SOLUTIONS FOR OPTIMIZING THE STREAM COMPACTION ALGORITHMIC FUNCTION USING THE COMPUTE UNIFIED DEVICE ARCHITECTURE

Alexandru Pîrjan[1]

Abstract

In this paper, I have researched and developed solutions for optimizing the stream compaction algorithmic function using the Compute Unified Device Architecture (CUDA). The stream compaction is a common parallel primitive, an essential building block for many data processing algorithms, whose optimization improves the performance of a wide class of parallel algorithms useful in data processing. A particular interest in this research was to develop solutions for optimizing the stream compaction algorithmic function that offers optimal solutions over an entire range of CUDA enabled GPUs: Tesla GT200, Fermi GF100 and the latest Kepler GK104 architecture, released on 22 March 2012. In order to confirm the utility of the developed optimization solutions, I have extensively benchmarked and evaluated the performance of the stream compaction algorithmic function in CUDA.

Keywords: parallel processing, CUDA, Kepler, threads, stream compaction.

Introduction

For a long time, graphics processing units (GPU) have been used solely as graphic accelerators, for graphic rendering specific functions. The increased necessity for improved high resolution three-dimensional rendering and a large number of frames per second have led to the GPUs' evolvement. By the end of the 1990s, these units have become programmable at a hardware level and in 2006, by unifying both software and hardware components, the NVIDIA company has released the Compute Unified Device Architecture (CUDA), a new parallel programming model based on the GPU's computational processing power for solving complex processing tasks more efficiently than using central processing units (CPUs) [1]. This development has made it possible to accelerate a broad class of data processing applications. In this context, the improvement of the data extraction and parallel processing can be achieved using high-performance basic functional blocks (and among them, the stream compaction algorithmic function), designed to offer optimum performance and efficiency, based on graphics processing units with multiple processing cores.

With the introduction of CUDA-C language, application developers are able to harness, using a standard programming language, the huge parallel processing power offered by the latest generation of graphics processing units. CUDA enables developers to specify how tasks are decomposed in order to be processed by many parallel threads and how are these tasks executed by the GPU [2]. The high level of control offered by the CUDA-C

[1] Ph D Candidate, Faculty of Computer Science for Business Management, Romanian-American University, 1B, Expozitiei Blvd., district 1, code 012101, Bucharest, Romania, e-mail: alex@pirjan.com

language facilitates the development of high-performance basic functions useful for optimizing a wide range of computational tasks that require high processing power.

The CUDA processing instructions flow consists of 4 stages (**Figure 1**):

1. In Stage 1, data is copied from the system's memory into the graphics processing unit's memory.
2. In Stage 2, the graphics processing unit receives the processing request instruction.
3. In Stage 3, every processing core of the GPU parallel processes its data and stores the result in the GPU's memory.
4. In Stage 4, the results are copied back from the GPU memory to the system memory

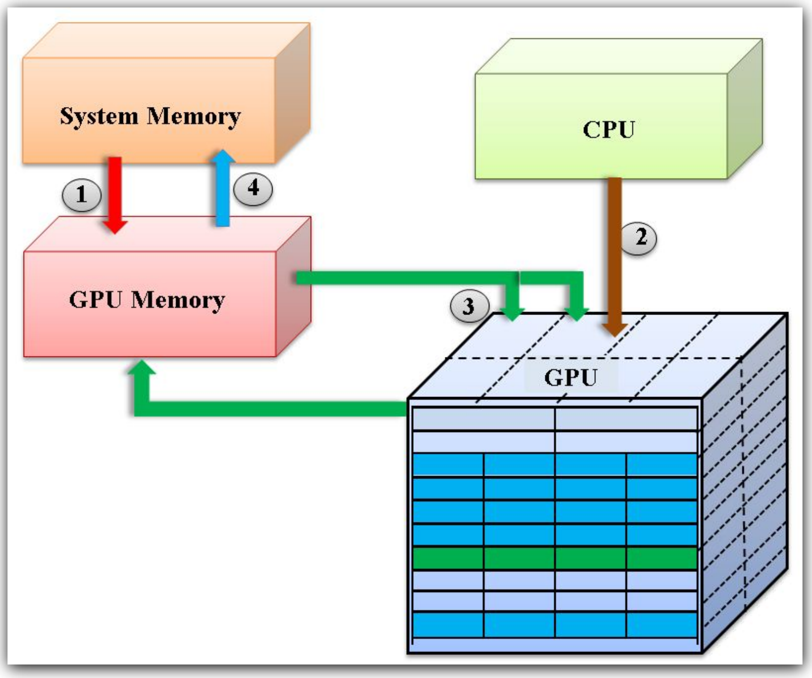


Figure 1. The CUDA processing instructions flow

The latest three CUDA-enabled graphic cards are GTX 280, GTX 480 and GTX 680. The GTX 280 from the Tesla GT200 architecture, launched on 16 Jun 2008, is based on 65 nm fabrication technology, has 1.4 billion of transistors, 240 CUDA cores and 30 streaming multiprocessors. The graphics clock runs at 602 MHz, the processor clock at 1296 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 a texture fill rate of 48.2 billion/sec, 80 texture units, 32 ROP units and the maximum board power (TDP) of 236 Watts.

The GTX 480 from the Fermi GF100 architecture, launched on 26 March 2010, is based on 40 nm fabrication technology, has 3.2 billion of transistors, 480 CUDA cores and 15 streaming multiprocessors. The graphics clock runs at 700 MHz, the processor clock at

1401 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 a texture fill rate of 42 billion/sec, 60 texture units, 48 ROP units and the maximum board power (TDP) of 250 Watts.

The newest CUDA graphic card, the GTX 680 from the Kepler GK104 architecture, released on 22 March 2012, is based on 28 nm fabrication technology, has 3.54 billion of transistors, 1536 CUDA cores and 8 streaming multiprocessors. The graphics clock runs at 1006 MHz, the boost clock at 1058 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 a texture fill rate of 128.8 billion/sec, 128 texture units, 32 ROP units and the maximum board power (TDP) of 170 Watts. 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.

In the following, I depict the stream compaction function, an overview regarding different approaches in designing it and an efficient algorithmic method for developing and implementing the stream compaction algorithmic function in the CUDA architecture.

The Stream Compaction Function

The stream compaction (stream reduction) is an important algorithmic function, a primitive building block, useful in many algorithms that benefit from the massive parallelism. Highly parallel algorithms often produce sparse data or data containing unwanted elements, especially in the situations when each input element produces a varying number of output elements. In order to maintain the overall performance, it is often necessary to compact the data before reaching the next processing steps. Using the stream compaction, it is obtained a compacted stream that provides a balanced computational load for all the processors.

Using the huge computational power provided by the CUDA parallel architecture, the stream compaction algorithmic function’s optimization is a solution to improve the performance of all data processing algorithms that require stream compaction. Therefore, the stream compaction function can improve the performance of a wide class of parallel algorithms useful in data processing [3].

In the following, I describe some situations in which the stream compaction is successfully applied. In the parallel breadth first tree traversal, after each traversal step, invalid nodes must be pruned from the list of open nodes because, otherwise, an exponential increase of the nodes number would take place [4]. Similar problems are encountered in other situations, for example those related to image processing techniques that simulate light interaction with various surfaces [5] or GPU-based collision detection issues [6].

The easiest implementation of the stream compaction is through a sequential algorithm, running on a uniprocessor machine: valid elements are moved from the input vector to the output one. On parallel architectures, the implementation of an efficient stream compaction is more difficult because for each input element of the stream, the output position critically depends on the previous state of the element, before applying the compaction. Although it seems to be the most appropriate, the implementation that synchronizes after each element has been processed is very inefficient.

In the literature, there are various approaches to overcome this problem, most of them being based on performing a parallel prefix sum [7], [8]. The parallel prefix sum is applied on a stream containing a "0" for each invalid element and a "1" for each valid element of the input vector. As a result of this operation, it is obtained a stream containing for each element, the number of valid elements preceding it. This information is useful for moving each valid element in the new location.

In 2005, this approach was implemented on a GPU [9]. In that moment, the graphics processing units did not provide random write access to memory and a binary search was used in order to find the input element corresponding to each output. In this case, the prefix sum has the $O(n)$ time complexity, the binary searches has the $O(log\, n)$ complexity, while the overall complexity is $O(n \cdot log\, n)$.

A similar approach was used later in [10]. In this case, the prefix sum has been improved by using a more efficient implementation, the binary search step remained unchanged and the total time complexity remained $O(n \cdot log\, n)$. Another approach [6], [11] uses a tree containing the valid elements. The binary search step is performed by searching into the tree for the correct input element corresponding to each output. The time complexity remains the same as in the previous cases, $O(n \cdot log\, n)$.

In order to improve the time complexity, the algorithm has been applied to smaller data fragments of fixed size [12]. The k-size fragments are compacted, using the previous algorithms, using $O(k \cdot logk)$ steps. Then, the compacted fragments are concatenated using the GPU. This version of the algorithm provides a $O(n)$ time complexity.

Nowadays, the graphics processing units incorporate specific hardware that facilitates the efficient implementation of stream compaction. In the literature there are a few results regarding this aspect, but the obtained performance is disappointing [12]. The modern graphics processing units also provide random write access to memory, which can be used to replace the binary search phase. This implementation has the time complexity $O(n)$ [9], [10].

The stream compaction is closely related to data sorting. A fast sorting technique that uses an efficient compaction algorithm is depicted in [13]. The stream compaction is useful in a variety of general purpose applications, including collision detection and sparse matrix compression, is the main method for transforming heterogeneous vectors, with elements of several types, in homogeneous vectors, with elements of the same type. This feature is particularly useful for those vectors in which only some elements are of interest for the analyzed problem. The stream compaction produces a smaller vector, containing only the elements of interest, that is being processed more efficiently and thus, the transfer times, particularly between the GPU and the CPU, are significantly reduced.

The stream compaction is a filtering operation: a subset of elements are selected from an input vector and used to build an output array of smaller dimension, containing only the elements of interest for the studied problem. Formally, the stream compaction starts with an input vector $v = (v_1, v_2, \dots, v_n)$ and a predicate p. The output vector contains only those elements of v for which $p(v_i)$ is true, keeping the order of the input elements [9] (**Figure 2**).

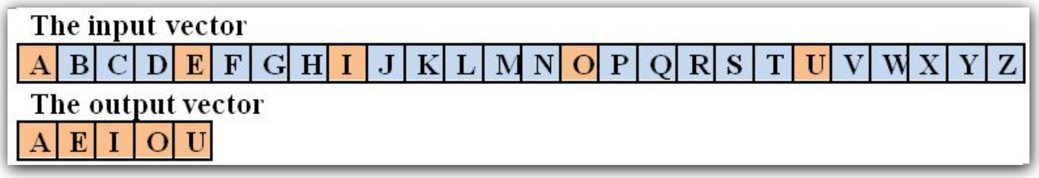


Figure 2. An example for the stream compaction algorithmic function

As it was mentioned above, the graphics processing units on which Horn implemented the stream compaction in 2005 did not provide random write access to memory, so Horn has replaced this write operation with a sequence of binary search steps [9]. The compaction of n elements was achieved by applying $log\,n$ binary search steps, but this technique required a large amount of resources. The native support for random write access in memory implemented in modern graphics processing units facilitates the design of stream compaction algorithms that are considerably more efficient.

Designing An Efficient Stream Compaction Algorithmic Function In Cuda

In the following, I present an efficient method for designing and implementing the stream compaction algorithmic function in the CUDA architecture. The stream compaction algorithmic function that I have developed is useful for architectures based on multiple processing cores, as SIMD (Single Instruction, Multiple Data). In this case, the multiple data streams and the single instruction stream facilitate the parallel processing of data. The processing units are acting on different data elements, executing the same instruction. These systems are useful in specialized graphics problems and in sound processing. When designing the stream compaction algorithmic function, I have considered a number of technical issues for optimizing the performance: maximizing concurrent execution, minimizing synchronizations, reducing the bandwidth and memory requirements. In the following, $v = (v_1, v_2, \dots, v_n)$ represents the input vector, p is a predicate, used to select the elements of interest from the input vector.

I have developed the stream compaction algorithmic function in three steps:

Step 1. First, it is generated a temporary array containing:
- a 1 for those elements of the input vector for which $p(v_i)$ is true;
- a 0 for those elements of the input vector for which $p(v_i)$ is false.

Step 2. The parallel prefix sum algorithmic function [14] is applied to the vector obtained in **Step 1**. The result of this operation is a vector containing, for each element, the number of elements from the input vector prior to that item, for which $p(v_i)$ is true. Thus, each element of this new vector offers information about its position in the output vector.

Step 3. The information obtained in **Step 2** is then used to move each valid element in its new location, benefiting from the random write access to memory available in the latest generation of CUDA graphics processing units (**Figure 3**).

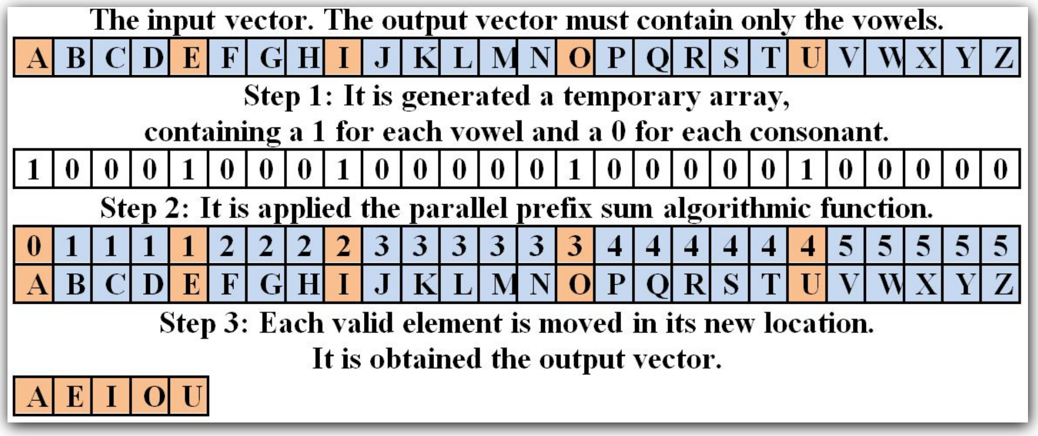
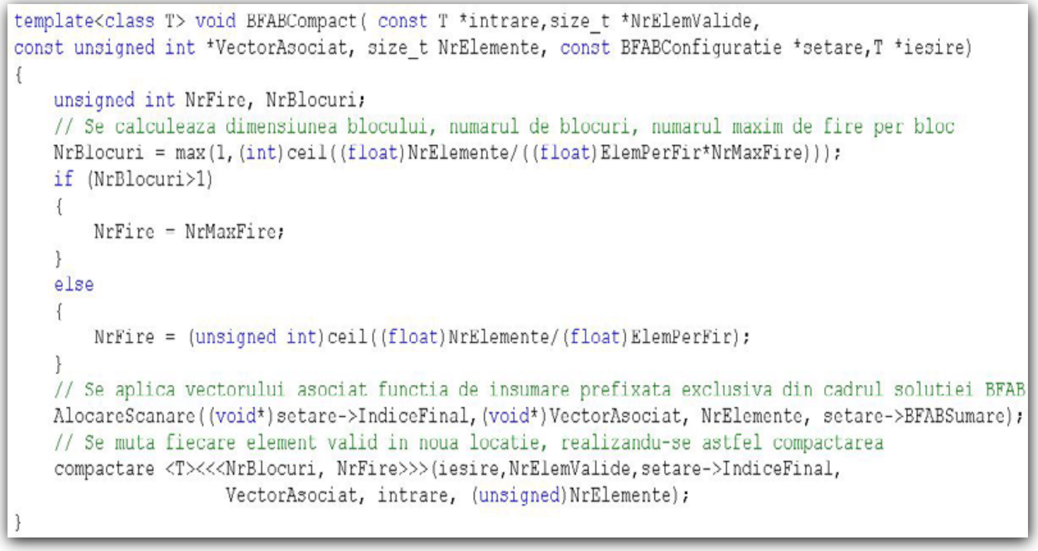

The input vector. The output vector must contain only the vowels.
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Step 1: It is generated a temporary array, containing a 1 for each vowel and a 0 for each consonant.
1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0
Step 2: It is applied the parallel prefix sum algorithmic function.
0 1 1 1 1 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 5 5 5 5 5
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Step 3: Each valid element is moved in its new location. It is obtained the output vector.
A E I O U

Figure 3. An example of applying the algorithmic steps of the stream compaction algorithmic function

In order to optimally process the stream compaction, I designed and developed the basic stream compaction algorithmic function as to first compute the thread block's size, the number of blocks and the maximum number of threads per block, depending on the total number of elements that have to be processed. These parameters are needed to fully exploit the huge computational power of the GPU, obtaining thus a more efficient data compaction process.

The function has been designed to allow the selection of the input array components, from one of the data types: integer, unsigned integer, float, double, long long or unsigned long long. The function also offers the possibility to set in advance the parameters of a configuration structure that are then passed as parameters to the BFABCompact function (**Figure 4**).

As in the Step 2 of the stream compaction algorithmic function, the parallel prefix sum algorithmic function is applied to the vector obtained in the Step 1, the overall performance of the stream compaction algorithmic function is significantly influenced by the performance of the parallel prefix sum algorithmic function.



```
template<class T> void BFABCompact( const T *intrare,size_t *NrElemValide,
const unsigned int *VectorAsociat, size_t NrElemente, const BFABConfiguratie *setare,T *iesire)
{
    unsigned int NrFire, NrBlocuri;
    // Se calculeaza dimensiunea blocului, numarul de blocuri, numarul maxim de fire per bloc
    NrBlocuri = max(1,(int)ceil((float)NrElemente/((float)ElemPerFir*NrMaxFire)));
    if (NrBlocuri>1)
    {
        NrFire = NrMaxFire;
    }
    else
    {
        NrFire = (unsigned int)ceil((float)NrElemente/(float)ElemPerFir);
    }
    // Se aplica vectorului asociat functia de insumare prefixata exclusiva din cadrul solutiei BFAB
    AlocareScanare((void*)setare->IndiceFinal,(void*)VectorAsociat, NrElemente, setare->BFABSumare);
    // Se muta fiecare element valid in noua locatie, realizandu-se astfel compactarea
    compactare <T><<<NrBlocuri, NrFire>>>(iesire,NrElemValide,setare->IndiceFinal,
                    VectorAsociat, intrare, (unsigned)NrElemente);
}
```

Figure 4. Computing the parameters and calling the stream compaction algorithmic function in CUDA

In the following section, I will outline several solutions for optimizing the performance of the stream compaction algorithmic function in CUDA, solutions that I have used for developing this function.

Solutions For Optimizing The Performance Of The Stream Compaction Algorithmic Function In Cuda

By consulting the scientific literature and using a thorough analysis of the current state of knowledge, I have identified and developed a series of solutions for optimizing the performance of the stream compaction algorithmic function in CUDA:

Solution 1 - calling the optimized parallel prefix sum function developed in CUDA. In the Step 2 of the stream compaction algorithmic function, I have used the optimized parallel prefix sum algorithmic function developed in CUDA [14]. This is the most important optimization technique applied in designing the stream compaction function and therefore a performance improvement of up to 45% was obtained compared to the case when running in this step a sequential algorithm, on the central processing unit.

Solution 2 – optimizing the management of resources used in the processing tasks. Another important solution that I have addressed in the development and design of the stream compaction algorithmic function, in order to optimize data processing, consists in the optimal management of resources used for the processing tasks. For improving the software performance in CUDA, when threads are being defined, one must take into account the high latency of global memory. While the CPU architecture uses large caches to hide memory latencies, CUDA generates and uses thousands of active threads [15]. First, I have computed the thread block's size, the number of blocks and the maximum number of threads per block, depending on the total number of elements that have to be processed. Therefore, I have obtained an optimal allocation of resources according to the tasks that must be processed, benefiting from the huge computational power provided by the GPU, improving the efficiency of the entire stream compaction process, which results in a substantial improvement in runtime compared to sequential implementations that are run on central processing units.

Solution 3 - maximizing the concurrent execution of tasks. The graphics processing units execute and manage hundreds of concurrent threads, avoiding computational overloading and reducing the memory latency, as they provide 240 CUDA processing cores in the case of the GTX 280 graphic card, 480 CUDA processing cores for the GTX 480 graphic card and 1,536 CUDA processing cores for the GTX 680 graphic card. Thus, taking into account that each thread can independently process a part of the source code (as it has its own private memory, private registers and program counter) and that there are required hundreds of execution threads in order to fully employ the processing cores, I have optimized the tasks assigned to each available thread using thousands of active threads offered by the CUDA architecture for improving the software performance of the stream compaction function. Taking into account that processing a single element per thread does not generate an enough computational load for reducing the memory latency, I have allocated eight input elements to each thread. This happens when data is loaded from global to shared memory. Each execution thread reads two groups of four elements and then sequentially processes them.

Solution 4 – managing shared memory bank conflicts. An important issue that I took into account when designing the stream compaction algorithmic function refers to managing shared memory bank conflicts. The stream compaction algorithmic function calls, during its execution, the optimized parallel prefix sum function developed in CUDA [14] and frequently uses the CUDA shared memory, composed of multiple memory banks (memory modules of equal size) [1]. Multiple data requests from the same memory bank generate shared memory banks conflicts. When a conflict occurs, the hardware device serializes memory operations and this makes all the threads wait until all memory requests have been fulfilled. This process consumes significant time resources. For solving this situation, I have designed the stream compaction algorithmic function as to use “warps” (groups of 32 threads) serialization markers, in order to divide the data blocks in more fragments that are being processed independently, using one warp for each fragment. I have preferred this technique because it helps avoiding shared memory bank conflicts and reduces the necessity of synchronization, as the synchronization is not required for sharing data within the same warp, unlike the case when the shared data belongs to different warps. The instructions are executed in SIMD mode (Single Instruction, Multiple Data).

Solution 5 – reducing the number of synchronization operations between parallel tasks. The parallel task synchronization represents real time coordination and is often associated with inter thread communication. Synchronization is usually implemented by setting a synchronization point in the application from which a task cannot continue until another task reaches a certain point. In the Step 2 of the stream compaction algorithmic function, I have used the parallel prefix sum function developed in CUDA [14]. As I have used the parallel prefix sum function within a warp when designing this function, I have eliminated the necessity of synchronization points, because inter threads communication occurs at the warp level, so I do not have to synchronize in order to share data within the same warp. Each of the warps' last elements is stored in the shared memory and finally a single warp sums the previously stored results.

Solution 6 – minimizing the number of used registers. The CUDA graphic processing units use multiple threads (multithreading) to reduce the memory latency. The number of executing threads that can be simultaneously used is often limited by their registry requirements. I have minimized the number of used registers, thus maximizing the available number of threads.

Solution 7 – optimizing the memory bandwidth. To design an efficient stream compaction algorithmic functions, I have first divided the input data into smaller fragments allocated to the thread blocks. Therefore, I have reduced the number of global memory calls and of the execution time, by optimally allocating thread blocks and using the GPU's shared memory instead of the global one.

Solution 8 - minimizing data transfers between the host and the device. Given that data transfers between the host and the device consume considerable computational resources, I have designed the stream compaction algorithmic function so that data transfers between the host and the device are minimal. I have reduced these transfers at the beginning, when data is being copied from host to the device memory and, in the end, when the compacted vector is copied back into the host memory.

Solution 9 - reducing the number of executed instructions. The stream compaction algorithmic function that I have developed is useful for multicore architectures such as SIMD (Single Instruction, Multiple Data), in which case instructions are synchronous within each warp. I have noticed that, as the stream compaction process progresses, the number of active threads decreases, and when the number of threads is less than or equal to 32, it means that only the last warp remained to be executed. Within this warp, there is no need to synchronize the threads. Therefore, I have decided to remove these instructions and thus I have obtained an improvement in the overall performance.

In the following I present a benchmark suite that I have developed in order to emphasize the performance of the stream compaction algorithmic function in CUDA, optimized using the above presented Solutions 1-9.

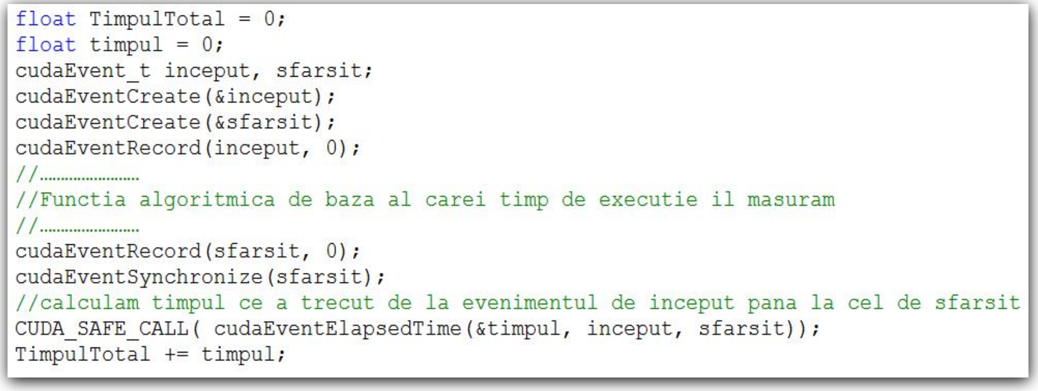
The Experimental Results And The Performance Analysis Of The Stream Compaction Algorithmic Function In Cuda

In this section I analyze the performance of the above described parallel stream compaction algorithmic function, using the Windows 7 64-bit operating system and the following configuration: Intel i7-2600K clocked at 4.6 GHz, with 8 GB (2x4GB) of 1333 MHz, DDR3 dual channel. I have used the NVIDIA graphic cards GeForce GTX 280, GTX480 and GTX 680. Programming and access to the GPUs used the CUDA toolkit 4.1, with the NVIDIA driver version 270.81 (for the GTX 280 and GTX 480) and 300.1 (for the GTX 680). In addition, all processes related to the graphical user interface have been disabled to reduce the external traffic to the GPU.

Measurements do not include the necessary time for data transfers between the central processing unit and the graphic processing unit, as the stream compaction algorithmic function is designed to be used as a building block for a large number of applications running on GPUs, so the transfer times will vary depending on the complexity of the specific application.

In order to compute the average execution time that the GPU spends for executing the stream compaction algorithmic function, I have used the CUDA application programming interface (API). I have preferred this option instead of those based on the CPU's or on the operating system's timers, because those methods would have included latency and variations from different sources. In addition, computations can be asynchronously performed on the host while the GPU kernel runs and the only way to measure the necessary time for the host computations is to use the CPU or the operating system timing mechanism.

A GPU time stamp recorded at user specified moment in time represents an event in CUDA. Because the time stamp is recorded directly by the GPU, I have not encountered the problems that could have appeared if I had tried to time the GPU execution using CPU timers. In order to time correctly the function's execution, I have created both a start and a stop event. Some of the kernel calls I make in CUDA C are asynchronous, the GPU begins to execute the code but the CPU continues the execution of the next code line before the GPU has finished. In order to safely read the value of the stop event I instruct the CPU to synchronize on the event using the API function "cudaEventSynchronize()", as depicted in **Figure 5.**



```
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 ce a trecut de la evenimentul de inceput pana la cel de sfarsit
CUDA_SAFE_CALL( cudaEventElapsedTime(&timpul, inceput, sfarsit));
TimpulTotal += timpul;
```

Figure 5. Measuring the execution time using CUDA events

In this way I have set the runtime to block further instructions until the GPU has reached the stop event so when calling the "cudaEventSynchronize()" I am sure that all the GPU work prior to the stop event has been completed and it is safe to read the recorded time stamp. In this way, I get a reliable measurement of the execution time for computing the above described stream compaction algorithmic function [1].

The first set of tests evaluates the execution times obtained by applying the stream compaction algorithmic function on vectors of various sizes and elements of float type. The vectors have been randomly generated as to cover a wide range of values. In **Table 1** I present the results of the experimental tests when running the stream compaction algorithmic function on the CPU and on the two GPUs mentioned above. The results represent the average of 10,000 iterations, and the unit of measure is milliseconds (ms).

Table 1. Experimental results for the stream compaction algorithmic function

Test No.	**Number of elements**	**Execution time on CPU (ms)**	**Execution time on GPU (ms)**		
			GTX 280	**GTX 480**	**GTX 680**
1	35	0.000251	0.163672	0.080994	0.069412
2	128	0.000881	0.165035	0.055889	0.102792
3	256	0.001751	0.150721	0.05853	0.068543
4	260	0.001802	0.164149	0.050703	0.069557
5	512	0.003764	0.167543	0.076934	0.068769
6	1000	0.00809	0.170796	0.070327	0.083748
7	1024	0.008283	0.162103	0.063049	0.080717
8	1030	0.008401	0.335697	0.117377	0.090358
9	32768	0.306007	0.34606	0.132379	0.144445
10	45555	0.425129	0.276806	0.138398	0.142196
11	65536	0.611791	0.389857	0.130423	0.144086
12	131072	1.231282	0.374864	0.158547	0.147817
13	262144	2.467049	0.441118	0.179466	0.152783
14	500111	4.770547	0.552596	0.248678	0.173436
15	524288	4.925248	0.599289	0.249158	0.166949
16	1048555	9.976362	0.747874	0.355804	0.217646
17	1048576	9.933178	0.712332	0.336078	0.218307
18	1048581	9.930643	0.86948	0.402963	0.238417
19	2097152	19.905175	1.216657	0.632166	0.336693
20	2097999	19.964916	1.254584	0.627385	0.329394
21	4194334	39.781245	1.961108	1.050636	0.561816
22	8388600	79.936226	3.212200	1.935706	0.947059

In **Figure 6** I present the obtained experimental results by running the stream compaction algorithmic function when the input array has a relatively low dimension (35-1030 elements). In this case I have noticed that the central processing unit has the best execution time, because it has not been generated an enough computational load in order to use the huge parallel processing capacity of the GPUs. In this case, the best result is obtained when running the stream compaction algorithmic function on the CPU.

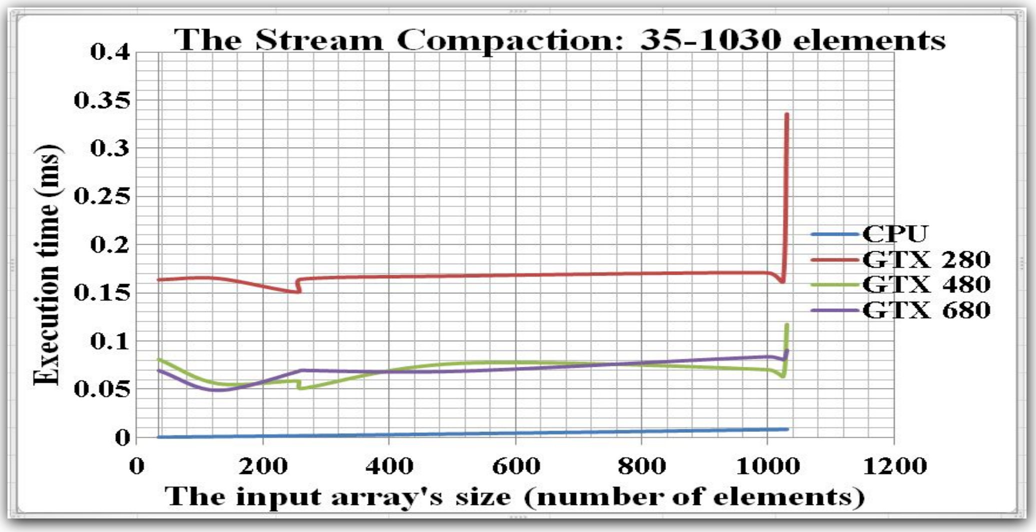


Figure 6. The stream compaction algorithmic function: 35-1030 elements of the input array

I have analyzed the obtained experimental results when running stream compaction algorithmic function on a large dimension input array (1030-8388600 elements) (**Figure 7).** In this case, I have noticed that the GTX 680 graphic card obtains the best execution time, because this time it has been generated a sufficient computational load to fully employ the huge parallel processing capacity of the GPU. Even when processing an input vector of 8388600, the GTX 680 offers an execution time of under 1 millisecond (0.947059 ms).

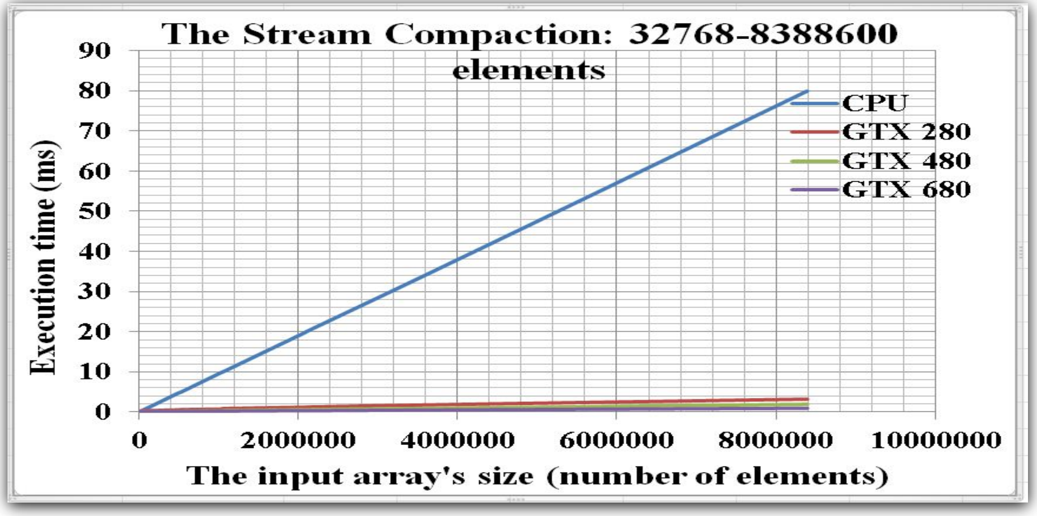


Figure 7. The stream compaction algorithmic function: 1030-8388600 elements

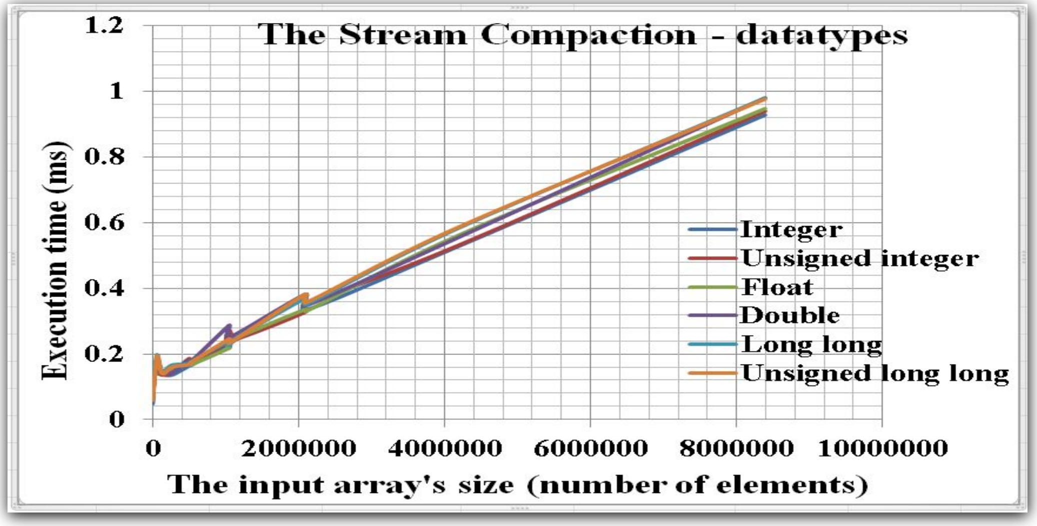


Figure 8. The influence of data types on the performance of the stream compaction algorithmic function

As the stream compaction algorithmic function has been designed to allow the selection of the input array components' data type, in the next set of tests, I evaluate the influence of the data types on its performance. In **Figure 8** there are presented the obtained experimental results when running the stream compaction algorithmic function on an input array of variable dimension (35-8388600 elements) and various type of input data, using the GTX 680 graphic processor. The results represent the average of 10,000 iterations. One can observe that the performance is comparable for all the types of input data, the execution time ranging between 0.049303 ms and 0.979995 ms.

The experimental results confirm the efficiency of the developed optimization solutions and the stream compaction algorithmic function's efficiency, which offers optimum results in different situations: when the number of elements of the input vector varies (35-8,388,600 elements); when the data types varies; on different graphics processors architectures. The function provides a high degree of performance in different situations and a great applicability potential in a wide range of data processing applications and algorithms.

Conclusions

In this paper I have researched and developed an efficient implementation of the stream compaction algorithmic function in CUDA, using different optimization solutions. I have first analyzed and designed the algorithmic function, highlighting the algorithm's steps. I have developed and then applied a series of solutions to improve the performance of the stream compaction algorithmic function (**Table 2**).

Table 2. Solutions for improving the performance of the stream compaction algorithmic function in CUDA

No.	**The solution for improving the performance of the stream compaction algorithmic function in CUDA**
Solution 1	Calling the optimized parallel prefix sum function developed in CUDA
Solution 2	Optimizing the management of resources used in the processing tasks
Solution 3	Maximizing the concurrent execution of tasks
Solution 4	Managing shared memory bank conflicts
Solution 5	Reducing the number of synchronization operations between parallel tasks
Solution 6	Minimizing the number of used registers
Solution 7	Optimizing the memory bandwidth
Solution 8	Minimizing data transfers between the host and the device
Solution 9	Reducing the number of executed instructions

I have analyzed the performance of the stream compaction 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 graphic processing unit I have used the CUDA application programming interface (API). After having analyzed the experimental results obtained by using the solutions for optimizing the performance of the stream compaction algorithmic function, I have noticed the following:

- When the stream compaction algorithmic function is run on the GTX 680 graphics processor and the input vectors have sizes ranging from 32,768 to 8,388,600 elements, I have recorded improvements of up to 84.40 x in terms of execution time (0.947059 ms compared to 79.936226 ms) compared to the stream compaction run on the central processing unit i7-2600K. I have progressively optimized the solutions in order to benefit from the huge computational power of the GPUs. When the input vector's size ranges from 35 to 1030, the CPU provides the best results (the lowest execution time), because there is not a sufficient computational load to ensure the full employment of the huge parallel capacity of the graphics processors. In order to reach the best performance, one must process low dimension input vectors using the CPU and large dimension input vectors using the GPU.
- When running the stream compaction algorithmic function on an input array of variable dimension (35-8388600 elements) and various type of input data, using the GTX 680 graphic processor, one can observe that the performance is comparable for all the types of input data, the execution time ranging between 0.049303 ms and 0.979995 ms. These results confirm the efficiency of the optimization solutions used for developing the stream compaction algorithmic function, providing a high level of performance regardless of the processed data types.

One particular interest of this paper was to research how the optimization techniques scale to the latest generation of general-purpose graphic processing units, like the GTX 280, the

GTX 480 and the GTX 680. The study demonstrates that the GTX 680, the latest CUDA-enabled GPU from the Kepler architecture is capable of efficient and accurate stream compaction processing. Analyzing the literature, I have noticed that none of the works so far (to my best knowledge) has studied how well the optimization solutions for the stream compaction algorithmic function scale to the main CUDA architectures, especially Kepler, the latest generation of GPU architectures. One important aspect to take into account is that the GTX 680 is a consumer-oriented graphic card and is not designed specifically for high performance scientifically computations, like the Quadro series. I have preferred the GTX 680 solution due to its reduced cost and wide accessibility.

The most important aim when designing and developing the optimization of the stream compaction 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, etc.) depending on the GPU's architecture. The developed solution offers a high degree of performance over 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 solutions for optimizing the stream compaction algorithmic function in CUDA have a high degree of applicability, confirmed by the high performance achieved on various GPUs architectures from different generations. The high performance that has been recorded in a variety of scenarios confirms the applicability and usefulness of the analyzed stream compaction algorithmic function. The Compute Unified Device Architecture offers viable solutions for developing efficient parallel software that runs on multithread processing cores architectures. Analyzing the obtained experimental results, the CUDA parallel programming model proves to be a very useful tool for designing scalable parallel applications and for developing solutions that optimize data processing.

References

1. Sanders J., Kandrot E., CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, New Jersey, 2010.
2. GPU Computing Gems Jade Edition, Wen-mei W. Hwu, Morgan Kaufmann, 2011.
3. Billeter M., Olsson O., Assarsson U., Efficient Stream Compaction on Wide SIMD Many-Core Architectures, Proceedings of the Conference on High Performance Graphics, pp. 159-166, August, 2009.
4. Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D., Fast BVH construction on GPUs, in Proceedings of the Eurographics Symposium on Rendering, Eurographics and ACM/SIGGRAPH, Mar. 2009.
5. Wald I., Gribble C. P., Boulos S., Kensler A., SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering, Sci Institute Tech. Rep. UUSCI-2007-012, 2007.
6. Gress A., Guthe M., Klein R., GPU-based Collision Detection for Deformable Parameterized Surfaces, Computer Graphics Forum 25, 3, pg 497–506, John Wiley and Sons, Sept. 2006.
7. Blelloch G. E., Prefix Sums and Their Applications, in Synthesis of Parallel Algorithms, Morgan Kaufmann, San Francisco, 2009

8. Chatterjee S., Blelloch G. E., Zagha M., Scan primitives for vector computers. in Proc. of the 1990 Annual International Conference on Supercomputing, pages 666-675, 1990.
9. Horn D., Stream reduction operations for GPGPU applications, in GPU Gems 2, chapter 36, pg. 573-589, Addison Wesley, Mar. 2005.
10. Sengupta S., Lefohn A. E., Owens J. D., A work-effcient step-efficient prefix sum algorithm, in Proceedings of the Workshop on Edge Computing Using New Commodity Architectures, pg. D-26-27, Chapel Hill, May 2006.
11. Ziegler G., Tevs A., Theobalt C., Seidel H.-P., GPU Point List Generation through Histogram Pyramids, Technical Reports of the MPI for Informatics, MPI-I-2006-4-002, June 2006.
12. Roger D., Assarsson U., Holzschuch N., Efficient Stream Reduction on the GPU, in Workshop on General Purpose Processing on Graphics Processing Units, Eds. D. Kaeli and M. Leeser, 2007.
13. Satish N., Harris M., Garland M., Designing efficient sorting algorithms for manycore GPUs, Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, Rome, pp. 1-10, IEEE Publisher, Washington, 23-29 may 2009.
14. Lungu I., Pîrjan A., Petroşanu D., 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, pg. 465-477, December 2011.
15. Pirjan A., Improving software performance in the Compute Unified Device Architecture, Informatica Economica Journal, vol. 14, pp. 30-47, no. 4/December 2010.