Multicore Processing for Classification and Clustering Algorithms

V. Vaitheeshwaran
School of Computing,
KL University Vijayawada, India

Kapil Kumar Nagwanshi
School of Computing,
KL University Vijayawada, India

T. V. Rao
School of Computing,
KL University Vijayawada, India

ABSTRACT

Data Mining algorithms such as classification and clustering are the future of computation, though multidimensional data-processing is required. People are using multicore processors with GPU's. Most of the programming languages doesn't provide multiprocessing facilities and hence wastage of processing resources. Clustering and classification algorithms are more resource consuming. In this paper we have shown strategies to overcome such deficiencies using multicore processing platform OpenCL.

Keywords

Parallel Processing, Clustering, Classification, OpenCL, CUDA, NVIDIA, AMD, GPU.

1. INTRODUCTION

CLUSTERING is an unsupervised learning technique that separates data items into a number of groups, such that items in the same cluster are more similar to each other and items in different clusters tend to be dissimilar, according to some measure of similarity or proximity. Pizzuti and Talia[1]presents a P-AutoClass technique for Scalable Parallel Clustering for Mining Large Data Sets Data clustering is an important task in the area of data mining. Clustering is the unsupervised classification of data items into homogeneous groups called clusters. Clustering methods partition a set of data items into clusters, such that items in the same cluster are more similar to each other than items in different clusters according to some defined criteria. Clustering algorithms are computationally intensive, particularly when they are used to analyze large amounts of data. A possible approach to reduce the processing time is based on the implementation of clustering algorithms on scalable parallel computers. This paper describes the design and implementation of P-AutoClass, a parallel version of the AutoClass system based upon the Bayesian model for determining optimal classes in large data sets. The P-AutoClass implementation divides the clustering task among the processors of a multicomputer so that each processor works on its own partition and exchanges intermediate results with the other processors. The system architecture, its implementation, and experimental performance results on different processor numbers and data sets are presented and compared with theoretical performance. In particular, experimental and predicted scalability and efficiency of P-AutoClass versus the sequential AutoClass system are evaluated and compared.

Different from supervised learning, where training examples are associated with a class label that expresses the membership of every example to a class, clustering assumes no information about the distribution of the objects and it has the task to both discover the classes present in the data set and to assign objects among such classes in the best way. A large number of clustering methods have been developed in several different fields, with different definitions of clusters and similarity among objects. The variety of clustering techniques is reflected by the variety of terms used for cluster analysis such as clumping, competitive learning, unsupervised pattern recognition, vector quantization, partitioning, and winner-take-all learning.

Most of the early cluster analysis algorithms come from the area of statistics and have been originally designed for relatively small data sets. Fayyad et al [2], found that the clustering algorithms have been extended to efficiently work for knowledge discovery in large databases and, therefore, to classify large data sets with high-dimensional feature items. Clustering algorithms are very computing demanding and, thus, require high-performance machines to get results in a reasonable amount of time. HunterandStates[3] gives classification algorithm on protein databases and experiences of clustering algorithms taking one week or about 20 days of computation time on sequential machines are not rare. Scalable parallel computers can provide the appropriate setting where to efficiently execute clustering algorithms for extracting knowledge from large-scale databases and, recently, there has been an increasing interest in parallel implementations of data clustering algorithms. There are variety of parallel approaches to clustering has been discovered by[4],[5],[6],[7].

Classification: Classification is one of the primary data mining tasks[8]. The input to a classification system consists of example tuples, called a *training set*, with each tuple having several *attributes*. Attributes can be *continuous*, coming from an ordered domain, or *categorical*, coming from an unordered domain. A special *class* attribute indicates the label or category to which an example belongs. The goal of classification is to induce a model from the training set, that can be used to predict the class of a new tuple. Classification has applications in diverse fields such as retail target marketing, fraud detection, and medical diagnosis (Michie, 1994). Amongst many classification methods proposed over the years [9][10]decision trees are particularly suited for data mining, since they can be built relatively fast compared to other methods and they are easy to interpret [11]. Trees can also be converted into SQL statements that can be used to access databases efficiently [12]. Finally, decision-tree classifiers obtain similar, and often better, accuracy compared to other methods [10].

Prior to interest in classification for database-centric data mining, it was tacitly assumed that the training sets could fit in memory. Recent work has targeted the massive training sets usual in data mining. Developing classification models using larger training sets can enable the development of higher accuracy models. Various studies have confirmed this[13]. Recent classifiers that can handle disk-resident data include SLIQ [14], SPRINT [15], and CLOUDS[16]. As data continue to grow in size and complexity, high performance scalable data mining tools must necessarily rely on parallel computing techniques. Past research on parallel classification has been focused on distributed-memory (also called shared-nothing) machines. Examples include parallel ID3 [17], which assumed that the entire dataset could fit in memory; Darwin toolkit with parallel CART [18]from Thinking Machine, whose details are not available in published literature; parallel SPRINT on IBM SP2 [15]; and ScalParC[19]on a Cray T3D. While distributed-memory machines provide massive parallelism, shared-memory machines (also called shared everything systems), are also capable of delivering high performance for low to medium degree of parallelismat an economically attractive price. Increasingly SMP machines arebeing networked together via high-speed links to form hierarchical clusters. Examples include the SGI Origin 2000and IBM SP2 system which can have a 8-way SMP as one *high* node. A shared-memory system offers a single memory address space that all processors can access. Processors communicate through shared variables in memory. Synchronization is used to co-ordinate processes. Any processor can also access any disk attached to the system. The SMP architecture offers new challenges and trade-offs that are worth investigating in their own right.

2. THE GPU ARCHITECTURE

All Initially intended as a fixed many-core processor dedicated to transforming 3-D scenes to a 2-D image composed of pixels, the GPU architecture has undergone several innovations to meet the computationally demanding needs of supercomputing research groups across the globe. The traditional GPU pipeline designed to serve its original purpose came with several disadvantages. Shortcomings such as the v limited data reuse in the pipeline, excessive variations in hardware usage, and lack of integer instructions coupled with weak floating-point precision rendered the traditional GPU a weak candidate for HPC. In November 2006 [20], NVIDIA introduced the GeForce 8800 GTX with a novel unified pipeline and shader architecture. In addition to overcoming the limitations of the traditional GPU pipeline, the GeForce 8800 GTX architecture added the concept of *streaming processor (SMP) architecture* that is highly pertinent to current GP-GPU programming. SMPs can work together in close proximity with extremely high parallel processing power. The outputs produced can be stored in fast cache and can be used by other SMPs. SMPs have instruction decoder units and execution logic performing similar operations on the data. This architecture allows SIMD instructions to be efficiently mapped across groups of SMPs. The streaming processors are accompanied by units for texture fetch (TF), texture addressing (TA), and caches. The structure is maintained and scaled up to 128 SMPs in GeForce 8800 GTX. The SMPs operate at 2.35 GHz in the GeForce 8800 GTX, which is separate from core clock operating at 575 MHz. Several GP-GPUs used thus far for HPC applications have architectures that are concurrent with the GeForce 8800 GTX architecture. However, the introduction of the Fermi by Nvidia in September 2009 [21]has radically changed the contours of the GP-GPU architecture, as we will explore in the next subsection.

GPU's amazing evolution on both computational capability and functionality extends application of GPUs to the field of non-graphics computations, which is so-called *general purpose computation on GPUs* (GPGPU) [22]. Design and development of GPGPU are becoming significant because of the following reasons:

1. *Cost-performance:* Using only commodity hardware is important to achieve high-performance computing at a low cost, and GPUs have become commonplace even in low-end PCs. Due to the hardware architecture designed for exploiting parallelism of graphics, even today's low-end GPU exhibits high-performance for data-parallel computing. In addition, GPU has much higher sequential memory access performance than CPU because one of GPU's key tasks is filling regions of memory with contiguous texture data. That is, GPU's dedicated memory can provide data to GPU's processing units at the high memory bandwidth.
2. *Evolution speed:* GPU's performance such as the number of floating-point operations per second has been growing at a rapid pace. Amazingly-evolving GPU capabilities have a possibility to enable the GPU implementation of a task to outperform its CPU implementation in the future. Modern GPUs have two kinds of programmable processors, *vertex shader*and*fragment shader*, on the graphics pipeline to render an image.

Figure 1 illustrates a block diagram of the programmable rendering pipeline of these processors. The vertex shader manipulates transformation and lighting of vertices of polygons to transform them into the viewing coordinate system. Polygons projected into the viewing coordinate system are then decomposed into fragments each corresponding to a pixel on the screen. Subsequently, the color and depth of a fragment are computed by the fragment shader. Finally, composition operations such as tests using depth, alpha and stencil buffers are applied to the outputs of the fragment shader to determine the final pixel colors to be written to the frame buffer.

It is emphasized here that vertex and fragment shaders are developed to utilize multi-grain parallelism in the rendering processes: the coarse-grain vertex/fragment level parallelism and the fine-grain vector component level parallelism. To exploit the coarse-grain parallelism at the GPU level, individual vertices and fragments can be processed in parallel. The fragment shaders (vertex shaders) of recent GPUs have several processing units for parallel-processing multiple fragments (vertices). For example, NVIDIA's high-end GPU, GeForce 6800 Ultra, has 16 processing units in the fragment shader, and therefore can compute colors and depths of up to 16 fragments at the same time. On the other hand, to exploit the fine-grain parallelism involved in all vector operations, they have SIMD instructions that can simultaneously operate on four 32-bit floating-point values within a 128-bit register. For example, one of the powerful SIMD instructions, the "multiply and add" (MAD) instruction, performs a component-wise multiply of two registers each storing four floating-point components, and then does a component-wise addition of the product to another register; the MAD instruction performs these eight floating-point operations in a single cycle. Due to its application-specific architecture, however, GPU does not work well universally. To exhibit high performance for a non-graphics application, hence, we ought to consider how to bind it to GPU's programmable rendering pipeline.

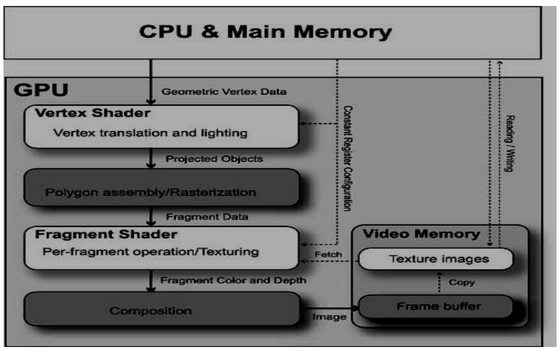

Fig 1 Overview of a programmable rendering pipeline

The most critical restriction in GPU programming for non-graphics applications is due to the restricted data flows in and between the vertex shader and the fragment shader. Arrows in Fig. 1 show typically-permitted data flows. Both vertex and fragment shader programs have to write their outputs to write-only dedicated registers; random access writes are not provided. This is severe impediment to effective implementation of many data structures and algorithms. In addition, the lack of loop-controls, conditionals, and branching is also serious for most of practical applications. Although the latest GPUs with Shader Model 3.0 support dynamic controls flows, there is some overhead to flow-control operations, and they can limit the GPU's performance. If an application imposes the restriction violation on the GPU programming model mentioned above, it is not a good idea to implement the application entirely on GPUs. For such an application, collaboration between CPU and GPU often leads to better performance, though it is essential to keep time-consuming data transfer between them minimum. From the viewpoint of data accessibility, the fragment shader is superior to the vertex shader because the fragment shader can randomly access the video memory and fetch data as texture colors.1 Furthermore, the fragment shader usually has more processing units than the vertex shader, and thereby the fragment shader has a great potential to exploit dataparallelism more effectively. Consequently, this paper presents an implementation of data clustering accelerated effectively using multi-grain parallel processing on the fragment shader.

2.1 GPU Computing with AMD/ATi Radeon 5870

The AMD/ATi's Radeon 5870 architecture [23]is very different compared to NVIDIA's Fermi architecture. The AMD/ATi Radeon 5870 used in our study has 1600 ALUs organized in a different fashion compared to the Fermi. The ALUs are grouped into five-ALU Very Long Instruction Word (VLIW) processor units. While all five of the ALUs can issue the basic arithmetic operations, only the fifth ALU can additionally execute transcendental operations. The five-ALU groups along with the branch execution unit and general-purpose registers form another group called the *stream core*. This translates to 320 stream cores in all, which are further grouped into *compute units*. Each compute unit has 16 stream cores, resulting in 20 total compute units in the ATi Radeon 5870. One thread can be executed on one stream core, thus 16 threads can be run on a single compute unit. In order to hide the memory latency, 64 threads are assigned to a single compute unit. When one 16-thread group accesses memory, the other 16-thread group executes on the ALU. Therefore theoretically, a throughput of 16 threads per cycle is possible on the Radeon architecture. Each ALU can execute a maximum of 2 single-precision Flops: multiply and add instructions per cycle. The clock rate of the Radeon GPU is 850 MHz; for 1600 ALUs this translates to a throughput of 2.72 TFlops/s. The Radeon 5870 has a memory hierarchy that is similar to the Fermi's memory hierarchy. The hierarchy includes a global memory, L1 and L2 cache, shared memory, and registers. The 1 GB global memory has the peak bandwidth of 153 GB/s and is controlled by eight memory controllers. Each compute unit has 8 KB L1 cache having an aggregate bandwidth of 1 TB/s. Multiple compute units share a 512 KB L2 cache with 435 GB/s of bandwidth between L1 and L2 cache. Each compute unit also has a 32 KB of shared memory, providing a total 2 TB/s aggregate bandwidth. The registers have the highest bandwidth, 48 bytes per cycle in each stream core (aggregate bandwidth of 48 ∗ 320 ∗ 850 MB/s, i.e., 13 TB/s). The 256 KB register space is available per compute unit, totaling 5.1 MB for the entire GPU.

3. THE K-MEANS ALGORITHM

In data clustering[24], multivariate data units are grouped according to their similarity or dissimilarity. MacQueen used the term *k-means* to denote the process of assigning each data unit to that cluster (of *k* clusters) with the nearest centroid. That is, *k*-means clustering employs the Euclidean distance between data units as the dissimilarity measure; a partition of data units is assessed by the squared error:

$$E[D] = \sum_{i=1}^{m} \begin{pmatrix} k \\ min\ \|x_i - y_i\|^2 \\ j = 1 \end{pmatrix} \qquad (3.1)$$

where$\boldsymbol{x}i\in R^d$, i= 1, 2, . . . ,m is a data unit and $\boldsymbol{y}j\in R^d$, j = 1, 2, . . . , k denotes the cluster centroid.

Although there are a vast variety of *k*-means algorithms [5], for the sake of explanation simplicity, this paper focuses on a simple and standard *k*-means algorithm summarized as follows:

1. Begin with any desirable initial states, e.g. initial cluster centroids may be drawn randomly from a given data set.
2. Allocate each data unit to the cluster with the nearest centroid. The centroids remain fixed through the entire data set.
3. Calculate centroids of new clusters.
4. Repeat Steps 2 and 3 until a convergence condition is met, e.g. no data units change their membership at Step 2, or the number of repetitions exceeds a predefined threshold.

At each repetition the assignment of *m* data units to *k* clusters in Step 2 requires *km* distance computations (and (*k* - 1)*m*distance comparisons) for finding the nearest cluster centroids, the so-called *nearest neighbor search*. The cost of each distance computation increases in proportion to the dimension of data, i.e. the number of vector elements in a data unit, *d*. The nearest neighbor search consists of approximately 3 *dkm*floating-point operations, and thus the computational cost of the nearest neighbor search grows at *O*(*dkm*). In practical applications, the nearest neighbor search consumes most of the execution time for *k*-means clustering because *m* and/or *d* often become tremendous. However, the nearest neighbor search involves massive SIMD parallelism; the distance between every pair of a data unit and a cluster centroid can be computed in parallel, and the distance computation can further be parallelized according to their vector components.

This motivates us to implement the distance computation on recent programmable GPUs as multi-grain SIMD-parallel coprocessors. On the other hand, there is no necessity to consider the acceleration of Steps 1 and 4 using GPU programming, because they require little execution time and further include almost no parallelism. In Step 3, cluster centroid recalculation consists of *dm*additions and *dk*divisions of floating-point values. Although most of these calculations can be performed in parallel, conditionals and random access writes are required for effective implementation of individually summing up vectors within each cluster. In addition, the divisions also require conditional branching to prevent divide-by-zero errors. Since the execution time for Step 3 is much less than that of Step 2, there is no room for performance improvement that outweighs the overheads derived from the lack of random access writes and conditionals in GPU programming. Therefore, we decide to implement Steps 1, 3, and 4 as CPU tasks.

4. DISCUSSION

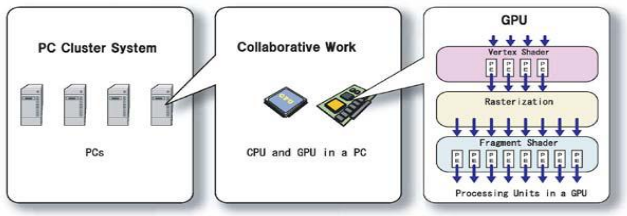

Fig. 2 Parallel processing of data clustering that uses GPU as a co-processor to exploit two kinds of data parallelism in the nearest neighbor search

Our preliminary analysis show that the data transfer from CPU to GPU at each rendering pass is not a bottleneck in the nearest neighbor search. This is because the large data set has already been placed on the GPU-side video memory in advance; only the geometry data of a polygon, including texture coordinates as a cluster centroid, are transferred at eachrendering pass. On the other hand, the data transfer from the GPU-side video memory to the main memory induces a certain overhead even when using the PCI-Express interface. Therefore, we should be judicious about reading data back from GPU, even in the cases of using GPUs connected via the PCI-Express interface. In our implementation scheme, the overhead of the data transfer is negligible except for trivial-scale data clustering because the data placed on the GPU-side video memory are transferred only once in each repetition. Accordingly, our implementation scheme of data clustering with GPU co-processing can exploit GPU’s computing performance without critical degradation attributable to the data transfer between CPU and GPU.

5. CONCLUSION

In this paper, further research tasks include using a cluster of GPUs for texture classification. This could be done including various GPUs on the same machine or dividing computations into a cluster of PCs, and would significantly increase the applicability of the architecture to complex industrial applications. Using this approach, a deeper analysis on the strategy of parallelization for multi-GPU introduced is needed. Undoubtedly, the increase of performance will not be proportional to the number of GPUs and will be damaged by factors such as data communication and synchronization among different hosts. we have proposed a three-level hierarchical parallel processing scheme for the k-means algorithm using a modern programmable GPU as a SIMD-parallel co-processor.

Based on the divide-and-conquer approach, the proposed scheme divides a large-scale data clustering task into subtasks of clustering small subsets, and the subtasks are executed on a PC cluster system in an embarrassingly parallel manner. In the subtasks, a GPU is used as a multigrain SIMD-parallel co-processor to accelerate the nearest neighbor search, which consumes a considerable part of the execution time in the k-means algorithm. The distances from one cluster centroid to several data units are computed in parallel. Each distance computation is parallelized by component-wise SIMD instructions. As a result, the parallel data clustering with GPU co-processing significantly improve the computational efficiency of massive data clustering. Experimental results clearly show that the proposed hierarchical parallel processing scheme remarkably accelerate massive data clustering tasks. Especially, acceleration of the nearest neighbor search by GPU co-processing is significant to save the total execution time in spite of the overhead of the data transfer from the GPU-side video memory to the CPU-side main memory. GPU co-processing is also effective to retain the scalability of the proposed scheme by accelerating the aggregation stage that is a non-parallelized part of the proposed scheme.

This paper has discussed the GPU implementation of the nearest neighbor search, compared with the CPU implementation to clarify the performance gain of GPU co-processing. However, the multi-threading approach has a possibility to allow both CPU and GPU to execute the nearest neighbor search in parallel without interrupting each other. In such an implementation, hence, GPU co-processing will always bring additional computing power even in the case where only a low-end GPU is available. The multi-threading implementation with effective load balancing between CPU and GPU will be investigated in our future work.

6. REFERENCES

[1] C. Pizzuti and D. Talia, "P-AutoClass: Scalable Parallel Clustering for Mining Large Data Sets," *IEEE Transactions On Knowledge And Data Engineering,*vol. 15, no. 3, pp. 629-641, 2003.

[2] U. Fayyad, G. Piatesky-Shapiro and P. Smith, From Data Mining to Knowledge Discovery: An Overview, NY: AAAI/MIT Press, 1996.

[3] L. Hunter and D. States, "Bayesian Classification of Protein Structure," *Expert,* vol. 7, no. 4, pp. 67-75, 1992.

[4] C. Olson, " Parallel Algorithms for Hierarchical Clustering," *Parallel Computing,* vol. 21, pp. 1313-1325, 1995.

[5] D. Judd, P. McKinley and A. Jain, "Large-Scale Parallel Data Clustering," in *Int'l Conf. Pattern Recognition*, New York, 1996.

[6] J. Potts, Seeking Parallelism in Discovery Programs, Arlington: Master Thesis : Univ. of Texas, 1996.

[7] K. Stoffel and A. Belkoniene, "Parallel K-Means Clustering for Large Data Sets," in *Parallel Processing*, UK, 1999.

[8] R. Agrawal, T. Imielinski and A. Swami, "Database mining: A performance perspective," vol. 5, no. 6, p. 914–925, Dec 1993.

[9] S. Weiss and C. Kulikowski, Computer Systems that Learn. ,, vol. 1, New York: Morgan Kaufman, 1991.

[10] D. Michie, Machine Learning, Neural and Statistical Classification, vol. I, NJ: Ellis Horwood, 1994.

[11] J. Quinlan, Programs for Machine Learning, vol. I, New York: Morgan Kaufman, 1999.

[12] R. Agrawal, "An interval classifier for database mining applications," in *VLDB Conference*, New York, Aug 1992.

[13] J. Catlett, Megainduction Machine Learning on Very Large Databases. PhD thesis,, vol. I, Sydney: Univ. of Sydney, 1991.

[14] M. Mehta, R. Agrawal and J. Rissanen, "SLIQ: A fast scalable classifier for data mining," in *5th Intl. Conf. on Extending Database Technology*, NJ, March 1996.

[15] J. Shafer, R. Agrawal and M. Mehta., "SPRINT: A scalable parallel classifier for data mining," in *22nd VLDB Conferenc*, NJ, Sept 1996.

[16] K. Alsabti, S. Ranka and V. Singh, "CLOUDS: A decision tree classifier for large datasets," in *4th Intl. Conf. on Knowledge Discovery and DataMining*, Aug 1998.

[17] D. Fifield, Distributed tree construction from large data-sets: Bachelor Thesis,, Australian Natl. Univ., 1992.

[18] L. Breiman, Classification and Regression Trees, Belmont: Wadsworth, 1984.

[19] M. Joshi, G. Karypis and V. Kumar, ScalParC: A scalable and parallel classification algorithm for mining large datasets, Intl. Parallel Processing Symp, 1998.

[20] "Technical Brief: NVIDIA GeForce 8800 GPU architecture overview," [Online]. Available: www.nvidia.com.

[21] "NVIDIA's next generation CUDA compute architecture: Fermi," [Online]. Available: http://www.nvidia.com/content/ PDF/fermi_white_papers/NVIDIAFermiComputeArchit ectureWhitepaper.pdf.

[22] Z. Fan, F. Qiu, A. Kaufman and S. Yoakum-Stover, "GPU cluster for high performance computing," NY, 2004.

[23] "ATI Mobility Radeon HD 5870 GPU specifications," [Online]. Available: http://www.amd.com/us/products/notebook/graphics/ati-mobility-hd-5800/Pages/hd-5870-specs.aspx.

[24] D. Judd, P. McKinley and A. Jain, "Performance Evaluation on Large-Scale Parallel Clustering in NOW Environments," in *Eighth SIAM Conf. Parallel Processing for Scientific Computing*, Mar 1997.