ESANN 2012 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 25-27 April 2012, i6doc.com publ., ISBN 978-2-87419-049-0.
Available from http://www.i6doc.com/en/livre/?GCOI=28001100967420.

A GPU-Accelerated Algorithm for Self-Organizing Maps in a Distributed Environment

Peter Wittek and Sándor Darányi *

Swedish School of Library and Information Science
University of Borås
Borås, Sweden
Email: peterwittek@acm.org,sandor.daranyi@hb.se

Abstract. In this paper we introduce a MapReduce-based implementation of self-organizing maps that performs compute-bound operations on distributed GPUs. The kernels are optimized to ensure coalesced memory access and effective use of shared memory. We have performed extensive tests of our algorithms on a cluster of eight nodes with two NVidia Tesla M2050 attached to each, and we achieve a 10x speedup for self-organizing maps over a distributed CPU algorithm.

1 Introduction

While recent advances in the programmability of GPUs have made graphics hardware more accessible for general-purpose computations, the process of developing efficient GPU implementations is still highly application-dependent. Many applications require significant re-structuring of the algorithms to realize high performance on GPUs, and the problem is even more convoluted if the GPUs are distributed across several computing nodes. In data-intensive tasks, MapReduce is a popular paradigm to distribute load and enable quick code development [1]. There have been some attempts to use MapReduce in the compute-intensive workloads that are typical to GPU-accelerated algorithms. We leverage on these to speed up a demanding neural network method, self-organizing maps [2] in a distributed GPU environment.

2 GPU-aware MapReduce

GPUs excel at compute-bound operations, but it is not always easy to formulate a problem in the data parallel fashion that is required by lower level frameworks to program GPUs. The core concept of GPGPU computing is the kernel. A kernel is a unit of code that will execute on a graphics device launched from the host CPU. The execution is asynchronous in most cases: the host may continue working on other tasks while the kernel is being executed. The graphics device cannot access the main memory, the data has to be copied to the memory of the device through the system bus. This is a costly operation, and if a sufficient level of parallelism cannot be achieved in the kernel, GPU-based computation

*This work was supported by Amazon Web Services.

ESANN 2012 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 25-27 April 2012, i6doc.com publ., ISBN 978-2-87419-049-0.
Available from http://www.i6doc.com/en/livre/?GCOI=28001100967420.

may decrease the overall performance due to this overhead [3]. There are several research attempts that port MapReduce to GPUs to help develop applications faster on this kind of hardware.

Mars was the first framework to program a GPU with the MapReduce paradigm [4]. While it did show good potential compared to a CPU-based MapReduce, it does not utilize the GPU efficiently. It has not been updated for a long time and it is not capable of using more than one GPU.

Inspired by Mars, GPMR is the latest attempt to harness the power of GPUs with MapReduce [5]. It can use any number of GPUs in a node and it is also capable of running in a distributed system. Both features are provided by MPI. The performance efficiency declines after about 16 GPUs in most of the tests. GPMR does not try to hide the complexity of GPU programming, and it allows full control over the individual GPUs. It is a compromise made to achieve maximum performance. The default MapReduce scheduler moves the data from the GPU to the main memory after each map step, and then pushes it back before reduction (if needed), which might be inefficient in certain applications.

Taking the approach of GPMR further, the MapReduce and GPU parts of an algorithm can be entirely decomposed. This way, for instance, one can use MR-MPI [6], which is a lightweight MapReduce framework built with MPI, or Hadoop [7], which is a more complex framework that also incorporates a distributed filesystem. These frameworks are good for a high-level load distribution and one can use GPU code only inside the map or reduce jobs. This is the approach we have taken.

3 Self-organizing maps

The self-organizing map (SOM) training algorithm constructs a nonlinear and topology preserving mapping of the input data set $X = \{x(t)|t \in \{t_{0,\ldots,t_f}\}\}$, where t_0 and t_f are the beginning and the end of the current epoch, onto a set of neurons $M = n_1, \ldots, n_k$ of a neural network with associated weight vectors $W = w_1(t), ..., w_k(t)$ at a given time step t. Each data point $x(t)$ is mapped to its best match neuron $\mathbf{bm}(x(t)) = n_b \in M$ such that $d(x(t), w_b(t)) \leq d(x(t), w_j(t)) \quad \forall w_j(t) \in W$, where d is the distance on the data set. The neurons are arranged on a two dimensional map: each neuron i possesses a set of two coordinates embedded in a two dimensional surface. Next the weight vector of the best match neuron and its neighbours are adjusted toward the input pattern using the following equation: $w_j(t+1) = w_j(t) + \alpha h_{bj}(t)[x(t) - w_j(t)]$, where $0 < \alpha < 1$ is the learning factor, and $h_{ck}(t)$ is the neighbourhood function that decreases for neurons further away from the best match neuron in grid coordinates. A frequently used neighbourhood function is the Gaussian: $h_{bj} = \exp(\frac{-||r_b - r_j||}{\delta(t)})$, where r_k and r_c stand for the coordinates of the respective nodes. The width $\delta(t)$ decreases from iteration to iteration to narrow the area of influence. It is assumed, that the SOM gives a mapping with minimal, or at least tolerable, topological errors [8]. The training might be repeated again on the same data set to increase the fit, a training cycle is referred to as an epoch.

ESANN 2012 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 25-27 April 2012, i6doc.com publ., ISBN 978-2-87419-049-0.
Available from http://www.i6doc.com/en/livre/?GCOI=28001100967420.

Eventually, the neighbourhood function decreases to an extent that training might stop. The time needed to train an SOM grows linearly with the dataset size and it also grows linearly with the number of neurons in the SOM.

4 Acceleration of self-organizing maps

A distributed, MapReduce-based SOM builds on the batch formulation of updating the weights [9]:

$$w_j(t_f) = \frac{\sum_{t'=t_0}^{t_f} h_{bj}(t')x(t')}{\sum_{t'=t_0}^{t_f} h_{bj}(t')}. \quad (1)$$

This implementation builds on MR-MPI. The key idea is that before the data set is partitioned by a map call, the current set of weight vectors is broadcast to all worker nodes. The update in Equation 1 is performed locally at each node, and the reduce step sums up the updates, and eventually the master nodes set the values of the new weight vectors. The process repeats with subsequent epochs.

We extended the MapReduce SOM algorithm by moving all calculations on local nodes to the GPU with the matrix-based Euclidean distance matrix and reduction algorithm described below. The chunk of X assigned to the node and the corresponding norms of X are kept in the GPU memory between subsequent epochs, and the weight vectors are copied to the GPU memory in each step after the node receives the broadcast of the update.

The most time-consuming part of the calculations is finding the best matching neuron. This involves calculating the pairwise distances between the elements of the data sets and the weight vectors. Euclidean distance is one of the most common distances used in SOMs. The distance between a data point and a weight vector in the batch formulation can be calculated by $d(w_j(t_0), x(t)) = \sqrt{\sum_{i=1}^{N}(x_i(t) - w_{ji}(t_0))^2}$, where N is the dimension of the space that embeds the data set, and $t \in \{t_0, \ldots, t_f\}$. The pairwise distances can be efficiently computed on a GPU if the distances are calculated on the same space [10]. The above formula is more general, and a much higher performance can be achieved if the entire distance matrix is calculated, and the steps are decomposed into matrix level operations (see Algorithm 1 [11, 12]).

Algorithm 1 Calculate a Euclidean distance matrix with matrix operations ($\circ$ is the Hadamard product)

1: $v_1 = (X \circ X)[1, 1 \ldots 1]'$
2: $v_2 = (W \circ W)[1, 1 \ldots 1]'$
3: $P_1 = [v_1 v_1 \ldots v_1]$
4: $P_2 = [v_2 v_2 \ldots v_2]'$
5: $P_3 = XW'$
6: $D = (P_1 + P_2 - 2P_3)$

ESANN 2012 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 25-27 April 2012, i6doc.com publ., ISBN 978-2-87419-049-0.
Available from http://www.i6doc.com/en/livre/?GCOI=28001100967420.

The algorithm does not calculate the Euclidean distances, but the square of the distances. Taking the square root is an expensive operation on the GPU, and since we seek the minimum of the distances, we can omit calculating it. The experimental results in [11] have shown a speed up of 15x on datasets which contain more than half million data points with the above algorithm on the GPU.

An important point to note is that as the norms of X do not change between subsequent epochs, these values can be computed before the main training loop starts. Step 5 is a BLAS matrix multiplication, which can be calculated on the GPU with a high-level CUBLAS call. The CUBLAS library is distributed with CUDA, and it may not be the fastest implementation at a given time, but it gives an optimized performance [13].

The other steps have to be implemented carefully to minimize global memory access and avoid bank-conflicts in the shared memory of SMPs. This is achieved by using a column-major representation of the matrices. The minimum is found by a multi-step reduction algorithm. Existing GPU-based SOM algorithms take a similar approach in finding the best matching unit, but they do not necessarily use matrix operations to derive the distance matrix [14, 15].

5 Discussion of experimental results

5.1 Cluster Configuration

We built a Beowulf cluster of eight nodes using Amazon Web Services (AWS). AWS ensures that cluster instances that are launched simultaneously are physically close and they are connected with a high-speed network. A cluster compute GPU instance in AWS has two Intel Xeon X5570 quad-core CPUs and 23 GB of memory, and it is equipped with two NVidia Tesla M2050. One Tesla card has 448 CUDA cores and 3GByte of device memory.

The implementation of the algorithms[1], except for creating the inverted index, used OpenMPI 1.5, the June 2011 version of MR-MPI, and CUDA 4.0. The inverted index was created using Lucene 3.4 and was not timed.

5.2 Data set

The collection consists of 84,283 PhD theses crawled from the database of the German Nationaly Library. The files are in PDF format and the total size is 498GByte. File sizes vary widely, averaging around 6Mbytes, but many files are over 100MBytes. The collection is multilingual, with the majority of documents being in English or German.

We created an inverted index with Lucene, applying PDF extraction. We did not use stemming or any other language specific processing step. The merged and optimized index was approximately 11GByte, with a total of 34,965,200 terms. We applied random projection [16] to reduce the number of dimensions to two hundred.

[1]The source code is available online: `https://github.com/peterwittek/mr-mpi-som-gpu`

ESANN 2012 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 25-27 April 2012, i6doc.com publ., ISBN 978-2-87419-049-0.
Available from http://www.i6doc.com/en/livre/?GCOI=28001100967420.

Method	Execution time	Speedup over CPU
CPU (64 cores)	2042s	-
GPU (16 Tesla)	433s	4.71x
CPU (64 cores) (One epoch)	1882s	-
GPU (16 Tesla) (One epoch)	194s	9.68x

Table 1: Execution time of self-organizing maps

5.3 Running time of self-organizing maps

The baseline distributed, multicore CPU implementation was based on MR-MPI-SOM [9]. We used all sixty-four physical cores across the eight nodes. When scaling up, we noticed a nearly linear scaling, which shows that the overhead of broadcasting the SOM weight vectors is not considerable.

We had to use all sixteen GPUs to fit the problem in the distributed device memories. Even then, the tensor describing the SOM at a given time was a serious limiting factor, and we had to limit our experiments to a small 10x10 map in both the CPU and GPU cases. The bulk of the memory is taken up by the dense matrix chunk that is assigned to a GPU. Since such a tiny map does not show emergent clustering features very well, in the future we intend to work on a variant of random projection that keeps the resulting structure sparse.

The total wall time of the CPU variant was 2041.71 seconds (Table 1). The wall time of the GPU implementation was 433.25 seconds, including initializing the CUDA context, and memory transfers before and after the SOM training. This translates to a speedup of 4.71x. Considering that this is a speedup over sixty-four cores of CPUs, we believe that this is a significant result.

Turning our attention to the execution time of one epoch, a CPU core finished its chunk in 1881.71 seconds. The GPU variant took 194.54 seconds, which means a speedup of 9.68x. Note that we only calculated the best matching units on the GPU, so there is room for further improvement.

Looking at the GPU occupancy results, we were not far from fully loading the stream processors. Steps 1 and 2 of Algorithm 1, that is, calculating the norms, were performed with 100% occupancy. Finding the minimum is also optimal. The CUBLAS dense matrix multiplication proved to be the weak point, with an occupancy varying between 33% and 83%.

6 Conclusions

MapReduce jobs are fairly easy to develop, a wide range of machine learning algorithms has already been adapted to this paradigm, hence developers of data mining applications will find it convenient to use. With rephrasing the compu-

ESANN 2012 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 25-27 April 2012, i6doc.com publ., ISBN 978-2-87419-049-0.
Available from http://www.i6doc.com/en/livre/?GCOI=28001100967420.

tational problems underlying self-organizing maps, we showed that it is feasible to leverage on GPUs in a distributed system, and achieved a speedup of close to 10x.

References

[1] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI-04, 6th International Symposium on Operating Systems Design & Implementation*, San Francisco, CA, USA, December 2004.

[2] T. Kohonen, S. Kaski, K. Lagus, J. Salojärvi, J. Honkela, V. Paatero, and A. Saarela. Self organization of a massive text document collection. *IEEE Transactions on Neural Networks*, 11(3):574–585, 2000.

[3] NVida Compute Unified Device Architecture C Best Practices Guide 4.0, 2011.

[4] B. He, W. Fang, Q. Luo, N.K. Govindaraju, and T. Wang. Mars: A MapReduce framework on graphics processors. In *Proceedings of PACT-08, 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269, Toronto, ON, Canada, October 2008.

[5] J.A. Stuart and J.D. Owens. Multi-GPU MapReduce on GPU clusters. In *Proceedings of IPDPS-11, 25th International Parallel and Distributed Computing Symposium*, Anchorage, AK, USA, May 2011.

[6] S.J. Plimpton and K.D. Devine. MapReduce in MPI for large-scale graph algorithms. *Parallel Computing*, 2011.

[7] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[8] T. Kohonen. *Self-Organizing Maps*. 2001.

[9] S.J. Sul and A. Tovchigrechko. Parallelizing BLAST and SOM algorithms with MapReduce-MPI library. In *Proceedings of IPDPS-11, 25th International Parallel and Distributed Computing Symposium*, pages 476–483, Anchorage, AK, USA, May 2011.

[10] D. Chang, N.A. Jones, D. Li, M. Ouyang, and R.K. Ragade. Compute pairwise Euclidean distances of data points with GPUs. In *Proceedings of CBB-08, International Symposium on Computational Biology and Bioinformatics*, pages 278–283, Orlando, FL, USA, November 2008. ACTA Press.

[11] Q. Li, V. Kecman, and R. Salman. A chunking method for Euclidean distance matrix calculation on large dataset using multi-GPU. In *Proceedings of ICMLA-10, 9th International Conference on Machine Learning and Applications*, pages 208–213, Washington, DC, USA, December 2010.

[12] K.E.A. van de Sande, T. Gevers, and C.G.M. Snoek. Empowering visual categorization with the GPU. *IEEE Transactions on Multimedia*, 13(1):60–70, 2011.

[13] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, and E.S. Quintana-Orti. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *Proceedings of IPDPS-08, 22nd International Symposium on Parallel and Distributed Processing*, pages 1–8, Miami, FL, USA, April 2008.

[14] K.S. Oh and K. Jung. GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[15] G. Strong and M. Gong. Browsing a large collection of community photos based on similarity on GPU. *Advances in Visual Computing*, pages 390–399, 2008.

[16] P. Kanerva, J. Kristofersson, and A. Holst. Random indexing of text samples for latent semantic analysis. In *Proceedings of CogSci-00, 22nd Annual Conference of the Cognitive Science Society*, volume 1036, Philadelphia, PA, USA, 2000.