IMPROVING PARALLEL PROGRAMMING IN THE COMPUTE UNIFIED DEVICE ARCHITECTURE USING THE UNIFIED MEMORY FEATURE

Alexandru PÎRJA[1]
Dana-Mihaela PETROŞANU[2]

ABSTRACT

One of the most important improvements within the Compute Unified Device Architecture (CUDA) 6.5 version, launched in August 2014, is the support for Unified Memory, a feature that simplifies the memory management, by providing a unified pool of managed memory, shared between the Central Processing Unit (CPU) and the Graphic Processing Unit (GPU). The system automatically migrates data allocated in the Unified Memory between the host memory and the device memory. In this paper, we analyze this new CUDA feature, its advantages and limitations, by developing two versions of an application that computes the scalar product: one using the Unified Memory concept and the other one using the classical approach based on the cudaMalloc instruction (for memory allocation) and cudaMemcpy (for transferring the data between the host and the device).

Keywords: CUDA, parallel programming, Unified Memory, Graphics Processing Unit, Central Processing Unit.

1. INTRODUCTION

The Compute Unified Device Architecture (CUDA), developed by the Nvidia Company, is a perfect symbiosis between a parallel computing platform and a programming model, designed to increase the overall computing performance by using the processing power of the Graphics Processing Units (GPUs). CUDA graphics processing units can be used for general-purpose computations and not exclusively for graphic rendering. Unlike central processing units, GPUs are based on a parallel transfer architecture that emphasizes processing through multiple concurrent execution threads. Nowadays many researchers, scientists and developers use the CUDA enabled GPU's huge available computing power in a wide range of applications or research papers from various fields such as medicine, aeronautics, finance, mathematics, informatics, chemistry, physics.

One of the main advantages offered by the CUDA enabled Graphics Processing Units is the fact that many programming languages (C, C++ and Fortran) can be processed directly by the GPUs, without requiring any assembly language. On the other hand, through these processors, worldwide developers have a huge opportunity to do general-purpose computations in scientific or engineering applications, on a wide range of platforms, at a spectacular computing speed [1]. Using a teraflop of floating point performance, the CUDA

[1] PhD, Faculty of Computer Science for Business Management, Romanian-American University, 1B, Expozitiei Blvd., district 1, code 012101, Bucharest, Romania, E-mail: alex@pirjan.com
[2] PhD, Department of Mathematics-Informatics I, University Politehnica of Bucharest, 313, Splaiul Independentei, district 6, code 060042, Bucharest, Romania, E-mail: danap@mathem.pub.ro

enabled GPUs incorporated in notebooks, personal computers, workstations, clusters or supercomputers are capable of managing much more than only graphics rendering [2].

In the following, we briefly present the history and evolution of the GPUs and of the GPU computing. First, the GPUs' solely purpose was graphic acceleration and they could manage only specific functions. Towards the late 1990s, the hardware has become programmable and this feature has been developed and has gradually increased, culminating in 1999, with the first NVIDIA GPU. This moment represents a real milestone in the GPU's history, as starting from this point, the GPU has become of paramount importance not only for game developers, but also for researchers, developers and scientists. Consequently, a new concept has appeared: the General-Purpose computation on Graphics Processing Units (GPGPU) [1], [3].

Initially, the GPGPU solution proved to be extremely difficult even for those researchers or developers who were acquainted with graphics programming languages. Developers were forced to formulate their scientific calculations only through problems that could be represented by polygons. GPGPU seemed to be almost impossible to use for those who did not know the latest graphics Application Programming Interfaces until a group of researchers from the Stanford University decided to change the concept of GPU, imagining it as a "streaming processor".

Researcher Ian Buck and his team presented in 2003 the first programming model that extends the C programming language with data-parallel elements. The compiler and runtime system named Brook, used concepts as reduction operators, kernels, streams and managed to harness the GPU as a general-purpose processor in a high-level language. The Brook code proved to be much easier to write than previous GPU code and was up to seven times faster than any of the existing similar codes [1].

In this context, in order to combine its extremely fast hardware with an intuitive software and the appropriate hardware tools, the NVIDIA Company invited researcher Buck to join them in the research of a perfect solution to run C code on Graphic Processing Units. Using a specialized hardware and software appropriate elements, NVIDIA launched in 2006 the Compute Unified Device Architecture, the first solution in the world for general purpose computing on GPUs.

Nowadays, more and more companies and researchers need to conduct their daily activities based on the huge parallel processing power of the GPUs. Many scientific researches are well suited for being further optimized on graphic processing units, such as the dynamic development and assembly of learning objects in a math learning environment [4], cryptographic solutions implemented by the use of the multilayered structural data sectors [5], the distributed systems [6]. In the future, taking into account that more and more people need to access remotely resources and information, the GPUs' impact on human's life will grow as more and more countries have Internet access [7].

In order to benefit from the GPUs' performance, users need to write their own code. In this purpose, a very useful tool provided by the Nvidia Company is the CUDA Toolkit [8], that offers a comprehensive development environment for C and C++ developers, includes a compiler, math libraries, specialized tools for optimizing software performance, programming guides, code samples, user manuals, API references and other documentation. There are also available solutions for other programming languages (Fortran, C#, Python).

Worldwide, the CUDA programming is studied in over 500 universities and colleges, Research and Training Centers or Centers of Excellence. In the latest list of the world's most powerful 500 supercomputers[3] one can see that 38 of them use Nvidia GPU-based accelerators.

In this paper, we analyze a CUDA feature available in the newest CUDA 6.5 version, the Unified Memory, its advantages and limitations, by developing two versions of an application that computes the scalar product: one uses the Unified Memory concept and the other one uses the classical approach based on the *cudaMalloc* instruction (for memory allocation) and *cudaMemcpy* (for transferring the data between the host and the device). In the following, we present the main enhancements brought by the version 6.5 of the CUDA Toolkit and we analyze the most important of them, the Unified Memory.

2. ENHANCEMENTS OF THE PARALLEL PROGRAMMING IN CUDA 6.5

Since its first version launched in 2006, CUDA has evolved significantly reaching its newest version, CUDA 6.5 that has been released in August 2014. This version brings a series of enhancements to the previous ones, thus leading to significant improvements regarding the parallel programming. In the following, we present some of the most important features available in the version 6.5 of the CUDA Toolkit[4]:

- The Unified Memory is a significant feature of the CUDA Toolkit 6.5. It enables applications to access both the Central Processing Unit's (CPU's) memory and the Graphic Processing Unit's (GPU's) memory, without requiring data to be manually copied between these two types of memory. Thus, the programming is simplified as some steps are automatically executed without needing explicit programming instructions from the developer.
- The development and recompilation of applications in order to run on ARM 64-bit systems with Nvidia GPUs.
- The Drop-in Libraries represent another important feature that is offered by the CUDA 6.5 version, as some of the existing Central Processing Unit libraries were replaced with equivalent GPU-accelerated ones. Thus, one has obtained improvements, accelerating up to 8X a wide class of applications that benefit from these libraries, such as Basic Linear Algebra Subprograms (BLAS) or implementations of the Fast Fourier Transform (FFT) algorithm.
- Another enhancement available in CUDA 6.5 is the Multi-GPU scaling. The new BLAS library scales the performance automatically over up to eight GPUs in a single node, thus providing over nine teraflops of double-precision performance per node and support for workloads of up to 512 GB. The new Fast Fourier Transform (FFT) library scales up to 2 GPUs in a single node, thus allowing larger transform sizes and higher throughput.

[3] http://www.top500.org/, accessed on 04.06.2014
[4] https://developer.nvidia.com/cuda-toolkit

- Enhanced support for the development tools: Microsoft Visual Studio 2013 support, enhanced debugging support for CUDA FORTRAN applications, replay feature in Visual Profiler and *nvprof*, the *nvprune* tool for optimizing the size of objects.

In the following, we present details regarding the most important improvement available in the CUDA Toolkit 6.5: the Unified Memory. Before CUDA 6, the memories of the Central Processing Unit (CPU) and of the Graphic Processing Unit (GPU) were distinct and separated by the PCI-Express bus. In order to share data between the CPU and the GPU, the programmers had to allocate the data in both memories and use explicit copy instructions from the host to the device and from the device to the host. These operations are time and resource consuming for the programmer.

The Unified Memory is a feature that eliminates the above mentioned limitation and simplifies the memory management in GPU-accelerated applications, by providing a unified pool of managed memory, shared between the Central Processing Unit (CPU) and the Graphic Processing Unit (GPU). The CPU and GPU can access the Unified Memory using a single pointer. Thus, the process can be summarized as follows: the system automatically transfers data from the Unified Memory, facilitating changes between the CPU (host) and GPU (device). On one hand, the Unified Memory acts as CPU memory when data inside it is required and used for CPU processing and on the other hand, it acts as GPU memory when data inside it is required and used for GPU processing.

From the programmer's point of view, there are two main benefits from using the Unified Memory [4]:

- The Unified Memory alleviates the parallel programming effort for developing CUDA applications. Using the Unified Memory, programmers can directly develop parallel CUDA kernels, without wasting time with details regarding memory allocation and data copying in the device memory. This significantly simplifies the programming within the CUDA platform and the porting of existing code to GPUs when necessary. Unified Memory simplifies the memory model and its management, making complex data structures much easier to use within the device code.
- By transferring data between the CPU and the GPU, the Unified Memory can provide to the GPU the same level of performance as for using local data, while providing access to globally shared data. All these features are achieved via the CUDA driver and runtime that give the advantage of faster CUDA kernels prototyping. A goal that must be attained when migrating to Unified Memory is to employ the full bandwidth of the CUDA processors.

One important aspect that must be taken into account for is that there are certain technical situations when a CUDA software using streams and asynchronous memory copies has a higher degree of performance than a software that uses solely Unified Memory. This happens because the CUDA runtime cannot have the same amount of information that a programmer has, concerning the data that must be processed. The CUDA developers have at their disposal a complex set of performance enhancing tools that must be extensively used in order to successfully performance tune a CUDA application: asynchronous memory copies, device memory allocation, CPU-GPU concurrency, etc. The Unified Memory concept must be looked upon as an added bonus to improve the parallel computing

productivity without sacrificing the powerful performance tuning solutions available to experienced programmers.

Starting with the CUDA 4 version, the Compute Unified Device Architecture has offered support for Unified Virtual Addressing (UVA). One should note that the Unified Virtual Addressing and the Unified Memory are two different concepts and should not be confused. In the following, we present some essential details regarding these two concepts, in order to differentiate them and eliminate the risk of confusion [4]:

- UVA offers a unique address space covering all the system's memory and enables pointers to be accessed from the GPU code, regardless of their position: in the host memory, in the device memory or in the on-chip shared memory.
- The Unified Virtual Addressing facilitates the usage of the *cudaMemcpy* and does not require specifying the input or output parameters' locations.
- UVA also facilitates the "zero-copy" host memory that represents a pinned part of the host memory that can be accessed directly through the device code, over the Peripheral Component Interconnect Express (PCI-Express) bus, without using a *memcopy* instruction. Although the "zero-copy" offers a number of advantages, similar to those of the Unified Memory, it has the disadvantage of having low bandwidth and generating high latency (that leads to a significant performance penalty).
- The Unified Virtual Addressing does not automatically ensure the physical transfer of data from one location to another, as it is the case for the Unified Memory. The changes of the CUDA runtime and of the device driver, made it possible for the Unified Memory to transfer data automatically between the host memory and the device memory.

The Unified Memory in CUDA 6.5 opens up new features and improvements. A significant benefit is that it offers to the developers the ability to write CUDA programs much easier. The *cudaMemcpy*( ) instruction, required until the CUDA 6 version for transferring the data between the host and the device, is no longer mandatory but it remains available to the developer as an important optimization tool. Using the new instruction *cudaMallocManaged*() for allocating Unified Memory, one is able to share complex data structures between the CPU and the GPU. Consequently, the CUDA programming is much easier to achieve because kernel programs can be written directly, instead of wasting time and resources in writing code for data management, for maintaining duplicates of all data in host and device memory. The "classical" CUDA instructions such as *cudaMemcpy*() or *cudaMemcpyAsync*( ) are still available and extremely useful when one wishes to obtain a high level of performance or to optimise the code, but their usage is no longer required.

For the subsequent CUDA versions, the Nvidia Company intends to bring more hardware and software improvements regarding the flexibility and performance of applications based on the Unified Memory, by adding features related to data prefetching, migration hints and support for a wider variety of operating systems[5].

[5] http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, accessed on 04.10.2014

In the following, we present a performance analysis of using the Unified Memory concept in contrast to the classical approach. We have developed two versions of an application that computes the scalar product: one using the Unified Memory concept and the other one using the classical approach based on the *cudaMalloc* instruction (for memory allocation) and *cudaMemcpy* (for transferring the data between the host and the device). Our purpose is to highlight some of the advantages and limitations of the Unified Memory.

3. PERFORMANCE ANALYSIS OF USING THE UNIFIED MEMORY CONCEPT IN CONTRAST TO THE CLASSICAL APPROACH

In this section, we analyze the performance of an application that computes the scalar product, using the following configuration: Intel i7-2600K operating at 3.4 GHz with 8 GB (2x4GB) of 1333 MHz, DDR3 dual channel and the GeForce GTX 680 NVIDIA graphic card from the Kepler architecture. Programming and access to the GPU used the CUDA toolkit 6.5, with the NVIDIA driver version 340.62. In addition, all processes related to graphical user interface have been disabled to reduce the external traffic to the GPU.

The GeForce GTX 680 NVIDIA graphic card, released in March 2012, has the following technical specifications: 28 nm fabrication technology, 3.54 billion of transistors, 1536 CUDA cores, 8 streaming multiprocessors, the graphics clock - 1006 MHz, the boost clock - 1058 MHz, 2048 MB of memory in the standard configuration having an effective clock of 6000 MHz, a 256-bit gDDR5 memory interface width and 192.2 GB/sec memory bandwidth, texture fill rate - 128.8 billion/sec, 128 texture units, 32 ROP units, the maximum board power (TDP) -170 Watts. The GTX 680's architecture, GK104, brings significant improvements regarding the streaming multiprocessors (named SMX units) delivering a high level of performance and power efficiency.

In order to compute the average execution time that the GPU and CPU spend for generating the sample arrays, computing the scalar product and the total execution time, we have used the "*StopWatchInterface*" offered by the CUDA application programming interface (API) for defining, creating and managing the timers. For computing correctly the average execution time, we have created separate timers for each of the events. Taking into account the fact that the execution is asynchronous (the GPU begins to execute the code while the CPU continues the execution of the next code line before the GPU has finished), we have used the "*cudaDeviceSynchronize*()" instruction to ensure that there are no more execution threads left before timing the event. In **Figure 1** is presented an example of a timer, used for computing the generating time of the sample data arrays.

```
StopWatchInterface *Generare = NULL;
sdkCreateTimer(&Generare);

printf("...allocating BOTH THE CPU AND GPU UNIFIED MEMORY.\n");
cudaMallocManaged(&A,DIMENSIUNE);
cudaMallocManaged(&B,DIMENSIUNE);

srand(123);
sdkResetTimer(&Generare);
sdkStartTimer(&Generare);

//Generating the sample DATA ARRAYS
for (i = 0; i < ELEM_N; i++)
{
    A[i] = AleatorFloat(0.0f, 1.0f);
    B[i] = AleatorFloat(0.0f, 1.0f);
}
sdkStopTimer(&Generare);
printf("Generating the sample data: %f msecs.\n",sdkGetTimerValue(&Generare));
```

Figure 1. Measuring the generating execution time using CUDA timers

The first set of tests computes the average execution times obtained by the GPU and CPU when generating the sample arrays ($\mathbf{GT_1}$), computing the scalar product ($\mathbf{SPT_1}$) and the total execution time ($\mathbf{TET_1}$) for the CUDA application developed using the classical approach, without the Unified Memory feature. We have randomly generated 1024 pairs of arrays having the size of 16384 float type elements.

In **Table 1**, we present the obtained experimental results when benchmarking the CUDA application. Each of the 1-10 lines of the table represents an average of 10,000 iterations, computed after having removed the first five results, in order to be sure that the GPU reaches its maximum clock frequency. The unit of measure is milliseconds (ms). For each case, we have also computed the difference $\mathbf{\Delta_1 = TET_1 - (SPT_1 + GT_1)}$, that is represented, in the developer's code, by the time spent when:

- allocating variables' memory on the CPU;
- allocating variables' memory on the GPU;
- copying data from the CPU to the GPU;
- copying the obtained results from the GPU back to the CPU.

In order to facilitate the result's interpretation and conclusions, we have also computed the average of all the obtained cases.

Table 1. The execution time without using the Unified Memory feature (ms)

No.	Generating the sample arrays on the CPU ($\mathbf{GT_1}$)	Computing the scalar product on the GPU ($\mathbf{SPT_1}$)	Total execution time on the GPU and CPU ($\mathbf{TET_1}$)	Difference ($\mathbf{\Delta_1}$)
1	862	1.20	971	107.8
2	871	1.19	995	122.81
3	920	1.23	1046	124.77
4	869	1.18	986	115.82
5	869	1.19	978	107.81
6	864	1.21	973	107.79
7	873	1.25	987	112.75

8	876	1.24	985	107.76
9	884	1.21	995	109.79
10	866	1.20	977	109.8
Average	**875.4**	**1.21**	**989.3**	**112.69**

In the second set of tests, we have computed the average execution times obtained by the GPU and CPU when generating the sample arrays ($\mathbf{GT_2}$), computing the scalar product ($\mathbf{SPT_2}$) and the total execution time ($\mathbf{TET_2}$) for the CUDA application developed using the new Unified Memory feature. The 1024 pairs of arrays that we have randomly generated have the size of 16384 float type elements.

In **Table 2**, we present the obtained experimental results when benchmarking the CUDA application. Each of the 1-10 lines of the table represents an average of 10,000 iterations, computed after having removed the first five results, in order to be sure that the GPU reaches its maximum clock frequency. The unit of measure is milliseconds (ms). For each case, we have also computed the difference $\mathbf{\Delta_2 = TET_2 - (SPT_2 + GT_2)}$, that is represented, in the developer's code, by the time spent when:

- allocating Unified Memory;
- all the other actions that are automatically processed by the CUDA runtime (copying data from the CPU to the GPU and copying the obtained results from the GPU back to the CPU).

In order to facilitate the result's interpretation and conclusions, we have also computed the average of all the obtained cases.

Table 2. The execution time using the Unified Memory feature (ms)

No.	**Generating the sample arrays on the CPU ($\mathbf{GT_2}$)**	**Computing the scalar product on the GPU ($\mathbf{SPT_2}$)**	**Total execution time on the GPU and CPU ($\mathbf{TET_2}$)**	**Difference ($\mathbf{\Delta_2}$)**
1	1221	127	1398	50
2	1168	128	1343	47
3	1163	128	1341	50
4	1130	127	1307	50
5	1215	127	1391	49
6	1157	127	1332	48
7	1151	127	1325	47
8	1170	127	1352	55
9	1168	132	1348	48
10	1131	128	1310	51
Average	**1167.4**	**127.8**	**1344.7**	**49.5**

In the following, we present and analyze a comparison between the experimental results that we have obtained by running the CUDA code of the scalar-product applications in the two cases: without and with the Unified Memory feature. First, we compare the average times spent by the CPU when generating the sample arrays in the two cases, $\mathbf{GT_1}$ and $\mathbf{GT_2}$ (**Figure 2).** Analyzing the results, we have noticed that, when randomly generating 1024 pairs of arrays having the size of 16384 float type elements on the CPU, we have recorded an average execution time of up to 1.33x higher when using the Unified Memory, than in the case when we have used separate variables for the host and the device. This difference

is accounted for if we take into account the supplementary operations that are being performed by the CUDA runtime in order to make data available to both the device and the host memory. In the case of the **GT1**, the data is available only to the host and it needs further copy operations in order to make it available to the GPU for computing the scalar product. In what concerns the **GT2**, as we have used the Unified Memory approach, the data is already available to the GPU at the end of the generating process.

We have compared then the average execution times spent by the GPU when computing the scalar product in the two cases, $\mathbf{SPT_1}$ and $\mathbf{SPT_2}$ **(Figure 3).** When comparing the $\mathbf{SPT_1}$ and $\mathbf{SPT_2}$ values we have noticed that, when computing on the GPU the scalar product of our generated sample data arrays, we have recorded an average execution time of up to 105.62x higher when using the Unified Memory, than in the case when we have used separate variables for the host and the device. This difference accounts for other operations that are being performed by the CUDA runtime in order to make data available to both the device and the host memory. In the case of the $\mathbf{SPT_1}$, the data is available only to the device and it needs further copy operations from device to host in order to make it available to the CPU. Regarding the $\mathbf{SPT_2}$, as we have used the Unified Memory approach, the data is already available to the CPU after the scalar product has been computed.

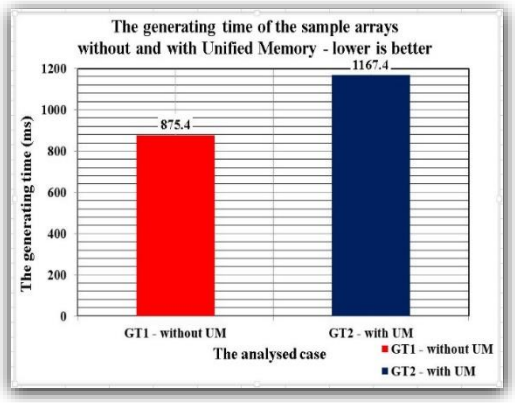

Figure 2. The average times spent by the CPU when generating the sample arrays

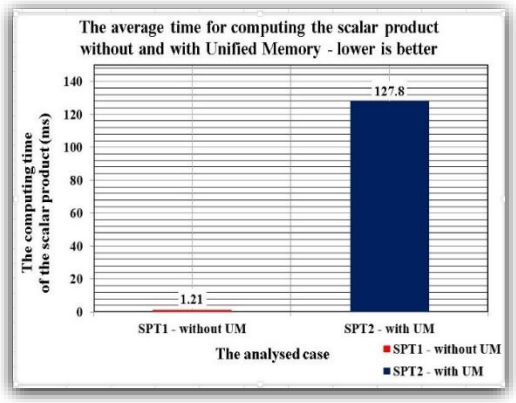

Figure 3. The average execution times spent by the GPU when computing the scalar product

Then we have compared the total execution times on the GPU and CPU obtained in the two analysed cases, $\mathbf{TET_1}$ and $\mathbf{TET_2}$ **(Figure 4).** Comparing the average total execution times on the GPU and CPU, we have noticed that the time recorded when using the Unified Memory is up to 1.36x higher than in the other case. Although $\mathbf{TET_1}$ is lower than $\mathbf{TET_2}$, the Unified Memory offers the advantage of faster prototyping of the CUDA kernels and therefore, of the entire application.

Finally, we have compared the differences $\mathbf{\Delta_1}$ and $\mathbf{\Delta_2}$ obtained in the two analysed cases (**Figure 5**). The $\mathbf{\Delta_1}$ and $\mathbf{\Delta_2}$ time differences cover different operations performed by the CUDA runtime but their comparison is mandatory and relevant in order to measure, compare and understand how the CUDA runtime handles the Unified Memory code.

Analyzing the results, we have noticed that the $\mathbf{\Delta_2}$ is up to 2.28x lower than $\mathbf{\Delta_1}$. One must note that $\mathbf{\Delta_1}$ includes several supplementary operations (e.g. copying data from the CPU to the GPU, copying the obtained results from the GPU back to the CPU). In the case of $\mathbf{\Delta_2}$, most of these operations have already been processed and timed during the previous steps and are included in their execution times (e.g. $\mathbf{GT_2}$ and $\mathbf{SPT_2}$).

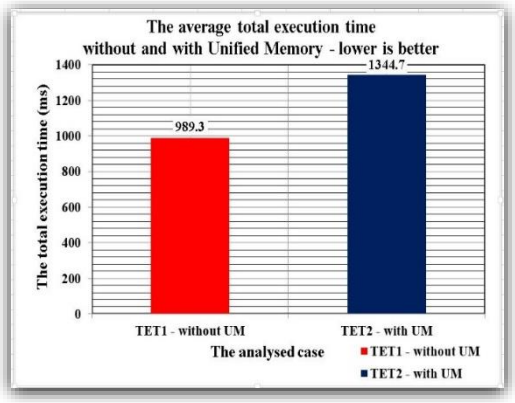

Figure 4. The average total execution times on the GPU and CPU

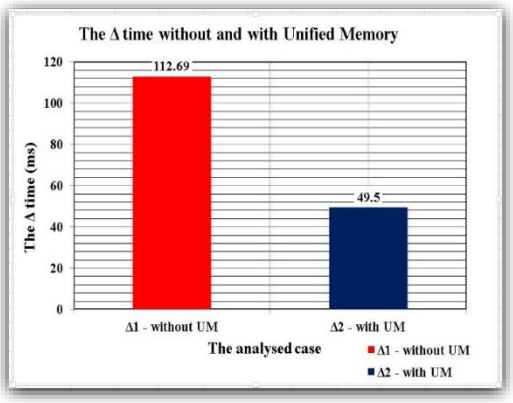

Figure 5. The Δ times

4. CONCLUSIONS

After having analyzed the obtained experimental results, we have concluded the following:

- the average generating time of the sample arrays on the CPU is 1.33x higher when using the Unified Memory than in the case when we have used separate variables for the host and the device (1167.4 ms compared to 875.4 ms);
- the average execution time of the scalar product on the GPU is 105.62x higher when using the Unified Memory, than in the case when we have used separate variables for the host and the device (127.8 ms compared to 1.21 ms);
- the average total execution time on the GPU and CPU is up to 1.36x higher when using the Unified Memory than in the other case (1344.7 ms compared to 989.3 ms);
- the average time difference is up to 2.28x lower when using the Unified Memory than in the other case (49.5 ms compared to 112.69 ms).

The execution times are, in all the cases lower for the "classical" approach, based on the *cudaMalloc* instruction (for memory allocation) and *cudaMemcpy* (for transferring the data between the host and the device). All of these noticeable differences between the execution times of our two approaches come from the fact that when using Unified Memory, the CUDA runtime automatically manages a lot of subsequent operations, spread across different execution stages – taking the burden from the programmer's shoulders and reducing the necessary time for developing the application.

One must not forget that there are certain technical situations when a CUDA application that uses streams and asynchronous memory copies has the potential to obtain a higher degree of performance than an application that uses only Unified Memory. It is obvious

that the CUDA runtime cannot have the same amount of information that a programmer has, concerning the data. In order to successfully performance tune a CUDA application, developers must make use of a complex set of performance enhancing tools that are available to them, such as: CPU-GPU concurrency, asynchronous memory copies, device memory allocation, etc. Developers must use the Unified Memory concept as an added bonus to improve the parallel computing productivity without sacrificing the powerful performance tuning solutions available to the experienced programmers.

The Unified Memory approach offers to the developer an undisputed advantage when factoring the CUDA kernels and even the whole application. Even in our applications, we have felt a significant improvement and a reduced time to program the Unified Memory application with ease and fluency. This improvement in the necessary development time for an application will certainly be more dramatic when dealing with complex data structures applications that need subsequent copies between the device and the host.

REFERENCES

[1] Sanders J., Kandrot E., CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, New Jersey, 2010.

[2] Hwu Wen-mei W., GPU Computing Gems Jade Edition, Morgan Kaufmann, 2011.

[3] Nickolls J., Buck I., Garland M., Skadron K., Scalable parallel programming with CUDA, Queue, vol. 6, no. 2/ March-April 2008, pp. 40-53.

[4] Stănică L., Crișan D., Dynamic development and assembly of learning objects in a math learning environment, Journal of Information Systems and Operations Management, vol. 6, no. 1, ISSN 1843-4711, Ed. Universitară, 2012.

[5] Tăbușcă A., A new security solution implemented by the use of the multilayered structural data sectors switching algorithm (MSDSSA), Journal of Information Systems & Operations Management, vol.4, no.2, ISSN 1843-4711, Ed. Universitară, 2010.

[6] Garais E., Maintenance phase in distributed application life cycle using UP Model, Proceedings of the 12th International Conference On Informatics In Economy (IE 2013), Education, Research & Business, ISSN 2284-7472.

[7] Tăbușcă, S., The Internet Access as a Fundamental Right, Journal of Information Systems and Operations Management, vol.4, no.2, ISSN 1843-4711, Ed. Universitară, 2010.

[8] ***, Nvidia CUDA Compute Unified Device Architecture - Programming Guide, Version 6.5, Nvidia Whitepaper, 2014.