Solutions for Optimizing the Relational JOIN Operator using the Compute Unified Device Architecture

Alexandru PIRJAN
The Romanian-American University, Bucharest, Romania
alex@pirjan.com

In this paper it is implemented the inner JOIN operator in the latest Pascal Compute Unified Device Architecture (CUDA), using two approaches developed in the CUDA Toolkit 8.0: a classical approach in which a thread selects one element from the first table and performs a binary search for the corresponding keys residing in the second table; a second approach that makes use of the dynamic parallelism feature of the Pascal architecture to solve the problem of task processing unbalance that may occur when the number of corresponding elements is different along the threads. The Compute Unified Device Architecture dynamic parallelism feature is used to invoke a supplementary kernel function in order to build in parallel the final output set of elements.

Keywords: *CUDA, Pascal architecture, Dynamic Parallelism, GPU, inner JOIN operator*

1 Introduction

In recent years, due to the unprecedented huge increase of the data volume that must be processed in a wide area of applications, scientists had to devise novel and efficient solutions for surpassing these difficulties. One of the most prominent breakthroughs in scientific computing is marked by the Compute Unified Device Architecture, released by the NVIDIA Company in 2007.

This solution allowed the developers of software applications to make use of the tremendous parallel processing power of the Graphics Processing Units (GPUs) to solve computational intensive problems efficiently and in a timely manner [1].

This major development had a massive impact on the industry, economy, and medicine and on many scientific research fields. The shift from the traditional sequential programming to the parallel processing one has opened up new paths and numerous possibilities in the information technology landscape, facilitating huge leaps and advances in technology, bringing many advantages through this new computing approach.

Until the release of the CUDA architecture, the primary role of the GPU has been solely to process graphics tasks, mainly in parallel. The introduction of this concept has provided the necessary means to the programmers for using the parallel processing power of the GPUs without having to know in detail specialized graphics programming.

The Graphics Processing Units are not targeting exclusively games developers anymore. Through the CUDA approach, the GPU becomes a general purpose programmable equipment that can be addressed by the developers in a wide area of applications.

In the last decade, there have been conducted a lot of researches that target the development of software optimization solutions using the Compute Unified Device Architecture [2], [3], [4]. Improving the software performance of data processing represents the main focus of researches from various fields, because of their multiple applications in: decision support systems [5], [6], electronic payments systems [7], [8], complex solutions for the office environment [9], temporal data mining [10].

The latest GPU Compute Unified Device Architecture, Pascal, released in the Spring of 2016, brings multiple improvements and new features to the previous versions, like: a new 16 nm Fin Field Effect Transistor (FinFET) production process that provides an improved performance and efficiency per

Watt, a new interconnect offering significant speeds up, support for a new type of memory architecture, an improved programming model and specifically optimized artificial intelligence algorithms. Thus, the Pascal architecture offers new opportunities for improving the software performance of compute intensive applications [11].

The newly developed software applications that have to process huge volumes of data differ significantly in terms of memory requirements and in the order of processing the instructions of the source code. This is the main reason why CUDA developers must take into account the hierarchical nature of parallelism, that is strictly tied to the tasks that have to be processed and the resulting processing time.

The research issues of significant importance consist in obtaining efficient and high performance parallel implementations in the Compute Unified Device Architecture of algorithms that handle complex data structures, scaling the problem to be solved according to the GPU features, thus obtaining an increased memory bandwidth and low execution time.

This article addresses the above mentioned issues in a specific situation regarding the implementation of the inner JOIN operator in the latest Pascal Compute Unified Device Architecture. The JOIN operator is a relational algebra operator that is frequently used in relational database applications. The inner JOIN operator processes two tables and returns a new one, using in the process one or more columns from each of the tables as a key and computes the Cartesian product for all the rows that correspond to the respective keys.

The article brings contributions to the current state of knowledge by developing and implementing two approaches in the latest Pascal Compute Unified Device Architecture using the CUDA Toolkit 8.0: a classical approach in which a thread selects one element from the first table and uses the method described in [1] to perform a binary search for the corresponding keys residing in the second table; a second approach that makes use of the dynamic parallelism feature to solve the problem of task processing unbalance that may occur when the number of corresponding elements is different along the threads.

Although there have been conducted extensive researches in the literature regarding the CUDA dynamic parallelism feature [12], [13], [14], to the extent of the available information, none of the works so far have analysed the impact of the Compute Unified Device Architecture dynamic parallelism feature when developing applications targeting the latest Pascal CUDA architecture.

In the following, the paper has the subsequent structure: in the 2nd section, there are presented the main features offered by the Pascal CUDA architecture; the 3rd section focuses on the main parallel programming issues that have been taken into consideration in developing the two approaches; in the 4th section are presented and compared the experimental results based on the two developed approaches; the 5th section presents the conclusions.

2 The main features offered by the Pascal CUDA architecture

When compared to the previous Maxwell and Kepler Compute Unified Device Architectures, one can observe that the most recent Pascal GP100 architecture offers substantial enhancements to the streaming multiprocessor (SM) such as: the level of occupancy afferent to the cores, and improved efficiency consisting in an enhanced performance per Watt metric.

The Pascal architecture offers important improvements, resulting in a higher overall performance than on all the other previous architectures. One can note that the Pascal GP100 architecture comprises 64 CUDA cores per each streaming multiprocessor, with a single precision of FP32 (**Fig. 1**).

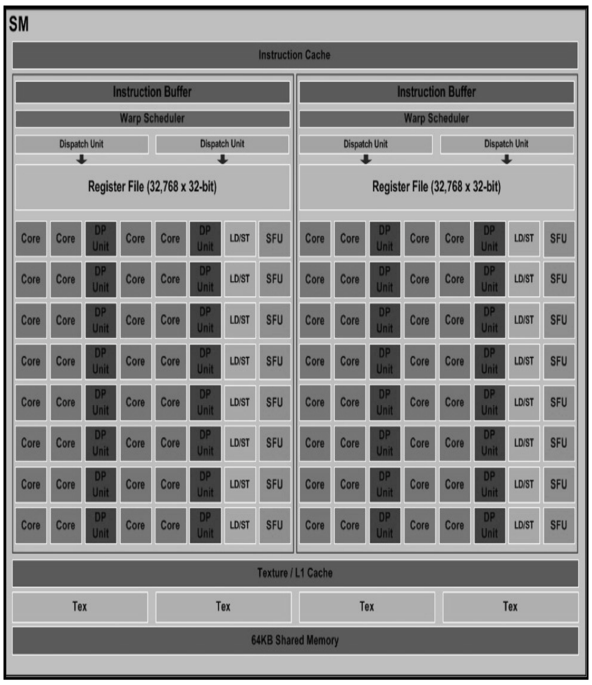

Fig. 1. A detailed insight of thelatest Pascal GP100 SM architecture[1]

In contrast with this, the previous Maxwell architecture comprises 128 CUDA cores (FP32 precision) within each streaming multiprocessor, while the Kepler CUDA architecture contains 192 CUDA cores (FP32 precision) in each of the streaming multiprocessors. In addition to this, the Pascal GP100 architecture incorporates 32 CUDA cores having a FP64 precision, thus resulting in half a rate when performing floating point computations with a FP64 precision. The Pascal architecture offers the technical possibility to incorporate in certain situations two operations having a precision of FP16 into a computing core that has a FP32 precision.

The Pascal GP100 SM architecture also contains two schedulers for warps, two buffers for instructions and two dispatching units per each processing block (**Fig. 1**). The Maxwell architecture incorporates a double number of cores than the Pascal one does, but the Pascal architecture maintains the Maxwell's register file's size and has the possibility to attain the same occupancy level of the warps and thread blocks.

The number of registers per streaming multiprocessor has remained unchanged when compared to the Maxwell and Kepler architectures but it brings a significant improvement. Although, the Pascal architecture offers a higher total amount of register memory because it has a higher number of streaming multiprocessors than the other CUDA architectures.

A comparison of the most popular Pascal architecture implementations (GP100 implemented in Tesla P100, GP102 implemented in Titan X, GP104 implemented in GeForce GTX1080) and their main technical characteristics are depicted in **Table 1**.

Table 1. A comparison between the technical features of the main Pascal architecture implementations[2]

NVIDIA GPU Architecture	GP100 - Tesla P100	GP102 - Titan X	GP104 - GeForce GTX1080
SMs	56	28	20
CUDA Cores	3584	3584	2560
Base Clock	1328 MHz	1417 MHz	1607 MHz
GPU Boost Clock	1480 MHz	1531 MHz	1733 MHz
GFLOPs	10600	10157	8873
Texture Units	224	224	160
Memory Bandwidth	732 GB/s	480 GB/s	320 GB/sec
Memory Interface	4096-bit HBM2	384-bit GDDR 5X	256-bit GDDR 5X
Memory Size	16 GB	12 GB	8 GB
L2 Cache Size	4096 KB	3072 KB	2048 KB
TDP	300 Watts	250 Watt	180 Watts
Manufacturing Process	16-nm FinFET	16-nm FinFET	16 nm FinFET

The graphic cards that implement the CUDA Pascal architecture cover a broad range of market segments, starting with game

[1]*The Figure has been created based on the figure provided by the official NVIDIA documentation sitehttps://devblogs.nvidia.com/parallelforall/inside-pascal/ , accessed on 10.14.2016, at 23:10*

[2]*The table has been created according to the official Nvidia documentation site https://devblogs.nvidia.com/parallelforall/inside-pascal/ , accessed on 10.15.2016, at 00:30*

oriented cards (GTX1080, GTX 1070, GTX 1060), up to scientific computational dedicated ones (Tesla P100). The novel CUDA Pascal architecture offers new features and innovations that provide the customers the possibility to solve problems that were previously impractical to approach, due to the huge computational requirements. Based on its undeniable advantages and prospects regarding the improvement of the parallel processing software performance, energetic efficiency and affordable price, the CUDA Pascal architecture is a viable option for developing solutions for optimizing database operations on huge datasets.

In the following section, there are analysed the main parallel programming issues that have been taken into consideration when developing the two approaches for implementing the inner JOIN operator in the latest Pascal Compute Unified Device Architecture (CUDA).

3 Analyzing important parallel programming aspects in order to develop the CUDA implementation

The most important aspects that had to be considered when developing the two approaches consisted in the appropriate management of the synchronization process, of race conditions, of atomic operations, avoiding memory leaks and dynamic parallelism.

The management of race conditions can be easily achieved when developing classical applications that run on central processing units and need only a single thread of execution. In such a situation the programmer only has to analyse the data flow in order to notice if a certain value has been retrieved from a variable before the latest updated value has been stored in it.

Nowadays, most of the existing compiling tools are able to signal and exactly point out these problems when developing single threaded applications. In the case of developing multi-threaded applications, these aspects must be meticulously analysed and prevented.

In the Compute Unified Device Architecture, the threading system automatically aims to attain the highest level of performance, often having as a result the fact that threads are executed without taking into account a certain chronology. For example, when processing an array in a certain program loop and the result from a certain step depends on the result obtained at a previous step, if the programmer allocates for every element a thread, the outcome will be correct only when the threads are executed in an ascending order and the result from the previous step has already been computed. If more threads are executed in parallel, the risks are high for the result to be incorrect or even the whole program may crash [1].

In some situations, randomly, the program may even produce the correct results if by chance a thread gets to finish the processing before another one needs the respective value. These particular issues illustrate the concept of a race condition, meaning that certain parts of a program are running in the same time to a certain execution point.

There are situations when a certain warp reaches the execution point and computes the result before another warp that needs the respective value reaches that point and there can also happen situations when the second warp reaches the execution point first, thus resulting in a computing error.

Therefore, a first characteristic for race conditions is that they manifest only in certain situations when particular conditions have been met, making it very hard for the developer to identify and pinpoint the problem.

Another important characteristic of these race conditions is that they are tightly related to the moment of executing. There are situations when introducing a breakpoint in the source code execution in order to debug the problem results in the altering of the warp's execution pattern and sometimes the error doesn't have the necessary chronology to happen.

In this situation, one has to disregard the place in the source code where the error manifests itself but has to analyse thoroughly how the threads are ordered and the pattern execution of blocks in order to pinpoint the trigger of the error.

A programmer that develops Compute Unified Device Applications must keep in mind the fact that the CUDA thread mechanism does not enforce a certain chronology in the execution of thread blocks or of the warps.

If there is even a single place in the source code where the programmer implements the logic of the program by presuming that a certain chronology will be followed by the thread blocks, then the whole application is faulty. There are certain situations when a programmer can and should state and create a certain order of the elements that are processed (for example, through sorting actions).

Nevertheless, the programmer must develop his application by taking into account that the order of execution in the equipment is indeterminate and thus, one must use a synchronization technique.

In the Compute Unified Device Architecture, the synchronization process makes it possible for the programmer to exchange data among the threads of the same block of threads or he can even exchange information among multiple blocks belonging to the same grid of blocks. Each thread has available a local memory region and its own register memory [1].

In order for the threads belonging to a certain block to be able to parallel process a dataset and exchange information with each other, they will have to store and retrieve data using the shared memory that is available at the block level.

In the Compute Unified Device Architecture, the warp has a size of 32 threads and offers to the device the possibility to schedule their execution. Therefore, in such a case the synchronization problem may arise. In the situation when the warps have the same execution paths, the operations are automatically serialized in the block, being processed in warps, at different moments of time. In spite of this, the pipelining of warps cannot be maintained consistent, due to external dependencies that may setback a warp for a period of time.

Within a block of threads, there can happen a situation when each warp inside the block retrieves data from the global memory. All the warps use the L1 cache memory except the final one that has to retrieve its data from global memory. In a situation like this, this warp will lag more iterations behind the others. It is obvious that without the implementation of carefully selected synchronization points, one cannot be certain that he obtains the correct results under all circumstances [1].

The synchronization process is mandatory in situations when the threads from different warps have to share data. When executing a CUDA program, the scheduling mechanism invokes considerable sets of block of threads that have their identifiers increase in a linear pattern. Only when a certain number of blocks have been freed from memory, the scheduling mechanism invokes new blocks of threads.

This was particularly useful when developing the two approaches for implementing the inner JOIN operator in the latest Pascal Compute Unified Device Architecture, as this made it possible to improve the access and availability of the L1 cache memory. Conversely, there is a risk of diminishing the state of the warps that are free and can be scheduled. The execution of warps and of thread blocks is spread at different points of the execution chronology and as a consequence it is absolutely necessary to assure that the computing has finished at certain points of an application.

In order to achieve this, when developing the two approaches, the "__syncthreads" primitive along with shared memory have been used to solve the race conditions and achieve correct synchronization. When developing the approaches, the synchronization was mandatory but was applied minimally as to ensure the obtaining of the correct results, avoiding the risk to keep the Graphics Processing Unit idle.

The fact that the chronology of operation is not assured also stands true for basic operations like read, write and update as one cannot be sure that these operations will finish in the same time in all the streaming multiprocessors of the Graphics Processing Unit. For that reason, when there are more threads that have to store their result in the same area of memory, the use of atomic operations assures the fact that different operations will be executed just as if they were a whole serial one.

Until recently, the problem of memory leaks was in strict conjunction with the CPU code. However, the same stands true when developing applications for the Compute Unified Device Architecture. Just like in the case of the CPU code, if a programmer allocates memory dynamically in CUDA, he must also deallocate it explicitly when the application no longer needs it.

There are some situations involving streams and events, where the Compute Unified Device Architecture runtime allocates the necessary memory the first time they are created. The programmer must use explicit instructions to deallocate the memory (cudaStreamDestroy, cudaEventDestroy), otherwise the Compute Unified Device Architecture runtime is not signalled to deallocate the memory [1].

When developing the two approaches, the "cuda-memcheck" tool has been used in order to identify and later solve problems related to memory leaks and memory usage.

The dynamic parallelism feature which is available on the Pascal architecture makes it possible for a CUDA kernel to invoke and synchronize additional child CUDA kernel functions. Until this feature was implemented in the Kepler architecture, the programmer had to invoke more kernel functions or to make sure that some threads within the block are left idle in order to be used later on. These techniques consumed high amount of resources and rendered inefficient results, especially when processing huge volumes of data.

The graphics processing unit was not used appropriately and the kernel functions couldn't store their data in the shared memory area because this type of memory exists only while the kernel does. In essence, a child kernel function can be invoked by a parent one and it is offered the possibility to synchronize the results when the child kernel has finished processing its task. The parent kernel function can make use of the result received from the child kernel function, with no implication of the Central Processing Unit.

A significant advantage that the dynamic parallelism feature brings to the developer consists in the fact that he no longer has to marshal and move the data that needs to be processed. Supplementary parallelism is obtained and can be made available dynamically to the Graphics Processing Unit's scheduling and load balancing mechanisms, in accordance to the volume of data that has to be processed. Up to the introduction of this feature, developers were compelled to remove recursion techniques when building algorithms and any other type of looping elements that did not comply to a single and flat-level of parallelism [12].

The Compute Unified Device Architecture dynamic parallelism feature makes it possible to set up and execute grids of thread blocks, in addition to delay action until the grids have completed the execution up to the threads that are already processing inside a grid of blocks. This means a certain thread that belongs to a grid of blocks and processes data can set up and execute another grid of blocks, called child grid, which will be owned by the parent grid of blocks.

A mechanism of nesting is in place, signifying that the finalization of the parent cannot be finished while waiting for the child grids to finalize. The Compute Unified Device Architecture runtime assures an implied synchronization among the parent kernel and the child kernel functions.

4 The CUDA implementation of the inner JOIN operator

The inner JOIN operator has been implemented in the latest Pascal CUDA architecture using two approaches developed in the CUDA Toolkit 8.0.

In the first classical approach, a thread selects from the first table one element and performs a binary search in parallel according to the method described in [1], in order to identify the corresponding keys that reside in the second table.

The second approach implements the dynamic parallelism feature of the Pascal architecture for solving the problem of task processing unbalance that is likely to occur when the number of corresponding elements is different along the threads.

The Compute Unified Device Architecture dynamic parallelism feature is used for invoking a supplementary kernel function that builds in parallel the final output set of elements. This approach uses the parent thread from the GPU to invoke a child kernel function. The CUDA dynamic parallelism feature has been implemented in the second approach instead of the parallel looping structures that were implemented in the classical one.

The child kernels are allocated dynamically by the parent threads in order to process in parallel the tasks. In contrast with the dynamic parallelism approach, in the classical approach, a loop structure is used by the threads within the warps to process the data in a different number of iterations, corresponding to the workload and the available resources.

One of the major advantages of the dynamic parallelism approach is that the resources of the graphics processing unit are better employed and a higher occupancy level of the GPU's resources is obtained because the parent threads invoke child kernel functions that process the tasks in parallel by means of minimal or even no control divergence.

A frequent problem when using this approach can arise due to a lack of parallelism (when the dataset has a small dimension), that makes it unfeasible to invoke the child kernel CUDA function. In this case, the processing takes place in the parent kernel function.

The patterns of memory access are different in the two developed approaches. In the classical approach, a thread accesses the memory using more loop iterations, while in the second approach, by using a single instruction, the threads of the child kernel function are contiguous and process the data in memory more efficiently, thus improving the alignment of memory and obtaining optimal coalesced memory operations along with an improved hit rate of the L1 cache.

In the case of the second approach, the Compute Unified Device Architecture dynamic parallelism feature allows to execute the same kernel function recursively, while in the case of the first approach the repeated execution of the same kernel function is achieved through multiple looped iterations.

The dynamic parallelism feature makes it possible for the parent kernel to invoke multiple child kernel functions separately that are processing the data in parallel. The execution of the child kernel functions is achieved by using a CUDA stream for every child launch in order to strengthen the chances of concurrent execution of the child kernel functions [13].

In both the approaches, the tasks are partitioned to multiple blocks of threads. In the experimental tests, different sizes were tested for both the number of blocks and threads within a block and the best results were obtained using the following allocation of resources: the number of allocated thread blocks is the smallest integer greater than or equal to the ratio between the number of records and 1024; if the number of thread blocks is greater than 1, the size of a thread block is 1024; otherwise, the number of threads per block equals the number of records that have to be processed.

In the dynamic parallelism approach, the best results are obtained when the thread block size is a multiple of a warp size as this avoids the occurring of divergent threads within the warps. If the dimension of the thread block is not a multiple of a warp size, the parent kernel function processes the rest of the threads.

The child kernels functions cannot retrieve data directly from the shared memory owned by the parent kernel function. Whenever a child kernel function has to retrieve the data from the parent kernel's shared memory, it can receive it as a kernel function argument, or the respective value can be stored into the global memory.

Both of these methods have their drawbacks, the first one cannot pass an increased number of elements as arguments of the function, while the second method suffers an enormous penalty due to the performance characteristics of the global memory.

In the second approach, the synchronization process was used only when strictly necessary, because even if the dynamic parallelism feature offers the possibility to synchronize among child kernel functions and parent kernel functions, the synchronization process affects the overall performance of the application tremendously.

As it is stated in the official NVIDIA CUDA C Programming Guide[3], even though a single thread synchronizes, the process affects all the other threads that reside within the same thread block, even if they didn't perform a synchronization operation.

A considerable penalty that the dynamic parallelism feature brings is due to the fact that the device must keep a detailed track of the execution and also due to the whole dynamic parallelism management mechanism. In the following section, there are presented the experimental results and it is made an analysis of the two developed approaches.

5 Experimental results and performance analysis of the developed approaches

In this section, it is analyzed the performance of the two developed approaches that implement the inner JOIN operator in the latest Pascal Compute Unified Device Architecture (CUDA). The following hardware and software configurations have been used in the testing methodology: Intel i7-5960x operating at 3.0 GHz with 32GB (4x8GB) of 2144 MHz, DDR4 quad channel and the GeForce GTX 1080 NVIDIA graphics card with 8GB GDDR5X 256-bit from the Pascal architecture, the Windows 10 Educational operating system, the CUDA Toolkit 8.0 with the NVIDIA developer driver.

The average execution time for both the classical approach and the dynamical parallelism approach has been calculated using the "StopWatchInterface" included in the Compute Unified Device Architecture application programming interface, in order to define, create and manage timestamps and timers.

The set of developed tests computes the average execution times obtained in the two approaches, when implementing the inner JOIN operator, when the input data tables have a varying number of records, ranging from 64 to 1,048,576 and the output data table is the one computed through the JOIN operator. The execution time (measured in milliseconds) is computed as an average of 10,000 iterations, that has been calculated after eliminating the first ten supplementary iterations, as to be sure that the Graphics Processing Unit has attained the highest

[3]*http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#ixzz4NGV8NesH accessed on 10.16.2016, at 19:45*

clock frequency. In **Table 2** are presented the registered experimental results.

Table 2. The registered experimental results

No.	NTIR (records)	CIT (ms)	DPT (ms)
1	64	0.015679	0.019357
2	128	0.015685	0.019364
3	256	0.016446	0.021415
4	512	0.017627	0.021592
5	1,024	0.017945	0.021749
6	2,048	0.021622	0.027614
7	4,096	0.024816	0.030771
8	8,192	0.028999	0.039590
9	16,384	0.035901	0.045851
10	32,768	0.040149	0.050384
11	65,536	0.046978	0.057401
12	131,072	0.055711	0.065138
13	262,144	0.065791	0.077154
14	524,288	0.078988	0.089885
15	1,048,576	0.095049	0.109880
Total execution time - 10,000 iterations (h)		**0.001604**	**0.001937**
The system's power (kW)		**0.261000**	
The total energy consumption (kWh)		**0.000419**	**0.000505**
A comparison between the economic efficiency of the two approaches		The total energy consumption is **17%** lower when using the first approach	

The second column of this table contains the number of total input records (NTIR), the third column contains the average execution times when developing the classical CUDA implementation of the inner JOIN operator (CIT), while the last column contains the average execution times when developing the dynamic parallelism CUDA implementation of the inner JOIN operator (DPT).

Of particular interest was to analyze the economic efficiency of the two approaches. Thus, it was computed the total number of processed records, the total CIT time and the total DPT time for all the tests, taking into account all the 10,000 iterations. Afterwards, using a Voltcraft Energy Logger 4000 meter that measures the consumption of energy, it has been measured the system's power (expressed in kW) and it has been computed the total energy consumption (measured in kWh) for each approach.

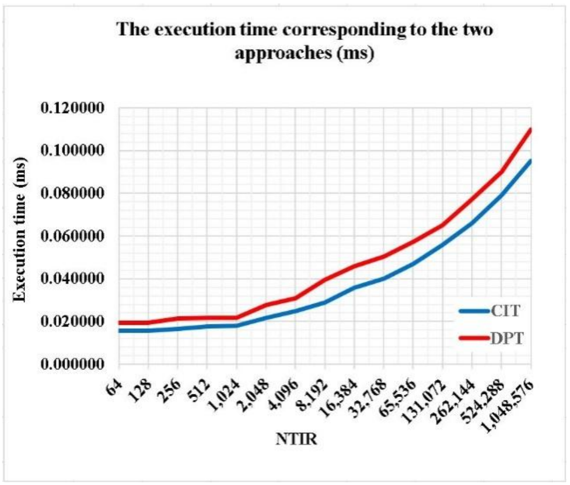

Fig.2. The execution time corresponding to the two developed approaches

After having executed the two approaches and having analysed the experimental results provided by the test suite, one can observe the following: in all the cases, the CIT value (corresponding to the classical approach) is lower than the DPT value (corresponding to the dynamic parallelism approach) (**Fig.2)**.

When running the test suite, the total execution time of all the 10,000 iterations and the corresponding system power consumption of the first approach were 17% lower than in the case of the second approach, thus the first approach offers an improved economic efficiency compared to the other one.

Although the dynamic parallelism feature allows the developer to make use of consecrated programming techniques, it suffers a considerable overhead due to the fact that the Graphics Processing Unit must monitor in detail the whole execution of the parent and child kernel functions and keep a detailed track of their execution, due to the way of how the management mechanism of the dynamic parallelism is implemented.

6 Conclusions

Both the developed approaches that implement in CUDA the JOIN operator in the Pascal architecture offer a high level of performance when processing high volumes of data (1,048,576 records processed in 0,1 milliseconds).

Although the dynamic parallelism feature allows a more robust implementation and makes it possible to generate work directly from the GPU, allowing the developer to tackle important programming techniques, like recursion, directly on the device, in the case of the inner JOIN operator the use of the dynamic parallelism in the Compute Unified Device Architecture creates a penalty on performance due to the overhead that is generated by invoking new child kernels.

The new Pascal Compute Unified Device Architecture offers an effective solution for processing huge data sets and data operators.

References

[1] S. Cook, *CUDA Programming. A Developer's Guide to Parallel Computing with GPUs,* 1st Edition, Morgan Kaufmann, San Francisco, 2012.

[2] D. M. Petroşanu and A. Pîrjan, "Economic considerations regarding the opportunity of optimizing data processing using graphics processing units", *Journal of Information Systems & Operations Management*, Vol. 6, Issue 1, May 2012, ISSN 1843-4711, pp. 204-215.

[3] A. Pîrjan, "Optimization Techniques for Data Sorting Algorithms", *Annals of DAAAM for 2011 & Proceedings of the 22nd International DAAAM Symposium*, pp. 1065-1066, Vienna, Austria, 23-26 November 2011.

[4] I. Lungu, A. Pîrjan and D. M. Petroşanu, "Optimizing the Computation of Eigenvalues Using Graphics Processing Units", *University Politehnica of Bucharest, Scientific Bulletin, Series A, Applied Mathematics and Physics*, Vol. 74, Issue 3, 2012, ISSN 1223-7027, pp.21-36.

[5] I. Lungu, G. Căruțașu, A. Pîrjan, S. V. Oprea and A. Bâra, "A Two-step Forecasting Solution and Upscaling Technique for Small Size Wind Farms located in Hilly Areas of Romania", *Studies in Informatics and Control*, Vol. 25, Issue 1, 2016, ISSN 1220-1766, pp. 77-86.

[6] I. Lungu, A. Bâra, G. Căruțașu, A. Pîrjan and S. V. Oprea, "Prediction Intelligent System in the Field of Renewable Energies Through Neural Networks", *Journal of Economic Computation and Economic Cybernetics Studies and Research*, Vol. 50, Issue1, 2016, ISSN online 1842– 3264, ISSN print 0424 – 267X, pp. 85-102.

[7] A. Pîrjan and D. M. Petroşanu, "Dematerialized Monies – New Means of Payment", *Romanian Economic and Business Review*, Vol. 3, Issue 2, 2008, ISSN 1842 – 2497, pp. 37-48.

[8] A. Pîrjan and D. M. Petroşanu, "A Comparison of the Most Popular Electronic Micropayment Systems", *Romanian Economic and BusinessReview,* Vol. 3, Issue 4, 2008, ISSN 1842–2497, pp. 97-110.

[9] A. Pîrjan and D. M. Petroşanu, "Solutions for Developing and Extending Rich Graphical User Interfaces for Office Applications", *Journal of Information Systems & Operations Management*, Vol. 9, Issue 1, May 2015, ISSN 1843-4711, pp. 157-167.

[10] A. Pîrjan, "The Optimization of Algorithms in the Process of Temporal Data Mining Using the Compute Unified Device Architecture", *Database Systems Journal*, Vol. I, Issue 1, 2010, ISSN 2069-3230, pp. 37-47.

[11] ***, Whitepaper NVIDIA Tesla P100, "The Most Advanced Datacenter Accelerator Ever Built - Featuring Pascal GP100, the World's Fastest GPU".

[12] J. Wang and Y. Sudhakar, "Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications", Proceedings of the *2014 IEEE International Symposium on Workload Characterization (IISWC),* Raleigh, North Carolina, USA, 26-28 October 2014.

[13] F. Wang, J. Dong and B. Yuan, "Graph-based Substructure Pattern Mining Using CUDA Dynamic Parallelism", *Intelligent Data Engineering and Automated Learning–IDEAL*, Springer, 2013, pp. 342–349.

[14] J. DiMarco and M. Taufer, "Performance Impact of Dynamic Parallelism on Different Clustering Algorithms", *SPIE Defense, Security, and Sensing*, International Society for Optics and Photonics, 2013, pp. 87.

Alexandru PIRJAN has graduated the Faculty of Computer Science for Business Management in 2005. He holds a BA Degree in Computer Science for Business Management since 2005, a MA Degreein Computer Science for Business since 2007, a PhD in Economic Informatics since 2012 and a Habilitation Certificate in the Economic Informatics field since 2016. He joined the staff of the Romanian-American University as a stagiary Teaching Assistant in 2005, a Lecturer Assistant in 2008 and a Lecturer since 2014. He is currently a member of the Department of Informatics, Statistics and Mathematics from the Romanian-American University. He is the author of more than 45 journal articles, 8 scientific books and a member in 8 scientific research projects. His work focuses on parallel processing architectures, parallel programming, database applications, artificial intelligence and software quality management.