Optimization Solutions for Improving the Performance of the Parallel Reduction Algorithm Using Graphics Processing Units

Ion LUNGU[1], Dana-Mihaela PETROSANU[2], Alexandru PIRJAN[3]
[1]Academy of Economic Studies, Bucharest, Romania
[2]University Politehnica of Bucharest, Romania
[3]Romanian-American University, Bucharest, Romania
ion.lungu@ie.ase.ro, danap@mathem.pub.ro, alex@pirjan.com

In this paper, we research, analyze and develop optimization solutions for the parallel reduction function using graphics processing units (GPUs) that implement the Compute Unified Device Architecture (CUDA), a modern and novel approach for improving the software performance of data processing applications and algorithms. Many of these applications and algorithms make use of the reduction function in their computational steps. After having designed the function and its algorithmic steps in CUDA, we have progressively developed and implemented optimization solutions for the reduction function. In order to confirm, test and evaluate the solutions' efficiency, we have developed a custom tailored benchmark suite. We have analyzed the obtained experimental results regarding: the comparison of the execution time and bandwidth when using graphic processing units covering the main CUDA architectures (Tesla GT200, Fermi GF100, Kepler GK104) and a central processing unit; the data type influence; the binary operator's influence.
Keywords: *GPU, Cuda, Kepler Architecture, Parallel Reduction, Thread Blocks*

1 Introduction

Initially, graphics processing units (GPUs) have been designed solely for graphic specific rendering purposes, but later, by the end of the 1990s, these processors became programmable at a hardware level. In November 2006, the NVidia company released the GeForce 8800 GTX, the first GPU to support the new CUDA (Compute Unified Device Architecture) by unifying both software and hardware components [1].

This new parallel programming model uses the huge parallel computational processing power of the GPU to solve complex processing tasks in a much more efficient manner than by using traditional processing methods based on central processing units (CPUs). This novel architecture offers several new components, specifically designed for alleviating the limitations of previous GPUs architectures and easing the processing of general-purpose computations through graphics processing units. Unlike previous GPU hardware architectures, the Compute Unified Device Architecture employs a unified implementation that makes it possible for the GPU to perform general-purpose computations.

In this context, the development of high performance optimization solutions using high-performance basic functional blocks (like the parallel reduction algorithmic function) leads to a tremendous improvement in the parallel data processing. In the scientific literature, this type of research is of great interest, many researchers studying the potential to optimize algorithmic functions using the CUDA architecture [2], [3], [4], [5], [6]. None of the works so far (to our best knowledge) has studied optimization solutions that scale in terms of resource allocation and performance on all the available CUDA architectures, especially on the latest Kepler CUDA architecture.

The latest three CUDA-enabled graphic cards are GTX 280 from the Tesla GT200 architecture, GTX 480 from the Fermi GF100 architecture and GTX 680 from the Kepler GK104 architecture [7].

The GTX 280 graphics processor, launched on 16 Jun 2008, is based on 65 nm fabrication technology, has 240 CUDA cores, 30 streaming multiprocessors and 1.4 billion

of transistors, the processor clock runs at 1296 MHz, the graphics clock at 602 MHz. It comes with 1024 MB of memory in the standard configuration, having an effective clock of 1107 MHz, a 512-bit GDDR3 memory interface width and 141.7 GB/sec memory bandwidth. It has the maximum board power (TDP) of 236 Watts, a texture fill rate of 48.2 billion/sec, 80 texture units and 32 ROP units.

The GTX 480 graphics processor, launched on 26 March 2010, is based on 40 nm fabrication technology, 480 CUDA cores, 15 streaming multiprocessors and 3.2 billion of transistors, the processor clock runs at 1401 MHz and the graphics clock at 700 MHz. It comes with 1536 MB of memory in the standard configuration, having an effective clock of 3700 MHz, a 384-bit GDDR5 memory interface width and 177.4 GB/sec memory bandwidth. It has the maximum board power (TDP) of 250 Watts, a texture fill rate of 42 billion/sec, 60 texture units and 48 ROP units.

The newest CUDA graphic card, the GTX 680, released on 22 March 2012, is based on 28 nm fabrication technology, 1536 CUDA cores, 8 streaming multiprocessors and 3.54 billion of transistors, the boost clock runs at 1058 MHz and the graphics clock at 1006 MHz. It comes with 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. It has the maximum board power (TDP) of 170 Watts, a texture fill rate of 128.8 billion/sec, 128 texture units and 32 ROP units. The GK104 poses significant differences regarding the streaming multiprocessors, that are now called SMX units and incorporates several important architectural changes in order to deliver an improved performance and power efficiency. Taking into consideration the above-mentioned technical specifications, we aimed to research, develop and study optimization solutions for the parallel reduction function that fully uses the processing power of CUDA enabled GPUs covering the main generations (Tesla GT200, Fermi GF100, Kepler GK104).

We paid particular attention to obtain a software solution that dynamically adjusts the number of thread blocks, threads per block and the number of processed elements per thread in order to harness the computational processing power of GPUs as to reach a performance peak.

2 Designing an Efficient Scalable CUDA Parallel Reduction Algorithmic Function

The reduction function is extensively used in many data parallel processing applications and algorithms, therefore its optimization improves the performance of all data processing algorithms and applications that implement the reduction function.

In the following we define the reduction function, using an associative binary operator *, defined on the set of real numbers. For an input n-dimensional vector $v = [a_0, a_1, \dots, a_{n-1}]$ with real components, the reduction operation, denoted by $red(*)$, produces an output real number, defined through:

$$red(*)v = a_0 * a_1 * \dots * a_{n-1} \quad (1)$$

The associative binary operator used in the definition of the reduction function can be the summation, maximum, minimum or the multiplication operator. In various scientific fields there are common applications that use the reduction function and the most common one is the computation of the scalar product of two vectors. Considering $x = [x_0, x_1, \dots, x_{n-1}]$ and $y = [y_0, y_1, \dots, y_{n-1}]$ two vectors with real components, their scalar product is defined as the real number:

$$x \cdot y = \sum_{i=0}^{n-1} x_i\, y_i \quad (2)$$

This computation requires n multiplication operations and $n-1$ summation operations. Since multiplication operations are independent of each other, the reducing operation defined by (2) facilitates its execution through parallel multiplication computations, followed by sequential summations. Consequently, the main advantage of using implementations based on the parallel reduction is that it converts

fragments of sequential computations in parallel equivalent ones.

The intuitive approach for designing the parallel reduction algorithmic function is to use only an execution thread that iterates and computes the sums in the shared memory starting from the input vector [1], but this method has a few disadvantages and limitations as the necessary execution time is proportional to the input vector's dimension and there are hundreds of idle execution threads.

A more advantageous designing method is based on the parallel execution of the reduction, because in this situation the execution time is proportional to the logarithm of the input vector's dimension. When the size of the input vector is large, we split it into multiple fragments and allocate their reduction to a thread block that processes the fragment and stores its partial result in the global memory. Afterwards, these partial results are reduced to a single element, obtaining the output. We have called the reduction function at every thread block's level depending on the input vector's size and the thread block's dimension, using the corresponding parameters in order to obtain the best performance.

The main direction that we have followed when designing and developing the optimized parallel reduction algorithmic function was to obtain a CUDA processing solution, self-adjustable and self-configurable (regarding the number of thread blocks, the number of threads in a block and the number of processed elements per thread), depending on the GPU's architecture. Our solution offers a high degree of performance on a wide range of CUDA GPUs architectures: Tesla GT200 (implemented in the GTX 280), Fermi GF100 (implemented in the GTX 480), Kepler GK104 (implemented in the GTX 680). Thus, we have designed the parallel reduction algorithmic function as to use: a maximum of 256 threads per block on the GTX 280; a maximum of 512 threads per block on the GTX 480 and GTX 680 GPUs.

We have developed the parallel reduction algorithmic function in 4 steps as we have also presented in [8]:

Step 1. If the thread block is of size 1, and the input vector has only one element, this element is copied as the output element. If the input vector has 2 elements, they are reduced using the corresponding binary operator. If the thread block has a size larger than 1, the processing continues with Step 2.

Step 2. Each thread reduces sequentially several elements (8/16/32 depending on the compute capability of the GPU's architecture) in the global memory, this process taking place in parallel, between the threads of each block (at intra-block level) and also between different blocks (at inter-block level). The obtained partial results are being copied from the global memory into the shared memory, where they are reduced in parallel, benefiting from the shared memory's increased performance and low latency.

Step 3. Each thread block's results are stored in the global memory. Therefore, it is obtained an output vector, in which every element represents the corresponding sum of each parallel thread block.

Step 4. Then, the vector's elements that are stored in the global memory are reduced using the host device, because the GPU would waste its resources in the final steps of the reduction as the dataset dimension is undersized. Therefore, the final computations will be processed by the CPU, summing the vector's components stored in the global memory. While the CPU completes the reduction, the GPU is already available to process other data.

The above-described parallel reduction's implementation is tree-type based and we have used it within each thread block. Thus, the reduction algorithmic function is designed to use multiple thread blocks for

processing large dimension input vectors, offering to each of the GPU's multi-processors an enough computational load in order to fully employ their parallel processing power. Each thread block will reduce a part of the input vector's elements.

After each reduction step, thread blocks communicate to each other and send their obtained partial results, in order to process the next step of the algorithm. For an efficient reduction of very large input vectors, we had to globally synchronize the results among all the thread blocks. Thus, after each thread block has produced its result, the process could continue recursively until the final output has been obtained. However, in the Compute Unified Device Architecture there is no possibility to globally synchronize the results using a direct instruction in the CUDA API. This happens because, on one hand there are hardware difficulties at the graphics processor level and on the other hand, the global synchronization would require developers to use only a few thread blocks, that would reduce the overall efficiency.

In order to overcome this technical limitation, we have split the function in several kernel functions. Each call of a kernel function provides a synchronization point and the advantage of the method is that it does not introduce a significant computational load that could affect the performance [5]. The CUDA kernel function running is asynchronous, requiring synchronization on the host machine in order to copy the data from the GPU's memory into the system's memory. The source code is the same at all the levels of the reduction algorithmic function and therefore we can recursively call the kernel functions. This is done only on the host machine, as a CUDA kernel function does not allow a recursive call.

In the following, we present several solutions that we have developed for optimizing the performance of the parallel reduction algorithmic function.

3 Optimization Solutions for Improving the Performance of the Parallel Reduction Algorithmic Function in CUDA

Using a thorough analysis of the CUDA programming guide and of the best practices depicted in the literature [2], [3], [4], [5], [6] for developing successful applications in CUDA, we have identified and developed a series of optimization solutions for the parallel reduction algorithmic function. The main optimization solutions that we have developed and applied, target aspects regarding: harnessing the GPU's performance; the optimal use of the shared memory bandwidth; the synchronization possibilities; the efficient cooperation between the GPU and the CPU. In the following we will describe our solutions and their progressive development.

Solution 1 - The interleaved addressing technique. Using this technique, we have first instructed every thread to load one element from the global memory into the shared memory. We have then performed the reduction phase in the shared memory (it had the advantage of a high throughput) and the obtained result was loaded back to global memory. We have found that this technique had the disadvantage of generating divergent warps in which the threads did not process the same tasks. These warps are inefficient in the reduction phase as they slow down the data processing; therefore, we have decided to manage the divergent branching in order to improve the performance.

Solution 2 - Avoiding the divergent branching. We have eliminated the above-mentioned limitations and we have improved the performance by replacing the divergent branching with a non-divergent one using a direct indexing for each element of the input vector that would be reduced. This technique has eliminated the divergent warps, but generated shared memory bank conflicts due to multiple requests from the same memory bank, that we had to manage using the sequential addressing technique.

Solution 3 - The sequential addressing technique. We have used thread's indexing instead of direct indexing in order to manage

the shared memory bank conflicts. Although we have obtained significant improvements regarding the execution time and bandwidth, we have observed that further optimizations are still possible as the GPU's resources are not fully employed. When each of the threads loads an element from the global memory into the shared memory, half of the threads within a block remain idle. Therefore, for solving this situation, we have decided to perform a reduction in the global memory before loading the data in the shared memory.

Solution 4 - Performing a reduction of data stored in the global memory before loading it into the shared memory. This solution solved the disadvantages of the sequential addressing technique and led to significant performance improvements. Even if the bandwidth increased considerably, further optimizations were still achievable. The reduction function does not involve an increased computational effort but the large amount of necessary synchronization operations caused a performance penalty. In order to overcome this penalty we have minimized the number of executed instructions.

Solution 5 - Minimizing the number of executed instructions. The parallel reduction algorithmic function that we have developed is useful for multicore architectures such as SIMD (Single Instruction, Multiple Data), in which case instructions are synchronous within each warp. We have noticed that, as the reduction process progresses, the number of active threads decreases, and when the number of threads is less than or equal to 32, only the last warp remained to be executed. Within this warp we did not have to synchronize or to validate the threads' indexes anymore. Consequently, we have decided to remove these instructions for obtaining an increase in the overall performance and to minimize as much as possible the number of synchronization operations.

Solution 6 - Processing multiple elements per thread. By reducing multiple elements per thread during the same reduction step (8 elements for GTX280, 16 elements for gtx480 and 512 elements for the GTX680), we have gained a considerable performance improvement for the reduction algorithmic function.

Solution 7 - Decomposing the reduction function in several smaller ones, in order to achieve global synchronization. After each reduction step, the blocks of threads have to inter-communicate their partial results, so we had to synchronize them. But in CUDA it is not technically possible to directly globally synchronize, so we have decided to decompose the reduction function into several smaller kernel functions. Each call of a kernel function provided a synchronization point and therefore we have achieved synchronization with minimal computational costs.

Solution 8 - Using the CPU to perform the final computations in the last step of the reduction algorithm. In the last step of the reduction algorithm the dataset dimension is undersized and thus, if we had used the GPU to perform the reduction in this case, its resources would have been wasted. Therefore, we have decided to reduce the array's elements from the global memory in the last step of the reduction using the CPU. The main advantage of this approach is that the GPU is already available to process other computations while the CPU finalizes the reduction operation.

By implementing these eight solutions, we have improved the parallel reduction algorithmic function's performance, regarding both the execution time and bandwidth. In the following, we present a benchmark suite that we have developed in order to analyze the performance of the parallel reduction function in CUDA, optimized using the above-described Solutions 1-8.

4 Experimental Results

We have analyzed the performance of the above described parallel reduction function, using the Windows 7 64-bit operating system and the following configuration: Intel i7-2600K operating at 4.6 GHz with 8 GB

(2x4GB) of 1333 MHz, DDR3 dual channel. We have used the NVIDIA graphic cards GeForce GTX 280, GTX480 and GTX 680. Programming and access to the GPUs have used the CUDA toolkit 4.1, with the NVIDIA driver version 270.81 (for the GTX 280 and the GTX 480) and NVidia 301.10 for the GTX 680. 270.81.

In order to reduce the external traffic to the GPU, all the processes related to the graphical user interface have been disabled. We have used the same measuring time method that we have used before, when we have developed other algorithmic function in CUDA [9], [10], [11].

The parallel reduction algorithm is designed to be used in a large number of applications running on GPUs, so the transfer times between the central processing unit and the graphic processing unit vary depending on the complexity of the specific developed application. Therefore, our measurements do not include the necessary time for data transfers.

We have used the CUDA event application programming interface (API) in order to compute the average execution time that the GPU spends for executing the parallel reduction algorithmic function. This method has many advantages compared to other methods that rely on operating system timers, because those methods usually include variations from different sources and latency, being more suitable for the CPU rather than the GPU.

We have used time events provided by the CUDA API to mark the moments of the function's execution. If we had tried to time the GPU execution using CPU timers, a series of problems could have appeared. For measuring the execution time using CUDA events, we have created a start and a stop event. Some of the kernel calls in CUDA C are asynchronous. The graphic processing unit begins to execute the code and, before the GPU has finished, the central processing unit executes the next code line.

We have used the CUDA API function "cudaEventSynchronize()" in order to synchronize and record accurately the value of the stop event. Thus, we have instructed the runtime to block all further instructions until the graphic processing unit has reached the stop event. When we call the "cudaEventSynchronize()" function, the GPU would have completed all processes before the stop event and the time stamp could be correctly recorded. Therefore, we have obtained a reliable measurement of the execution time in the benchmark suite of the parallel reduction algorithmic function, as described in the source code below:

```
float TimpulTotal = 0;
float timpul = 0;
cudaEvent_t inceput, sfarsit;
cudaEventCreate(&inceput);
cudaEventCreate(&sfarsit);
cudaEventRecord(inceput, 0);
//……….……
//Functia algoritmica de baza al carei
timp de executie il masuram
//……….……
cudaEventRecord(sfarsit, 0);
cudaEventSynchronize(sfarsit);
//calculam timpul dintre evenimentul de
inceput si cel de sfarsit
CUDA_SAFE_CALL(
cudaEventElapsedTime(&timpul,   inceput,
sfarsit));
TimpulTotal += timpul;
```

The first set of tests evaluates the execution times obtained by applying the parallel reduction algorithmic function on vectors of various sizes (35-60,000,000 elements) of float type elements. We have used the summation as the associative binary operator. The vectors were randomly generated, as to cover a wide range of values. To obtain more accurate results, we have computed the average of 10,000 iterations. We have highlighted the execution time (measured in milliseconds) and the bandwidth (measured in gigabytes per second) corresponding to each input vector, when running the reduction function on the three NVIDIA graphic cards and the CPU. In Table 1 and Table 2 we present the results of the experimental tests of the reduction algorithmic function run on the CPU and the GPUs.

Table 1. A comparison between the results obtained on the CPU, GTX 280, GTX 480 and GTX 680 – the running time (float data type)

Test no.	Number of elements	Execution time (ms)			
		CPU	GTX 280	GTX 480	GTX 680
1	35	0.000147	0.074442	0.028267	0.021783
2	128	0.000505	0.072493	0.028867	0.021660
3	256	0.001012	0.077201	0.031434	0.033880
4	260	0.001082	0.073116	0.031420	0.021636
5	512	0.001981	0.072784	0.025498	0.021726
6	1,000	0.003883	0.145758	0.055812	0.036067
7	1,024	0.003985	0.146853	0.060125	0.025151
8	1,030	0.003981	0.140329	0.062212	0.054064
9	32,768	0.129467	0.143899	0.067184	0.054242
10	45,555	0.179772	0.146500	0.052708	0.050089
11	65,536	0.258083	0.147387	0.064974	0.049220
12	131,072	0.520887	0.149085	0.060253	0.076134
13	262,144	1.02914	0.155884	0.063449	0.116000
14	500,111	1.962163	0.159747	0.074067	0.060836
15	524,288	2.062592	0.160366	0.077766	0.062369
16	1,048,555	4.077351	0.178414	0.083419	0.075952
17	1,048,576	4.089703	0.176097	0.090081	0.077899
18	1,048,581	4.092077	0.178931	0.087715	0.091198
19	2,097,152	8.24243	0.208023	0.125539	0.108628
20	2,097,999	8.273636	0.215445	0.129343	0.103861
21	4,194,334	16.688135	0.281550	0.179433	0.160004
22	8,388,600	32.999443	0.414526	0.282094	0.270689
23	16,000,000	61.752274	0.660247	0.530292	0.532815
24	32,000,000	123.678261	1.190001	0.935515	0.891583
25	48,000,000	185.674133	1.696008	1.367220	1.326497
26	60,000,000	231.584732	2.105889	1.671947	1.643266
Total execution time – 10.000 tests (h)		1,909	1.909	0.025	0.017
The system's power (kW)		0,198	0.198	0.306	0.358
Total energy consumption (kWh)		0,378	0.378	0.008	0.006
The GPU's consumption compared to the CPU's			47 times lower	63 times lower	75 times lower

Table 2. A comparison between the results obtained on the CPU, GTX 280, GTX 480 and GTX 680 – the bandwidth (float data type)

Test no.	Number of elements	Bandwidth (GB/s)			
		CPU	GTX 280	GTX 480	GTX 680
1	35	0.9524	0.0019	0.0050	0.0064
2	128	1.0139	0.0071	0.0177	0.0236
3	256	1.0119	0.0133	0.0326	0.0302
4	260	0.9612	0.0142	0.0331	0.0481
5	512	1.0338	0.0281	0.0803	0.0943
6	1,000	1.0301	0.0274	0.0717	0.1109
7	1,024	1.0279	0.0279	0.0681	0.1629
8	1,030	1.0349	0.0294	0.0662	0.0762
9	32,768	1.0124	0.9109	1.9509	2.4164
10	45,555	1.0136	1.2438	3.4572	3.6379
11	65,536	1.0157	1.7786	4.0346	5.3260
12	131,072	1.0065	3.5167	8.7014	6.8864
13	262,144	1.0189	6.7266	16.5263	9.0394
14	500,111	1.0195	12.5226	27.0086	32.8826
15	524,288	1.0168	13.0773	26.9675	33.6249
16	1,048,555	1.0287	23.5084	50.2790	55.2220
17	1,048,576	1.0256	23.8181	46.5615	53.8428
18	1,048,581	1.0250	23.4410	47.8176	45.9914
19	2,097,152	1.0177	40.3254	66.8207	77.2233
20	2,097,999	1.0143	38.9519	64.8817	80.8003
21	4,194,334	1.0053	59.5892	93.5020	104.8557
22	8,388,600	1.0168	80.9464	118.9476	123.9592
23	16,000,000	1.0364	96.9334	120.6882	120.1167
24	32,000,000	1.0349	107.5629	136.8230	143.5649
25	48,000,000	1.0341	113.2070	140.4309	144.7421
26	60,000,000	1.0363	113.9661	143.5452	146.0506

We have computed the total execution time for the 10.000 iterations related to each of the 26 dimensions of vectors. Using an energy consumption meter device we have measured the system's power (kW) and then calculated the total energy consumption in each of the four analyzed cases (running the tests on the CPU and on the three GPUs). The system consumes 47 times less power when the test suite is run on the GTX 280 GPU compared to the i7-2600K CPU. The power consumption is 63 times better for the GTX 480 GPU and 75 times better for the GTX 680 than for the i7-2600K CPU.

In Figure 1 and Figure 2 we present the obtained experimental results by running the parallel reduction algorithmic function on the CPU and on the three GPUs, when the input array has a relatively low dimension (35-1,030 elements). In this case we have noticed that the central processing unit offers the best execution time and bandwidth, because in this case it has not been generated an enough computational load in order to use the huge parallel processing capacity of the GPU.

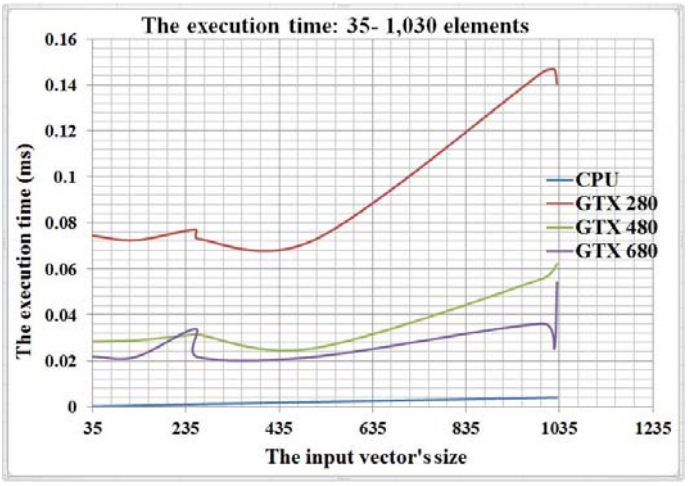

Fig. 1. The execution time for 35 - 1,030 elements of the input array

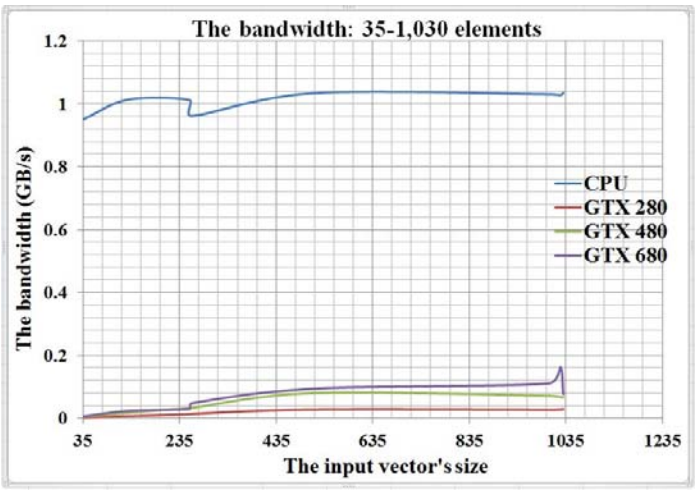

Fig. 2. The bandwidth for 35 - 1,030 elements of the input array

In Figure 3 and Figure 4 there are presented the obtained experimental results when running the parallel reduction algorithmic function on a large dimension input array (32,768-60,000,000 elements). In this case we have noticed that the GTX 680 graphic card offers the best results (lower execution time, higher bandwidth), followed by the GTX 480, the GTX 280 and the CPU, because this time it has been created a sufficient computational load to fully employ the huge parallel processing capacity of the GPUs and to use at maximum the 512/256/128 threads per block we have allocated for each of the graphics processors. The CUDA implementation offers a high degree of performance whether the vector's dimension is a power of two or it is not.

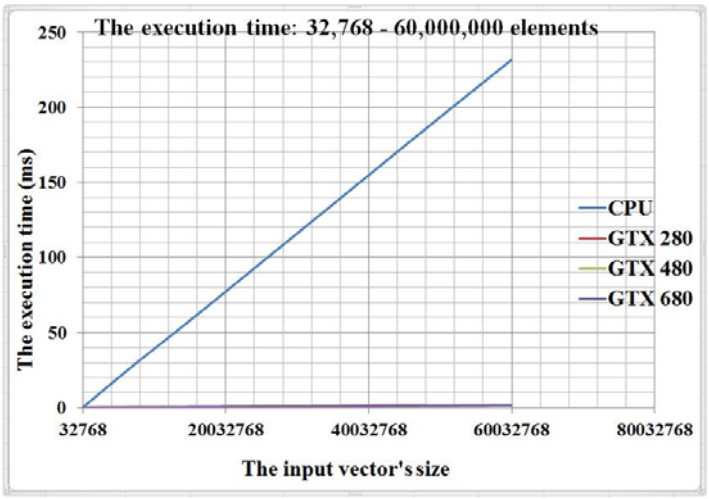

Fig. 3. The execution time for 32,768-60,000,000 elements of the input array

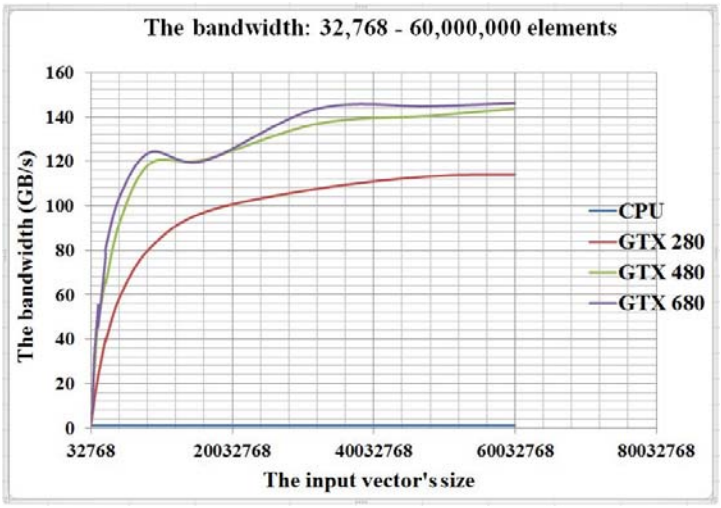

Fig. 4. The bandwidth for 32,768-60,000,000 elements of the input array

The next set of tests evaluates the influence of the data types on the performance of the above described parallel reduction algorithmic function. The function has been designed to allow the selection of the data type for the input array components, which can be one of the following types: integer, unsigned integer, float, double, long long or unsigned long long. Below are presented the obtained experimental results when running the parallel reduction algorithmic function on the GTX 680 graphic processor, using an input array of variable dimension (35-60,000,000 elements). The results represent the average of 10,000 iterations.

One can observe in Figure 5 that the performance is comparable when the input data is of integer, unsigned integer or float type, the execution time ranging between 0.021783 ms and 1.665599 ms. In the case when the input data is of double, long long or unsigned long long type, the performance is comparable but the execution times are generally higher than in the three previous cases, ranging between 0.021628 ms and 3.226117 ms. This is justified taking into account the amount of necessary memory to store the analyzed data types.

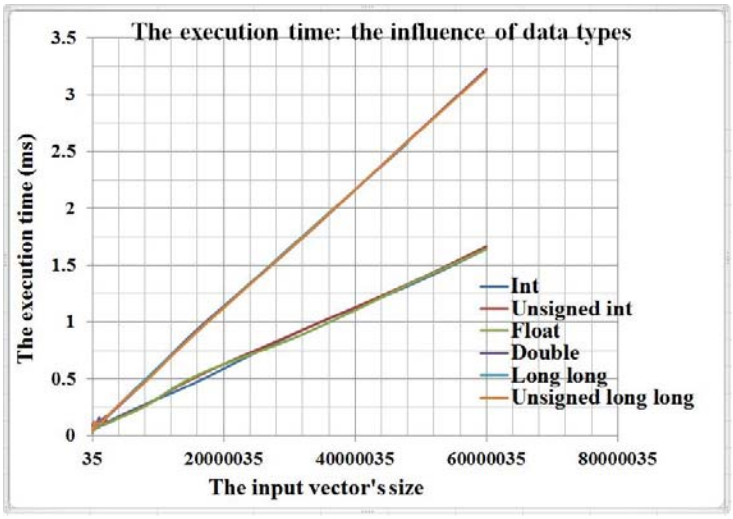

Fig. 5. The influence of data types on the execution time

We have highlighted the bandwidth variation depending on the input vector's size for various data types (Figure 6). Regarding the bandwidth, the performance is comparable in all of the six analyzed cases, reflecting the efficiency of the optimization solutions for improving the performance of the parallel reduction algorithm using graphics processing units that provide constant data processing speed regardless of the data type considered.

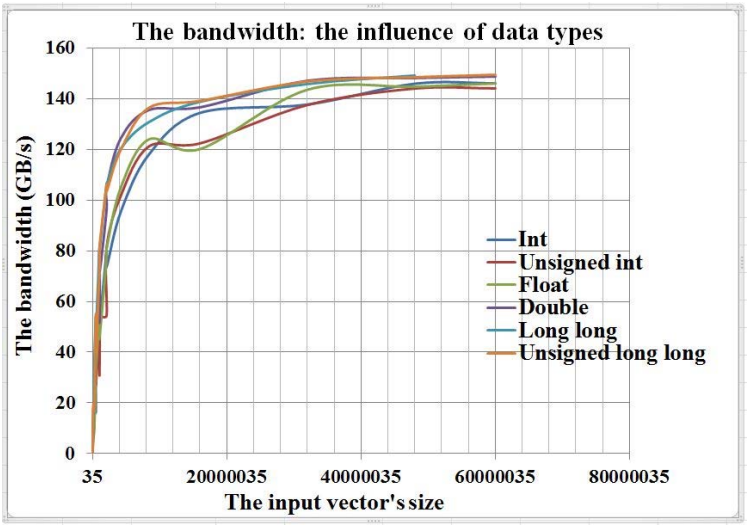

Fig. 6. The influence of data types on the bandwidth

We have analyzed next the influence of the associative binary operator used when defining the parallel reduction function through equation (1), on the performance of the function. This function has been designed to allow the selection of the binary operator that can be one of the following: summation, maximum, minimum or multiplication. In Figure 7 there are presented the experimental results when running the parallel reduction function on an input array of variable dimension (35-60,000,000 elements). We have chosen float type elements for the input array. The results represent the average of 10,000 iterations. One can observe that the performance is comparable in all four cases of binary operators, the execution times ranging between 0.021348 ms and 1.680986 ms, thus confirming the efficiency of the solution, no matter what binary operator is used.

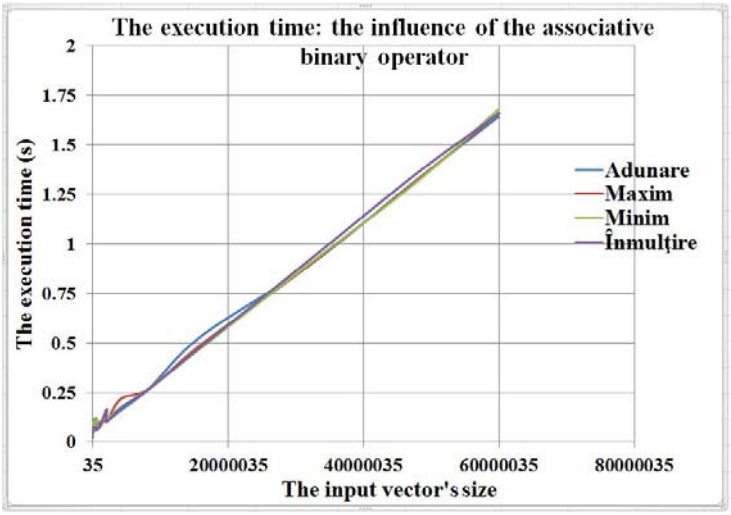

Fig. 7. The influence of the associative binary operator on the execution time

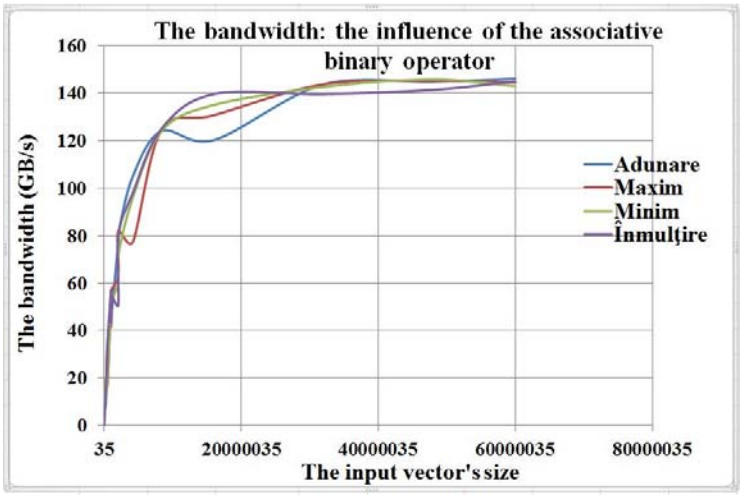

Fig. 8. The influence of the associative binary operator on the bandwidth

When we highlighted the bandwidth variation for different dimensions of float data type for the input vector, we have noticed that the performance is comparable for the four types of associative binary operators, the bandwidth ranging from 0.0037 GB/s to 146.0506 GB/s (Figure 8).

The experimental results confirm the solutions' efficiency, which offer optimum results in different situations and thus they can be implemented in a wide range of algorithms, without being influenced by the chosen binary operator.

5 Conclusions

In this paper, we have researched, developed and analyzed an efficient implementation of the parallel reduction algorithmic function in CUDA, using different optimization solutions. We have first developed the algorithmic function, highlighting the algorithm's steps. We have identified, developed and implemented a set of solutions to improve the performance of the parallel reduction algorithmic function: the interleaved addressing technique; avoiding the divergent branching; the sequential addressing technique; performing a reduction of data stored in the global memory before loading it into the shared memory; minimizing the number of executed instructions; processing multiple elements per thread; decomposing the reduction function in several smaller ones, in order to achieve synchronization; using the CPU to perform the final computations in the last step of the reduction algorithm. In order to maximally benefit from the huge

computational power of the GPU, we have paid particular attention to continuously improve and optimize the solutions.

We have analyzed the performance of the parallel reduction algorithmic function in CUDA, using a series of experimental tests and compared it with an alternative approach run on the central processing unit. In order to compute the average execution time of the GPU we have used the CUDA application programming interface. After having analyzed the experimental results obtained by using the developed solutions for optimizing the performance of the algorithmic function, we have noticed the following:

- When the parallel reduction function is run on the GTX 280 graphics processor, the system consumes 47 times less energy than when the function is run on the central processing unit i7-2600K. For the GTX 480, the power consumption is 63 times smaller and for the GTX 680, 75 times smaller than for the central processing unit i7-2600K.
- We have obtained improved execution times and larger bandwidth when processing large dimension input arrays on the GTX 680 graphics processing unit (32,768 – 60,000,000 elements) than using traditional central processing units. The GTX 680 recorded an improvement of up to 140.93x in both execution time (1.643266 ms versus 231.584732 ms) and bandwidth (146.0506 GB/s versus 1.0363 GB/s) compared to the i7-2600K processor.
- The CPU offers the best results (lower execution time, higher bandwidth) when the input array has a relatively low dimension (35 – 1030 elements) that does not generate an enough computational load in order to fully use the huge parallel processing capacity of the GPU. This aspect imposes the necessity of using a hybrid solution in order to obtain the best in class performance for different scenarios: a solution that uses the CPU when the input vector's dimension is low and the GPU for large data volumes.
- A particular interest was to research how well the optimization techniques scale to the latest generation of GPUs from the Kepler architecture (implemented in the GTX 680) in contrast to previous GPUs architectures (like the GTX 280 and GTX 480). The optimization solutions of the parallel reduction function scaled well across different GPU architectures, confirming their efficiency. The function proves to be applicable and useful in a wide range of algorithms and data processing applications, providing optimal results in various situations.
- When running the parallel reduction algorithmic function on the GTX 680 processor, using integer, unsigned integer or float input data types, we have recorded a comparable performance in terms of execution times. Processing double, long long or unsigned long long input data types, has led to a comparable performance, but the execution times increased up to 1.96x (1.642558 ms compared to 3.226117 ms). The performance is comparable in all of the six considered cases regarding the bandwidth. By analyzing these results we found that, regardless of the type of data being processed, the efficiency of the optimization solutions developed and applied to the parallel reduction function is confirmed.
- Using various types of associative binary operators (summation, maximum, minimum or multiplication), we found that the optimization solutions used in developing the parallel reduction function offer a high level of performance and we have recorded comparable results in all these situations.
- All the obtained experimental results confirm the efficiency of the parallel reduction algorithmic function that offers optimal results in different scenarios without being significantly influenced by the type of input data or the chosen binary operator, and therefore it can be implemented in a wide range of data parallel algorithms.

There has been a lot of interest in the literature lately for the optimization of the parallel reduction algorithmic function, but none of the works so far (to our best knowledge) tried to validate if the optimizations techniques can be applied to a GPU from the Kepler architecture. The study demonstrates that the GTX 680, the latest CUDA-enabled GPU from the Kepler architecture is capable of efficient and accurate reduction.

One important aspect to take into account is that the GTX 280, GTX 480 and GTX 680 are consumer-oriented graphic cards that are not designed specifically for high performance scientifically computations, like the Quadro series. The three graphics processors are especially optimized for graphic processing and rendering in video games and not for scientific computations. We have preferred these three processors due to their reduced cost, high level of performance and wide accessibility. The registered results, including those on the GeForce GTX 280 architecture launched four years ago, far exceed those obtained on the last generation central processing unit, Sandy Bridge i7-2600K (even if the CPU has been overclocked at 4.6 GHz).

The most important goal when designing and developing the parallel reduction algorithmic function was to obtain a CUDA processing solution that is self-adjustable and self-configurable (regarding the number of thread blocks, number of threads per block, number of elements processed per thread) depending on the GPU's architecture. The experimental results proved that the developed solution offers a high degree of performance on an entire range of CUDA enabled GPUs: the Tesla GT200 architecture, launched on 16 Jun 2008; the Fermi GF100 architecture, launched on 26 March 2010 and the Kepler GK104 architecture, released on 22 March 2012. The high level of performance achieved on various GPUs architectures from different generations confirms the efficiency and the high degree of applicability for the optimization solutions of the parallel reduction algorithmic function in CUDA. As this high performance has been recorded in a variety of scenarios, the developed reduction function proves its applicability and usefulness in a wide range of algorithms. Moreover, the Compute Unified Device Architecture represents a novel approach for developing efficient parallel software on multithreaded architectures, a very useful tool for developing solutions that optimize the data processing at low-costs and with a spectacular performance.

References

[1] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Boston: Addison-Wesley Professional, 2010, pp. 79-94.

[2] G. E. Blelloch, "Prefix Sums and Their Applications", in *Synthesis of Parallel Algorithms*. San Francisco: Morgan Kaufmann, 2009, pp. 35-59.

[3] S. Sengupta, M. Harris and M. Garland, "Efficient Parallel Scan Algorithms for GPUs", *Nvidia Technical Report NVR-2008-003*, December 2008.

[4] N. Satish, M. Harris and M. Garland, „Designing efficient sorting algorithms for manycore GPUs, in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, 2009, pp. 1-10.

[5] S. Sengupta, M. Harris, Y. Zhang and J. D. Owens, "Scan primitives for GPU computing", in *Graphics Hardware 2007 ACM SIGGRAPH/Eurographics Symposium proceedings*, San Diego, 2007, pp. 97-106.

[6] W. W. Hwu, *GPU Computing Gems Jade Edition*, San Francisco: Morgan Kaufmann, 2011, pp. 1-560.

[7] GeForce Graphics Cards, *http://www.nvidia.com/object/geforce_family.html*, accessed at 10 July 2012.

[8] I. Lungu, D. M. Petrosanu D. and A. Pirjan, "Solutions for optimizing the parallel reduction algorithm using the Compute Unified Device Architecture, in *Proc. of The Eleventh International Conference on Informatics in Economy*

– *Education, Research & Business Technologies IE 2012*, ASE Bucharest, 2012, pp. 7-11.
[9] A. Pirjan, "Improving software performance in the Compute Unified Device Architecture", Informatica *Economica*, vol. 14, no. 4, pp. 30-47, December 2010.
[10] A. Pirjan, "Solutions for optimizing the stream compaction algorithmic function using the Compute Unified Device Architecture", *Journal of Information Systems & Operations Management*, vol. 6, no. 1, pp. 216-231, June 2012.
[11] I. Lungu, A. Pirjan and D. M. Petrosanu, "Solutions For Optimizing The Data Parallel Prefix Sum Algorithm Using The Compute Unified Device Architecture", *Journal of Information Systems & Operations Management*, vol. 5, no. 2.1, pp. 465-477, Dec. 2011.

Ion LUNGU is a Professor at the Economic Informatics Department at the Faculty of Cybernetics, Statistics and Economic Informatics from the Academy of Economic Studies of Bucharest. He has graduated the Faculty of Economic Cybernetics in 1974, holds a PhD diploma in Economics from 1983 and, starting with 1999 is a PhD coordinator in the field of Economic Informatics. He is the author of 41 books in the domain of economic informatics, 57 published articles (among which 20 articles ISI indexed or included in international databases) and 39 scientific papers published in conferences proceedings (among which 5 papers ISI indexed and 15 included in international databases). He participated (as director or as team member) in more than 20 research projects that have been financed from national research programs. He is a CNCSIS expert evaluator and member of the scientific board for the ISI indexed journal Economic Computation and Economic Cybernetics Studies and Research. He is also a member of INFOREC professional association and honorific member of Economic Independence academic association. In 2005 he founded the master program Databases for Business Support, who's manager he is and in 2010 he founded Databases Journal. His fields of interest include Databases, Design of Economic Information Systems, Database Management Systems, Decision Support Systems, Executive Information Systems and Business Intelligence.

Dana-Mihaela PETROSANU has graduated the Faculty of Mathematics and Computer Science, from the University of Bucharest, in 1988. She holds a PhD diploma in Mathematics since 2010. She was a teaching assistant from 1990 and currently she is a senior lecturer at the Faculty of Applied Sciences from the University Politehnica in Bucharest. She is the author of 2 books and 21 journal articles, in the fields of interest: Applied Mathematics, Geometry, Mechanics, Differential Equations and Complex Analysis. She was a member in 9 scientific research projects.

Alexandru PIRJAN has graduated the Faculty of Computer Science for Business Management in 2005. He holds a MA Degree in Computer Science for Business from 2007. He joined the staff of the Romanian-American University as a stagier teaching assistant in 2005 and a Lecturer Assistant in 2008. He is a PhD candidate since 2009 at the Doctoral School from the Bucharest Academy of Economic Studies. 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 28 journal articles and a book. He was a member in 6 national scientific research projects. His work focuses on parallel processing, database applications, artificial intelligence and quality of software applications.