Scalability of Self-organizing Maps on a GPU cluster using OpenCL and CUDA

Sabine McConnell, Robert Sturgeon, Gregory Henry, Andrew Mayne, and Richard Hurley

Department of Computing and Information Systems, Trent University, Peterborough, On, Canada
E-mail: sabinemcconnell@trentu.ca

Abstract. We evaluate a novel implementation of a Self-Organizing Map (SOM) on a Graphics Processing Unit (GPU) cluster. Using various combinations of OpenCL, CUDA, and two different graphics cards, we demonstrate the scalability of the SOM implementation on one to eight GPUs. Results indicate that while the algorithm scales well with the number of training samples and the map size, the benefits from using the data-parallel approaches offered by the GPU are severely limited when combined with the Message Passing Interface (MPI) in this setting, and comparable to speedups of GPU-based implementations as compared to optimized sequential code. Speedups achieved range from 3 to 32, for various map and training data sizes. We also observed a performance penalty for the OpenCL implementation as compared to CUDA.

Since the introduction of NVIDIA’s release of the Compute Unified Device Architecture (CUDA) in November 2006, which enabled programmers to bypass traditional graphics Application Programming Interfaces (APIs) such as OpenGL for General-Purpose GPU computing, the suitability of graphics cards for computationally intensive data-parallel applications has been demonstrated by a wide range of applications. Recent examples include the calculation of powder diffraction patterns [5] and real-time stream scheduling that utilizes both GPU and CPU [14].

Early on, the medical community recognized the potential for the use of Graphic Processing Units in, for example, organ segmentation [6]. With the advent of CUDA and the removal of the need for substantial background knowledge in graphics, the number of applications increased steadily where today, the use of GPUs has a strong base in medical computation. In addition to, but separate from the use of GPUs, the medical community has also taken advantage of data mining, which is typically defined as the extraction of novel information from large dataset. Successful applications of this include an early evaluation of neural networks and genetic programming for classification and generalization on multiple diagnosis problems by Brameir and Banzhaf [3], as well as the diagnosis of cardiopulmonary disease [4], breast cancer [1], and diabetes [2].

With the exponential growth of datasets over the past decade and the increased use of medical data-mining applications, the data-mining community has moved into high-performance settings...
including accelerators which are characterized as hardware that performs certain computations faster than the CPU. Examples for such accelerators include Field Programmable Gate Arrays, the Cell Broadband Engine Architecture (CBEA), and GPUs. Implementations of data-mining techniques on GPUs include a neural network for bankruptcy prediction [12], Expectation Maximization for Gaussian Mixture Models [8], and Support Vector Machines [10]. Liao et al. [10] present a parallel implementation evaluated on six different datasets, achieving a maximum speedup of 212 for the kernel alone, and a speedup of up to 94 as compared to the CPU version when taking memory transfers into account. To the best of our knowledge, no implementations of data-mining techniques exist to date for clusters of GPUs. In this paper, we evaluate the scalability of Self-Organizing Maps, a particular data-mining technique, on the SHARCNET cluster Angel, which supports both Open Computing Language (OpenCL) and CUDA. We investigate the scalability of the implementations with respect to the size of the input dataset, the size of the map, and the effects of the different programming environments.

The remainder of this paper is organized as follows. In Section 1, we introduce the concepts of data mining and self-organizing maps. We describe the implementation along with the experimental setup and environment in Section 2, followed by a discussion of the results in Section 3. Section 4 summarizes our findings and points to future work.

1. Data Mining and Self-Organizing Maps

1.1. Data Mining

Over the past two decades, the field of data mining as a tool to extract novel information from typically large datasets has matured to a point where it is used to answer a wide variety of questions in a large range of application areas. In order to do so, a range of techniques that can be categorized into predictive and descriptive techniques, are utilized. Predictive techniques include Decision Trees, Rule-Based Approaches and Support Vector Machines and develop a model from data containing predefined classes and class labels. The model is then used to predict class labels for previously unclassified data. In contrast, descriptive techniques such as k-means clustering and Expectation Maximization detect structure inherent in the data without the use of classes and labels.

One particular challenge in data mining (other than the need to analyze data sets on the order of Tera- and Petabytes) is its interactive nature, together with the wide range of available techniques. Typically, the model-building stage that uses a single particular technique follows a preprocessing step in which the data is transformed to the format expected by the selected technique (for example, numerical values only for Support Vector Machines). This means that if the learning technique is changed, the preprocessing step has to be modified and repeated. In addition, the data set itself is often refined iteratively. For all but small data sets, this interactive process results in prohibitively large run times. For that reason, the data-mining community has turned towards parallel implementations using a variety of environments, focusing most recently on accelerator architectures including GPUs and the CBEA.

1.2. Self-Organizing Maps

Self-organizing maps, originally proposed by Kohonen [7] are variants of Artificial Neural Networks, which themselves date back to the proposal by McCulloch and Pitts [11] that used artificial neurons as a basic building block in modeling a complex structure. A SOM is a typically two- or three-dimensional map of neurons that consist of weights (one for each dimension of the input data), which represents a higher-dimensional data space in fewer dimensions in order to employ human pattern recognition capabilities. During the training stage, the input data is presented to the map repeatedly and the winning neuron (the closest map neuron in terms of Euclidean distance) is determined, then updated along with its neighbourhood to more closely represent the input data. Both sequential and batch training algorithms are available. In a
sequential approach, the map updates are calculated based on the distance between the winning node and the input sample, according to the rule $m_i(t+1) = m_i(t) + \alpha h_{ci}(t)[x(t) - m_i(t)]$, where $m_i(t)$ denotes the $m_{th}$ neuron at iteration $t$, $\alpha$ the learning rate, $h_{ci}$ the neighborhood function, and $x$ the input vector. Over time, both the neighborhood function and the learning rate decrease. Convergence of the map occurs often over a substantial number of iterations through map updates calculated from the complete input data set. In contrast, the batch training algorithm replaces the node updates that are based on Euclidean distances with weighted averages of the input samples that are assigned to the map nodes. In both cases, the neighborhood function is often reduced to updating only the winning node. The computationally significant portion of the learning algorithm is the determination of the Euclidean distance between each input sample and all map nodes, while the required synchronization of map updates is the limiting factor for any distributed-memory implementation.

2. Experimental Setup and Environments
In this section, we describe the implementation of the SOMs and the environments under which it was evaluated. The data-parallel version using the GPU performs the calculations of the distance and map updates using the streaming processors, while the CPU is responsible for kernel calls and memory transfers. The map is accessed frequently, and, as a result, resides in shared memory of the GPU in our implementation. Each thread is responsible for calculation of the distance of one particular input datum to the map, determining the winning map node for its assigned input datum. The updates to the map therefore need to be atomic, which results in stalls for threads if two input data share a winning node. If more than one GPU is used, then the training dataset is distributed over the processors and partial map updates are collected and processed at the root, followed by a broadcast of the updated map at each iteration.

We used two different APIs that are currently candidates for a developing standard for General Purpose GPU programming: CUDA and OpenCL. CUDA has undergone nine releases as of the time of this writing and was designed by NVIDIA specifically for their GPUs. In contrast, OpenCL is designed for use across heterogeneous platforms, and is being developed by the Kronos Group, a not-for-profit industry consortium. Both use the concept of a kernel to represent a unit of code used for general purpose computation. The memory hierarchy model is conceptually similar in both, only differing slightly in the names of hierarchical levels. Allocation and management of device memory is similar in concept as well.

OpenCL mainly differs from CUDA in its approach to managing and executing instructions on the GPU. OpenCL employs a model which abstracts the program flow, logic and synchronization required for computation from device and vendor specific API calls used to carry out the computation. As such, OpenCL programs are required at run time to explicitly manage and determine which devices will be used. The choice of devices then dictates how the underlying interactions between the device and host will be carried out. Another major difference is the way in which OpenCL programs typically use a run-time compiler to produce kernel binaries, for a specified device, from raw strings of code. In contrast, CUDA kernels are compiled along with the main program for a specific target device/system. Compiled OpenCL kernels can be stored to disk and used in subsequent instances of the program. However, depending on the OpenCL implementation the kernel may be stored in an intermediate language (device specific) instead of executable binary.

We used two different environments for our evaluation: a Zotac GeForce GT 220 with 1GB DDR2 and the Sharcnet cluster Angel. The former is hosted by an AMD Athlon 64 X2 Dual Core Processor 5400+ with 2GB memory and a clock speed of 2.8 GHz. Angel consists of 11 NVIDIA Tesla S1070 GPU servers; each server contains 4 GPUs and 16GB global memory, with a total peak performance of over 345 GFlops (double precision). The GPU servers are connected via a DDR Infiniband and GigE copper network in a standard star topology with a single switch.
Figure 1. Runtime as a function of varying training set size on a single CPU. Shown are results for a map size of 64 nodes with 4 (o) and 8 (□) attributes) and 256 nodes and 4 attributes (x).

3. Results
In order to determine the scalability of our implementation, we present the following results. Section 3.1 shows how our implementation scales on a single GPU, using two different graphics cards in combination with OpenCL and CUDA. For comparison, we include results obtained from implementation on a CPU only. Section 3.2 discusses the run times and speedups for the SOM using MPI only, and up to eight processors. Finally, in Section 3.3, we combine MPI with CUDA to determine the scalability of the hybrid implementation, using up to 8 GPUs. For each section, the data sets range in size from 16 384 to 2 097 152 samples of simulated data, with 4 or 8 attributes, and a map size of either 64 or 256 nodes. All results are averaged over five trials, for 500 iterations.

3.1. Scalability on a Single GPU
Figure 1 shows the scalability of the SOM implementation on a single CPU, for varying training set sizes and map sizes, on a log-log plot. Results indicate that the implementation scales well with the number of samples and map size, but that the run times are prohibitive for interactive data mining beyond 10 000 samples. In addition, the confidence intervals are too small to show on the figure.

Next, we evaluate the scalability of our implementation using varying numbers of training samples and map sizes, as well as two graphics cards in combination with OpenCL and CUDA. Shown are results for a map size of 64 nodes with 4 (o) and 8 (□) attributes and 256 nodes and 4 attributes (x), for Figures 2 through 4.

Figure 2 shows a log-log plot of the run times for the CUDA implementation on a single GPU using the GT 220, for three different map sizes and varying size of training data up to 2 Million samples. In general, the algorithm scales close to linear with the training set size. For example, increasing the number of training samples by a factor of 128 resulted in a runtime increase of a factor of 125 for a map with 64 nodes, and eight attributes. Results for other training set sizes are similar, given a fixed map size. In addition, the implementation scales well with the map size (increasing the number of nodes in the map by a factor or four results in less than threefold increase in runtime), but less with the number of attributes of the training data and the map. That is, doubling the number of attributes for a given size of the map resulted in an increase by roughly a factor of three, for all training set sizes. We believe that the increase in memory transfers for updates of the larger map is responsible for this effect.
The speedups achieved using CUDA on the GT 220 vary for the different map sizes and increase slightly as the training size is increased, for a fixed map size. For example, the speedup increases from 11.14 to 11.39, for the largest map, as the size of the training data increases by a factor of $2^7$. The average speedups for the two smaller maps are 6.90 (4 attributes) and 4.79 (8 attributes), both with small standard deviations.

Figure 3 shows a log-log plot of the run times for the CUDA implementation on a single GPU on Angel, for three different map sizes and varying size of training data, up to 2 Million samples. While trends similar to those discussed for Figure 2 are visible, of particular interest is the fact that there is a substantial decrease in runtime in general, for all training set and map sizes, as compared to the results obtained by the GT 220. For example, in the smallest map, runtime reductions fall between 3.20 and 3.54, while for the largest map the runtime decrease by a factor that ranges from 4.67 to 5.91. The reductions in runtime for the CUDA implementation on Angel increase slightly with the size of the training data and are summarized in Tables 1 through 3.
Figure 4 shows a log-log plot of the run times for the OpenCL implementation on a single GPU on Angel, for three different map sizes and varying size of training data up to 2 Million samples. While the implementation again scales with the size of the training data, we observed substantial runtime penalties as compared to the implementation using CUDA, which were most prominent for the largest map. While the speedup is roughly reduced by a factor of two for the smaller maps, there is a four to five fold drop in performance, depending on the size of the training set, for the largest map.

Table 1. Speedups for OpenCL and CUDA using two GPUs as compared to the sequential implementation, for a map with 64 nodes (4 attributes) and varying sizes of training data

| Training Data | GT 220(CUDA) | Tesla(CUDA) | Tesla(OpenCL) |
|---------------|--------------|-------------|---------------|
| 16 384        | 6.62         | 21.21       | 8.17          |
| 32 768        | 6.80         | 23.01       | 9.37          |
| 65 536        | 6.88         | 24.06       | 9.69          |
| 131 072       | 6.95         | 24.12       | 9.60          |
| 262 144       | 6.97         | 25.03       | 9.85          |
| 524 288       | 6.98         | 24.53       | 9.75          |
| 1 048 576     | 6.98         | 24.76       | 9.74          |
| 2 097 152     | 6.99         | 24.71       | 9.73          |

Table 2. Speedups for OpenCL and CUDA using two GPUs as compared to the sequential implementation, for a map with 64 nodes (8 attributes) and varying sizes of training data

| Training Data | GT 220(CUDA) | Tesla(CUDA) | Tesla(OpenCL) |
|---------------|--------------|-------------|---------------|
| 16 384        | 4.72         | 22.77       | 11.05         |
| 32 768        | 4.77         | 23.83       | 11.36         |
| 65 536        | 4.80         | 24.24       | 11.37         |
| 131 072       | 4.80         | 24.61       | 11.58         |
| 262 144       | 4.81         | 24.10       | 11.45         |
| 524 288       | 4.82         | 24.27       | 11.41         |
| 1 048 576     | 4.81         | 24.23       | 11.44         |
| 2 097 152     | 4.82         | 24.38       | 11.46         |

Table 3. Speedups for OpenCL and CUDA using two GPUs as compared to the sequential implementation, for a map with 256 nodes (4 attributes) and varying sizes of training data

| Training Data | GT 220(CUDA) | Tesla(CUDA) | Tesla(OpenCL) |
|---------------|--------------|-------------|---------------|
| 16 384        | 11.14        | 52.01       | 12.91         |
| 32 768        | 11.25        | 63.83       | 13.46         |
| 65 536        | 11.33        | 66.16       | 13.87         |
| 131 072       | 11.36        | 66.87       | 13.87         |
| 262 144       | 11.38        | 67.30       | 13.91         |
| 524 288       | 11.38        | 67.06       | 13.90         |
| 1 048 576     | 11.39        | 67.05       | 13.87         |
| 2 097 152     | 11.39        | 67.37       | 13.86         |
3.2. MPI only

In this section, we discuss the results for the implementation using MPI only. We evaluate the implementation of the SOM using a single map size (256 nodes, 4 attributes) with varying numbers of training samples, up to 2 Million samples. Using Angel, we vary the number of processors from 1 to 8 over 500 iterations through the training data.

Figures 5 and 6 show the runtimes for the four smallest and largest datasets, respectively. The data depicted in the figures show that there is a trend towards better performance for larger datasets, as expected, where the costs for initial data distribution and communication of partial updates after each iteration are better amortized. The smaller datasets also show more variation in the runtimes for each of the trials than the larger datasets, which is caused by the network characteristics. In all cases, the confidence intervals are too small to show on the figures.

Table 4 shows the attained speedups for the MPI-only version of the Self-Organizing Map. The maximum speedup is achieved by the largest data set, with a speedup of 7.83 for eight processors. With few exceptions, we observe that, as expected, the speedup grows consistently with the training set size, for a given number of processors. For a fixed-size dataset, the speedup

![Figure 5. Runtimes for the Four Smallest Datasets: 16 384 (o), 32 768 (∇), 65 536 (x), and 131 072 (+) samples.](image1)

![Figure 6. Runtimes for the Four Largest Datasets: 262 144 (o), 524 288 (∇), 1 048 576 (x), and 2 097 152 (+) samples.](image2)

| Training Data | 2   | 3   | 4   | 5   | 6   | 7   | 8   |
|---------------|-----|-----|-----|-----|-----|-----|-----|
| 16 384        | 1.99| 2.99| 3.97| 1.27| 1.38| 3.91| 6.45|
| 32 768        | 2.00| 2.53| 3.97| 4.69| 1.27| 6.66| 7.11|
| 65 536        | 2.00| 2.98| 3.97| 4.53| 3.11| 6.45| 7.62|
| 121 072       | 1.99| 2.32| 3.97| 4.94| 5.65| 6.60| 5.41|
| 262 144       | 1.99| 2.51| 3.90| 4.47| 5.63| 6.58| 6.30|
| 524 288       | 1.94| 2.98| 3.89| 4.94| 5.93| 6.81| 7.52|
| 1 048 576     | 1.99| 2.99| 3.97| 4.76| 5.93| 6.78| 7.51|
| 2 097 152     | 2.00| 2.99| 3.97| 4.95| 5.91| 6.90| 7.83|

Table 4 shows the attained speedups for the MPI-only version of the Self-Organizing Map. The maximum speedup is achieved by the largest data set, with a speedup of 7.83 for eight processors. With few exceptions, we observe that, as expected, the speedup grows consistently with the training set size, for a given number of processors. For a fixed-size dataset, the speedup
grows close to linear with the number of processors, which is especially evident for the larger datasets.

3.3. Multiple GPUs
Finally, we evaluated the performance of the implementation using CUDA in combination with MPI. Figures 7 through 9 show the run times of the hybrid code, for various map sizes and 500 iterations. Shown are the results for 262 144 (*), 524 288 (o), 1 048 576 (□), and 2 097 152 (x) samples. We observe the following trends: For the larger datasets, our implementation scales fairly well with the number of processors when combining MPI with CUDA, for all map sizes, as compared to the results obtained with a single GPU. The efficiencies are highest for two GPUs, but then drop off consistently. However, for the three largest datasets, the speedup as compared to using a single GPU does not drop below for a value of 7, for eight processors. The overhead in our implementation stems in part from the initial distribution of the data, but, more importantly, also from the required synchronization of the map updates.

The achieved speedup increases with the size of the training set, for fixed number of processors and map size; the largest performance improvement when combining multiple GPUs was 7.55
and occurred for the largest training set and map size. For all combinations of number of processors and size of training data, the speedup increases when increasing the dimensionality of the data from 4 to 8 attributes. In general, the increase in speedup when increasing the number of attributes is less than 10%, with a few exceptions. The largest increase in speedup when increasing the dimensionality of the data was observed for 32 768 samples and 7 processors, with a performance improvement of 40%. In addition, there is no general trend towards either a performance drop or improvement when increasing map size from 64 to 256 nodes. Overall, the speedup achieved as compared to the sequential version when combining CUDA with MPI ranges from 3.30 (single GPU, smallest dataset) to 32.45 (8 GPUs, second-largest data set).

For comparison, Table 5 shows the relative speedup when adding CUDA to the MPI implementation of the Self-Organizing Map, for different training set sizes and number of processors. The low speedups are similar to those found by Lee et al. [9], who compared GPU implementations to optimized sequential code. With the exception of the second largest data set for 7 processors, all performance gains are below a factor of 5. This relatively poor performance of the combination of MPI and CUDA stems from the relatively low computational requirements of the learning algorithm at each iteration, combined with the overhead at each iteration.

Table 5. Relative Speedups for Distributed Version With and Without CUDA for Increasing Training Data Size and Number of Processors, for a map size of 256 nodes (4 attributes) and 16 384 to 2 097 152 samples (shown in rows), using up to 8 processors (shown in columns).

| Training Data | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
|---------------|----|----|----|----|----|----|----|----|
| 16 384        | 3.30 | 2.66 | 2.45 | 2.07 | 6.82 | 6.48 | 2.35 | 1.40 |
| 32 768        | 3.70 | 3.23 | 2.49 | 2.69 | 2.65 | 10.90 | 2.18 | 2.15 |
| 65 536        | 4.79 | 3.31 | 3.18 | 3.10 | 2.74 | 3.60 | 1.79 | 2.40 |
| 131 072       | 4.18 | 3.99 | 4.45 | 3.88 | 2.73 | 3.30 | 2.59 | 4.60 |
| 262 144       | 4.16 | 4.06 | 4.57 | 4.27 | 3.91 | 3.43 | 3.55 | 4.43 |
| 524 288       | 4.22 | 4.36 | 3.99 | 4.30 | 4.09 | 3.84 | 2.84 | 4.01 |
| 1 048 576     | 4.30 | 4.21 | 3.94 | 4.13 | 4.36 | 3.82 | 3.89 | 4.32 |
| 2 097 152     | 4.50 | 4.43 | 4.03 | 4.39 | 4.31 | 4.09 | 4.20 | 4.06 |

4. Conclusions and Future Work
We implemented and evaluated a Self-Organizing Map on the Sharcnet GPU cluster Angel. An MPI-only version scaled well with the number of processors. As well, the GPU-only version, implemented using CUDA and OpenCL and evaluated using two different graphics cards, achieved speedups of up to 67 on the Tesla GPU server. However, the hybrid MPI/CUDA version showed an increase in speedup of roughly a factor of four only, caused mainly by the synchronization requirements of the algorithm. One possible avenue for future work would be the use of GPU Direct using CUDA 4, which allows memory access to remote GPUs from within kernels but was not supported by the cluster we used. We also observed a performance penalty for the OpenCL implementation as compared to CUDA, and therefore prefer the use of the latter. Moreover, OpenCL is designed for a more heterogeneous set of devices, as opposed to CUDA’s GPU-specific approach. Future work will include the implementation of additional data-mining algorithms on the GPU cluster, as well as the evaluation of pre-processing techniques required by various learning techniques, with the overall goal of providing a data-mining toolkit for GPU clusters based on both CUDA and OpenCL.
Acknowledgments
Acknowledgments. This work was made possible by a Fellowship Award from the Shared Hierarchical Academic Research Computing Network (SHARCNET:www.sharcnet.ca).

References
[1] Abdelaal, M., and Farouq, M., Using data mining for assessing diagnosis of breast cancer, Proceedings of the International Multiconference on Computer Science and Information Technology pp. 1117
[2] Bose, I., Data Mining in Diabetes Diagnosis and Detection, Data Warehousing and Mining: Concepts, Methodologies, Tools, and Applications, J. Wang (Ed.), pp. 1817-1824, 2008
[3] Brameier, M. and Banzhaf, W., A comparison of linear genetic programming and neural networks in medical data mining, IEEE Transactions on Evolutionary Computation, 5(1), pp. 17-26. Feb 2001
[4] Flietstra, B., A Data Mining Approach for Acoustic Diagnosis of Cardiopulmonary Disease, MSc Thesis, Massachusetts Institute of Technology, June 2008
[5] Geliso L., Azanza Ricardo C. L., Leoni, M. and Scardi, P., Real-space calculation of powder diffraction patterns on graphicsprocessing units, Journal of Applied Crystallography 43:647-653, 2010.
[6] Grady, L., Schiwietz, T., and Ahron, S., Random walks for interactive organ segmentation in two and three dimensions: Implementation and validation, Proceedings of MICCAI 2005 II, 2005
[7] Kohonen, T.: The Self-Organizing Map. Proceedings of the IEEE, Vol. 78(9), pp. 1551-1558.
[8] Kumar, N., Satoor, S. and Buck, I., Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs Using CUDA, 11th IEEE International Conference on High Performance Computing and Communications (HPCC 09), pp. 103-109, 25-27 June 2009
[9] Lee, V. et al., Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU, in ISCA10:Proceedings of the 37th Annual International Symposium on Computer Architecture. New York, NY, USA: ACM, 2010, pp. 451460.
[10] Liao, Q., Wang, J., Webster, Y., and Watson, IA., Journal of Chemical Information and Modeling, 2009 Dec; 49(12):2718-25.
[11] McCulloch, W. and Pitts, W., A logical calculus of the ideas immanent in nervous activity Bulletin of Mathematical Biophysics 5:115-133, 1943
[12] Ribeiro, B., Lopes, N., and Silva, C., High-Performance Bankruptcy Prediction Model using Graphics Processing Units, IEEEWCCI 2010 IEEE World Congress on Computational Intelligence July, 18-23, 2010 - CCIB, Barcelona, Spain
[13] Singh, J. and Aruni, I., Accelerating Power Flow studies on Graphics Processing Unit, Proceedings of the Annual India Conference 2010 (INDICON), pp 1-5, Dec. 2010
[14] Verner U., Schuster A., and Silberstein, M., Processing data streams with hard real-time constraints on heterogeneous systems, ICS11, to appear