Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 225 (2009) 379–389
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Evaluating Computational Performance of Backpropagation Learning on Graphics Hardware[1]

Hiroyuki Takizawa[2] Tatsuya Chida[3]

Graduate School of Information Sciences
Tohoku University
Sendai, Japan

Hiroaki Kobayashi[4]

Information Synergy Center
Tohoku University
Sendai, Japan

Abstract

Although volunteer computing with a huge number of high-performance game consoles connected to the Internet is promising to achieve large-scale data mining, the programming models of such game consoles for data mining tasks are restricted. As the game consoles have high-performance graphics hardware for state-of-the-art video games, a key to exploit their computation power for data mining is how effectively the data mining is mapped to the hardware as graphics processes.
In this paper, therefore, a popular data mining tool called the backpropagation learning neural network is implemented as an application running on graphics hardware. Since the recent graphics hardware has many vector processing units and high memory bandwidth, it is promising to accelerate the backpropagation learning task involving a lot of data-parallel computations. The evaluation results have demonstrated the great potential of our prototype implementation for massive backpropagation learning tasks. The graphics hardware can efficiently work especially if the task is implemented so as to use data-parallel instructions supported by the hardware.

Keywords: Backpropagation learning neural networks, programmable graphics processing unit, general-purpose computation on graphics hardware.

[1] This research was partially supported by Grants-in-Aid for Scientific Research on Priority Areas #18049003, Young Scientists(B) #19700020, and Strategic Information and Communications R&D Promotion Programme (SCOPE-S) #061102002.

[2] Email: tacky@isc.tohoku.ac.jp

[3] Email: tatsuya@sc.isc.tohoku.ac.jp

[4] Email: koba@isc.tohoku.ac.jp

doi:10.1016/j.entcs.2008.12.087

1 Introduction

Today, enormous amounts of data are being produced and accumulated everyday, at all times. As a result, it is very difficult to manually retrieve valuable information and useful knowledge from a huge sea of data. Because of the social demand, the computer technologies for knowledge discovery in databases, so-called *data mining* [4], have been receiving increasing interests. Although data mining is helpful to limit the scope of our information search, a data mining system has to process a large number of data in a practical time. As the database is growing rapidly, we need more and more computing power for future data mining.

The SETI@home project [1] has demonstrated the tremendous potential of volunteer computing, which uses idle computing resources on the Internet, to realize a large-scale data mining system. Furthermore, the latest game consoles have excellent computing power. In the near future, hence, a huge number of idle game consoles will ubiquitously exist on the Internet. Effectively using such idle game consoles, volunteer computing is expected to realize an unprecedented scale data mining task.

Considering the requirements for recent video games, it is definite that the game consoles are equipped with high-performance graphics hardware, so-called graphics processing units (GPUs). However, due to the application-specific architecture, programming non-graphics applications for GPUs is likely to be restricted. Therefore, one assured way to exploit their high performance for a data mining project is to effectively map the data mining computations to the graphics processes.

In this paper, the learning task of a backpropagation neural network [11], which is one of the most popular tools for data mining [2], is implemented as an application running on GPU. As the backpropagation learning algorithm involves massive data parallelism, GPU is promising to accelerate the learning tasks. As the first step to establish such an effective implementation scheme, therefore, this paper shows the great potential of the GPU implementation for massive backpropagation learning tasks.

The outline of this paper is as follows. Sections 2 and 3 briefly review the backpropagation learning algorithm and GPU, respectively. Then, we propose an implementation scheme of the backpropagation learning task using GPU. Section 4 shows our experimental results to evaluate the performance of the GPU implementation. Finally, Section 5 gives concluding remarks and our future work.

2 Backpropagation Learning

Artificial neural networks are the basic tools for data mining [2]. Their learning algorithms can roughly be categorized into supervised learning and unsupervised learning algorithms; supervised learning assumes that a set of ideal input-output vector mappings, a training data set, is given, while unsupervised learning does not. The backpropagation algorithm being discussed below is a typical supervised learning algorithm for multi-layered feed forward neural networks [11].

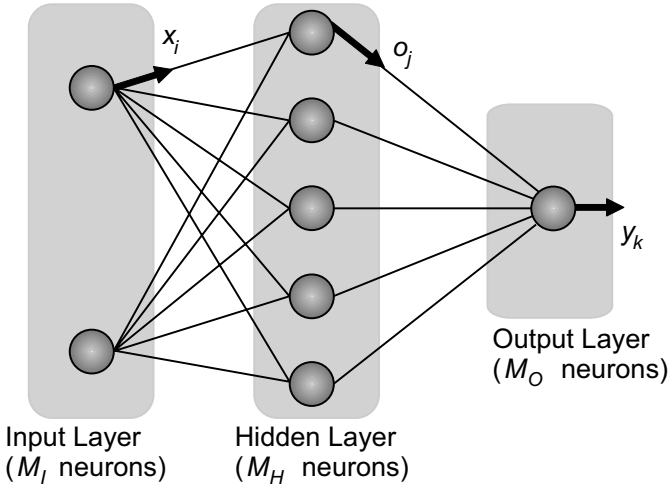


Fig. 1. A multi-layered feed forward neural network.

A multi-layered neural network consists of some layers of artificial neurons, and those layers are connected via weighted links. Suppose a 3-layered neural network consisting of one input layer, one hidden layer, and one output layer. Let M_{I}, M_{H}, and M_{O} be the numbers of neurons in the input layer, the hidden layer, and the output layer, respectively. Figure 1 illustrates such a neural network.

An M_{I}-dimensional input vector in a training data set $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$ is denoted by $\boldsymbol{x}_l = \{x_{1l}, x_{2l}, \ldots, x_{M_{\mathrm{I}}l}\}$. Each component of an input vector, x_{il}, is given to an input neuron, and propagated to hidden neurons via weighted links. The output value of a hidden neuron, o_{jl} is calculated by

$$o_{jl} = f(\sum_{i=0}^{M_{\mathrm{I}}} w_{ij} x_{il}), \tag{1}$$

where w_{ij} is the weight of the link connecting the i-th input neuron and the j-th hidden neuron, and $f(\cdot)$ is a non-linear transfer function, e.g. sigmoid function $f(x) = 1/(1 + \exp(-x))$. Here, x_{0l} and o_{0l} are always equal to 1, and hence w_{0j} represents the threshold of the j-th hidden neuron.

The output values of hidden neurons are again propagated to output neurons to calculate the outputs of the neural network:

$$y_{kl} = f(\sum_{j=0}^{M_{\mathrm{H}}} w_{jk} o_{jl}), \tag{2}$$

where y_{kl} is the output value of the k-th output neuron, M_{H} is the number of hidden neurons, and w_{jk} is the weight of the link connecting the j-th hidden neuron and the k-th output neuron. In this way, the input signals propagate through the neural network in a forward direction to calculate M_{O}-dimensional output vectors.

The backpropagation learning algorithm is an iterative gradient descent algorithm to optimize the link weights so as to minimize the mean squared error between

the current actual outputs and ideal outputs:

$$E(\boldsymbol{W}) = \frac{1}{2}\sum_{l=1}^{N} |\boldsymbol{y}_l(\boldsymbol{W}) - \boldsymbol{t}_l|^2, \tag{3}$$

where N is the number of training data in the data set, $\boldsymbol{y}_i$ and $\boldsymbol{t}_i$ are the actual and ideal output vectors for the i-th training data, respectively. The backpropagation learning algorithm adjusts link weights, $\boldsymbol{W}$, by

$$\begin{aligned} \boldsymbol{W} &:= \boldsymbol{W} - \epsilon \cdot \nabla E(\boldsymbol{W}) \\ &= \boldsymbol{W} - \epsilon \cdot \sum_{l=1}^{N} (\boldsymbol{y}_l(\boldsymbol{W}) - \boldsymbol{t}_l) \nabla \boldsymbol{y}_l(\boldsymbol{W}), \end{aligned} \tag{4}$$

where ϵ is the learning rate that specifies how much a weight value is modified in one iteration.

Accordingly, the backpropagation learning algorithm consists of three phases: the forward propagation of input signals, the backward propagation of errors, and the weight updating. By propagating input signals in a forward direction, the backpropagation learning algorithm first calculates the outputs for each input vector in the training data set. Then, it accumulates the weight modifications for each input vector. To calculate the weight modification of a link not directly connected to any neurons in the output layer, the errors at the output neurons propagate in a backward direction from the output layer to the input layer. Finally, all the weights are updated according to Equation (4).

3 Graphics Processing Unit

GPU has many parallel processing elements and high-speed dedicated memory to efficiently perform fine-grain data parallel tasks commonly involved in computer graphics applications [9]. This drives many researchers to use GPU's computing power even for non-graphics applications. The use of GPUs for non-graphics computation is called *general-purpose computation on GPUs* (GPGPU) [10]. However, GPU cannot work well universally due to its application-specific architecture. Therefore, one important challenge is how to effectively map a non-graphics application to GPU's programmable graphics rendering pipeline.

Modern GPUs have two kinds of programmable processors, *vertex shader* and *fragment shader* on the graphics pipeline to render an image. These processors have SIMD [5] instructions similar to SIMD streaming extensions (SSE) of Intel CPUs that can simultaneously operate on four 32-bit floating-point values within a 128-bit register. Figure 2 illustrates the programmable parts of the GPU and their typical data flows. A 3-dimensional object is represented by a set of polygons, and the vertex shader first projects each vertex of a polygon onto the viewing

[5] SIMD is short for Single Instruction, Multiple Data.

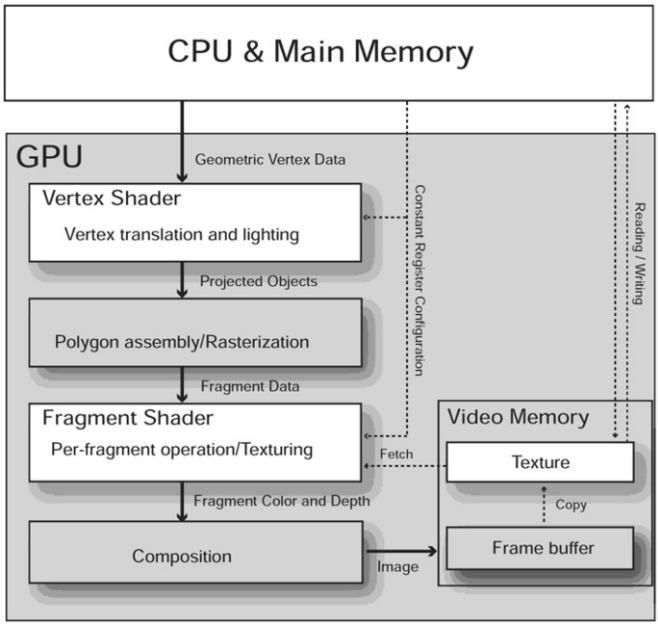


Fig. 2. Programmable Graphics Processing Unit.

coordinate system. Then, the projected polygon is decomposed into fragments, each of which corresponds to a pixel on the screen. Subsequently, the fragment shader performs per-fragment operations to determine the color and depth of each fragment. Finally, composition operations are applied to the outputs of the fragment shader to determine the final pixel colors on the screen. Rendering a polygon once in the above way is called a *rendering pass*. As GPU is originally designed to quickly determine pixel colors on the screen, its architecture is well-suited for SIMD parallel calculation of independent components in a large data array.

Non-graphics applications on GPUs usually exploit the programmability of the fragment shader. The fragment shader can operate colors of multiple fragments in parallel. The fragment shader can also fetch data from texture images on the video memory; the texture images can be used as the source operands of computing on the fragment shader. Moreover, SIMD instructions of the fragment shader are available to simultaneously operate four color channels: red, green, blue and alpha (RGBA) channels[6]. Therefore, we can see the fragment shader as a multi-grain SIMD parallel processor that can calculate multiple fragment colors in parallel, and further can use fine-grain SIMD parallel instructions for the calculations. As a result, the fragment shader can efficiently execute the component-wise matrix operations that independently determine each matrix component in a SIMD parallel fashion, such as addition of two huge matrices.

If four array elements are packed into one texel[7], the fragment shader can use SIMD instructions to simultaneously operate the four values, and hence perform component-wise matrix operations much more effectively. Packing four values into

[6] The alpha value indicates the opacity of the pixel.

[7] A texel is a minimum component of a texture image.

one texel is called *data packing* [8]. Data packing is necessary for multi-grain parallel processing by the fragment shader. Since data packing can reduce either the number of fragments or rendering passes to quarter, it can significantly accelerate many GPGPU applications.

4 Backpropagation Learning on GPU

This paper proposes an efficient mapping scheme for the backpropagation learning on GPU. A multi-layered neural network needs to accumulate all the inputs of each neuron as written in Equations (1) and (2); for the forward propagation, signals from the previous layer and their corresponding weight values are multiplied and then accumulated just like matrix multiplication [3]. As the multiplications are performed in parallel, GPU can accelerate the forward propagation. Similarly, the backward propagation of errors can also be accelerated by simultaneously performing multiplications of the errors at output neurons and their corresponding weight values. Furthermore, as the weight modifications of links between each pair of two layers can be computed in parallel, the weight updating also involves a lot of data-parallel computations. However, it is difficult for GPU to accumulate those values in a single rendering pass because the number of instructions executable in a single pass is limited. In our implementation scheme, therefore, most vector operations in the backpropagation learning algorithm are combined into fewer matrix-matrix component-wise operations, and multi-pass rendering approach is used for matrix-matrix additions to achieve the accumulations.

To calculate the signal propagation of N input vectors from input neurons to hidden neurons, each rendering pass calculates all of N signals from one input neuron to M_{H} hidden neurons. Thus, a single rendering pass corresponding the i-th input neuron performs $N \times M_{\mathrm{H}}$ multiplications:

$$\begin{array}{cccccc} w_{i1}x_{i1}, & w_{i1}x_{i2}, & w_{i1}x_{i3}, & \ldots, & w_{i1}x_{iN}, \\ w_{i2}x_{i1}, & w_{i2}x_{i2}, & w_{i2}x_{i3}, & \ldots, & w_{i2}x_{iN}, \\ \multicolumn{5}{c}{\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots} \\ w_{iM_{\mathrm{H}}}x_{i1}, & w_{iM_{\mathrm{H}}}x_{i2}, & w_{iM_{\mathrm{H}}}x_{i3}, & \ldots, & w_{iN_H}x_{iN}. \end{array} \tag{5}$$

Including calculation of hidden neuron thresholds, hence, $(M_{\mathrm{I}}+1)$ rendering passes are required to sum up all the inputs to each hidden neuron, $\sum_{i=0}^{M_{\mathrm{I}}} w_{ij}x_{il}$. Then, $N \times M_{\mathrm{H}}$ output values are computed in parallel from the accumulated values in another rendering pass, as described in Equation (1). As in the case of hidden neurons, output values of output neurons are computed in $(M_{\mathrm{H}}+2)$ rendering passes in total. Consequently, the forward propagation of N input vectors through the neural network is accomplished in $(M_{\mathrm{I}}+M_{\mathrm{H}}+4)$ rendering passes. The number of input vectors, N, determines the sizes of texture images used in the rendering passes, but does not affect the number of rendering passes required for the forward propagation.

Updating all the weights between the hidden layer and the output layer needs N rendering passes to achieve the accumulation in Equation (4). Thus, a single rendering pass performs $M_{\mathrm{O}} \times (M_{\mathrm{H}}+1)$ parallel tasks to calculate the weight modifications related to one training data:

$$\begin{array}{c} \Delta w_{01}, \quad \Delta w_{02}, \quad \Delta w_{03}, \quad \ldots, \quad \Delta w_{0M_{\mathrm{O}}}, \\ \Delta w_{11}, \quad \Delta w_{12}, \quad \Delta w_{13}, \quad \ldots, \quad \Delta w_{1M_{\mathrm{O}}}, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Delta w_{M_{\mathrm{H}}1}, \ \Delta w_{M_{\mathrm{H}}2}, \ \Delta w_{M_{\mathrm{H}}3}, \ \ldots, \ \Delta w_{N_H M_{\mathrm{O}}}, \end{array} \tag{6}$$

where

$$\Delta w_{jk} = -\epsilon(y_{kl} - t_{kl})\frac{\partial y_{kl}}{\partial w_{jk}}. \tag{7}$$

If the sigmoid function $f(x) = 1/(1+\exp(-x))$ is used as the transfer function, the partial derivative in Equation (7) is calculated by

$$\frac{\partial y_{kl}}{\partial w_{jk}} = y_{kl}(1 - y_{kl})o_{jl}. \tag{8}$$

Then, the backward propagation of errors is done to update all the weights between the input layer and the hidden layer, because calculation of those modifications for training data l requires the sum of the errors at output neurons:

$$\Delta w_{ij} = -\epsilon o_{jl}(1 - o_{jl})x_{il}\sum_{k=1}^{M_{\mathrm{O}}} w_{jk}(y_{kl} - t_{kl})y_{kl}(1 - y_{kl}). \tag{9}$$

As the backward propagation of errors at one output neuron to all the hidden neurons is performed in a single rendering pass as with the forward propagation, the backward propagation needs M_{O} rendering passes in total. Updating all the weights between the input layer and the hidden layer also needs N rendering passes. Accordingly, GPU can execute one learning step of the backpropagation learning algorithm in total $(M_{\mathrm{I}} + M_{\mathrm{H}} + M_{\mathrm{O}} + 2N + 4)$ rendering passes.

Data packing stores four signals into one texel [3]. In our prototype implementation, this packing is used to reduce the size (the number of fragments) of each matrix in the forward and backward propagation phases to quarter, and the packing also allows each per-fragment program to fetch four matrix components by sampling a texture once. Furthermore, it can reduce the number of rendering passes to $(M_{\mathrm{I}} + M_{\mathrm{H}} + M_{\mathrm{O}} + N/2 + 4)$, because the numbers of rendering passes in the weight updating phase is reduced to quarter. Therefore, the packing is significantly effective to improve the computational performance of the GPU implementation.

5 Performance Evaluation

This section evaluates a prototype implementation of the backpropagation learning algorithm running on GPU. For performance evaluation, we have implemented a 3-layered neural network. The program code is written in C++ and OpenGL [12] with NVIDIA's extensions, and compiled by Microsoft Visual Studio 2003 compiler. All of its shader programs are written in a high-level programming language, called *C for Graphics* [5].

First, the performance gain due to the data packing is evaluated using AMD Athlon64 3500+ and NVIDIA GeForce 7800GTX, whose popular edition is used in the latest game console. In the case of $M_{\mathrm{I}} = 2$, $M_{\mathrm{H}} = 64$, $M_{\mathrm{O}} = 1$, and $N = 1,024$, GeForce 7800GTX needs only 0.008 seconds to execute one learning step of the GPU implementation with the packing, while it needs 0.026 seconds for that without the packing. In addition, the GPU implementation with the packing can perform a larger learning task than that without the packing. This is because the maximum texture size is limited and the packing can reduce the required texture sizes used in the forward and backward propagation phases to quarter.

Figure 3 shows the execution time per learning step measured changing M_{H} and fixing the others. For a small network of few neurons, the conventional CPU implementation is superior to our GPU implementation because there is a certain overhead to launch the processing on GPU. However, the computational performance of the GPU implementation improves as the number of hidden neurons increases, because the overhead becomes relatively small for a larger network. As a result, the GPU implementation outperforms the CPU implementation for a large neural network of many neurons. These results indicate that the GPU implementation is well suited for high-dimensional data mining, e.g. high-resolution image data mining, which needs many neurons at least in the input layer.

In general, the GPU implementation with the packing outperforms that without the packing. However, their difference becomes small as the network size increases, because performance gain by reduction in the number of rendering passes becomes small relative to the total execution time. As the number of hidden neurons increases, the network size becomes a dominant factor in the number of rendering passes. Although the packing can reduce the texture sizes used in the forward and backward propagation phases, it cannot reduce rendering passes required in the phases. The *efficiency*, i.e. the ratio of effective performance to peak performance, generally degrades as the texture size decreases, even though the total execution time decreases with the texture sizes. Accordingly, these experimental results clarify that the reduction in the number of rendering passes can accelerate the GPU implementation more significantly than the reduction in the texture sizes. For improving the efficiency, the data packing should be used so as to reduce the rendering passes rather than the texture sizes, if possible.

Figure 4 shows the execution time per learning step measured changing N and M_{H}. As shown in the figure, the superiority of the GPU implementation with the packing becomes remarkable as N increases. The reduction in the number of

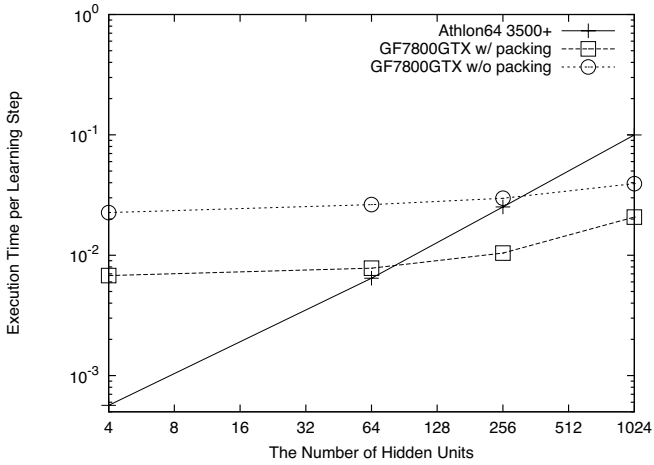
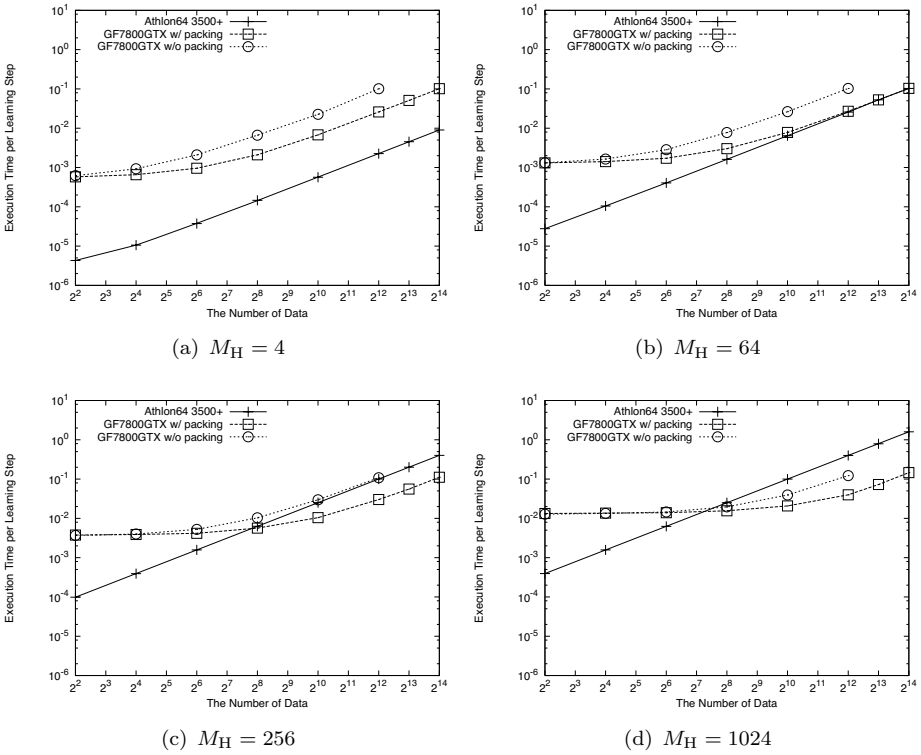


Fig. 3. Execution Time per Learning Step Changing the Network Size.



(a) $M_{\mathrm{H}} = 4$ (b) $M_{\mathrm{H}} = 64$

(c) $M_{\mathrm{H}} = 256$ (d) $M_{\mathrm{H}} = 1024$

Fig. 4. Execution Time per Learning Step Increasing the Number of Data.

rendering passes that is attained by the packing is $3N/2$, and depends only on N. Furthermore, the sizes of texture images used in the forward propagation phase become larger as N increases, and the efficiency improves as a result.

For a larger learning task with many data and/or many neurons, our prototype GPU implementation can outperform the conventional CPU implementation. Therefore, we conclude that the GPU implementation is a promising approach to

highly efficient volunteer computing with many game consoles for massive data mining tasks. One may claim that our GPU implementation should work faster because the theoretical peak performance of GeForce 7800GTX in single precision floating-point operations is much better than that of Athlon64 3500+. Indeed, our GPU implementation is still inefficient in view of GPU's peak performance. This is mainly due to the memory bandwidth shortage as mentioned in [3]. A more sophisticated implementation scheme, which takes the GPU memory hierarchy into account [7], will be required to overcome this inefficiency problem.

6 Concluding Remarks

The goal of our project is to realize a large-scale data mining system by volunteer computing of game consoles with high-performance GPUs. As the first step, this paper has discussed the GPU implementation of one popular data mining tool, the backpropagation learning neural network. Then, our prototype implementation has clearly demonstrated the high performance of the GPU implementation especially for large-scale learning tasks.

The superiority of our GPU implementation greatly depends on the data size and parameter configuration. We need to estimate the performance of the GPU implementation for a given task, and should use GPU only if it can work better than CPU. A promising alternative is complementary use of both CPU and GPU with carefully considering the load-balance between them [6]. Another important research issue is how efficiently the data mining algorithm can be implemented under the data size limitation of GPU. The video memory is generally smaller than the main memory, and the maximum texture size is also limited. Therefore, efficient use of such a limited memory space is necessary to execute a task whose data size exceeds the limitation. These issues will be further investigated and discussed in our future work.

References

[1] Anderson, D. et al., *SETI@home: An experiment in public-resource computing*, Communications of the ACM **45** (2002), pp. 56–61.

[2] Berry, M. and G. Linoff, "Data Mining Techniques; For Marketing, Sales, and Customer Support," John Wiley & Sons, Inc., 1997.

[3] Fatahalian, K., J. Sugerman and P. Hanrahan, *Understanding the efficiency of GPU algorithms for matrix-matrix multiplication*, Graphics Hardware 2004 (2004), pp. 133–138.

[4] Fayyad, U. M., G. Piatetsky-Shapiro and P. Smyth, *From data mining to knowledge discovery in databases*, AI Magazine **17** (1996), pp. 37–54.

[5] Fernando, R. and M. J. Kilgard, "The Cg Tutorial," Addison-Wesley, 2003.

[6] Gierlinger, T. and P. Prabhu, *Towards load balanced computations using GPU and CPU*, 2004 ACM workshop on general-purpose computing on graphics processors(GP2) (2004), pp. C–14.

[7] Govindaraju, N., S. Larsen, J. Gray and D. Manocha, *A memory model for scientific algorithms on graphics processors*, in: *the 2006 ACM/IEEE conference on Supercomputing (SC06)*, 2006.

[8] Hall, J., N. Carr and J. Hart, *Cache and bandwidth aware matrix multiplication on the GPU*, Uiuc technical report, UIUCDCS-R-2003-2328 (2003).

[9] Kilgariff, E. and R. Fernando, *The GeForce 6 series GPU architecture*, in: *GPU Gems 2: programming techniques for high-performance graphics and general-purpose computation*, Addison-Wesley, 2005 pp. 471–491.

[10] Owens, J. D. et al., *A survey of general-purpose computation on graphics hardware*, Eurographics 2005, State of the Art Reports (2005), pp. 21–51.

[11] Rumelhart, D. E., G. E. Hinton and R. J. Williams, *Learning representations by back-propagating errors*, Nature **323** (1986), pp. 533–536.

[12] Woo, M., J. Neider, T. Davis and D. Shreiner, "OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2," Addison-Wesley, 1999, 3rd edition.