Using GPUs for Machine Learning Algorithms

Dave Steinkraus[1], Ian Buck[2*], Patrice Y. Simard[1]
[1] Microsoft Research, One Microsoft Way, Redmond, WA 98056
[2] Stanford University, 282 Gates Bldg, Stanford, CA 94305
steinkraus@hotmail.com, patrice@microsoft.com, ianbuck@nvidia.com

Abstract

Using dedicated hardware to do machine learning typically ends up in disaster because of cost, obsolescence, and poor software. The popularization of Graphic Processing Units (GPUs), which are now available on every PC, provides an attractive alternative. We propose a generic 2-layer fully connected neural network GPU implementation which yields over 3X speedup for both training and testing with respect to a 3GHz P4 CPU.

1. Introduction

OCR and on-line handwritten recognition are computationally expensive. Training time is a major bottleneck for improving handwriting recognition (3+ weeks, depending on the language). Printed OCR, being off-line, is also limited by test time (currently at about 1000 character per second).

Using dedicated hardware to do machine learning most often ends up in disaster. The hardware is typically expensive, unreliable, without libraries, poorly documented, and obsolete within a few years. The machine learning software typically only implements a few algorithms in an obscure language under very constrained architectural conditions. The results cannot be shared with other researchers, let alone customers, who do not have the hardware, patience, or interest. One of the authors has witnessed or participated in several such misadventures with analog chips, FPGAs, and coarse-grained parallel computers.

The situation has changed recently with the popularization of Graphic Processing Units (GPUs) [1]. The GPU is a single-chip processor that is designed to accelerate the real-time three-dimensional (3D) graphics that are displayed to a user. Initially a feature of high-end graphics workstations, the GPU has found its way onto the personal computer bus as an accelerator of graphics functions for which a conventional central processing unit (CPU) was ill-suited or simply too slow.

Current trends in GPU design and configuration have given them larger dedicated memory, higher bandwidth to graphics memory, and increased internal parallelism. In addition, current GPUs are designed with ever-increasing degrees of programmability. With the introduction of programmability, the GPU has gained enough flexibility to find use in non-graphics applications. Furthermore, the data-parallel architecture of GPUs delivers dramatic performance gains, compared to CPUs for computationally-intensive applications. Extensions to alternative graphics algorithms and scientific computing problems have been explored in a number of instances [2][3][4].

GPU cards are now mass produced for the consumer market and are therefore inexpensive ($200-$500 for a high-end graphics card). They are programmable through languages such as DirectX or OpenGL. The graphics primitives still use triangles, but the hardware also allows the instructions to render each pixel to be specified by a program, which can be loaded before the triangle(s) is (are) rendered. These programmable triangle renderers are called "pixel shaders". The instructions of the program in the shaders are close to assembly language, since each has a direct hardware implementation. The new flexibility introduced by pixel shaders allows not only naturalistic rendering of surfaces, but also brings the GPU closer to a general purpose parallel processor. It is the latter aspect that we will exploit in this paper.

We have explored several machine learning algorithms for potential implementation on a GPU. We eliminated SVMs, HMMs, Decision Trees, Nearest Neighbors, Boosting, and various search optimization, because of their high memory access requirements. Neural networks, however, seem particularly well

* Currently at NVDIA Corporation, MS 14, 2701 San Tomas Expressway, Santa Clara, CA95050

Proceedings of the 2005 Eight International Conference on Document Analysis and Recognition (ICDAR'05)
1520-5263/05 $20.00 © 2005 IEEE

suited for GPU implementation, provided that the weights reside on the GPU. In this paper we describe how to implement a 2-layer fully connected network on a GPU, but other architecture such as convolutional networks (SDNN and TDNN) are possible.

2. General architecture

The CPU and GPU communicate through the Accelerated Graphics Port (AGP) Bus. The CPU-to-GPU bandwidth is 1GB/s, but the GPU-to-CPU bandwidth is very limited. Because data transfer is slow and asymmetric, the learning parameters reside on the GPU, and only the training data or testing data are transmitted to the GPU in the recognition loop. The new PCI Express cards promise a much better bandwidth in both directions.

During training, the learning parameters can be transferred back to the CPU at regular, but long intervals to verify the learning progress without introducing overhead. During testing, the classification results, which typically comprise a small amount of data, can be sent back to the CPU is small batches with little overhead. The current AGP bus bandwidth favors applications where the number of multiply-adds done on the GPU is much greater than the number of bytes transmitted over the bus. Because of this limitation, we have chosen to implement neural network training to illustrate the use of GPU. For simplicity, we have implemented a 2-layer fully connected network that we train on the MNIST dataset. With N hidden units, we have about 3N multiply-adds for each input number transferred across the AGP bus during training with vanilla backpropagation. The number of hidden units N must be at least 300 to circumvent the AGP bottleneck (see results section). We have not tested the PCI Express bus, but we expect that this limit will be lowered significantly.

The general architecture is illustrated in Figure 1. The first step is to load the shaders, initial weights, and other learning parameters on the GPU. The CPU pre-loads as much as possible onto the GPU before entering the training loop. In our example, the learning parameters are the weights of each of the neural network layers, and the thresholds of each unit. The learning parameter is a single scalar called the learning rate. The programs P are for the different shaders used for the forward and backward propagation, and for the weight updates.

In the next step, the CPU starts to loop on the training data, and accumulates groups of training data. The reason for the groups is that there is a cost of initiating a data transfer between CPU and GPU. Transferring the data by groups of several patterns at a time is more efficient. In our training session, the groups consist of 1000 patterns. The patterns are pairs of 28-by-28 pixel images (X) and their target labels (T).

We then have a preprocessing step where X is transformed into X' and X' is sent to the GPU instead of X. The preprocessing step can have many different functions, such as putting the data in better form (normalization), extracting intelligent or complex features, generating new examples by distorting existing ones (enriching the data set), etc. In theory, the preprocessing could be done either on the GPU or the CPU. In practice, it is much easier to program on the CPU than the GPU. This means that if the preprocessing is computationally inexpensive, it is much easier to run it on the CPU. In our case, the preprocessing is used to generate artificial data (translation, rotation, etc) from the original data. This process is known to improve generalization [5].

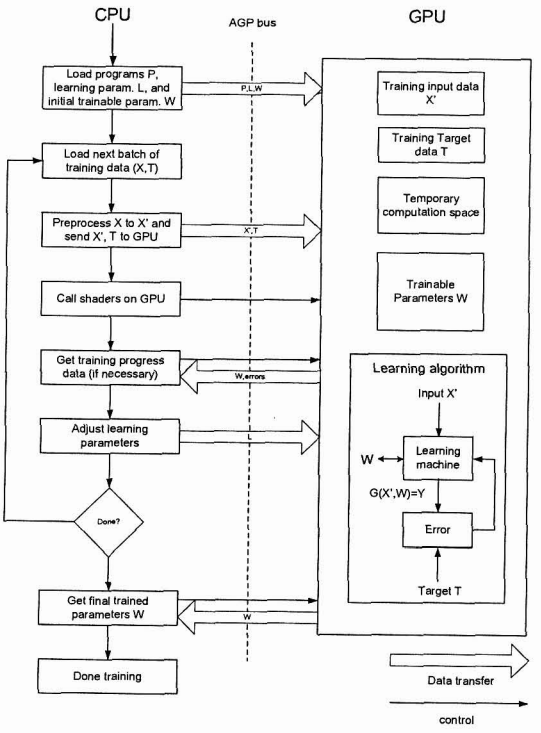


Figure 1. Left, data flow diagram for training machine learning algorithms on the GPU. Right, corresponding data flow during testing.

Once the training data has been loaded on the GPU, the CPU instructs the GPU to run the various shaders that make up the learning algorithm. A description of the shaders can be found in the next section. A typical

learning algorithm is represented in a box inside the GPU. The learning algorithm computes a function G(X',W) as a function of the preprocessed input X' and the trainable parameters W. The goal is to make this output as close as possible as the target value T. An error between G(X',W) and T is computed, and error signals (e.g. gradient with respect to W) are sent back to the learning machine. The weights are then updated in order to reduce the error between G(X',W) and T.

For training our 2-layer neural networks, the forward and backpropagation correspond to about twenty different shaders (some of which are called multiple times). The number and complexity of shaders can of course vary depending on the algorithm used. The shaders are called for each pattern in a group (1000 times in our case). For stochastic gradient descent, the learning parameters are updated after processing each pattern in the group. For batch gradient descent, the learning parameter gradients are accumulated over several patterns before the learning parameters are updated. Whether to use stochastic or batch gradient descent depends heavily on the application and the learning algorithm (and for some learning algorithms, such as SVM, the questions does not arise). For handwriting recognition and neural networks, stochastic gradient descent is preferable, and this is what we implemented.

Figure 1 shows that it is possible to get information (such as error or weights) back from the GPU during the training loop. This is useful to adjust learning parameters (e.g. learning rate), but it must not be done too often if the cost of transfer over the bus is high.

Finally, when the training session is completed, after a fixed number of iterations, or when a desired error threshold has been achieved, the training is stopped and the learning parameters are downloaded to the CPU and saved.

The testing architecture is very similar to the training architecture, except that the classification results must be transferred back to the CPU. For the MNIST database, the classification results are so small (10 probability numbers for each of the classes), that the result can be sent back to the GPU after each presentation if desired for interactive use.

3. Using pixel shaders for machine learning computation

In the Direct3D component of DirectX 9, there are two elements, called vertex shaders and pixel shaders, that are highly programmable. Both types of shaders are concerned with the rendering of triangles (the building blocks of graphics objects) to an output device. Vertex shaders can be used for tasks like spatial transformation and animation of the vertices of triangles (hence the name). Pixel shaders, which are of greater interest in this context, are used to render (that is, to calculate the color values of) the individual pixels in one triangle at a time.

A pixel shader is expressed as a series of instructions in DirectX shader assembly language, which is a limited, hardware-independent language defined by DirectX. The code in a shader is executed once for each pixel in a triangle being rendered, and its only effect is to set the values of the 4-vector for that pixel. The limitations of the shader language, and the lack of side effects, mean that the GPU is free to render pixels in any order and using as much parallelism as its hardware can support, resulting in very high performance. The fact that a pixel is a 4-vector affords yet another kind of parallelism; each execution of a pixel shader can calculate four elements (e.g. four adjacent elements of a vector) at once.

Many of the facilities that an assembly-language programmer would expect can be used within pixel shaders, including constants, registers, addition, subtraction, multiplication, reciprocal, a small set of transcendental functions, and so on.

Machine learning algorithms typically use simple primitives such as:

1. Inner products (between vectors or matrix and vector)
2. Outer products (between vectors)
3. Linear algebra (e.g. addition, subtraction, multiplication by a scalar) on vectors or matrices
4. Non-linearity (e.g. tanh, sigmoid, thresholding) applied to a vector or a matrix
5. Matrix transpose

We will show how to implement each of these operations with pixel shaders, and illustrate how they can work together with the example of an end-to-end implementation of a fully connected 2-layer neural network, for both training and use in a real setting. However it should be clear that the concept can be used with other learning algorithms made out of the same primitives.

3.1 Implementing point-to-point operation

All of the operations above can be implemented using one or more pixel shaders. The first challenge is

to make a shader calculate a result that is a rectangle in GPU memory – in other words, a 2-dimensional matrix of floating-point values. As stated before, pixel shaders render all pixels in a given triangle. However, we can also specify a rectangular viewport, and the GPU will only calculate pixels within the intersection of the viewport and the triangle (Figure 2, right).

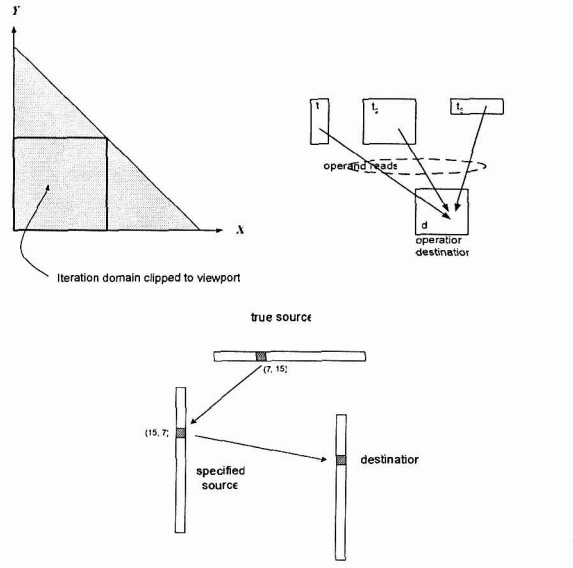


Figure 2. Top left, triangle clipped to rectangular region. Top right, use of texture mapping to do operation with multiple operand. Bottom, transposition.

In the degenerate case, the viewport is a row or a column to access memory vectors. Using viewports, we can implement unary operations like x = F(x) for vectors and rectangular matrices.

DirectX allows us to allocate a rectangular region of memory as a workspace. On current hardware we can allocate rectangles inside a square region of at least 2048 x 2048 pixels (each 4-valued), depending on memory.

Pixel rendering depends on multiple regions because of surface texture, reflection, transparency, etc. DirectX provide a mechanism – texture mapping – to map more than one source rectangle to the current destination. At least 8 such mappings can be used at once in current hardware. With multiple sources, we can implement operations such as *(vector A - vector B -> vector C)*. At each pixel in C, we fetch the texture-mapped values from A and B, perform elementary math on register values, and store the result.

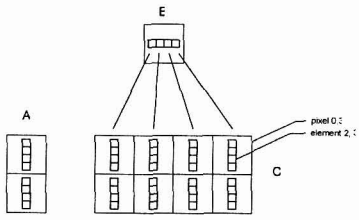
We can extend the usefulness of textures by using arithmetic on register values inside a shader. Registers are local variables which the shader can use to render a given pixel. Their values cannot be shared between pixels (this would break the parallelism assumption) but can be used locally for intermediate results. For example, we can transpose any array or vector while copying its values to a new location. Suppose we have a source rectangle whose left, right, top, and bottom coordinates are *l*, *r*, *t*, and *b*. We would specify a texture rectangle whose coordinates are *t*, *b*, *l*, and *r*. Then, inside the pixel shader, we would swap the x and y texture coordinates before using them to fetch a value from the source and copy it to the destination. At the end of rendering, the destination will contain the transpose of the source. In figure 2, left, the rendering of the current pixel of the destination will retrieve texture coordinates (15, 7) from the texture rectangle; but before fetching the texture value, we will reverse the row and column coordinates so that we actually read the value from location (7, 15), which is located inside the actual vector we are transposing.

Using the four coordinate planes

The GPU workspace has 4 planes – each pixel consists of x, y, z, and w values. These values can be accessed as rows of pixels (0:x, 0:y, 0:z, 0:w, 1:x, 1:y...) or columns (0:x, 1:x, 2:x, ...), where 0:x means pixel 0, plane x. This layout does not affect copy or vector transpose operations, but certain operation such as matrix transpose and mathematical operations must take plane indexing into account.

Outer Product

The outer product of two vectors can be accomplished with use of all four planes, but another new technique must be introduced, which we will designate the *indexer texture*. This is a way to select the value in just one of the four planes of a vector which has been mapped to elements as described above.



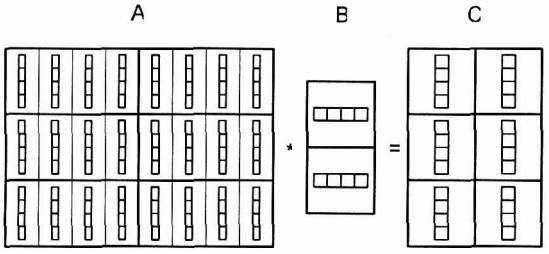
$$C_{0,0} = A_0 * B_0$$
$$C_{1,0} = A_1 * B_0$$
$$C_{2,0} = A_2 * B_0$$
$$C_{3,0} = A_3 * B_0$$

Figure 3. Outer dot product. Each pixel has values in 4 planes (left). Individual plane values must be extracted and reused for outer product (right).

In figure 3, we compute the outer product C = A X B. The problem is that B0 (N:x for pixel N, plane x) must be accessed and reused for A0, A1, A2, and A3. This is not a point-to-point operation (the strength of pixel shaders). B0 can be extracted by using a mask [1,0,0,0] and using the dot operation: B0 = dot([B0,B1,B2,B3],[1,0,0,0]). The value B1 can be accessed using [0,1,0,0], and so on.

Inner product

Inner products present a problem because they require that we accumulate a sum while looping over several values. This goes against the shader architecture which assumes there is no communication between adjacent pixels. So an inner product cannot be implemented as a single pixel shader. However, we can accomplish it by using a series of shaders and some additional memory.



$$C_{00} = A_{0,0} * B_{0,0} + A_{0,1} * B_{0,1} + A_{0,2} * B_{0,2} + A_{0,3} * B_{0,3}$$
$$C_{10} = A_{1,0} * B_{1,0} + A_{1,1} * B_{1,1} + A_{1,2} * B_{1,2} + A_{1,3} * B_{1,3}$$

Figure 4. Inner product. The dot product is done as a point to point partial product. Darker lines indicate how cells are multiplied. Some of the columns of A are collapsed by the dot operation. Rows of C must be further summed, with successive "reduce" operations to complete the dot product.

We show in figure 4 how the inner product can be computed by point–to-point operations in C. The rows of C must then be summed by doing successive summation passes, each corresponding to a distinct shader.

Results

We have implemented a generic 2-layer fully connected neural network using the Pixel Shader 2.0 (ATI) and 3.0 (NVIDIA) languages for the GPU. This implementation is truly versatile because preprocessing and postprocessing can easily be done on the CPU. These typically represent a very small fraction of the computation for the end-to-end system. Our test application is the MNIST handwritten digit database. We have experimented extensively with different memory layouts and different hardware. Low precision floating point operations (24 bits, with 16-bit mantissa) limits the training accuracy of the ATI Radeon X800 card[2]. The reason is that weight updates need a large mantissa due to the cumulative effect of small gradients. The ATI GPU error rate after training is about twice as high as with the CPU, as a result of 24 bits weight updates. This is not good enough for commercial applications, but may be sufficient for rapid testing of new algorithms. For testing/deployment, 24 bits is more than enough accuracy, and GPU and CPU yield identical results. The NVIDIA GeForce 6800 Ultra GPU, which supports 32-bit precision, is as accurate as the CPU for both testing and training. Our CPU implementation used full dot products with compile SSE optimization enabled on a 3GHz P4. We also report results obtained with the Intel Math Kernel Library (MKL). The results are summarized in Table 1.

Table 1: The 2nd column indicates the number of pattern updates per second as a function of the number of hidden units. The next 3 columns show the relative speedup for the Intel Math Kernel Library (MKL), the ATI Radeon X800, and the NVIDIA GeForce 6800 Ultra. In each cell, 1st line is for training and 2nd line is for deployment. The best usable numbers are represented in bold. The results in parenthesis are not usable for training.

Hidden units	CPU	MKL	ATI GPU	NVIDIA GPU
200 (train)	500	**2.1**	(2.1)	1.3
200 (test)	2011	**1.1**	0.9	0.6
600 (train)	165	2.2	(5.9)	**2.3**
600 (test)	723	1.1	**2.4**	1.2
1000 (train)	89	2.2	(7.0)	**3.3**
1000 (test)	440	1.1	**3.4**	1.4

These results must be interpreted with caution. Memory access speed plays a significant role on the performance. Small network that entirely fit in the L2 cache (512K) are faster in terms of multiply-add per seconds (for both CPU and MKL). For networks that do not fit in the cache (as in the table), MKL yields a 2X speedup for training but not for testing. This is because the back propagation requires a transpose of the weight matrix. The basic CPU implementation is slow on back propagation because the transpose

[2] The next generation is rumored to support 32 bits.

generates memory accesses in a direction not favored by the cache. MKL has an optimization which performs the transpose by small block which yields a 2X speedup for the whole training update. The MKL implementation takes full advantage of the P4 cache architecture. Neither NVIDIA nor ATI has released the necessary information to make the corresponding optimizations on GPUs. At this point, we can only second guess how the GPU cache is organized and our reported performances are well below what NVDIA and ATI advertise.

On small fully connected neural network (200 hidden units), MKL is the winner with a 2X speedup for training. For medium and large network, the NVDIA is best for training, yielding up to 3.3X speedup over the basic implementation. The ATI 24 bits training, although faster, does not converge to a usable solution. For test or deployment, however, the ATI yields the best results, with 2.4X to 3.4X speedup, depending on the size of the network.

Conclusion

We have implemented a generic and versatile end-to-end learning algorithm on the GPU. This has already yielded a better than 3X speedup in both training and test time (although on different cards). Unlike dedicated machine learning hardware, there is no risk of obsolescence for our system and software. The GPUs are mass produced off-the-shelf hardware, and they are quickly and regularly replaced by new generation GPUs which not only run faster, but support the code written for the previous generation. Graphics has brought SIMD computing to the masses. GPU performance is currently increasing faster than CPU performance, which means that our implementation's speedup will increase with each new generation of GPU (the last generation upgrade yielded a 2X speedup). Programmability of SIMD/pixel shaders is become easier with each generation (e.g. Pixel shader 3) and the graphic card manufacturers have great incentive to make GPU more general purpose. It is likely that the next generation of ATI cards will support 32 bits floating point.

OCR and on-line handwritten recognition are prime target applications for GPUs. Training time is a major bottleneck for handwriting recognition (1-3 weeks, depending on the language). Printed OCR, being off-line, is limited by test time (currently at about 1000 character per second). In both case, GPUs bring a much needed speedup. The current paper describes the implementation of a fully connected 2-layers network on a GPU. We are currently working on a GPU implementation of convolutional networks.

Acknowledgement

We would like to thank Kumar Chellapilla for useful discussions, suggestions, and getting MKL results.

References

[1] M. Macedonia, "The GPU Enters Computing's Mainstream," IEEE *Computer*, 2003, pp. 106-108.

[2] T.J. Purcell, I. Buck, W. Mark, & P. Hanrahan, "Ray Tracing on Programmable Graphics Hardware," *ACM Transactions on Graphics*, 21 (3), 2002, pp. 703-712.

[3] J. Kruger, and R. Westermann, "Linear Operators for GPU Implementation of Numerical Algorithms," *Proceedings of SIGGRAPH*, San Diego, 2003, pp. 908-916.

[4] J. Bolz, I. Farmer, E. Grinspun, & P. Schroder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid,", *Proceedings of SIGGRAPH*, San Diego, 2003, pp. 917-924.

[5] P. Y. Simard, D. Steinkraus, & J. Platt, "Best Practice for Convolutional Neural Networks Applied to Visual Document Analysis," *International Conference on Document Analysis and Recognition (ICDAR)*, IEEE Computer Society, Los Alamitos, 2003, pp. 958-962.