The Optimization of Algorithms in the Process of Temporal Data Mining Using the Compute Unified Device Architecture

Alexandru PIRJAN
The Bucharest Academy of Economic Studies, Romania
alex@pirjan.com

Considering the importance and usefulness of real time data mining, in recent years the concern of researchers to discover new hardware architectures that can manage and process large volumes of data has increased significantly. In this paper the performance of algorithms for temporal data mining that are implemented in the new Compute Unified Device Architecture (CUDA) from the latest generation of graphics processing units (GPU) will be analyzed and reviewed. The performance will be evaluated taking into account the type of algorithm, data access, the problems` size, the GPU's processor generation, the number of threads processed.
Keywords: *Temporal data mining, MapReduce, CUDA, GPU, Fermi, thread, kernel.*

1 Introduction

Real time data mining will enable scientists to develop researches on a scale that seemed unimaginable until recently. Both hardware architectures and data mining algorithms must properly manage and process huge volumes of data, otherwise data analysis risks becoming irrelevant in certain fields such as that of neuroscience. A possible solution to overcome these difficulties is the development and implementation of new parallel processing algorithms and novel hardware architectures.
New techniques and data mining methods have been developed in recent years, used to discover new patterns, clusters and to classify different types of data. In order to optimize a data mining algorithm one should aim to improve the quality of the data extraction process and to streamline it by reducing the response time.

Parallel hardware architectures have proved to be viable solutions in this respect. Graphics processing units (GPUs) have a real potential in optimizing the data mining process, as they are multithreaded and multicore processing units. Unlike central processing units (CPU's), the cores of a GPU are virtualized at a hardware level and its threads are hardware managed, so the programs' scalability and portability improves substantially. A GPU has a computational capacity and memory bandwidth far beyond than those of a CPU, which help accelerate most of the databases operations and streamline the entire data mining process. These graphics processing units combine hundreds of simplified parallel processing cores, which can be very useful in the data mining process, reducing the necessary time for extracting knowledge from data analysis. This high computational power is currently being used successfully in various scientific fields: image processing, geometric processing and database, overcoming the most powerful CPUs. Another essential aspect is the performance per watt consumed, obtained from the GPU when compared to the CPU processors.

Based on their high performance, low cost and on the increasing number of features offered, GPU processors are powerful tools capable of solving an increasingly wide range of applications. In this paper, the research is focused on the study of the temporal data mining process. This technique is becoming increasingly important and widely used in applications from different fields such as: financial data prediction, telecommunication control, neuroscience, medical data analysis, even if temporal data mining is a relatively new field.

For example, researchers in neuroscience may determine how neurons are connected and related to each other in the human brain.

For this purpose, besides traditional methods, whose main disadvantages are the restricted area of the brain on which you can get information, modern fast methods can be used, which offer real-time images and information. These lead to a series of huge opportunities, patients can be screened, diagnosed and operated by extremely rapid procedures, based on huge GPU processing performance.

In this paper the performance of algorithms for temporal data mining will be analyzed and reviewed considering that the algorithms are implemented on the new Compute Unified Device Architecture (CUDA) from the latest generation of graphics processing units (GPU). The performance will be evaluated taking into account the type of algorithm, data access, the problems` size, the GPU's processor generation and the number of threads processed.

Research shows that GPU processors can provide the desired performance, but it is required to address specific technical issues for each temporal data mining problem. Taking into account the size of the problem and the type of the algorithm implemented on the GPU, one can determine the optimal algorithm, the data access model and the number of threads that are necessary to achieve the desired performance. These results confirm but also contrast with previous research about the temporal data mining, implemented on graphics processors, such as research on MapReduce algorithms [1]. While most papers provide conclusions based on optimum choice of configurations, this article presents some general characterizations that help explain how data mining applications can benefit from the parallel architecture of the latest graphics processors.

2. Compute Unified Device Architecture

For a long time, GPU processors have been used to accelerate graphics rendering on computers. Following the increasing need for improved three-dimensional rendering at a high resolution and a large number of frames per second, the GPU has evolved through specialized architecture, from one-purpose components to multiple purposes complex architectures, able to do much more than just provide video rendering. This development allowed the acceleration of a broad class of applications. The architecture and characteristics of NVIDIA GPUs are summarized in Figure 1.

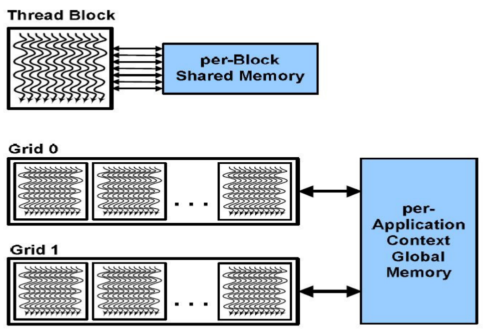


Fig. 1. NVIDIA Compute Unified Device Architecture (CUDA) [2]

CUDA is a software and hardware architecture that allows the NVIDIA graphics processor to execute programs written in C, C++, FORTRAN, OpenCL, Direct Compute and other languages. A CUDA program calls parallel program kernels. A kernel executes in parallel a set of parallel threads. The programmer or compiler organizes these threads into thread blocks and grids of thread blocks. The GPU processor instantiates a kernel program on a grid containing parallel thread blocks. Each thread from the block executes an instance of the kernel and has an unique ID associated to registers, to thread's private memory from the thread block [2].

The Compute Unified Device Architecture hierarchy of threads is mapped to the hierarchy of the graphics processing units hardware processor; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; the CUDA cores contained in the SM run the threads within blocks. SM can perform up to 32 groups of threads called warp. Regarding memory hierarchy, each multiprocessor contains a set of 32-bit registry with a zone of shared memory, which is easily accessible for each core of the multiprocessor but hidden from other multi-processors. Depending on the generation of a GPU, the number of registry and the size of shared memory varies. Besides shared memory, a multiprocessor contains two read-only memory cache, one for texture and another one for constants.

3. The Optimization of Algorithms In the Process Of Temporal Data Mining Using the Compute Unified Device Architecture

When algorithms are developed in the CUDA programming model, the basic concern of developers is to divide the work required in fragments that can be processed by a x number of thread blocks, each containing n threads. For optimum performance, it is recommended that the number of thread blocks matches the number of processors, although the threads within a block will be executed by more cores within a multiprocessor SM. The repartition of tasks to be performed between the x thread blocks is the most important factor in achieving performance.

A single thread block can be considered as equivalent to a PRAM model (parallel random-access-machine) which allows processors to behave arbitrarily asynchronous CRCW (concurrent-read, concurrent-write) [3].

Thus, PRAM algorithms are most efficient at block level [4] and they have to be decomposed into separate kernels because of the need for global synchronization of data flows, synchronization that can be achieved only by successive calls of the kernel.

The technique of data mining through association is an usual method used to discover how certain subsets of elements are associated with other subsets. Temporal data mining is a restricted version of that technique, in which temporal relationships between elements are taken into account.

A specific problem of temporal data mining is the mining of frequent episodes in which we find sequences of frequent items (episodes) appearances in a timed ordered database.

An episode is defined as a partially ordered set of events for consecutive time intervals, embedded in a sequence [4]. The frequent episode mining is defined below [5]:

- $\boldsymbol{D} = \{\boldsymbol{d}_1, \boldsymbol{d}_2, ..., \boldsymbol{d}_n\}$ is a database of ordered items;
- $\boldsymbol{d}_i$ is an element of the alphabet $\boldsymbol{I} = \{\boldsymbol{i}_1, \boldsymbol{i}_2, ..., \boldsymbol{i}_n\}$;
- an episode $\boldsymbol{A}_j$ is a sequence of k elements $< \boldsymbol{i}_{j_1}, \boldsymbol{i}_{j_2}, ..., \boldsymbol{i}_{j_k} >$;
- it is defined an appearance in the database $\boldsymbol{D}$ of the episode $\boldsymbol{A}_j$ if there is a sequence of indices $< \boldsymbol{r}_1, \boldsymbol{r}_2, ..., \boldsymbol{r}_k >$ in ascending order so that $\boldsymbol{i}_{j_1} = \boldsymbol{d}_{r_1}, \boldsymbol{i}_{j_2} = \boldsymbol{d}_{r_2}, ..., \boldsymbol{i}_{j_k} = \boldsymbol{d}_{r_k}$;
- the total number of appearances of $\boldsymbol{A}_j$ in $\boldsymbol{D}$ is called the count of an episode, $\boldsymbol{Number}(\boldsymbol{A}_j)$;
- the purpose of frequent episodes mining is to find all episodes $\boldsymbol{A}_j$ for which $\boldsymbol{Number}(\boldsymbol{A}_j)$/n is greater than a threshold α .

In the following, the standard algorithm for frequent episodes mining is presented.

- Input: the threshold α and the sequential database $\boldsymbol{D} = \{\boldsymbol{d}_1, \boldsymbol{d}_2, ..., \boldsymbol{d}_n\}$;
- Output: the set of frequent episodes $\boldsymbol{A} = \boldsymbol{A}_1, \boldsymbol{A}_2, ..., \boldsymbol{A}_m$;
- Stages:

```
1. on generate candidate episode for each level
       $k \leftarrow 1, S \leftarrow \phi$
       level $k \leftarrow 1, A'_k \leftarrow \{\{i_1\},\{i_2\},...,\{i_m\}\}$
2. the count of candidate episodes
       while $A' \neq \phi$ do
       is calculated $Number(A'_{k_j})$ for all episodes $A'_{k_j}$ of $A'_k$
3. non-frequent episodes are eliminated
       $[Number(A'_j)]/n \leq \alpha$ from the set $A'_k$
4. frequent episodes are stored in the set $S_A$:
       $S_A \leftarrow S_A \cup A'_k$
5. on generate candidate episode for next level
       $A \leftarrow A + A', \quad k \leftarrow k+1$
       end while
6. it is returned the set $S_A$ which contains frequent episodes:
       return $S_A$
```

While the elimination phase and generating steps include only the relevant subsets, the counting step may increase exponentially in respect with the size of a subset A_j and the alphabet I. So, the potential number of episodes of length k is $\frac{n!}{(n-k)!}$ for every $k \in \{1,2,...,n\}$.

Run time can be reduced by the use of advanced algorithms and hardware, implemented on parallel processing architectures in order to increase computational power. Although a number of data mining algorithms have already been implemented on graphics processing units, very few are for temporal data mining. An example is presented in [6].

In the following four algorithms based on CUDA programming model [6] will be presented. They are based on the MapReduce programming model (which will be presented below) and for each of them some kind of parallelism is implemented. In Algorithm 1, each thread is looking for a single episode using data stored in graphics memory. In Algorithm 2, each thread is looking for a single episode, but uses shared memory to create first a data buffer. In Algorithm 3, threads in a block are looking for the same episode, but different blocks are looking for different episodes using data from the graphic card memory. In Algorithm 4, threads in a block are looking for the same episode and different blocks are looking for different episodes using shared memory to create first a data buffer.

MapReduce is a software framework developed and implemented by Google [1], which provides programmers the necessary means to process large sets of data using large parallel systems. It is not necessary for the programmers, which use the MapReduce framework to have advanced knowledge in the field of parallel systems. In achieving real-time data mining, the ability to process data sets in parallel is extremely useful.

The MapReduce algorithm uses two functions: "map" and "reduce." The first one, the function "map" applies to a set of input, which consists of a key/value pair in order to create a set of pairs of intermediate key/value. The function "reduce" applies to all pairs of intermediate key/value containing the same key intermediate to produce a set of outputs. Each of the two functions "map" and "reduce" can be parallel executed in order to use the available resources in large data centers (Figure 2).

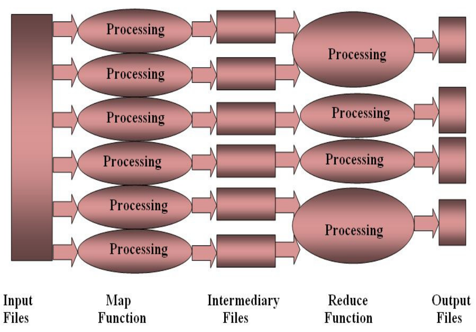


Fig 2. The parallelism in MapReduce algorithm.

MapReduce was originally developed and optimized by Google to run on its private computer data centers. Currently there are different versions of the MapReduce framework for multicore processors, for the Cell processor and for graphics processing units as well. Achieving high performance in these frameworks is quite difficult.

The four algorithms analyzed in this article follow the MapReduce programming model to efficiently benefit from the parallel processing advantage. The "map" function returns the number of appearances of an A_j within a database D. The "reduce" is applied differently, considering if parallelism of threads or parallelism of thread blocks is used.

The first two algorithms implement thread level parallelism to assign a thread for searching an episode A_j in the database D. Using a thread for searching an episode makes the "reduce" function to become the identical application, which returns the value given by the "map" function itself as an output.

Algorithm 1 (without buffering). As each thread will scan the entire database, the first algorithm places the database in the texture memory, so each thread can use the high bandwidth of the GPU. Consequently, threads are allocated in thread blocks one by one until the maximum number of threads per block is reached. For example, if the maximum number of threads per block is 256, then threads from 1 to 256 are allocated to the first block of threads, those from 257 to 512 correspond to the second block of threads and so on until all threads have been used.

Algorithm 2 (with buffering). The second algorithm also uses thread level parallelism, but instead using the texture memory, this algorithm loads a block of data from the database into a buffer of shared memory, processes data from the buffer, then loads another block of data in the buffer and the process is repeated until the entire database has been processed. Thread allocation within the thread blocks is achieved in the same way as in Algorithm 1.

The Compute Unified Device Architecture programming model implements also a block level parallelism. The two algorithms assign a block of threads to find an episode. Within a block, threads collaborate in searching so every thread is looking in a portion of the database.

Algorithm 3 (without buffering). Similar to Algorithm 1, threads within each block access data through the texture memory. Unlike the first algorithm, threads within a block are starting at different positions within the database, while threads with the same ID from different blocks are starting from the same position.

Algorithm 4 (with buffering). The fourth algorithm analyzed in this article uses block-level parallelism with shared memory database buffering. The starting point for each thread of Algorithm 4 depends on buffer size and not on the size of the database (as in Algorithm 3). A thread will always access the same shared memory area during all searches, but data from the shared memory will change when buffer updates.

4. Experimental results

In order to analyze the performance of implementing the characteristics of MapReduce algorithms within the graphics processing units, the various existing NVIDIA CUDA properties should be taken into account.

Table 1. Characteristics of graphics cards used.

Graphics Card	8800 GTS 512	9800 GX2	GTX 280	GTX 480
GPU	G92	2xG92	GT280	GF100
Memory (MB)	512	2x512	1024	1536
Memory Bandwidth (GBps)	57.6	2x64	141.7	177.4
Multiprocessors	16	16	30	48
Cores	128	128	240	480
Processor Clock (MHz)	1625	1500	1296	1401

Compute Capability	1.1	1.1	1.4	2
Registers per Multiprocessor	8196	8196	16384	32768
Registers per thread	10	10	16	21
Threads per Block (Max)	512	512	512	512
Active Threads per Multiprocessor (Max)	768	768	1024	1536
Active Warps per Multiprocessor (Max)	24	24	32	48

For this purpose four different NVIDIA GeForce graphics cards have been subjected to a series of tests. These cards were chosen to represent the latest developed technologies nowadays. A brief description of the chosen graphics cards is presented in Table 1.

The most relevant experimental results on the performance of algorithms (presented in the previous section), implemented on CUDA architecture, running on the four graphics cards in Table 1, are presented below [6], [7]. This article aims to study, compare and analyze these results, highlighting the specific characteristics of each selection.

In tests the following configuration has been used: E4500 Intel Core2 Duo operating at 2.2 GHz with 4 GB (2x2GB) of 200 MHz DDR2 SDRAM (DDR2-800). Programming and access to the GPUs used the CUDA toolkit and SDK 2.0 with NVIDIA driver version 197.75. In addition, all processes related to graphical user interface have been disabled to reduce the external traffic to the GPU.

The experimental results and interpretations on the performance of algorithms mentioned above, using different graphics cards for episodes at different levels with different numbers of threads per block are presented below.

At the L level of an episode, an algorithm searches an episode of length L. In the considered cases, L can be 1, 2 or 3.

The alphabet in which the searching is performed consists of capital English alphabet letters and the database contains 393,019 letters. Different scenarios have been chosen: the first level contained 26 episodes, level 2 contained 650 episodes and level three contained 15,600 episodes [6].

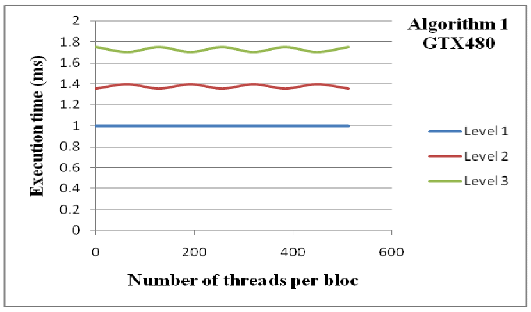
A test consists of selecting an episode's level, an algorithm, a graphics card and the block size. The execution period (measured in milliseconds) is considered the period of time between the moment when the kernel is invoked and the moment when it returns the answer.

Although the GPU's access to graphics has been limited by disabling all non-essential services, each test was performed ten times and the average time obtained during the tests was calculated.

In the following some characteristics that result from these tests on the three criteria (the chosen level, the algorithm and graphics card used) and their impact on the execution time are presented.

The effect of level selection on execution time

To assess the impact of the problem's size on execution time, a series of tests have been done, in which the hardware and the algorithm remained stable and the level L was varied. Because the number of episodes that must be searched increases exponentially when L increases (as noted earlier), the scalability of an algorithm regarding the problem's size is important (Figure 3).



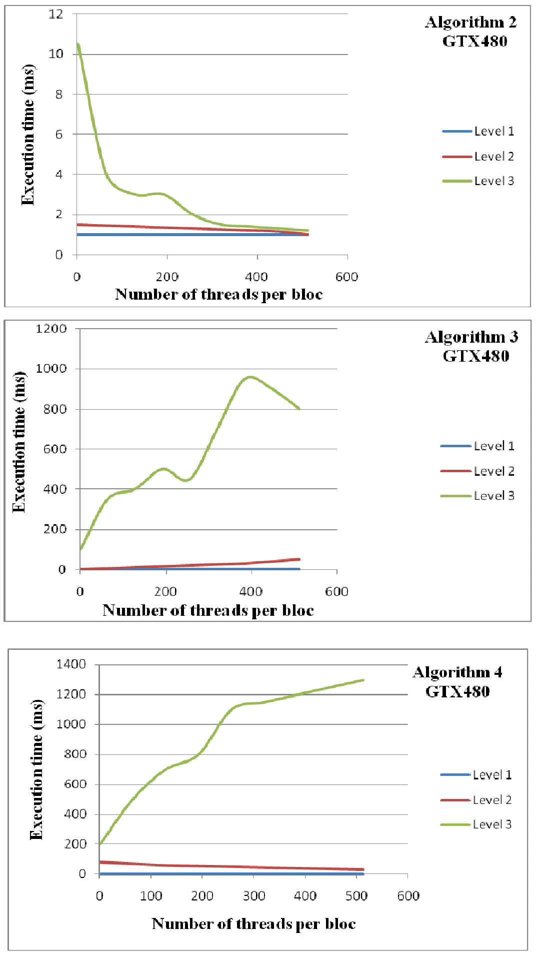


Fig. 3. The effect of level selection on execution time.

a) Parallel thread algorithms provide constant time per episode. This is the case of Algorithms 1 and 2. Regardless of the number of searches, the time required to complete each individual search is essentially the same. A search for each episode is completely independent of other searches and every search is assigned to a thread. The complexity of searching a single episode in a set of data remains constant regardless of the chosen level. For this reason, the execution time is spent entirely for the execution of the "map" function across the entire database. Since these algorithms require a constant time per search, if we consider the parallel processing capability, one can observe that time remains constant even if a large number of searches are executed via the graphics processing unit. Even if there are 30 or 700 or thousands of searches, the process requires the same period.

b) The increasing time in parallel thread processing caused by buffering may be amortized. Algorithm 2 uses a buffer zone to combine the memory bandwidth of all threads in a memory block to reduce the texture load. This implies a long execution time because only one block can be resident on a multiprocessor at a time during the loading phase and other processing cannot be done. As more threads are added to a block, the execution time for Algorithm 2 decreases exponentially. This feature shows that Algorithm 2 is able to use the processing power of a large number of threads. As the number of threads increases, more results will be quickly calculated since all threads can access the shared memory block without additional resource consumption (until the moment when planning a large number of processes on the multiprocessor exceeds the total calculation time).

c) The execution time increases along with the number of threads, when using Algorithm 4 and the 3rd level. Unlike Algorithm 2, Algorithms 3 and 4 lose performance per episode as they increase the number of threads per block and the episode's length. Studying the experimental results, we can observe an increase in the execution time along with the number of threads when using Algorithm 4 and the level 3. Therefore, the execution time increases when switching from the first to the second level and from the second to the third. These two trends are due to the complexity of finding the episodes and due to the increased resources consumption when loading more blocks that can be active simultaneously on the graphics card.

4.2. The effect of algorithm selection on execution time

It is very important that the chosen algorithm matches the size of the considered problem. However, a programmer often wants to solve a problem of a certain size but has access only to a certain type of hardware. The programmer can modify only the algorithm and the number of threads that he uses within this algorithm. In this situation, he will want to use the fastest algorithm for the problem. Tests were done on GTX480 graphics card, because from all the tested cards, it has the highest computational capability (Figure 4).

d) One thread per episode is not enough for small problems (L = 1). When small lower levels problems are assessed, there are not enough episodes to generate sufficient threads to use the graphic card's resources. As the number of episodes is fixed and there is just one thread per episode, in the case of Algorithms 1 and 2 one can observe the tendency to increase the execution time along with the number of threads. Therefore, for these algorithms, the search time slowly decreases. Algorithm 4 on GTX480 obtains a search time of a milliseconds order. Therefore, it is noted that when using the GTX480, real-time data mining can be achieved and this becomes a certainty for servers incorporating more of these parallel cards and for future GPU architectures, when dealing with significant size databases.

e) Block level depends on its size for medium size problems (L = 2). Algorithms 1 and 2 describe the number of blocks while the number of threads per block increases, because there are a fixed number of episodes and thus a fixed number of threads, so the number of blocks changes in the same time with the number of threads per block. At the second Level, the number of blocks will vary depending on the number of threads per block.

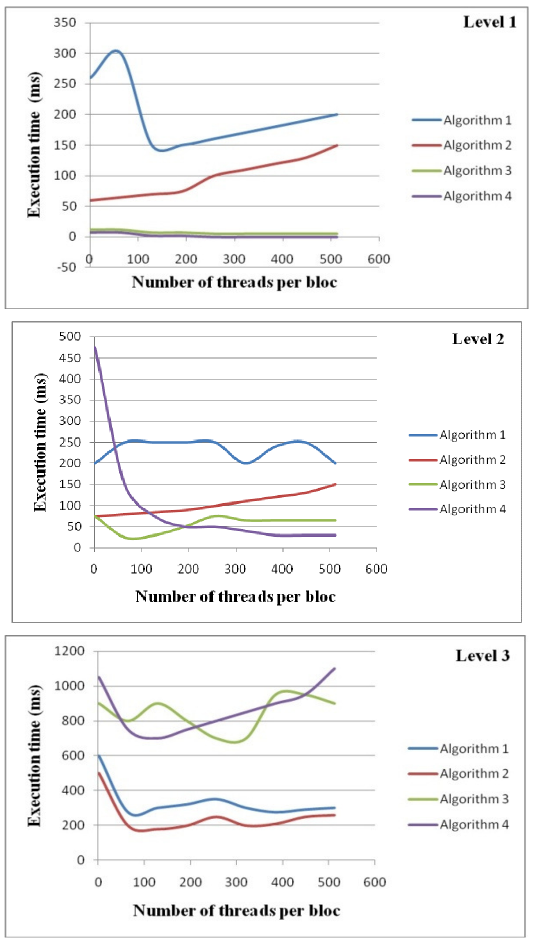


Figure 4. The effect of algorithm selection on execution time

f) Thread level parallelism is enough for large problems (L = 3). The graphic card used, GTX480 has 48 microprocessors, a maximum number of 1,536 active threads per multiprocessor and a total of 73,728 active threads available. For L = 3, there are 25,230 episodes to search. Parallel thread processing algorithms (Algorithms 1 and 2) are much faster than block-level algorithms (Algorithms 3 and 4) because the Algorithms 1 and 2 have to search simultaneously for more episodes than Algorithms 3 and 4 for a certain number of threads per block. Algorithms 3 and 4 are limited to 384 episodes that can be searched due to the limitation of eight blocks per each of the 48 available multi-processors in GTX480 and each block is searching for a single episode. Algorithms 1 and 2 may search more episodes because each thread within a block will look for one episode. Practical resources used by each thread and the available resources per each multiprocessor determine the number of active episodes for Algorithms 1 and 2.

4.3.The effect of graphics card selection on execution time

The hardware configuration is one of the major influencing factors of the algorithms' performance (Figure 5).

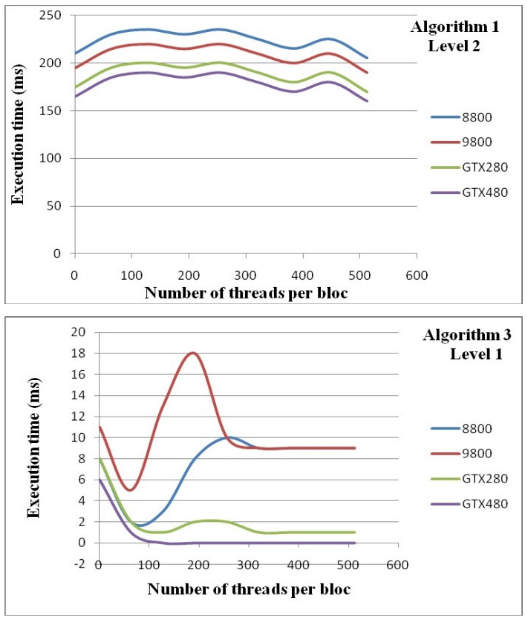


Fig. 5. The effect of graphics card selection on execution time.

g) The frequency of CUDA processors implemented in GPU influence performance for small and medium problems. Algorithms 1 and 2 depend heavily on the CUDA processing cores frequency for small or medium size problems. The frequencies of the four graphic cards in question are 1401 MHz (for GTX480), 1296 MHz (for GTX280), 1500 MHz (for 9800GX2) and 1625 MHz (for 8800 GTS512). For the first two levels, there are no important differences, but starting from the third level, the 480 cores of the GTX480 significantly exceed those 256 of GX2 and those 128 of 8800GTX.

h) Block-level algorithms are affected by memory bandwidth. Algorithm 3 is considerably affected by the memory requirements when large sets of data are processed. The total number of threads that require memory is given by the total number of episodes related to the total number of threads in a block. The algorithm needs to store a large number of threads per multiprocessor over a long period of time and this produces a huge memory consumption. GTX480 gets the highest performance due to the 177.4 Gbps bandwidth followed by GTX280 with a bandwidth of 141 Gbps.

5. Conclusions

In this article, we analyzed and compared temporal data mining algorithms implemented on the latest NVIDIA CUDA architectures. As expected, the best execution time for the analyzed algorithms is the one obtained on the latest architecture, Fermi, that was released by NVIDIA on March 26, 2010. As experimental results outlined, an implementation based on the MapReduce framework must dynamically adapt the type and parallelism level in order to obtain an increased performance.

In order to design efficient temporal data mining algorithms implemented on CUDA parallel processing architectures, one must take into account the eight criteria mentioned above:

- parallel thread algorithms provide constant time per episode;
- the increasing time in parallel thread processing caused by buffering may be amortized;
- the execution time increases along with the number of threads, when using Algorithm 4 and the 3rd level;
- one thread per episode is not enough for small problems (L = 1);
- block level depends on its size for medium size problems (L = 2);

- thread level parallelism is enough for large problems (L = 3);
- the frequency of CUDA processors implemented in GPU influence performance for small and medium problems;
- block-level algorithms are affected by memory bandwidth.

There are many difficulties regarding the practical implementation of data mining algorithms on a GPU architecture. A CUDA programmer must have thorough knowledge of how threads work and how thread blocks are mapped, must know in detail six different areas of memory and especially inter-threading communication. Software development for the CUDA architecture began to be facilitated by new development environments such as NEXUS, but programmers are still forced to write source code for low-level resources and kernels control for each processing operation that is implemented on the GPU, which requires a large amount of time.

There are also other limitations on the performance of data mining algorithms described above. Most important of them are the limitations in memory size and the transfer time between the GPU and the memory. Current NVIDIA cards support memory sizes up to 6 GB, the size being extended from 4GB with the launch of the new Fermi architecture, but even this is far below from the required size when it comes to huge dimensions data warehouses. Transfer of memory blocks between the CPU and GPU still consume a considerable amount of execution time which influences the performance when applying temporal data mining algorithms.

Although the results offer a much improved performance compared to conventional architectures based on CPUs and a tremendous potential for improving the performance of temporal data mining process, there are hardware issues that have obviously limited the implemented algorithms' performance, limitations that can be overcome since the new Fermi hardware architecture has been launched. An important limitation that has a direct impact on algorithms runtime performance happens when dealing with dynamically accessed arrays (which cannot be accessed directly by an index at compile time). Dynamically accessed arrays are automatically stored in local memory and cannot be stored in the registry memory within the CUDA programming model. Since local memory is an abstraction that refers to memory in the scope of a single thread, it has the same latency time as global memory of GPU and is up to about 140 times slower than registry memory [8]. In the above presented algorithms, this type of arrays addressing is frequently needed and the fact that the registry memory cannot be used is a significant restriction.

Also, certain functions in CUDA, such as "atomicAdd ()" are implemented only for integer values. The support for other types of data would facilitate communication between thread blocks.

Considering the possibilities offered by CUDA (depicted in the official documentation "Official CUDA Programming Guide" [2]) the limitation of memory can be managed in two ways. The first option is the memory paging technique used in the algorithms above, successively moving portions of memory and then processing them. Another way to manage memory limitation is to use the CUDA direct access memory option, "zero-copy ".

Besides the fact that the bandwidth available for this technique is very low, the memory should be declared "pinned", thus allowing the memory pages to be maintained in real memory all the time. In practice, both the GPU and the operating system have limits concerning the pinned memory that is under 4 GB and thus makes this method less effective than the paging one.

Fermi, the new generation of NVIDIA architecture, can overcome all the limitations mentioned above. Even when writing this article significant efforts are made to develop the CUDA programming environment to provide the necessary facilities for the typical programmer. An unified memory hierarchy address space makes it possible to run genuine C++ code on Fermi GPUs. Dynamic arrays can also be accessed in registry memory. Enhancements made in this new architecture allow improved execution times for the algorithms.

The small size of the memory supported by the GPU is a significant limitation of the hardware, which is indeed increased but the 6GB is still insufficient considering that in practice many databases` sizes are of the order of terabytes or even petabytes. TESLA products based on the new Fermi architecture use 40 bits of space address and thus allow addressing of up to a terabyte of memory, but until the first benchmarks will be available, it remains only a theoretical statement. Real time temporal data mining becomes slowly but surely a reality.

References

[1] J. Dean, S. Ghemawat - *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, 2004.

[2] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide, Version 3.1, 2010.

[3] C. Martel, R. Subramonian, A. Park http://portal.acm.org/author_page.cfm?id=81363604006&coll=GUIDE&dl=GUIDE&trk=0&CFID=89531225&CFTOKEN=84483493 - *Asynchronous PRAMs are (almost) as good as synchronous PRAMs*, Proceedings of the 31st Annual Symposium on Foundations of Computer Science, Pages: 590-599 vol.2, 1990.

[4] N. Satish, M. Harris, M. Garland - *Designing Efficient Sorting Algorithms for Manycore GPUs*, Proc. 23rd IEEE International Parallel and Distributed Processing Symposium, 2009.

[5] R. Agrawal, T. Imielinski, A. N. Swami - *Mining Association Rules between Sets of Items in Large Databases*, Proc. ACM SIGMOD International Conference on Management of Data, 1993.

[6] J. Archuleta, Y. Cao, W. Feng, T. Scogland - *Multi-Dimensional Characterization of Temporal Data Mining on Graphics Processors*, Technical Report TR-09-01, Computer Science, Virginia Tech, 2009.

[7] W. Fang, K. Lau, M. Lu, X. Xiao, C.Lam, P. Yang Yang, B. He1, Q. Luo, P.Sander, K. Yang - *Parallel Data Mining on Graphics Processors*, Technical Report HKUSTCS0807, 2008.

[8] P. Bakkum, K. Skadron - *Accelerating SQL Database Operations on a GPU with CUDA*, Vol. 425, Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pg. 94-103, 2010.