MANAGING GRAPHICS PROCESSING UNITS' MEMORY AND ITS ASSOCIATED TRANSFERS IN ORDER TO INCREASE THE SOFTWARE PERFORMANCE

Alexandru Pîrjan [1*]

ABSTRACT

This paper is focused on analyzing a key aspect, the management of the Graphics Processing Units' (GPUs) memory, which is of paramount importance when developing a software application that makes use of the Compute Unified Device Architecture (CUDA). The paper tackles important technical aspects that can affect the overall performance of a CUDA application such as: the optimal alignment in memory of the data that is to be processed, obtaining optimal memory access patterns that facilitate the retrieving of instructions; aligning to the L1 cache line according to its size, taking into account the balance achieved between single or double precision and the effect on how much memory is being used; joining more kernel functions into a single one in certain situations, benefiting from the increased speedup offered by putting into use the shared and cache memory, adjusting the code to the available memory bandwidth by taking into account the memory latency and the need to transfer data between the host and the device.

KEYWORDS: *CUDA, GPU, Memory, Kernel Function, Software Performance*

1. INTRODUCTION

One of the most important technical aspects that affect the performance of most software applications are the memory bandwidth and latency. The memory bandwidth measures the quantity of data that can be transferred to or from a certain point in a certain amount of time. The memory latency targets the time needed for an operation to respond to a certain request and have the data available. In the case of the Graphics Processing Units that offer support for the Compute Unified Device Architecture, the appropriate management of these two quintessential technical aspects affects to a great extent the performance of the developed application.

Many scientific articles have made determined efforts to devise optimization solutions that target the Compute Unified Device Architecture enabled graphics processing units [1], [2], [3]. The general purpose graphics processing unit (GPGPU) offers new possibilities to overcome the limitations of traditional processors by offering a huge parallel processing power potential, which can be harnessed to optimize data processing. Achieving data processing at high speeds with low costs is of great importance in a large

[1*] corresponding author, Lecturer PhD, Faculty of Computer Science for Business Management, Romanian-American University, 1B, Expozitiei Blvd., district 1, code 012101, Bucharest, Romania, alex@pirjan.com

number of applications: in processing electronic payments [4], [5], in developing cryptographic algorithms [6], in implementing high-performance web solutions [7], in complex office solutions [8], in Artificial Neural Networks [9], [10], in Resource Description Framework query languages [11].

The classical central processing unit implements a memory model that is described in the literature as being linear and flat [12]. This model implies that all of the central processing unit’s cores have almost unrestricted access to memory, no matter where it resides. Most of the modern central processing units offer three levels of cache memory: L1, L2 and L3. Programmers who focus on developing optimization solutions for the central processing unit have a good knowledge of these three levels and make an extended use of their characteristics.

In spite of this, there are a lot of programmers who regularly or frequently are ignoring these types of memory when developing their applications, mainly due to the tendency that has been induced in recent years by modern programming languages. These languages, in the desire to achieve as much abstraction as possible, have a tendency to detach the programmer from the hardware. Time has proven that this approach often results in an increased productivity, because evolved compilers deal with the task of abstracting, thus allowing problems to be tackled without losing precious time with this matter. But the current state of technology makes it mandatory to fully comprehend the hardware and its features in order to achieve the highest possible level of performance when harnessing the parallel processing power of an architecture.

The graphics processing unit offers more types of memory that can be used to store and retrieve data. Each of these memory types differ in terms of performance (bandwidth, latency). The CUDA threads can access data from multiple memory addresses during the execution. Each thread contains a local private memory area. Each thread block offers a shared memory area accessible by all the threads within the block. The lifetime of the shared memory is the same with the lifetime of the thread block, in which it resides. All the threads have access to the same global memory. The architecture also provides two supplementary memory types that are read-only and are accessible by all the threads: the constant memory and the texture memory. The lifetime of the global, constant and texture memory is the same with the kernel function's lifetime [13].

In what concerns the memory hierarchy, each streaming multiprocessor (SM) contains a set of 32-bit register memory, together with a shared memory area that can be easily accessed by every core of the streaming multiprocessor, but it is not visible to the other streaming multiprocessors. The size of the shared memory varies depending on the graphics processor architecture and the available number of registers. Besides the shared memory, a streaming multiprocessor offers two levels of cache memory, one for texture and another one for constants [14].

In order to improve the software performance for a CUDA developed application, the programmers must optimize the number of active threads and to balance their memory resources: the number of registers and threads used per multiprocessor, the global memory bandwidth and the percentage of memory allocated to each thread. When one develops a CUDA application, he must employ a progressive approach, programming his application in the first phase using only global memory. In a second phase, he can consider implementing shared, constant, register and other types of memory.

In this paper there are analyzed important technical aspects that can influence the overall performance of an application developed for CUDA enabled GPUs: the increased speedup offered by putting into use the shared and cache memory; the alignment of data in memory; optimal memory access patterns; aligning to the L1 cache line; the balance achieved between single or double precision and its effect on memory usage; joining more kernel functions into a single one; adjusting the code to the available memory bandwidth in accordance with the memory latency and the necessity to transfer data between the host and the device.

In the following, the mechanism of caching data is analyzed on both the central and the graphics processing unit, highlighting their common characteristics as well as their main differences.

2. AN ANALYSIS OF THE CACHE MECHANISM

In the case of programs that process data sequentially, especially in the case of sequential programs that make extensive use of cycle operators, after a certain function has been called, the odds are high that in the near future the function will be used again. The chances are very high for certain memory locations to be used frequently for storing or retrieving information. The principle of locality states that it is very probable that one will need to access again, in a short time-frame, a certain memory address or a certain portion of a source code after having already accessed it once [12].

When compared to the processing speed of a central processing unit, the Dynamic Random-Access Memory (DRAM) performance lags several orders of magnitude behind. If there had not been for the cache memory to compensate the low speed of the DRAM, most part of the central processing unit’s processing power would have been wasted as the processor would be bound to the DRAM’s bandwidth and latency. The same situation stands true for the graphics processing unit as some memory processes are insubstantial when compared to the necessary minimum resources that can be allocated to handle them, thus resulting in a reduced peak memory efficiency.

The cache memory actually consists in a type of memory that offers very high speeds and is situated in close proximity to the processing core. The production process of this type of memory is very expensive and influences the final selling price of the processor. Depending on the types of cache memory, the sizes and speeds can vary significantly. For example, the L1 cache has a size of around 16 to 64 K and offers the highest level of performance. This type of cache memory is assigned to a certain core of the central processing unit. The L2 cache's size varies between 256 and 512 K, but it operates at a lower speed than the L1 cache. The L3 cache has been introduced most recently, having a size of a few megabytes and just like the L2 cache it can be used by more processor cores or assigned to specific ones. On typical central processing units, the L3 cache is allocated among the processor cores, thus facilitating the communications between the cores at high speeds.

In the case of the CUDA-enabled GPUs, the Fermi architecture brought for the first time a portion of cache memory which is entirely hardware managed. This architecture also introduced a L1 cache associated with each streaming multiprocessor that offers the possibility to be managed by the programmer or automatically, by the hardware. Along with this L1 cache, the architecture also implemented a L2 cache that is allocated among the streaming multiprocessors. This addition facilitates the communication between the threads belonging to the same processor, not having to use the low speed global memory when sharing small amounts of data. The atomic operations benefit a lot from the introduction of this type of memory, as a homogenous value at a certain memory address is available to the streaming multiprocessors. Up till this type of memory was introduced a streaming multiprocessor had to store data into the low speed global memory and retrieve it afterwards in order to be certain of the cohesion among the CUDA cores.

Developers are primarily monitoring the global memory's bandwidth while memory latency is compensated on the Compute Unified Device Architecture by invoking threads originating in different warps. Like it has been depicted before, a CUDA device implements more types of memory, each type having a distinctive scope, different lifetimes within the application and its unique caching characteristics. The global memory is represented by the device's Dynamic Random Access Memory (DRAM) and is especially used for transferring data among the host and the device in addition to inserting data and retrieving the output results of a kernel.

The qualifier global ascertains the scope of the memory, meaning that this type of memory is accessible by both the device and the host system. In order to declare a variable as being stored in global memory, one can use the "__device__" qualifier or to allocate it dynamically with the instruction "cudaMalloc()". The lifetime of this type of memory is equivalent to the lifetime of the application, a variable that has been declared using the global memory qualifier exists during the lifetime of the application. For devices offering support for certain compute capabilities the global memory can be stored in the cache memory of the chip.

The process of grouping the threads into warps is beneficial to the processing performance as it optimizes the inserting and retrieving of data in global memory. Global memory is coalesced by the device, thus reducing the number of transactions and optimizing the bandwidth. Different Compute Unified Device Architectures determine different sizes for the memory transactions. For example, on older compute 1.x compatible devices the size of the coalesced memory transaction begins at 128 bytes for each memory access. Afterwards, this value will be cut down to 64 or 32 bytes if the memory area that is being accessed has a low dimension and resides in the same aligned block of threads, having the dimension of 32-bytes. Because this memory is not cached, it would have affected negatively the bandwidth if the threads had not accessed memory addresses that are consecutive.

Therefore, if a programmer does not pay careful attention at how different types of memory are used, he risks losing most of the potential bandwidth he could have obtained on the graphics processing unit. In contrast with compute 1.x compatible devices, on more advanced CUDA architectures, the technical possibilities are enhanced. Devices based on the Fermi architecture retrieve memory in transactions varying in size between 32 and 128 bytes. It is not offered support for sizes of 64- bytes, implicitly each memory transaction has a 128-byte cache size. The main advantages resulting from this enhancement consist in the fact that strided access within the bounds of 128 bytes is cached and another retrieve from memory is no longer necessary. The whole CUDA programming model starting with the Fermi architecture has become significantly simpler than the one implemented by the preceding architectures.

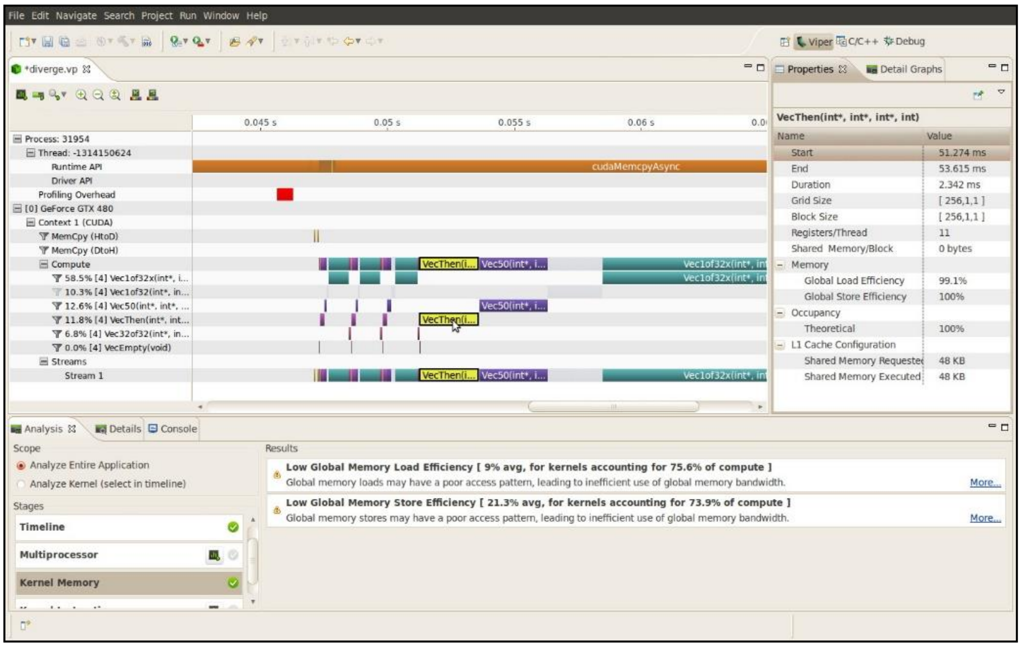
The number of ongoing transactions is a significant factor that affects the overall performance of a CUDA application. The memory transactions are queued and afterwards are executed one by one. This mechanism implies a consumption of the processing resources. It is more efficiently for a thread to perform a read of three integers in a single pass than to perform three separate reads. According to the official NVIDIA documentation, in order to reach the peak bandwidth for Kepler and Maxwell, the programmer has to employ several strategies. One strategy consists in filling the processor up with warps until the near full occupancy is reached.

Another strategy consists in performing 64 or 128 bit reads using vector types like float2/int2 or float4/int4, the occupancy would be lower than in the case of the first strategy, but the memory bandwidth would still reach its peak. Practically this strategy issues less transactions of greater sizes that can be processed more proficiently by the equipment. The use of vector types also inserts a specific instruction-level parallelism due to the fact that a thread processes more elements. Nevertheless, the programmer must pay particular attention to the fact that the use of vector types is essentially connected with an alignment of 8 and 16 bytes, depending on the used data type. In [12] it is stated that processing four elements per thread leads to an improved performance due to a moderate register usage, an increased opportunity for instruction level parallelism and higher memory bandwidth.

3. AN ANALYSIS OF THE LIMITING FACTORS

The memory latency and memory bandwidth, as well as the latency existing at the instruction level are significant causes for capping the kernel functions performance. The programmer must identify exactly what causes the deficiency and solve it so that the performance can reach its full potential. In order to identify the key points in the code that must be addressed the programmer must analyze the existing arithmetic instructions. One to one assignment provides satisfactory results if the outputs have a one to one correspondence with the inputs. After performing this analysis, if the percentage of processing time is spent mostly on arithmetic operations, then the application is arithmetically bound. However, if the overall performance remains unaffected, the application is memory bound [12].

One way of solving the issues is to use the Parallel Nsight or Nsight Eclipse Edition tool to profile the kernel function (**Figure 1**).



VecThen(int*, int*, int*, int)

Name	Value
Start	51.274 ms
End	53.615 ms
Duration	2.342 ms
Grid Size	[256,1,1]
Block Size	[256,1,1]
Registers/Thread	11
Shared Memory/Block	0 bytes
Memory	
Global Load Efficiency	99.1%
Global Store Efficiency	100%
Occupancy	
Theoretical	100%
L1 Cache Configuration	
Shared Memory Requested	48 KB
Shared Memory Executed	48 KB

Scope
[✓ Analyze Entire Application]
Analyze Kernel (select in timeline)

Stages
Timeline ✓
Multiprocessor
Kernel Memory ✓

Results

Low Global Memory Load Efficiency [9% avg, for kernels accounting for 75.6% of compute]
Global memory loads may have a poor access pattern, leading to inefficient use of global memory bandwidth. More...

Low Global Memory Store Efficiency [21.3% avg, for kernels accounting for 73.9% of compute]
Global memory stores may have a poor access pattern, leading to inefficient use of global memory bandwidth. More...

Figure 1. An example of profiling a kernel function using the Nsight Eclipse Edition [1]

By applying the Analysis function, one should inspect the report regarding the statistics of instructions. If the report signals that the kernel's memory pattern exhibits poor coalescing and the graphics processing unit has to process the instruction stream serially in order to offer support for scatter memory reads or writes, one has to correct these issues.

First the programmer should try to change the allocation of the memory pattern so that the graphics processing unit is able to coalesce, according to the thread, the access pattern. The best possible case would be when the data pattern generates an access pattern that is column based according to the thread. If the data pattern cannot be repositioned, the programmer can try to modify the thread pattern by using the threads to store the data that is about to be processed in shared memory. By using this technique, the programmer does not have to worry about the process of coalescing when using the shared memory.

Another solution consists in enlarging the dimension of the output dataset, thus obtaining a higher number of processed elements per thread. This technique has the potential to improve both the arithmetic and memory bound situations. It is best to duplicate the code and avoid inserting a new loop within the thread. The read instructions should be positioned at the beginning of the kernel function so that they have time to gather the data before being used. This mechanism will rise the amount of needed registers, so it is important to closely monitor the amount of warps that have been involved, in order to assure that their number is not cut down.

In the case of arithmetic bound kernels, the programmer should analyze the generated Parallel Thread Execution (PTX) code. The programmer must weigh the advantages and

[1] The Figure has been downloaded from the official Nvidia documentation site https://developer.nvidia.com/nsight-eclipse-edition , accessed on 09.29.2016, at 21:00.

disadvantages of unrolling the existing loops and he can also activate floating-point arithmetic on 24-bits, that can greatly improve the overall performance.

4. AN OPTIMAL HANDLING OF THE MEMORY

Obtaining a high level of performance in a CUDA developed application is often conditioned by obtaining the right memory pattern. Central processing unit applications have a tendency to organize the data in memory by rows. Although the latest CUDA architectures can handle noncoalesced memory operations, older devices cannot. The developer must try to obtain, for both the global and shared memory, a pattern that is accessed in columns by successive threads. In the case of the "cudaMalloc" function, the alignment process must not be performed as the function generates memory by grouping it in aligned blocks of 128-byte size.

If the programmer needs to use in his application a structure that stretches over this limit, he can insert padding bytes/words directly into the structure or he can call the "cudaMallocPitch" function instead. The alignment of memory determines if there is a need to fetch the transactions or cache lines. There are situations when one can access a structure containing a certain header at the beginning and the first thread will process a memory address with a different offset than zero [12]. In this situations a single fetch instruction will not be enough to provide the data to all the 32 threads of the warp.

Moreover, a supplementary transaction will be generated, having a 128-byte size. The succeeding warps will be affected by this problem like a domino effect, the whole memory bandwidth being reduced in half due to a single header at the beginning of the structure. On older devices compatible with the compute 1.x, the supplementary fetch instruction is simply disposed instead of being used to fill in advance the cache memory. A possible solution to this problem would be to explicitly store the header in a different memory address, thus making it possible to align the memory for subsequent operations.

Another solution consists in the addition of padding bytes within the structure in order to assure the desired alignment of the header as to correspond to the 128-byte limit. If the structure is not being used afterwards to produce an array, the repositioning of its elements will suffice. After assuring that the threads are in accordance with the optimal memory pattern, the bandwidth will increase significantly. Regarding the read and write instructions, several reads originating from the same address will provide good results from the performance point of view as the graphics processing unit will allocate the value to the following threads of the warp without having to use supplementary memory fetch instructions. The same cannot be stated about several write instructions to the same memory address, as in this case the write instructions must be serialized.

Another practice in order to achieve optimal alignment consists in inserting a padding value that does not affect the outcome of the processing in any way. For example, when adding up values over an array, the padding with the zero value will have no effect on the final result but it can help to obtain the desired memory pattern and assure the optimal execution path within the warp. When the programmer works with bins that have a fixed size, it becomes easier to generate the dataset and manipulate it in columns instead of rows. In order to obtain coalesced operations in regards to the global memory, one can create a buffer using the shared memory.

An important metric that must be taken into account for when developing Compute Unified Device Architecture applications is represented by the ratio between the number of memory operations and the number of arithmetic ones. In the scientific literature, it has come to the conclusion that this ratio should be at least 10 to 1 [12], meaning that for each fetch instruction issued to the global memory there have to be processed ten or even more operations. These operations can be of different types, ranging from computing the arrays indexes, branches, evaluating conditional expressions, computing loops etc. Each of these operations should influence positively the final result, moreover the loop instructions that in most of the cases should be unrolled, otherwise they tend to consume a lot of resources and generate an overhead of instructions.

From the architecture point of view, one can observe that within a streaming multiprocessor there are implemented special odd and even dispatchers that issue the warps to multiple cores. The latest Pascal GP100 streaming multiprocessor architecture brings significant improvements in areas such as the Compute Unified Device Architecture cores occupancy level, the performance per watt, thus ensuing noteworthy enhancements that translate in an increased overall performance when compared to previous architectures. The Pascal streaming multiprocessor includes 64 Compute Unified Device Architecture cores that have a single precision FP32 (**Figure 2**). The previous CUDA architectures incorporate 128 cores in a Maxwell streaming multiprocessor and 192 cores in a Kepler one, in both cases the precision being FP32.

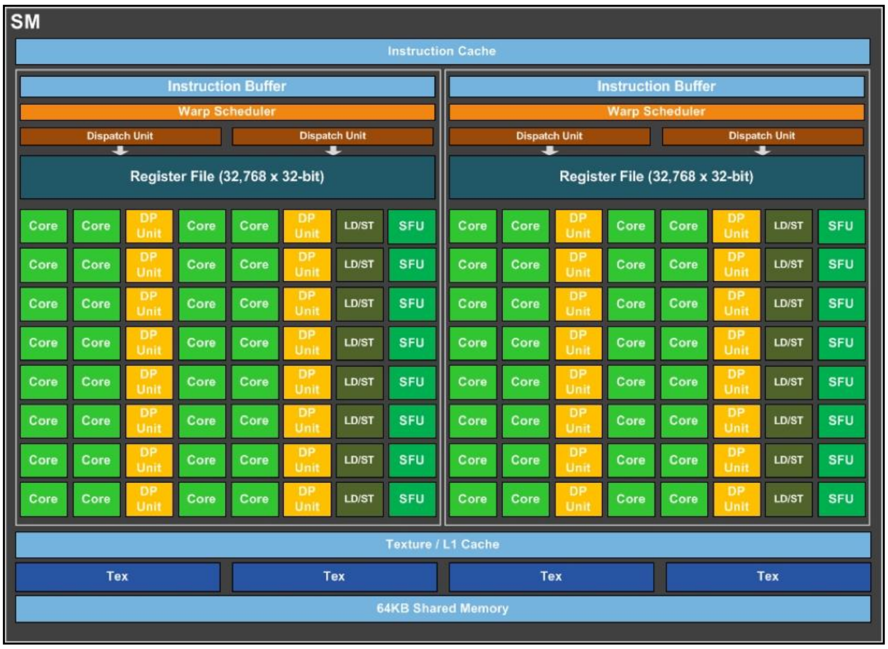


Figure 2. An overview of the latest Pascal GP100 SM architecture [1]

[1] The Figure has been downloaded from the official Nvidia documentation site https://devblogs.nvidia.com/parallelforall/inside-pascal/ , accessed on 09.30.2016, at 14:05

The streaming multiprocessor of the latest Pascal GP100 architecture comprises two processing blocks that offer Compute Unified Device Architecture processing cores with 32 single precision, two instruction buffers, two warp schedulers and four dispatch units, two per each processing block (**Figure 2**). Even if the streaming multiprocessor from the Pascal architecture contains only half the number of cores when compared to the previous Maxwell architecture, the Pascal architecture streaming multiprocessor preserves the similar size for the register file and can sustain the same level of occupancy regarding the thread blocks and warps. The Pascal GP100 streaming multiprocessor provides the same amount of registers like in the cases of Maxwell GM200 and Kepler GK110 architectures, but with a major difference: the Pascal architecture offers as a whole considerably more streaming multiprocessors, consequently putting forward more registers than the previous CUDA architectures had to offer.

When developing applications targeting the Pascal Architecture, the programmer must make use of the fact the Pascal graphics processing unit’s threads can access far more registers than on previous architectures and there are more thread blocks, threads and warps available that can run concurrently. The higher number of streaming multiprocessors also determines an increased overall shared memory size and the aggregate bandwidth of the shared memory is in fact more than two times higher than on the previous Maxwell architecture. All of these aspects make the Pascal GP100 streaming multiprocessor more efficient when processing the source code as the scheduler can select from a higher number of available warps, the available shared memory bandwidth for each threads is considerably improved and more data can be loaded into memory.

Table 1 depicts the main technical advantages offered by the Pascal GP100 architecture, implemented in the Tesla P100, when compared to the previous Maxwell GM200, implemented in the Tesla M40 and the Kepler GK110, implemented in the Tesla K40 graphics card.

Table 1. A comparison between the technical features of the latest CUDA architectures[1]

NVIDIA GPU Architecture	GK110 (Kepler) implemented in Tesla K40	GM200 (Maxwell) implemented in Tesla M40	GP100 (Pascal) implemented in Tesla P100
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA Cores / SM	192	128	64
FP32 CUDA Cores / GPU	2880	3072	3584
FP64 CUDA Cores / SM	64	4	32
FP64 CUDA Cores / GPU	960	96	1792
Base Clock	745 MHz	948 MHz	1328 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz
Peak FP32 GFLOPs	5040	6840	10600
Peak FP64 GFLOPs	1680	210	5300
Texture Units	240	192	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts
Manufacturing Process	28-nm	28-nm	16-nm FinFET

In contrast with the Kepler architecture, the GP100 streaming multiprocessor provides a plainer data path structure that can handle more efficiently the data transfers. The Pascal architecture offers the possibility to overlap load and store instructions far better than before, along with an improved scheduling process, thus delivering an increased level of performance with lower power consumption. There is one warp scheduler per processing block that can dispatch two warp instructions per clock [15]. A novel addition to the Pascal GP100's Compute Unified Device Architecture cores is the facility of processing instructions and data that have a precision of both 16-bit and 32-bit, while the amount of FP16 processed operations is up to two times the throughput of FP32 operations.

5. THE KERNEL FUSION TECHNIQUE

When multiple kernels are running sequentially, it can often happen that one can identify certain elements within the kernels that can be joined together (fused). The programmer must be very thorough when applying this technique because the process of creating two kernels in series produces an implied synchronization between the two CUDA functions. It is a common practice, when one develops CUDA kernel functions, to divide the needed operations into several phases or steps. For instance, during the first step, one can compute the results associated with the entire dataset and during the second step the data can be filtered according to specific criteria.

[1] The table has been created according to the official Nvidia documentation site https://devblogs.nvidia.com/parallelforall/inside-pascal/ , accessed on 09.30.2016, at 17:55

If the second step takes place inside a block of threads, the first phase along with the second one can be joined together within the same kernel function. By doing so, the supplementary resources, necessary to invoke the second kernel, are removed along with the writing instructions of the first kernel into the global memory and the succeeding reading instructions of the second kernel. The first kernel function has the possibility to store its results using the shared memory and use them in the second step of the processing, thus eliminating completely the need to access global memory.

The operations that perform reductions can take advantage from this technique because the results produced by the subsequent steps are typically less than they were during the first step, as a consequence the consumption of memory bandwidth is substantially reduced. The rationale behind this technique’s success lies in the ability of data reuse. The fetching operations applied to global memory are very slow, on the average of 500 clock cycles. This is the reason why it is better to read bigger segments of data and store them in memory, preferably into faster memory types. In the Compute Unified Device Architecture it is better to retrieve the data in segments of up to 16 bytes per thread and not to use words or single bytes. After every thread has processed without errors one element, it is best to shift processing to two elements per thread. From this point forwards, one can experiment and analyze the obtained results when processing four or even six elements per thread. As soon as the desired data has been obtained, one must try to store it into the faster memory types, like the shared or register memory and re-call it as often as it must.

6. CONCLUSIONS

The management of the Graphics Processing Unit memory is essential in order to obtain a high level of performance when developing a CUDA application. The developer must take into account technical aspects that have a significant impact on the overall performance, such as: an analysis of the cache mechanism as to benefit from the increased speedup provided by the shared and cache memory; aligning the data in accordance to the cache memory supported by the targeted architecture; facilitate the retrieving of instructions through optimal memory access patterns; obtaining an optimal memory occupancy in accordance with the processed data types; a thorough understanding of the performance limiting factors and their correction; an optimal handling of the memory according to the processing architectures and their technical characteristics, using optimization techniques such as merging more kernel functions into a single one.

REFERENCES

[1] Petroşanu Dana-Mihaela, Pîrjan Alexandru, *Economic considerations regarding the opportunity of optimizing data processing using graphics processing units*, JISOM, Vol. 6, Nr. 1/2012, pp. 204-215, ISSN 1843-4711.

[2] Pîrjan Alexandru, *Optimization Techniques for Data Sorting Algorithms*, The 22nd International DAAAM Symposium, Annals of DAAAM for 2011 & Proceedings of the 22nd International DAAAM Symposium, Vienna, 2011, pp. 1065-1066, ISSN 1726-9679, ISBN 978-3-901509-73-5.

[3] Lungu Ion, Pîrjan Alexandru, Petroşanu Dana-Mihaela, *Optimizing the Computation of Eigenvalues Using Graphics Processing Units*, Scientific Bulletin, Series A, Applied Mathematics and Physics, Vol. 74, Number 3/2012, pp.21-36, ISSN 1223-7027.

[4] Pîrjan Alexandru, Petroşanu Dana-Mihaela, *Dematerialized Monies – New Means of Payment*, Romanian Economic and Business Review, Vol. 3 Nr. 2/2008, pp. 37-48, ISSN 1842 – 2497.

[5] Pîrjan Alexandru, Petroşanu Dana-Mihaela, *A Comparison of the Most Popular Electronic Micropayment Systems*, Romanian Economic and Business Review, Vol. 3, Nr. 4/2008, pp. 97-110, ISSN 1842–2497.

[6] Tăbușcă Alexandru, *Established ways to attack even the best encryption algorithm*, JISOM, Vol., No.2.1/2011 – December 2011, pages 485-491, ISSN 1843-4711.

[7] Tăbușcă Alexandru, *HTML5 - A new hope and a dream*, JISOM, May2013, Vol. 7 Issue 1, p49, ISSN 1843-4711.

[8] Pîrjan Alexandru, Petroşanu Dana-Mihaela, *Solutions for developing and extending rich graphical user interfaces for Office applications*, JISOM, Vol. 9, Nr. 1/2015, pp. 157-167, ISSN 1843-4711.

[9] Lungu Ion, Căruțașu George, Pîrjan Alexandru, Oprea Simona-Vasilica, Bâra Adela, *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, No. 1/2016, pp. 77-86, ISSN 1220-1766.

[10] Lungu Ion, Bâra Adela, Căruțașu George, Pîrjan Alexandru, Oprea Simona-Vasilica, *Prediction intelligent system in the field of renewable energies through neural networks*, Journal of Economic Computation and Economic Cybernetics Studies and Research, Vol. 50, No. 1/2016, pp. 85-102, ISSN online 1842– 3264, ISSN print 0424 – 267X.

[11] Altar Samuel. A., Costin A., Enache D., *RDF & RDF Query Languages - Building blocks for the semantic web*, JISOM, CNCSIS B+, vol. 9, nr.l, 2015, ISSN 1843-4711.

[12] Cook Shane, *CUDA Programming*, 1st Edition, A Developer's Guide to Parallel Computing with GPUs, Morgan Kaufmann, 2012, ISBN 9780124159334.

[13] Sanders J., Kandrot E., *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, 2010, ISBN-10 0-13-138768-5.

[14] Schneider S., Yeom, J, Nikolopoulos D., *Programming multiprocessors with explicitly managed memory hierarchies*, in IEEE Computer Society, Volume 45 , Issue 5, October 2009, ISSN 0018-9162.

[15] Whitepaper, NVIDIA Tesla P100 The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU