Bull. Nov. Comp. Center, Comp. Science, 29 (2009), 139–150
© 2009 NCC Publisher

Implementation of the algorithm of hierarchical cluster analysis on GPU by means of CUDA technology

N. B. Zverev, F. A. Murzin, S. A. Poletaev

Abstract. In the paper, the algorithm of the hierarchical cluster analysis is considered and the method is proposed to transfer this algorithm onto the parallel multiprocessor system used on modern graphics processing units (GPUs). Within the frameworks of some natural assumptions, we have estimated the run time of the algorithm in a sequential case, in a parallel case for some abstract parallel machine and for GPU. The algorithm is implemented on CUDA, allowing us to carry out the hierarchical cluster analysis much faster, than on CPU.

1. Introduction

In the paper, the algorithm of the hierarchical cluster analysis is considered [1, 2] and the method of implementation of this algorithm onto the parallel multiprocessor system used on modern graphics processing units (GPUs) is proposed. To implement the algorithm, we used the CUDA (Compute Unified Device Architecture) technology developed by the NVIDIA company [7].

Let us notice that GPUs are intended for high-performance intensive computing. The model of computation in CUDA assumes that a programmer first breaks a problem into independent parts (blocks), which can be processed in parallel. Then each block is broken into a set of streams (threads) executed in parallel but, probably, depending on each other. CUDA is based on an extension of the C ++ language for a parallel start of threads, which are carrying out the same function (kernel). The maximal size of a kernel is 2 million instructions. Streams are aggregated into blocks (up to 512 streams), and blocks are aggregated into grids.

Now it is possible to say that the architecture of parallel computation NVIDIA CUDA allows many researchers, applying C++, to use the GPU computing capacities when solving difficult computational problems. Thereby it is attractive that this equipment is accessible to many specialists, in view of its low price as compared to that of supercomputers.

Within the frameworks of some natural assumptions, we have estimated the performance of our algorithm in a sequential case, in a parallel case for some abstract parallel machine and for GPU.

The procedure of constructing a matrix of distances between clusters is implemented on GPU by means of the CUDA system, which in case of a high-dimensional problem is much (more than 60 times) faster than the same C++ program implemented on CPU. The procedure of search of the minimal element of a matrix, considerably reducing data exchange between GPU and host-computer, is also implemented on GPU by means of the CUDA system. The results of the algorithm testing are represented.

2. Basic notions of the cluster analysis

The cluster analysis (data clustering) is a problem of splitting the set of objects (situations) into non-intersecting subsets named clusters so that each cluster consists of similar objects, and objects of different clusters are essentially different. It is possible to see applications of this approach in [5].

Input data can have various forms. The first variant is when each object is described by a set (vector) of characteristics named signs. Generally, signs can be numerical or non-numerical. Non-numerical data are often called categorical [3, 4] which, as a rule, are also digitized.

The purposes of clustering can be various.

1. Understanding of the data structure. Partition of a set of data into groups of similar objects allows us to understand better the data structure.

2. Data compression. If an initial sample is redundant, then it is possible to reduce it taking the most typical representative object from every cluster.

3. Novelty detection. New objects are determined, which themselves might not be members of the given clusters.

The procedure of the hierarchical cluster analysis [1, 2] provides grouping of objects (lines of a matrix of data) and variables (columns of the matrix). The matrix here considered is such that its lines correspond to objects. Thereby each line contains a vector of characteristics (signs) of the object associated with this line.

At the beginning of the clustering process, all objects are considered as separate clusters which are then aggregated.

A key component of the analysis is a repeated calculation of distance measures between objects and between clusters, thus objects are grouped into clusters. The outcome is represented graphically as a dendrogram.

The dendrogram is a standard graphical representation of the results of hierarchical cluster analysis. This is a tree-like plot where each step of hierarchical clustering is represented as a fusion of two branches of the tree into a single one. The branches represent clusters obtained at each step of hierarchical clustering.

As a result of successful analysis, there appears a possibility to allocate clusters as separate branches and give them a natural interpretation.

Distances (or in a more general case, affinity measures) between points and between clusters can be defined in different ways. For example, we can use Euclidian or Chebyshev's metrics, a supremum-norm, etc.

3. Data clustering methods

Thus, at the first step, we have metrics or an affinity measure between separate points in some multidimensional space. Further, in order to continue the process of the hierarchical cluster analysis, we should define the distance, or an affinity measure between clusters. For this purpose, different methods are applied.

1. Single connection (a method of the nearest neighbor). In this method, the distance between two clusters is defined by the distance between two closest objects (the nearest neighbors) in the given clusters.

2. Full connection (a method of the most remote neighbors). In this method, the distance between clusters is defined by the greatest distance between any two objects in the given clusters.

3. Not weighed paired average. In this method, the distance between two different clusters is calculated as the average distance between all pairs of their objects.

4. Weighed paired average. The method is identical to the method of not weighed paired average, except that the size of the corresponding clusters (i.e. the number of their objects) is used as a weight factor.

5. Not weighed centered method. In this method, the distance between two clusters is defined as the distance between their centers of gravity.

6. Weighed centered method (median). This method is identical to the previous one except that weight is used for the difference estimation between the sizes of clusters (i.e. the number of their objects).

7. Ward's Method. This method differs from all other methods as it actually uses the methods of disperse analysis for estimation of distances between clusters. The method minimizes the sum of squares of distances. More precisely, as the distance between clusters, we use the sum of squares of distances between objects and the centers of clusters obtained after their aggregation.

After execution of the next step of agglomerative procedure, it is necessary to find out that desirable agglomeration is reached. There are various methods to define the procedure stop criterion: the a priori defined number of clusters is obtained; all clusters contain more elements than it was predefined; clusters possess a required parity of internal uniformity and heterogeneity among them.

4. CUDA technology

CUDA technology is the hardware-software computing architecture [7] developed by the NVIDIA company on the basis of some extension of the C language, which gives us a possibility to organize access to the set of instructions of the graphics accelerator and to manage its memory for organization of parallel computing. CUDA helps us to implement algorithms on graphics processors of GeForce video accelerators of the eighth generation and later (series GeForce 8, GeForce 9, GeForce 200), and also Quadro and Tesla.

The CUDA software consists of several parts: a hardware driver, an application programming interface (API), and two high-level mathematical libraries CUFFT and CUBLAS described in [8, 9].

Though difficulties of programming on GPU by means of CUDA are still quite serious, they are not as challenging as those encountered with early GPGPU decisions. Such programs require partition of computations among several multiprocessors, which is similar to MPI, but without data sharing, which are stored in the general video memory. Because CUDA programming for each multiprocessor is similar to that in OpenMP, it requires good understanding of memory organization. But certainly, complexity of data processing and transfer strongly depends on the program.

A toolkit for developers contains a set of examples of a well-documented code. The tutorial course requires about two or four weeks for those already familiar with OpenMP and MPI. The extended C language is the basis of API. For translation of a code from this language onto the structure of CUDA, SDK includes nvcc – a command line compiler created on the basis of the open compiler Open64.

5. Implementation of the algorithm with CUDA

5.1. The structure of the algorithm

The algorithm of the hierarchical cluster analysis is iterative. Each iteration contains several stages: construction of a matrix of distances; search for the minimal element of the matrix; aggregation of clusters the distance between which is minimal.

The triangular matrix of distances $M = (m_{ij}),\ 0 \leqslant i, j < N$ is filled only in its top part, i.e. for m_{ij} with $i > j$. Otherwise $m_{ij} = 0$.

Let $M_N = M$. Further in the process of cluster aggregation, there appears a sequence of matrices of distances $M_N, M_{N-1}, \ldots, M_{N-s}$. We use it to construct a corresponding sequence $A_N, A_{N-1}, \ldots, A_{N-s}$ of matrices, which reflects the membership of elements to clusters.

Let $A_N = E$ be an identity matrix of dimension $N \times N$.

If A_k is already constructed, then A_{k-1} is constructed as follows.

In a matrix M, we search for such a pair (i, j) that $M(i, j)$ is minimal (or $m_{ij} = \min\{m_{kl} : 0 \leqslant k, l < N\}$). Notice that several pairs (i, j) may exist, on which minimum is reached. In this case, we choose one of them.

Having received the result, we walk through the i-th and j-th columns of the matrix A and replace the elements of the i-th column by disjunction of the corresponding elements of columns i and j. We replace the j-th column by the k-th column. Thus k should be not equal to j. Then we delete the k-th column of the matrix. It means that at the next iteration, we will have a matrix M of smaller dimensions. Hence, the number of clusters will decrease by 1.

Let us also notice that in the k-th iteration, we work with a matrix $M_k = M_{N-k+1}$. It means that in this case we work with $N'_k = (N_k^2 - N_k)/2$ elements. Now, similar to [6], we will estimate the efficiency of the algorithm for different situations.

5.2. Estimation of run time of the sequential algorithm

Step 1. Construction of a matrix of distances

In a **sequential** case, it is necessary to calculate $N'_k = (N_k^2 - N_k)/2$ elements. We assume that time necessary for calculation of one element of a matrix is equal to the constant c_1. Thus, total time necessary for matrix construction is

$$T_1^k = c_1 \cdot N'_k = c_1 \cdot (N_k^2 - N_k)/2.$$

Let us notice that this time is not static. At any technique of definition of a distance between clusters, in view of a growing size of clusters, the number of operations also increases. In other words, as the number of clusters decreases step by step simultaneously with iterations, the volumes of clusters grow, and c_1 will also grow. Therefore, let T_p^k be time necessary for calculation of one element of a matrix in the k-th iteration. Thus we have

$$T_1^k = T_p^k \cdot N'_k.$$

It is possible to write that

$$c_1 \leqslant T_p^k \leqslant c_1\, k^2.$$

The left border corresponds to one-element clusters. The right border corresponds to the fact that in the k-th iteration, clusters consist of no more than k points.

Step 2. Finding the minimal element

Analogously, we should consider the elements of the matrix located above the main diagonal (M (i, j) | j>i). Time necessary for comparison of any element with the minimal element is assumed to be constant and denoted by c_2. Thus

$$T_2^k = c_2 \cdot N_k' = c_2 \cdot (N_k^2 - N_k)/2.$$

Step 3. Aggregation of clusters

This step can be divided conditionally into two parts:

1. Preliminary aggregation of clusters

The corresponding columns of a matrix A_k are considered, i.e. it is necessary to make N_k iterations. Accordingly, time is equal to

$$T_{3.1}^k = c_{3.1} \cdot N_k.$$

2. Deletion of one of the clusters

Deletion is carried out by means of zeroing the values of the k-th columns, therefore

$$T_{3.2}^k = c_{3.2} \cdot N_k.$$

Taking into account these two subparagraphs, it is possible to say that $T_3^k = c_3 \cdot N_k$, where

$$c_3 = c_{3.1} + c_{3.2}.$$

The total time in the sequential case is

$$T = \sum_{i=1}^{3} T_i^k = T_p^k \cdot N_k' + c_2 \cdot N' + c_3 \cdot N.$$

5.3. Estimation of run time of the parallel algorithm for an abstract machine

Let us make a rough estimate of an overall performance of the abstract machine for parallel computation. We suppose that this machine can simultaneously process the unlimited number of streams.

Step 1. Construction of the matrix of distances

Distances between clusters are calculated simultaneously. Therefore, the time of calculation and the acceleration factor are

$$t_1^k = c_1' = c_1,$$

$$\lambda_1^k = T_1^k / t_1^k = (N_k^2 - N_k)/2.$$

Step 2. Search for the minimal element

Let us consider the most obvious way, i.e. a pairwise comparison of elements of a matrix in the form of a tree. In that case we have

$$t_2^k = {}_2' \cdot \log_2 N_k'.$$

It is possible to assume that $c_2' = c_2$, which results in

$$\lambda_2^k = T_2^k / t_2^k = N'_k / \log_2 N'_k.$$

Step 3. Aggregation of clusters

We can make all replacements simultaneously, therefore we obtain

$$t_3^k = c'_3 = c_3,$$

$$\lambda_3^k = T_3^k / t_3^k = N_k.$$

5.4. Estimation of run time of the parallel algorithm for GPU

General facts

Blocks of size $n \times n$ are processed. In our case $n = 16$. The number of blocks simultaneously processed is k(we usually have $k = 4$).

Actually the situation is more difficult. There exists a parameter characterizing the minimal number of streams: $warp = 32$. The major part of parallel implementation of the program is hidden from the programmer. Our equipment allows us to process $M = k \times n = 1024$ streams in parallel.

Step 1. Construction of the matrix of distances

Simultaneously we can calculate distances for 4 blocks, therefore

$$t_1^{k,CUDA} = c_1^{CUDA} \cdot T_1 / M.$$

Let c_1^{CUDA} be the proportionality coefficient reflecting the difference in speed of the central and graphics processors $c_1^{CUDA} = \tau_{CPU} / \tau_{GPU}$.

In a typical situation, $\tau_{CPU} > \tau_{GPU}$ (for example, $\tau_{CPU} / \tau_{GPU} = 16$), i.e. the graphics processor is 16 times slower than the central processor. Parallelization gives a more than 16 times increase in efficiency, thus making its application reasonable.

Step 2. Search for the minimal element

2.1. Search for the minimum is made simultaneously in the first 4 blocks, then in the next 4 blocks, etc. Search for the minimal element in a block is made by a pairwise hierarchical comparison of elements. As a result, we have

$$t_{2.1}^{k,CUDA} = c_{2.1}^{CUDA} \cdot T_1 / M.$$

2.2. Formation of a new matrix containing minima is shown below.

Block (0,0)	Block(1,0)
Block(0,1)	Block(1,1)

The input matrix

Min(0,0)	Min(1,0)
Min(0,1)	Min(1,1)

The matrix obtained at the next step

Both matrices are formed in the internal memory of the graphics processor.

Minima are selected from k blocks. The coordinates of minima are stored in another matrix of the same dimension. Accordingly, we have

$$t_{2.2}^{k,CUDA} = c_{2.2}^{CUDA} \cdot N_k/k.$$

2.3 Then we return to step 2.1, using already a newly obtained matrix. We repeat this operation until we get a matrix of dimension 1×1. Thus, we will obtain the minimal element of the matrix. Unfortunately, it is hard to make an exact estimation, though a rough estimate can be obtained. If we suppose that we are processing not a half of the matrix but the whole matrix, we would have:

$$N_k^1 = N_k,$$

$$N_k^2 = N_k/n^t,$$

$$\ldots\ldots\ldots$$

$$N_k^{t+1} = N_k/n^t.$$

Since we consider approximately only a half of the matrix, we have

$$t_2^{k,CUDA} = c_2^{CUDA} \cdot (1 + 1/n + 1/n^2 + \ldots) \cdot N/2M.$$

Step 3. Aggregation of clusters

It was shown in practice, that this process is more appropriate to be carried out on the central processor in view of small volumes of computation.

6. Program implementation

Step 1. Construction of the matrix of distances

To construct the matrix of distances with CUDA, the algorithm of clusters aggregation was separated into a kernel-module, and the calculation of distances into a separate_device_procedure. Thereby, the possibility of flexible use of various methods of the cluster analysis was reached.

The program contains the modules performing the following functions:

1. GPU memory allocation for the initial matrix A and coordinate data.
2. Copying of the data mentioned above to GPU.
3. GPU memory allocation for the result.
4. Execution of the kernel-module.
5. Host memory allocation for the result.
6. Copying the result from GPU to host.
7. Search for the minimal element

Contrary to our expectations, integration of code created with CUDA into the initial code has not produced especially notable results for the following reasons: a bulky enough search for the least element remained implemented in the first version of our program without use of CUDA and copying of data from the device memory (GPU) onto a host computer appeared to be too time-consuming. Approximate distribution of blocks and streams in the matrix of distances can be seen below (Figure 1). For simplicity, the matrix of size 6×6 is shown.

0	d10	d20	d30	d40	d50
-	0	d21	d31	d41	d51
-	-	0	d32	d42	d52
-	-	-	0	d43	d53
-	-	-	-	0	d54
-	-	-	-	-	0

Figure 1. Partition of the matrix elements into blocks

Step 2. Search for the minimal element

At this stage, the following problems are considered.

1. Minimization of data transfer from GPU memory to host memory.
2. The use of CUDA for searching the minimal element of a matrix.

The second problem is complicated by one of the disadvantages of CUDA, namely, shared memory can have only streams inside one block. Different blocks cannot exchange data. Thus computations on the graphics processor are made only partially:

1. The matrix of distances is divided into submatrices. Each submatrix is processed by a separate stream.
2. In each submatrix, there is a minimal element which we write in a file in GPU memory.
3. This file is moved into host memory.
4. On a host, we find the minimal element by standard means.

The submatrix dimension was determined according to the dimension of a block at the first step of the algorithm, i.e. 16×16.

Unfortunately, the result of this implementation was unsatisfactory. The goal was not reached because of the performance losses while processing cycles on GPU. Thereby, the time for the minimal element search was not reasonably reduced.

A new recursive algorithm to search the minimum has been considered:

1. One stream is allocated for each element of the given matrix. Streams are joined into blocks of size 16×16.

2. For each block, an array of size 16×16 is allocated in a shared memory (a memory which is shared among streams of a block). Thus, the content of the element [bx*BLOCK_SIZE+tx] [by*BLOCK_SIZE+ty] of the initial matrix is written into this array.

3. In the shared memory, a variable vmin is allocated, which is compared with the elements of the array of size 16×16 mentioned above.

4. Collecting all vmin from all blocks, we construct a new matrix B (Figure 5) which contains the values of this variable.

5. Then we return back to step 1. We continue this process until the dimension of the matrix B becomes equal to 1.

The process can be seen below (Figure 2). For simplicity, the matrix of the size 6×6 and blocks of the size 4×4 are shown.

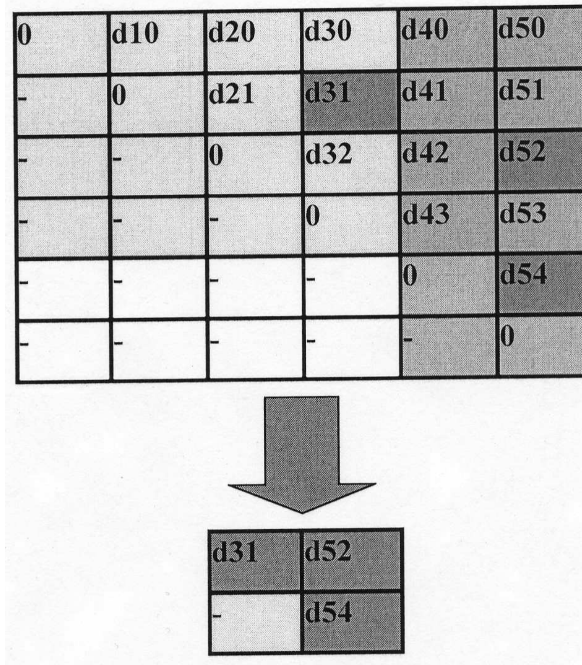

Figure 2. Construction of the next matrix

It is necessary to notice that both purposes of optimization have been reached: the matrix M moves no longer from GPU memory into host memory, and thereby the number of local iterations is reduced to minimum.

7. Conclusion

The algorithm of the hierarchical cluster analysis has been implemented with CUDA, allowing us to carry out the hierarchical cluster analysis much faster than on CPU. The comparative results of running the algorithm with the same initial data on CPU and GPU can be seen below in the table and graphical form (Figure 3). Time is given in seconds.

N	**CPU**	**GPU**	**Acceleration coefficient on GPU**
64	0,28	0,13	2
128	1,17	0,30	4
192	3,91	0,39	10
256	10,11	0,53	19
320	21,98	0, 90	24
384	42,09	1,16	36
448	73,34	1,35	54
512	122,72	1,82	67

Comparative results of performance on CPU and GPU

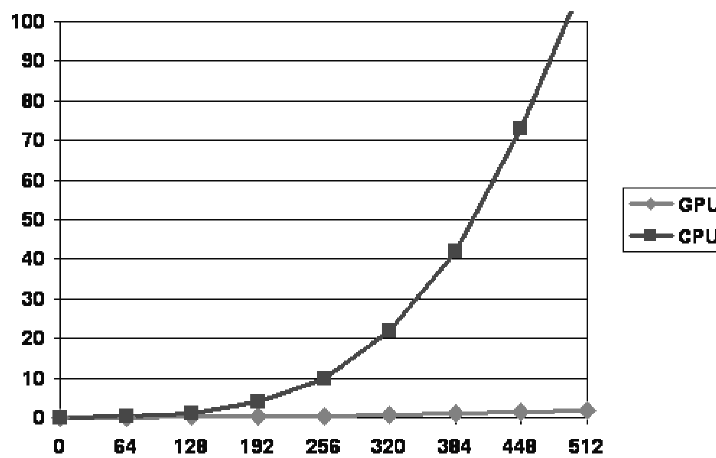

Figure 3. The comparative diagram of the algorithm run-time on GPU and CPU; the horizontal axis corresponds to the number of points and the vertical axis is time in seconds

Resuming, we can say that we have obtained the following results:

1. The method to transfer the algorithm onto the parallel multiprocessor system used on modern GPUs is proposed.

2. Within the frameworks of some natural assumptions, we obtained the estimates for the algorithm run-time in a sequential and parallel case, for an abstract parallel machine, and for GPU.

3. The procedure to construct the matrix of distances between clusters is implemented on GPU by means of the CUDA system, which in case of a high-dimensional problem is much faster than the same C++ program implemented on CPU.4. The search for the minimal element of the matrix, which considerably reduces data exchange between GPU and host, was also implemented on GPU by means of the CUDA system, and it has given a major efficiency increase.

References

[1] Bradley P., Fayyad U., Reina C. Scaling Clustering Algorithms to Large Databases // Proc. 4th Intl Conf. Knowledge Discovery and Data Mining. – AAAI Press, Menlo Park, Calif., 1998. – P. 8–15.

[2] Zhang T., Ramakrishnan R., Livny M. An Efficient Data Clustering Method for Very Large Databases // Proc. ACM SIGMOD Intl Conf. Management of Data. – ACM Press, 1996. – P. 103–114.

[3] Huang Z. A fast clustering algorithm to cluster very large categorical data sets in Data Mining // Research Issues on Data Mining and KDD. – 1997. – P. 367–370.

[4] Ganti V., Gerhke J., Ramakrishan R. CACTUS – Clustering Categorical Data Using Summaries// Proc Int. Conf. on Knowledge Discovery and Data Mining. – 1999. – P. 73–83.

[5] Murzin F.A., Poplevina N.V., Semich D.F. Algorithms of the determination of the oil saturated layers on the basis of data of the radioactive wells logging // 7th Intl Conf. of memory of academician A.P. Ershov, "Perspectives of system informatics", Working seminar "High technology software". – Novosibirsk, 2009. – P. 199–206. (In Russian)

[6] Kalinnikov P.A., Murzin F.A., Pletneva T.A. Some algorithms of image processing and their reflection onto multiprocessor systems // Joint Bull. of NCC&IIS. Ser. Comput. Sci. – Novosibirsk, 2008. – Iss. 28. – P. 67–78.

[7] NVIDIA CUDA Compute Unified Device Architecture. CUDA Programming guide (v. 2.2). – NVIDIA Corporation, 2009. – 125 p.

[8] CUFFT Library. – PG-00000-003 (v.1.1), NVIDIA Corporation, 2007. – 17 p.

[9] CUBLAS Library. – PG-00000-002 (v.1.0), NVIDIA Corporation, 2007. – 80 p.