__' version of the *exp()* function. According the the Appendix B.2 from the CUDA Programming Guide [5], *"these functions are the less accurate, but faster versions"* of the non-underscored calls. This most probably implies that all SPUs will be capable of doing the calculation. This accuracy loss is certainly acceptable trade to gain more performance.

The first integer approximation implementation, referred to as *linear*, is simply a linear, piecewise approximation. It contains nine linear pieces. The code assigning the slope and offset uses is divergent, but still more efficient then it 'collapsed' counterpart. Slopes are all calculated as shifts. Input and output was to be floating point numbers, so cast both ways consumed more clock cycles. This could be improved with tighter implementation of code.

The second integer approximation, referred to as *second*, is a second order Taylor polynomial expansion, clamped at [-4.4] with two pieces about the origin:

$$y = \begin{cases} \frac{(x/4+1)^2}{2} & , \quad x < 0 \\ \frac{1-(1-x/4)^2}{2} & , \quad x \geq 0 \end{cases} \tag{11}$$

Both clamp code and parts of the actual calculation could be

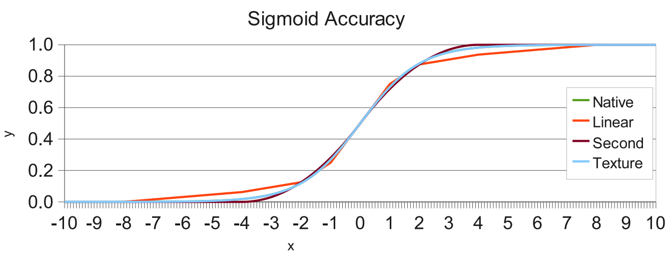

Figure 3: A plot of the sigmoid reconstruction of the various CUDA implementations.

further optimized in assembly with predicated assignments, which upon further investigation of PTX assembly, the compiler does not do, even with highest optimizations enabled.

Lastly, to reuse and explore the CUDA optimized hardware, a texture lookup sigmoid was implemented (*texture*). The values were precalculated on the CPU and loaded onto the texture. The result range was normalized to automatically by GPU to [0,1] and the input was clamped to [-10, 10], also by the GPU, producing a promising one liner:

```
return tex1D(tex, x*scale + offset);
```

The texture address calculation could further be optimized in assembly to multiply-add – this would require 4 clock cycles and is not currently done. The advantage of textures, is that they are cached, interpolated and auto-clamped. As the access pattern is unknown, cache is very important to aid optimization.

A plot of the sigmoid reconstruction of these four methods are shown in Fig. 3.

4. RESULTS

4.1 Platform

The software baseline benchmark was written in C++. It is compiled with g++ version 4.1.2 with optimization level 3 (-O3). An Intel Core2 Quad core processor running Debian at 2.83GHz with 4GB of DDR2 RAM is the baseline machine.

The software baseline benchmark was used as a shell with each of the kernels replaced with their respective CUDA implementation. The GTX280 NVIDIA graphics card was used which has 240 processor cores and runs at 1.3GHz with 1GB of memory.

4.2 Metrics

For comparing two different implementations of the same architecture, the *update period* is a simple and effective metric. The update period is the time it takes for the implementation to complete a single batch of data. The *speed-up* will be measured by the ratio described in Eq. 12, where S is the speed-up, and T_{hw} and T_{sw} are the update periods for the hardware and software implementations, respectively.

$$S = \frac{T_{sw}}{T_{hw}} \qquad (12)$$

However, an absolute measure of performance is also desirable. Although it is unable to account for the differences in neural network architectures, a common metric for computational performance is the number of Connections per Seconds (CPS) that an implementation can compute [7] – described by Eq. 13, where n is the layer size count and T is the update period.

$$\text{CPS} = \frac{n^2}{T} \qquad (13)$$

For the software program, the function *gettimeofday()* in the standard C *time.h* library is used to time stamp the software implementation at the beginning and end of every batch.

The error in weights was not measured since two different random number generators were used for each implementation, which would result in divergent weight updates making their comparison impossible.

4.3 Analysis

To make our comparison of GPU vs. CPU, we devised a RBM with the following properties:

- 512 nodes in visible layer
- 512 nodes in hidden layer
- 256k single-precision floating point weights

In our testing (Fig. 4-5), we had discovered CUDA implementation best CPU implementation for every component and fully integrated RBM. First, we will analyze the performance of individual components. The matrix operations provided varying levels of speedup with the matrix multiply providing the greatest performance. For the random number generator, increased performance is achieved by generating more numbers – this should be expected since the overhead of running *dcmt0.4* is amortized over a greater portion.

As for the sigmoid functions, the CUDA __expf() was the best performer, though only slightly outperforming the integer approximations. It is also the most precise out of the four implemented versions. Note that this function is in itself an approximation and not calculated by a FPU directly, as specified by Appendix B.2 of the CUDA Manual.

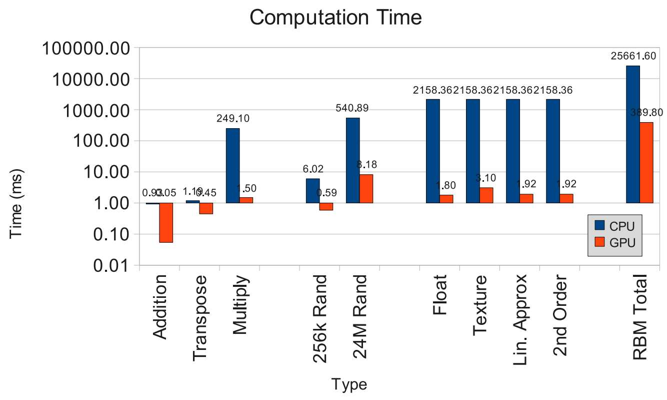

Figure 4: The computational time for each of the kernels.

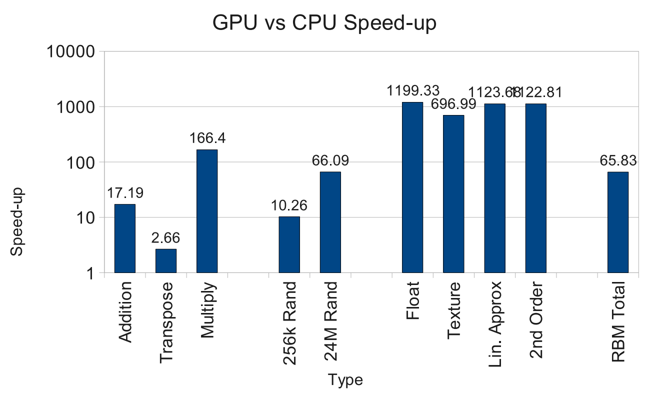

Figure 5: A speedup for each of the kernels.

Both integer implementations were essentially the same, with the decision coming from what is preferred: smaller error or larger range. It is not entirely fair to compare them to __expf(), as the later was optimized at assembly level. Significant performance improvements could be gained if these two functions were written in assembly themselves. Of interest, note that all operation can be easily and cheaply implemented and optimized in hardware; CUDA cannot achieve such custom integration.

Lastly, the texture implementation is the slowest. The error is negligible for our application. The performance loss is attributed to the memory access time. Nevertheless, while textures were slower in this context, their performance is still comparable.

For our integration code, we chose to use the pure floating point implementation.

It is worth to note that, while component-wise speedup was up to thousands of times faster than CPU implementations, the RBM only gained approximately 66-fold speedup, resulting in a computational speed of 672MCUPS. It showed that significant overhead had gone into the memory copies and synchronization of threads.

We also took a look at the scalability of GPU designs (Fig. 6). The computational time of the same CUDA program, at half the dimensions, showed 1.67 times speedup. While, by doubling the dimensions (quadruple the size), the computational time had increased five folds. This led us to believe that our CUDA implementation may not be capable of being efficiently scaled. However, we cannot rule out the fact we had coded our program for above listed properties for comparison purposes. Thus, it is entirely possible that our program can be further optimized for more dynamic dimensions.

5. CONCLUSIONS

This project showed that CUDA implementations can be well suited for neural network applications. The implementation was built around the design of three computational kernels: matrix operations, random number generators and sigmoid functions. Each of these kernels provided significant performance benefit. A speed up of 66-fold that achieved a computational speedup of 672MCUPS for a network size of

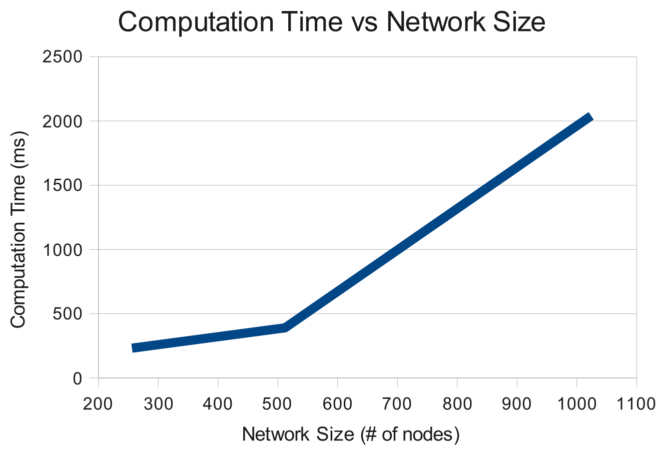

Figure 6: The computational time vs network size.

512 × 512 is significant and comparable to other work done in this field. However, further analysis suggests that this GPU implementation might not be infinitely scalable.

6. REFERENCES

[1] Y. Freund and D. Haussler, “Unsupervised Learning of Distributions on Binary Vectors Using Two Layer Networks,” *Neural Information Processing Systems Conference (NIPS)*, pp. 912–919, 1992.

[2] R. Raina, A. Madhavan, and A. Y. Ng, “Large-scale Deep Unsupervised Learning using Graphics Processors,” *Proceedings of the Twenty-Sixth International Conference on Machine Learning*, 2009. To appear.

[3] H. H. Jang, A. J. Park, and K. C. Jung, “Neural Network Implementation Using CUDA and OpenMP,” *Computing: Techniques and Applications*, pp. 155–161, 2008.

[4] D. Ly and P. Chow, “A High-Performance FPGA Architecture for Restricted Boltzmann Machines,” *ACM International Symposium on FPGAs*, pp. 73–82, 2009.

[5] NVIDIA, “CUDA Programming Guide v2.0,” 2008.

[6] M. Tommiska, “Efficient digital implementation of the sigmoid function for reprogrammable logic,” *IEE Proceedings – Computers and Digital Techniques*, pp. 403–411, 2003.

[7] Y. Liao, “Neural Networks in Hardware: A Survey,” tech. rep., Santa Cruz, CA, USA, 2001.