GPGPU optimized parallel implementation of AES using C++AMP

Gabriel Munteanu, Ştefan Mocanu, Daniela Saru

Faculty of Automatic Control and Computer Science , University Politehnica of Bucharest, 060042, Romania, (gabriel.munteanu2909@yahoo.com, stefan.mocanu@upb.ro, daniela.saru@aii.pub.ro)

Abstract: Nowadays, the characterization of a computing system using attributes like "single core" is, for most applications, deprecated. Multiprocessor or multi-core platforms are, now, widespread and serve for solving more and more complex problems in shorter execution times. Video cards make no exception to this rule since, for the past years, they are based on powerful GPUs with high parallelism architectures and extremely fast memories. In addition, new development languages and platforms became available for the programmers. This way, the processing power of the GPU (Graphics Processing Unit) can now be used even for non-video or non-graphics applications that imply a serious amount of parallel processing. This paper presents a comparative study of AES algorithm implementation on CPU and two different GPGPU platforms. Similar studies involving GPGPUs are based on Nvidia's CUDA platform but this approach imposes a severe limitation over the application portability. In our approach a platform independent application was designed and implemented using C++ AMP, the latest C++ extension oriented to parallel programming. Tests were conducted over two GPGPU platforms, one from NVidia and one from AMD and a multi-core CPU from Intel. Results show that cross-platform portability was achieved while the performances are similar or better as compared to similar studies.

Keywords: graphic processing unit, GPGPU, encryption, AES, parallel processing, C++ AMP

1. INTRODUCTION

Processing time, when it comes to solving a problem, is reverse proportional with the available processing power. For this reason, efforts for improving actual computing techniques and facing new arriving challenges were made.

As one well knows, the heart of any intelligent device is the processor. Preoccupations regarding the increase of mathematical operations made in the same time unit have lead, in the digital era, to dramatic changes both in architecture and computing philosophy. Until 2006, engineers made huge efforts to increase the working frequency of the processors. This approach has reached it's technological limits due to the problems derived from high working frequencies such as high power consumption and huge amount of dissipated heat which required more and more advanced and expensive cooling systems. In this scenario, unfortunately, most of the consumed electric energy was converted into heat and then, lost, without having more than a marginal positive impact over the processing power.

The solution was represented by developing the processor with 2 cores (and more, soon afterwards) onto the same silicon chip, working at low frequencies, but capable of running, in real mode, 2 tasks in the same time. This way, real parallelism was possible which had a significant repercussion over lowering execution times.

Just like the central processors (CPU – Central Processing Unit), video cards had an interesting evolution over the time and the most significant moment, from this study's point of view, was the one when the use of these devices for other purposes than the pure graphics ones was aimed. From that moment on, new programming languages, frameworks and extensions were created with the sole goal of offering the programmer the possibility of using the GPU (Graphics Processing Unit) for solving general purpose high complexity problems. This is encouraged by the GPU's highly parallelized architecture, the processor itself embedding hundreds or even thousands of cores. In addition, fast memories were placed on the video card, close to the GPU, allowing ultrafast communication between those two. That was the moment a new concept was born: GPGPU – General Purpose Graphic Processing Unit which is nothing but a GPU with processing capabilities that can be exploited for non-graphic applications.

Among the programming languages/frameworks allowing GPGPU programming one may notice: CUDA (offered by NVidia in 2007), OpenCL (2008) and C++ AMP (released by Microsoft at mid 2012).

This paper presents an original optimized implementation of AES (Advanced Encryption Standard) on two different GPGPU platforms and a CPU platform. Due to its' functionality and the way data is used, AES algorithm is suitable for a parallel implementation so the obtained results are relevant.

By using the "young" C++ AMP extension dedicated to oriented parallel programming, a high portability was aimed. Unlike similar studies based on CUDA (Luken et al, 2009),

the developed application can be used on any GPGPU platform under various versions of Microsoft Windows, without considerable modifications or even no modifications at all. Other authors tried to achieve platform independence by using FPGA (Hoang and Nguyen, 2012).

The rest of paper is organized as follows. Section 2 describes related work focusing on AES implementations on CUDA but also investigates some existing alternatives. Section 3 presents implementation details and optimizations brought to AES. In Section 4, actual results and performance evaluation tests are presented. Conclusions and ideas for future developments and improvements are presented in Section 5.

2. RELATED WORK

The popularity and potential of GPGPUs dramatically increased as soon as new programming languages/frameworks sustained programmers' efforts to develop complex, parallel applications.

From 2007, GPGPUs have been used for developing complex applications for various fields where parallelism is fitted: image processing (Morar et al., 2012), biology (Benso et al., 2010), medicine (Morar et al., 2012), data compression, encryption and others. The common characteristic of these fields is that data can be segmented and independently processed and partial results can be aggregated at the end. Results prove that, in certain conditions, GPGPU offers far better performance than the CPU and rival more complex parallel processing structures, such as computer clusters or supercomputers, at infinite lower costs.

The first initiative was represented by CUDA (Compute Unified Device Architecture) (Nvidia, consulted 2014) offered by Nvidia for programming their proprietary GPGPUs. The core of CUDA is C/C++ to which specific APIs were added. Basically, CUDA offers the possibility to use the GPGPU but writing code for the CPU is possible within the same application. This way, the programmer has the possibility of using both CPU and GPGPU according to the application's demands.

There are, however, some severe drawbacks: the developed applications are platform dependent, meaning they can not be executed on different types of GPGPUs. Especially when it comes to unknown hardware configurations, this imposes severe limitations which can result in partial or total incapacity of running the application.

An alternative for CUDA is represented by OCL (Open Computing Language, OpenCL). OCL is a framework that allows parallel implementations for various processing units such as: CPUs, GPGPUs, DSPs (Digital Signal Processor), FPGAs (Field Programmable Gate Array) and so on. Initially OCL (OpenCL, 2014) was supported by Apple and was presented to other companies with the purpose of implementing the specification on the proprietary hardware. Soon, a working group named Khronos was born. Intel, AMD, Nvidia, Sun and others joined Apple and started to offer OCL support. Unlike CUDA, in case of OCL the portability issue was addressed.

Recent literature presents several implementations of AES on CUDA platforms. In (Luken et al., 2009), CPU and CUDA implementations of AES and DES algorithms are depicted. The results reveal the fact that GPGPU implementation surpasses the CPU implementation as soon as the data to be encrypted is larger than 100 KB. For even bigger data, the AES GPGPU implementation was about 3.5 time faster than the CPU implementation while the DES GPGPU one was about 4.5 times faster than DES CPU. In (Iwai et al., 2010) authors report a performance gain of 10 times when running the CUDA based AES encryption on NVIDIA GeForce GTX285. A similar study (Manavski, 2007). reports a performance gain of almost 20 times in favor of NVIDIA GeForce 880 GTX as opposed to an Intel Pentium IV CPU. However, the result was obtained only for a specific length of the input data (8 MB) and, also, the investigated range of input data volume was rather short (2KB to 8 MB).

Several studies approached the GPGPU based AES encryption using OpenCL. In (Gervasi, Russo and Vella, 2010) a comparative study regarding the AES implementation on a multicore CPU and GPGPU using OpenCL is reported. As authors conclude, the GPGPU OpenCL implementation was superior to CPU sequential and parallel similar implementations. The study was conducted over an Intel CPU and two different GPGPUs: one from AMD (AMD Firestream 9270) and one from Nvidia (Nvidia GeForce 8600 GT).

In (Xingliang et al., 2011) the authors present an AES implementation based on OpenCL as opposed to a similar implementation based on CUDA. Actual result show the OpenCL implementation is superior to CPU serial and parallel programming but a little slower than CUDA implementation. However, the study concludes that a loss in performance is acceptable if the implementation exhibits better portability.

In this paper, AES implementation based on C++AMP is presented. The comparative study is focused on a multi-core CPU and two different GPGPU platforms. Portability as well as performance improvements are aimed. Details will be provided in the following sections.

3. IMPLEMENTATION DETAILS

3.1 C++ AMP

C++ Accelerated Massive Parallelism (or, shortly, C++ AMP) is a technology developed and maintained by Microsoft that exploits parallel hardware architecture (CPU, GPU, etc.) for accelerating applications written in C++. The AMP extension of C++ has an open specification, meaning it is created and maintained by Microsoft but it is available for free to all interested programmers and, even more, anybody can bring optimization proposals.

C++ AMP was released in mid 2012 which makes it one of the youngest development tools. It was included in Microsoft Visual Studio 2012 and it is based on DirectX11, therefore all hardware devices which run C++ AMP code must support DirectX11 framework. With very few exceptions, most of the devices of interest fulfil this demand. If C++AMP code will run on Window 7 (or newer) and the computer does not have a proper video card, the application may still be executed on a device named Microsoft Basic Render Driver or Warp (Windows Advanced Rasterization Platform) which is nothing but an emulated device on the CPU based on SIMD (Single Instruction Multiple Data) instructions.

One of the strongest points of C++AMP is represented by the possibility of defining and using templates which allows the design and use of a highly reusable code. Another strong point is represented by the native support for developing parallel applications. There are, however, some drawbacks: the inexistence of new and delete operators, the impossibility of converting pointers or the inexistence of throw-try-catch. A programming difficulty is given by the restriction in using pointers: the code running on CPU can only access RAM memory while the code running on GPGPU can only access the video memory.

The major advantage of a C++ AMP implementation is given by its portability. In this study, the same application was run over different hardware processing platforms without the need of re-writing the code. Although, at this point, a minor restriction is given by the operating system which is limited to Microsoft Windows (regardless the version), at the beginning of September 2014 Microsoft and AMD announced the release of a C++ AMP compiler with Linux Support. For this reason, we appreciate there is an even lower portability limit when it comes to application developed with C++ AMP.

3.2 AES

AES (Advanced Encryption Standard) is an encryption/decryption standard defined by the US governmental institutions. The standard is described in the Federal Information Processing Standard 197 (AES, 2001) and was requested due to the security issues identified in DES (Data Encryption Standard) and the unacceptable speed of 3DES. The replacement for DES had to support: symmetric encryption, various size encryption keys (128, 192 and 256 bits) and, maybe most important, both hardware and software implementation.

There were 50 candidates for the initial selection but only 5 of them qualified for the final turn. Based on its strong security, efficiency, performance and ease of implementation, Rijndael was declared winner and officially became AES. Details about all 5 finalists can be found in (Ichikawa et al., 2000; Schneier and Whiting, 2000).

AES is based on repetitive calls of 4 well defined functions: SubBytes(), ShiftRows(), MixColumns() and AddRoundKey(). The first three are dedicated to confusion and diffusion, two operations that aim to prevent breaking the encryption by cryptanalysis means. The fourth is in charge with effective encryption. In other words, SubBytes() scrambles the bits of each byte, ShiftRows() scrambles each row, MixColumns() scrambles each column and AddRoundKey() encrypts the data. Full details of how AES works are presented in (AES, 2001).

Encryption algorithms can choose one of the three most popular (Dworkin, 2001) operation modes:

1) *ECB (Electronic Codebook)* – this is the easiest operation mode. Each data block is individually encrypted using the same encryption key, as presented in Figure 1. The main problem is that identical blocks will generate, after encryption, identical results which make cryptanalysis process easier. However, the probability of having a considerable amount of identical blocks of data is low and, furthermore, this mode is one of the few that can be parallelized. This is why it was chosen for this study.

2) *CBC (Cipher block chaining), PCBC (Propagating cipher-block chaining (PCBC), Cipher feedback (CFB), Output feedback (OFB)* – these modes are sequential dependent since the current block encryption can be made only if all previous blocks were encrypted. Although all offer better confidentiality, they are not proned to parallel implementation. This mode requires a key which will only be used for encrypting the first block of the message as one can see in Figure 2.

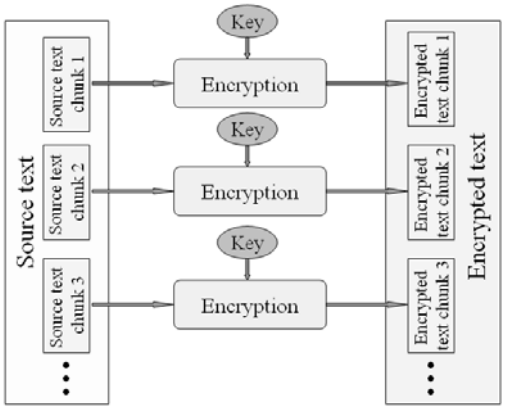


Fig. 1. ECB mode.

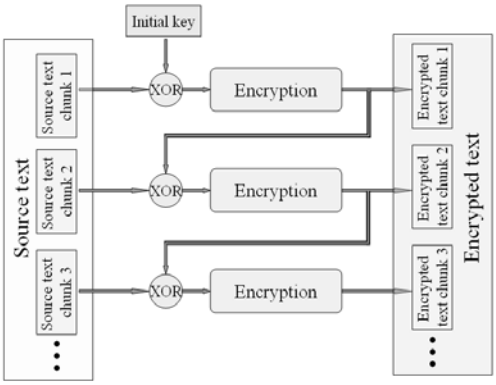


Fig. 2. CBC mode.

3) *CTR (Counter)/ICM (Integer counter mode)* – this mode is using a starting value for encryption process. This value can be obtained from the beneficiary of the encrypted data which aims to receive safe data through a public network or channel. Since blocks of data are independently encrypted, this mode can also be parallelized. In addition to what ECB performs, the encryption process modifies the initial key for each block of data according to a known procedure, usually an incremental one as depicted in Figure 3. This way, identical blocks of data will no longer generate identical results after encryption. The method used for this study can be easily modified in order to implement this operation mode.

Giving the fact that AES was designed before the "modern parallel era" clearly it natively works as a sequential algorithm. This is why it's main drawback is the encryption speed which is considered rather low. As Vincent Rijman himself stated, performance comes with a lack of speed.

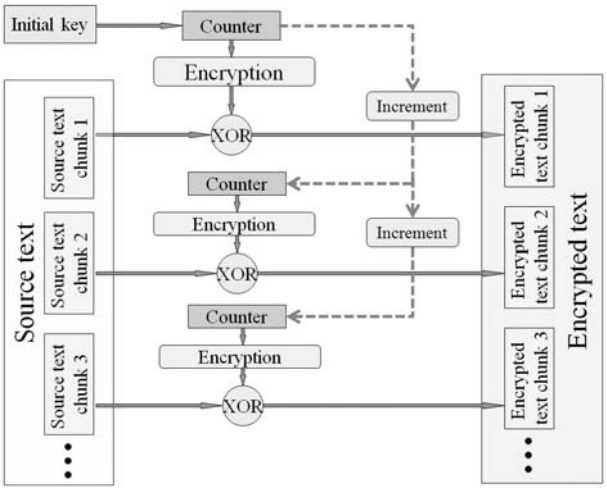


Fig. 3. CTR mode.

3.3 Actual implementation

The main objective of this study was to use a software platform in order to develop a parallel application exploiting different types of processing devices. Secondary objectives involved optimizations of AES implementation and increase its' portability over heterogeneous hardware platforms.

The low data dependency in AES algorithm allows us to consider that a parallel implementation is possible and, furthermore, to expect superior results as opposed to a sequential implementation. This, by itself, can be considered an optimization to classic AES since low speed is considered the major problem of the original AES. Previous implementations based on CUDA (Manavski, 2007) seem to confirm our hypothesis.

The AES implementation presented in this study is based on *The Design of Rijndael* (Daemen and Rijmen, 2002). Several parameters were eliminated, for example the variability of the data block and the encryption key. It is important to remember that the purpose of this study was not only AES optimization but the use of C++ AMP in an attempt to develop a fast, cross-platform project.

After a first, basic implementation, several questions were obvious: what happens if we deal with large data, is there a limitation regarding the number of active threads? And what happens if we deal with small data, is the GPGPU parallel implementation always better than a CPU parallel or even sequential implementation ? The results section will provide the answers to these questions.

The original sequential encryption procedure from (Daemen and Rijmen, 2002) was implemented and adapted for GPGPU as follows:

1) Current thread is identified and data to be processed is located within it. The linear data is copied into a local matrix in order to maintain the conformance with the encryption algorithm which works with a 4x4 *unsigned int* matrix unlike *unsigned char* that is described in (Daemen and Rijmen, 2002).

2) Due to programming language restrictions, all data types were converted from *unsigned char* to *unsigned int*.

3) Globally declared structures embed the encryption key and are available for the GPGPU specific code which is executed by threads.

Implementing a parallel version of AES using C++ AMP may seem both easy and fast. This is partially true because, so far, modularity issues were not discussed and the "of the box" algorithm was not tested in difficult conditions.

One of the biggest problems is represented by the inexistence of *char (byte)* type in C++ AMP. This may seem a little surprising since many applications need access at byte level. Graphics processing applications are just one example of such applications. As a consequence, the data was transmitted to the GPGPU as *unsigned int* and a procedure to implement bit operations for extracting byte values from the original data array was developed starting from Gregoy and Miller (2012). A major problem was given by the fact that *char* is represented on 8 bits and the *unsigned int* is represented on 32 bits therefore it is imperative to work with groups of 4 bytes (32 bits). Moreover, since the encryption block has 128 bits (16 bytes), the input data must be a multiple of 16. In our approach, if the above condition is not met missing values are filled with zeros.

The parallelization aims to encrypt each data block of 128 bits on a different thread. Having the input data in a linear form, as an array, data to be processed by each thread must be brought to a matrix form as mentioned above. The start of corresponding segment for each thread is obtained by multiplying the thread order by 4. Extraction of data based on this procedure revealed the fact that bytes are stored, in memory, in reverse order. This is explained by the *little endian* format (IBM, consulted 2014) in which a 4 bytes integer is represented by some processing units. Having this in mind, for every *unsigned int* value, bit operations and masks were used to extract data in reverse order as depicted

in Figure 4. The entire input text is transferred to video RAM before encryption and, similarly, the encrypted text is transferred to system's RAM at the end.

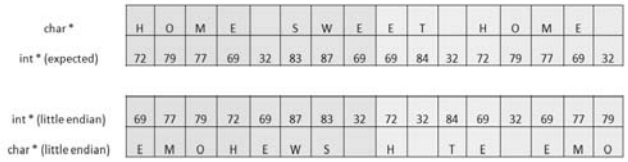


Fig. 4. Extracting data from Little endian format.

Another problem that had to be solved was given by the TDR (Timeout Detection & Recovery) function of Windows Operating System (earlier than Vista) that prevents exhaustive use of the GPGPU. The utility of this function is obvious when a general application makes excessive use of the graphic processor, sometimes even preventing a simple image to be displayed. The solution offered by TDR consists in resetting video cards if they fail to respond to a request within 2 seconds.

Since this study aimed to fully exploit the GPGPU processing power in order to improve the AES algorithm without any video requirements, the TDR function limited our initiative. More precisely, as soon as the input data exceeded 200 MB, the TDR stopped the encryption by resetting the video card.

Although the TDR function can be disabled by modifying several keys in the Operating System's registry, this was not considered a valid option since it reduces the portability of the application (Other systems may not have the TDR disabled and administrator privileges are needed to do that). In order to avoid TDR, in our approach the data which is encrypted by the GPGU was, initially, divided into segments of 50MB, each segment being transferred alone from RAM to GPGPU memory. The 50MB limit was chosen in order to allow the application to run on any GPGPU regardless the age, speed or memory.

However, for later generations GPGPUs, this limit may prove to be too small therefore it may induce unwanted delays due to the sub-optimal degree of data fragmentation. This is why this limit should be maximized according to the individual performance and capabilities of each GPGPU the application will run on. This would have a positive impact over the data fragmentation and reconstruction procedures, thus the overall performance of the application.

In order to take full advantage of the system's specific GPGPU, a dynamic procedure to determine the optimum data segment size that is passed at once to the GPGPU to be encrypted was designed and implemented. Answers to several questions had to be found first: how to accurately measure execution time on the GPGPU, how to estimate the optimum volume of data that a GPPGU can process before TDR resets the video card and, finally, check if the procedure must be applied for all chunks of data or only when the program is started.

According to (Gregoy and Miller, 2012), chapter 7, *Optimization*, the peak performance can be achieved only if data copy procedure to video RAM is completely separated from data processing itself. The possibility of using system's RAM directly by the GPGPU was investigated in order to determine if the copying procedure of data to video RAM was absolutely necessary. Although system's RAM can be accessed by the GPGPU using C++AMP, this approach did not lead to any improvements. There are several reasons that can explain this result: graphic memory bandwidth is bigger than RAM's while the latency is smaller. In addition, the PCI bus itself introduces some delays in data exchange. Under these circumstances, this direction was dropped.

The execution time can be accurately determined by using a method presented in (Gregoy and Miller, 2012). A system call activated before and after the processing on the GPGPU is used to determine the exact elapsed time between those calls.

As mentioned earlier, a testing procedure was implemented in order to determine the optimum (maximum) volume of data that a GPPGU can process before TDR is activated. An initial value is set as a reference. In our case, this value is set to 50 MB for the reasons mentioned above. After the kernel call on GPGPU, we check if the processed data is as big as expected according to the reference. If so, the elapsed time is determined and, by a simple proportional rule, the maximum amount of data that can be processed within 1.8 seconds is calculated. The interval of 1.8 seconds represents 90% of the TDR interval (2 seconds). An error of 10% (0.2 seconds) was considered in this case. If the system is used only for data encryption, as it was the case of this study, the determined value is relevant so it can be used for future use. This is why the value is stored in a local file becoming the new reference. In case multiple GPGPUs are present, multiple values will be stored and used.

This procedure can be applied each time an encryption is required so, regardless the GPGPU generation or capabilities, the application will run at its peak performance. However, its' efficiency is high if there is a big volume of data to be encrypted and the procedure is executed only once, at the beginning. Preferably, the stored reference value should be used but this approach is fine only if the GPGPU suffers no changes in load between encryption requests. If the GPGPU gets busy (processing video or graphic information, for instance) the reference value is no longer relevant. This situation was not investigated in this study.

4. PERFORMANCE EVALUATION AND RESULTS

The first testing scenario was based on the following system:

- CPU Intel Core i7 960 @2GHz, 4 physical cores, 8 virtual processors
- RAM 8GB, DDR3, 1066MHz
- GPGPU Nvidia GeForce GTX 480, 1482 MB DDR3, 448 cores

The second testing scenario was based on the following system:

- CPU Intel i5-4200M @ 2.5Ghz, 2 physical cores, 4 virtual processors
- 8GB Ram DDR3 1600Mhz
- AMD Radeon HD 8750M, 2048MB DDR3, 384 cores

The other hardware components are not relevant since all test data was generated in RAM in order to avoid any delays given by slow components. As one can observe, the testing hardware configurations are not even close in performance. For this reason, results will be presented separately. However, it must be pointed that the study did not aim to compare two different GPGPU platforms. Instead, one of the main goals was to achieve a real portability for different GPGPU platforms.

Given the architecture differences between the GPGPU and the CPU, in case of the latter, a load balancing procedure was implemented for a fair distribution of the processing effort to all available cores.

In order to provide a relevant evaluation, a reference had to be set. In our case, the reference is represented by two implementations of AES algorithm on CPU, the first purely sequential and the second, parallel. To eliminate any hazard 10 different tests were made and the results were averaged.

Table 2. Test results for the first hardware system.

	CPU S ms	CPU P ms	GPU ms
1KB	0.1	0.8	23.5
2KB	0.2	0.8	22.4
4KB	0.4	0.9	21.5
8KB	0.7	0.9	22.1
16KB	1.4	1	21.7
32KB	2.8	1.3	21.5
64KB	5.6	2.3	21.6
128KB	11.1	3.2	23.1
256KB	22.2	6.1	22.2
512KB	44.4	11.4	22
1MB	88.7	22.7	24.2
2MB	177.5	46.2	26.9
4MB	348.3	79.7	32.4
8MB	695	166.3	41.5
16MB	1386.6	319.4	56.2
32MB	2761.7	631.1	85.6
64MB	5514.9	1261.3	146.6
128MB	11018.9	2520.1	261.2
256MB	22053.6	5049.5	500.3
384MB	33083.9	7554.8	746.8
512MB	44136.8	10073.1	982.9
640MB	55009.2	12580.8	1212.2
768MB	66086.7	15101.1	1446.2
896MB	77290.8	17628	1739.3
1024MB	88288.1	20204.7	1972

Table 2 presents the data collected after running the application on the first testing system. CPU S and CPU P stand for CPU Sequential and CPU Parallel implementation while GPU stands for Nvidia's GPGPU. The testing data were chosen in order as many real case scenarios as possible. The maximum size, set to 1GB, was more than enough to reveal the differences between CPU an GPGPU behaviour. The first test may appear useless but, in fact, it had the purpose of initializing the GPGPU since a delay in reaction was observed for the first processing. More or less, it should be interpreted as a "wake up" signal for the GPGPU.

Data from Table 2 confirm some expectations but also reveal several interesting aspects. For small data (<8KB), sequential implementation on CPU outperforms the parallel CPU implementation (as shown in Figure 5) and, also, the GPU implementation.

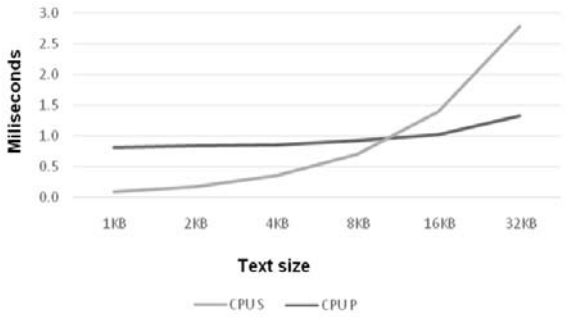
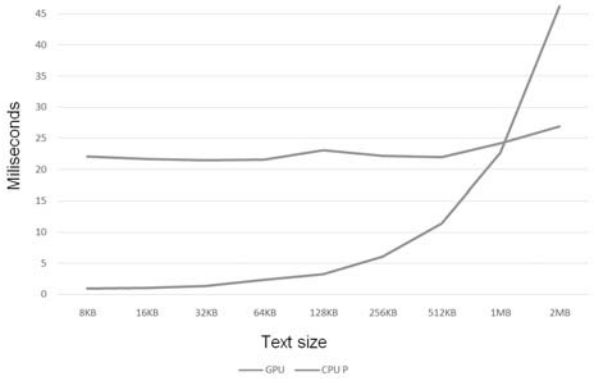
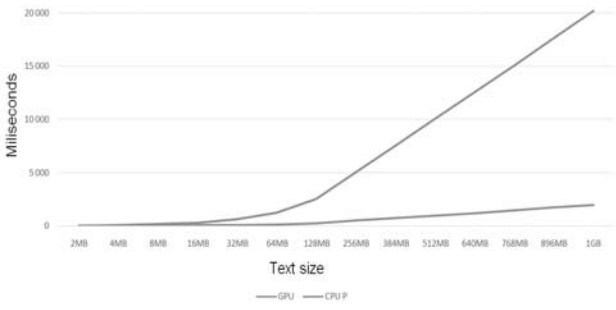


Fig. 5. CPU S vs. CPU P implementation.

As soon as input data exceeds 8KB, the CPU S exhibits worse results CPU P. After 256 KB, CPU S behaves worse than both its competitors. For a better observation of CPU P and GPU implementations, the CPU S will no longer appear in following two figures.



a.



b.

Fig. 6. CPU P vs. GPU implementation.

Figure 6a presents the encrypting times for data ranging from 8KB to 1MB. Within this interval, one can observe a better behaviour for the CPU P as opposed to GPU. Similar executing times were obtained for input data of 1 MB. After the crossing point observed at 1 MB of data, the GPU implementation clearly outperforms the parallel implementation on CPU (Figure 6b). A dramatic difference in performance is observed as soon as the input data exceeds 32 MB.

The overall behaviour of AES implementation on Nvidia's GPGPU as opposed to the sequential and parallel CPU implementations is revealed in Figure 7.

Performance ratio was defined as CPU processing time/GPU processing time. Perf S represents the performance ratio of GPU execution compared to CPU S execution while as Perf P represents the performance ratio of GPU execution compared to CPU P execution. For high volumes of input data (>128MB), figures reveal a Perf S of 45 and a Perf P of 10.

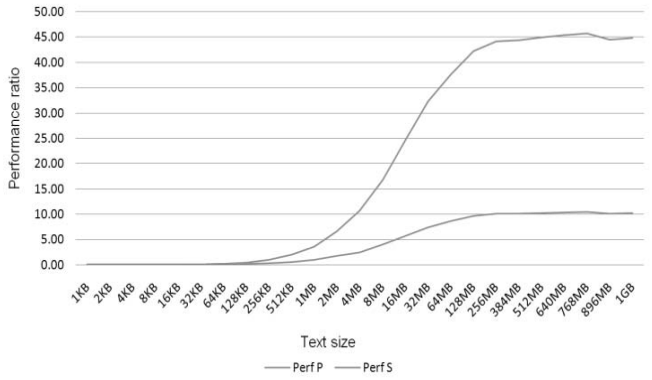


Fig.7. Performance ratio, first testing system

Table 3 presents the data collected after running the application on the second testing system. CPU-2 S and CPU-2 P stand for CPU Sequential and CPU Parallel implementation while GPU-AMD stands for AMD's GPGPU. The testing data were identical to those used in the first testing scenario. As previously stated, a direct comparison between different GPGPU platforms was not aimed. Instead, we consider that GPGPU's performance relative to system's CPU is relevant in all cases.

Similar behaviours were observed as in the first testing scenario. Figure 8 presents the performance of sequential and parallel implementations of AES on the second's system CPU. Measured execution times are slightly different (which was expected) but the trend is the same. The parallel implementation catches up the serial one as soon as input data exceeds 8 KB just like in the previous testing scenario.

In Figure 9a encrypting times for data ranging from 8KB to 8MB are presented. A better behaviour for the CPU-2 P as opposed to GPU-AMD can be observed but, in this case, similar executing times were obtained for input data of 8 MB. Comparing with Figure 6a, this behaviour can be explained by the lower performance of second GPGPU and higher performance of the second CPU. However, one can observe that trends are similar. Just like in the previous test, GPU-AMD's performance dramatically increases for large input data, this time bigger than 32 MB (Figure 9b).

Table 3. Test results for the second hardware system

	CPU-2 S ms	CPU-2 P ms	GPU-AMD ms
1KB	0.0001	0.0004	0.1827
2KB	0.0001	0.0005	0.1962
4KB	0.0003	0.0005	0.1809
8KB	0.0006	0.0006	0.1845
16KB	0.0013	0.0010	0.2064
32KB	0.0025	0.0017	0.1786
64KB	0.0048	0.0027	0.1767
128KB	0.0093	0.0048	0.1747
256KB	0.0191	0.0093	0.1738
512KB	0.0369	0.0176	0.1701
1MB	0.0766	0.0571	0.1856
2MB	0.1526	0.0681	0.1820
4MB	0.2971	0.1363	0.1998
8MB	0.6045	0.2690	0.2212
16MB	1.2020	0.5388	0.2488
32MB	2.4047	1.1077	0.2854
64MB	4.8478	2.2115	0.3760
128MB	9.7037	4.3964	0.6431
256MB	18.9783	8.7755	1.0113
384MB	27.1826	13.1759	1.4071
512MB	36.6942	17.2230	1.8321
640MB	49.4872	21.4604	2.4205
768MB	53.9524	25.8264	2.6469
896MB	62.9661	30.0756	3.1884
1024MB	71.8683	34.3292	3.4801

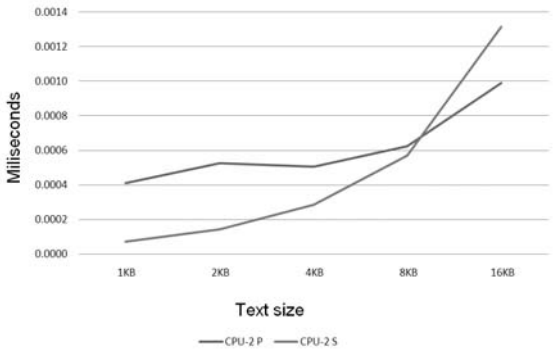
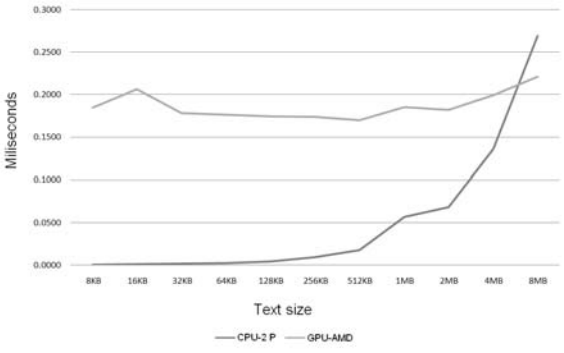
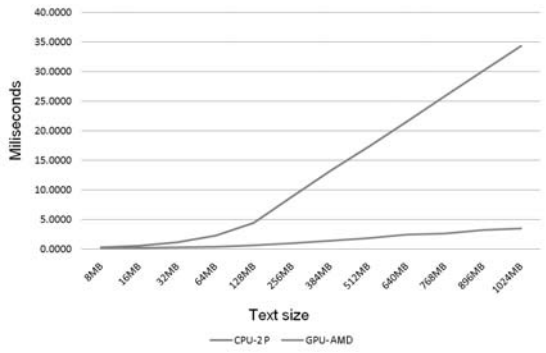


Fig. 8. CPU-2 S vs. CPU-2 P implementation.



a.



b.

Fig. 9. CPU-2 P vs. GPU-AMD implementation.

Performance ratio was determined using the same criteria as in the previous configuration and results are presented in Figure 10. Perf-2 S and Perf-2 P exhibit the same trend as in Nvidia's GPGPU case.

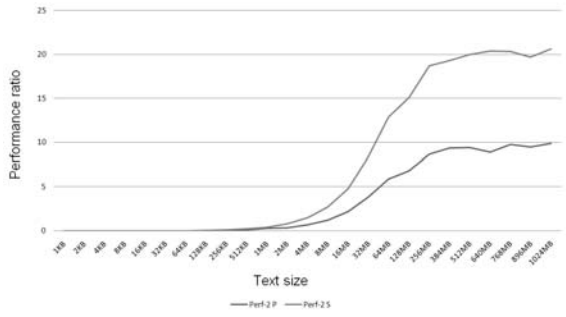


Fig. 10. Performance ratio, second testing system.

However, if the Perf-2 P is very close to Perf P from the first scenario, one can observe that Perf-2 S is considerably lower than Perf S (20 as opposed to 45).

5. CONCLUSIONS AND FUTURE WORK

In this study, an optimized version of AES was implemented on GPGPU platforms from different manufacturers using C++ AMP. The original AES was slightly modified for parallel implementation. For each hardware testing platform, CPU based sequential and parallel versions were tested and used as references.

Previous studies covering various domains report differences in execution times of CPU as opposed to GPGPU up to hundreds of times (Govindaraju et al., 2008; Mocanu et al., 2014). In their controversial paper (Lee et al, 2010) the authors report smaller differences of only 2 or 3 times. For input data bigger than 128MB, our study reveals actual differences ranging from 20 and 45 times in case of non-optimized GPGPU implementations vs. sequential CPU implementations. A performance ratio of 10 in favor of GPGPU implementation was observed when compared to a parallel CPU implementation. For input data within 2MB and 128 MB, the GPPGU still outperforms the CPU but the performance ratio is smaller, ranging from 1.7 to 10. The results are very close to similar studies where AES was implemented on GPGPU using different frameworks.

Even if the results are not the same for all testing systems, the most important conclusion is that all systems exhibit the same trend. It is safe to assume that collected and presented data are relevant for any present system based on components from the same generation.

Another very important aspect is related to the application's portability. In fact, as stated from the beginning, this was one of the major goals of this study since the literature does not report similar achievements. As presented, the application was tested over two GPGPUs from different manufacturers and over two different CPUs. The only request is that DirectX 11 (Microsoft DirectX, consulted 2014) should be supported by the video hardware. This is a very lax requirement since all major video cards manufacturers offer this support for years now. For instance, Nvidia included support for DirectX 11 staring with September 2009 when GeForce GT430 was released. In the same month, ATI (bought by AMD) released Radeon 5000 Series, also providing support for DirectX 11.

An important achievement of the study is represented by the procedure that determines the maximum amount of data that the GPGPU can process before TDR is activated. From the beginning, the procedure was designed to be adaptive, meaning it can work considering previous results and the present state (load) of the GPGPU. Since, in this study, the load of the GPGPU was constant (there were no other requests but the encryption) the procedure was called only once and the resulted value was used for the entire encryption process. However, in case of heavy loaded systems, the procedure will be called for every step of the encryption. This way, the optimum amount of data that can be processed at that moment by the GPGPU without being disturbed by the GPGPU will be determined.

Another direction that will be investigated is the improvement of AES algorithm. One of the improvements that can be easily achieved is replacing the ECB mode with CTR mode. In fact, the only difference consists in adding an incremental block to current implementation. This way, stronger encryption will be possible without considerable performance loss.

Since AES is a symmetric algorithm, decryption will be implemented with small efforts. Moreover, security and robustness of AES can be increased by adding all key lengths indicated by the standard (192b and 256b) which will be very important especially for very sensitive data.

In addition to the performance gain offered by the parallel architecture of the GPGPUs, algorithms' optimization will be addressed in our future studies. First step may be represented by the implementation of a dynamic selection and scheduling mechanism. A study dedicated to scheduling of heterogeneous processors is presented by Noureddine et al, (2008). In our case, the mechanism should choose between the CPU and GPGPU implementations based on several

parameters: size of input data, hardware configuration and performance and hardware availability.

REFERENCES

AES (2001), http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, consulted 2014

Benso, A., Di Carlo, S., Politano, G., Savino, A., and Scionti, A. (2010). GPU cards as a low cost solution for efficient and fast classification of high dimensional gene expression datasets, *Journal of Control Engineering and Applied Informatics*, ISSN 1454-8658, vol. 12, no.3, pp. 34-40

Daemen, J., and Rijmen, V. (2002). *The Design of Rijndael*. Springer, Heidelberg, Germany, ISBN: 3-540-42580-2

Dworkin, M. (2001). *Recommendations for Block Cipher Modes of Operation. Methods and techniques*, NIST Special Publication 800-38A, 2001 Edition, Washington, DC

Gervasi, O., Russo, D., and Vella, F. (2010). The AES implantation based on OpenCL for multi/many core architecture, *Proceedings of the 2010 International Conference on Computational Science and Its Applications*, 23-26 March, Fukuoka, Japan, pp. 129-134, ISBN: 978-0-7695-3999-7

Govindaraju, N.K., Lloyd, B., Dotsenko, Y., Smith, B., and Manferdelli, J. (2008). High performance discrete Fourier transforms on graphics processors, *IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*, 15-21 November, Austin, Texas, USA, pp.1-12, ISBN: 978-1-4244-2834-2

Gregoy, K., and Miller, A. (2012). *C++ AMP - Accelerated Massive Parallelism with Microsoft® Visual C++®*, Microsoft Press, 1 edition, September 25, ISBN-10: 0735664730

Hoang, T., and Nguyen, V.L. (2012). An Efficient FPGA Implementation of the Advanced Encryption Standard Algorithm, *IEEE International Conference on Computing and Communication Technologies, Research, Innovation and Vision for the Future (RIVF),* Feb. 27 2012-March 1 2012, Ho Chi Minh City, Vietnam, pp. 1-4, ISBN: 978-1-4673-0307-1

IBM, http://www.ibm.com/developerworks/ aix/library/au-endianc/index.html?ca=drs-, consulted 2014

Ichikawa, T., Kasuya, T., and Matsui, M. (2000). Hardware Evaluation of the AES Finalists, *The third AES Candidate Conference*, 13-14 April, New York, USA, pp. 279-285

Iwai, K., Kurokawa, T., and Nishikawa, N. (2010). AES encryption implementation on CUDA GPU and its analysis, *Proceedings of 2010 First International Conference on Networking and Computing*, 17-19 Nov., Higashi-Hiroshima, pp. 209-214, ISBN 978-1-4244-8918-3

Lee, V., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. (2010). Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *Proceedings of 37th International Symposium on Computer Architecture (ISCA'10)*, June 19-23, Saint-Melo, France, pp. 451-460, ISBN: 978-1-4503-0053-7

Luken, B., Ouyang, M., and Desoky, A.H. (2009). AES and DES Encryption with GPU, *Proceedings of the ISCA 22nd International Conference on Parallel and Distributed Computing and Communication Systems (PDCCS 2009)*, September 24-26, 2009, Louisville, Kentucky USA, pp. 67-70

Manavski, S.A. (2007). Cuda compatible GPU as an efficient hardware accelerator for AES cryptography. *IEEE International Conference on Signal Processing and Communications (ICSPC 2007)*, 24-27 November 2007, Dubai, United Arab Emirates, pp. 65-68, ISBN: 978-1-4244-1235-8

Microsoft DirectX - https://msdn.microsoft.com/en-us/library/windows/desktop/ee663275%28v=vs.85%29.aspx consulted 2014

Mocanu, Ş., Din, R., Saru, D., and Popa, C. (2014). Using Graphics Processing Units for Accelerated Information Retrieval, *Studies in Informatics and Control*, ISSN 1220-1766, vol. 23 (3), pp. 249-257, 2014.

Morar, A., Moldoveanu, F., Asavei, V., Moldoveanu, A., and Egner, A. (2012). Multi-GPGPU Based Medical Image Processing in Hip Replacement, *Journal of Control Engineering and Applied Informatics*, ISSN 1454-8658, vol. 14, no.3, pp. 25-34

Noureddine, L., Yahia, H., and Borne, P. (2008). Multi-objective Scheduling onto Heterogeneous Processors System Using Ant System & Fuzzy Logic Controller, *Studies in Informatics and Control*, vol. 17 (1), pp. 95-106, 2008, ISSN 1220-1766

Nvidia CUDA ZONE, https://developer.nvidia.com/cuda-zone, 2014

OpenCL, https://www.khronos.org/opencl, 2014

Schneier, B., and Whiting, D. (2000). A Performance Comparison of the Five AES Finalists, *The third AES Candidate Conference*, 13-14 April, New York, USA, pp. 123-135

Xingliang, W., Li, X., Zou, M. and Zhou, J. (2011). AES finalists implementation for GPU and multi-core CPU based on OpenCL, *IEEE International Conference on Anti-Counterfeiting, Security and Identification (ASID)*, 24-26 June, Xiamen, China, pp. 38-42, ISBN: 978-1-61284-631-6