EAFIT University

Final semester project

CUDA capable GPU as an efficient co-processor

Author:
Julian Ortega
jortegac@gmail.com

Advisors:
Christian Trefftz
Dr. Helmuth Trefftz

Medellin - Antioquia - Colombia
November 26, 2010

Abstract

This paper presents a study of implementing the AES block cipher cryptographic algorithm making use of a GPU with the CUDA platform by Nvidia, in order to demonstrate its general applicability as a co-processor in the above task, making a comparative analysis to single core CPUs and multi-core CPUs. Advanced Encryption Standard or AES is a symmetric-key encryption standard adopted by the U.S. government [NIS01] which makes it widely spread and documented. Two different machines were used for testing and comparing. Machine 1 used an Intel Core i5 M 450 @ 2.4 ghz and a Nvidia Geforce GT 330M. Machine 2 used an Intel Core i7 960 @ 3.2 ghz and a Nvidia Quadro FX 1800. The developed implementations showed a maximum speedup on the GPUs, over the sequential executions on the CPUs, of `10.70485x` on machine 1 and a maximum speedup of `9.85299x` on machine 2, were achieved for total execution times. Regarding encryption times, maximum speedups on the GPUs, over sequential execution on the CPUs, of `19.08824x` and `18.31696x` were achieved, for machine 1 and 2, respectively.

Keywords: AES, CUDA, GPU, GPGPU, Cryptography

Contents

1 Introduction

Graphics Processing Units or GPUs have been the subject of extensive research and numerous advances in recent years. Nowadays, thanks to recent efforts of GPU manufacturing companies, they have been successfully used for general purpose applications that are outside the domain of graphics.

General-Purpose computation on Graphics Processing Units or GPGPU, is a field in which high performance Graphics Processing Units (GPU) with many processing cores are capable of very high data calculations and data transfers, according to [gpg10]. Once designed especially for computer graphics and difficult to program, today's GPUs are general purpose processors with support for accessible programming interfaces and industry standard languages such as C. Developers who port their applications to the GPU often achieve optimizations of various order of magnitude compared to the optimization achieved by implementations on the CPU.

Computer systems are going from performing the "central processing" in the CPU to perform "co-processing", processing shared between the CPU and GPU. To enable this new computational paradigm, NVIDIA has invented CUDA [Cor10f], a parallel computing architecture, which is available for GeForce, ION, Quadro and Tesla GPUs. This represents a considerable installed base for application developers, allowing the execution of CUDA capable application for a considerable amount of users, considering that a large amount of computer users have access to dedicated GPUs.

With the increasing use of computational power offered by the technologies related to graphics cards, this project is an effort to implement a cryptographic algorithm making use of a GPU, in order to demonstrate its general applicability as a co-processor in the above task, making a comparative analysis to single core CPUs and multi-core CPUs.

Rijndael [NIS01], best known as Advanced Encryption Standard, was chosen as block cipher cryptographic algorithm for this project, because the implementation is well documented and widespread. It should be noted that even though the AES works with keys of 128, 192 and 256 bits, in this project, only the 128 bit key will be used.

The remainder of this document is organized as follows: Section 2 describes some related work. Section 3 presents a theoretical framework of parallelization, CUDA and the AES algorithm, and describes the 3 different implementations created. In Section 4 performance analysis and experiment results are provided. It also describes some special considerations and information worth noting from the results. Section 5 contains the conclusions to this project. And Section 6 some desired future work.

2 Related work

2.1 CUDA compatible GPU as an efficient hardware accelerator for AES cryptography:

In [Man07] the performance of the AES algorithm using a Nvidia GeForce 8800 is compared with those of the sequential implementations running on an Intel Pentium IV 3.0 GHz CPU, achieving a `19.60x` speed-up for the portion of the AES executed on the GPU with an input size of 8MB and a `5.92x` speed-up on the total time, that includes the time needed for the download and read-back operations (that means copying data from the host memory to the GPU device and back).

The authors considered the speed-up achieved in the overall process of executing the algorithm, and also the speed-up of only the porting of the code that performs the encryption. A similar approach will be taken in this project where speed-up will be considered for the total time and the encryption time.

2.2 AES Encryption Implementation and Analysis on Commodity Graphics Processing Units:

In [HW07] the authors presented new approaches to solving AES block cipher encryption on pre G80 GPU hardware. They compared each approach's resulting performance to each other and to standard CPU implementations. They achieved rates of up to `870.8Mbits/s` using a Raster Operations Unit based approach and `313.84Mbits/s` using a fragment processor based XOR simulation on a GeForce 7900GT. They proceed to compare to some sources of AES performance figures for optimised implementations on standard CPUs and discover that the GPU approaches under perform. It was demonstrated that the GPU performs best using large packet sizes and thus suits applications which require bulk data encryption/decryption.

The approach taken here by the authors is significantly different from the approach in this project, the most appreciable characteristic being the usage of a GPU that is not CUDA-enabled.

2.3 Practical Symmetric Key Cryptography on Modern Graphics Hardware:

In [HW08] the authors present a new block based conventional implementation of AES implementation using CTR mode of operation on an Nvidia G80 GPU. This approach is shown with `4-10x` speed improvements over CPU implementations and `2-4x` speed increase over the previous fastest AES GPU implementation. The paper also establishes that the GPU is suitable for bulk data encryption and can also be employed in a general manner while still maintaining its performance in many circumstances for both parallel and serial modes of operation messages.

This project uses an AES implementation that relies on simplicity and does not include special modes of the AES, nonetheless, CTR mode of the AES is used by the authors, while improving performance of previous attempts at parallelizing the AES algorithm.

2.4 Design of a Parallel AES for Graphics Hardware using the CUDA framework

In [BBAP09] the authors propose an effective implementation of the AES-CTR symmetric cryptographic primitive using the CUDA framework. They provide quantitative data for different implementation choices and compare them with the common CPU-based OpenSSL implementation on a performance–cost basis. In order to make throughput comparisons, they chose Intel's Core 2 Quad Q6600 and Xeon Clovertown E5335 as two current highend processors and both AMD's Athlon 64x2 3800+ and Intel's Pentium D 540 as two low-end but widely deployed ones. With respect to previous works, they focus on optimizing the implementation for practical application scenarios, and they provide a throughput improvement of over 14 times, with a maximum of 12.4 Gb/s.

2.5 Serpent Encryption Algorithm Implementation on Compute Unified Device Architecture (CUDA)

In [NHA09] the authors present a methodology for the transformation of CPU-based implementation of Serpent encryption algorithm (in C language) on CUDA to take advantage of CUDA's parallel processing capability, achieving a throughput performance of up to `100MB/s` or more than `7X` performance gain by integration of multiple block encryption in parallel.

2.6 CUDA-based AES Parallelization with Fine-Tuned GPU Memory Utilization

In [MJJ10] the authors proposed an efficient approach to parallelize AES and fine-tune the memory utilization in GeForce 9200M GS. The influences of different memories, amount of blocks and number of threads in each block on performance are provided. Experimental results have shown impressive performance gains. The throughput of these experiments has reached the upper limit of bandwidth for GeForce 9200, which is 6.4GB/s.

They implement AES in three ways. The first sequential algorithm is implemented for a 32-bit CPU. The second version is implemented using Pthread: 4 CPU threads are used. The third implementation is based on the algorithm they propose.

2.7 A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA:

In [CBM+08], a comparison of the performance of CPU and GPU implementations of six naturally data-parallel applications was performed. The experiments used CUDA language and compared performance on an NVIDIA GeForce GTX 260 engineering sample with that on an Intel dual-core, hyperthreaded Xeon based system with OpenMP. The GPU implementations of their applications obtained impressive speedups and were able to offload work to the GPU responsible for `95.8 - 99.7%` of the applications' original, single-threaded execution time excluding disk I/O.

The approach seen is similar to the one of this project, where a comparison of single-threaded execution will be compared to a OpenMP-based implementation of the algorithm and then to a CUDA implementation, in order to determine speed-ups.

3 Approach

3.1 Theoretical framework

This section is a brief overview of some concepts required to better understand this project. First, a quick overview of what parallel computing means and its benefits. Next, a short look into the CUDA technology from NVIDIA. Last, an overlook at the AES algorithm.

3.1.1 Parallelization

[Bar10] states that parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs

Parallel computing differs from, traditional, serial computation [Bar10], given that the serial approach has the following characteristics:

- A problem runs on a single computer having a single Central Processing Unit (CPU)
- A problem is broken into a discrete series of instructions
- Instructions are executed one after another
- Only one instruction may execute at any moment in time

There are a number of reasons to explore parallel approaches to computational problems [Bar10]:

- Save time and/or money shortening the time it takes to complete a task by using more resources.

- Solve larger problems that are impractical or impossible to solve on a single computer.
- Provide concurrency, allowing multiple computing resources to do many things simultaneously.
- Use of non-local resources to perform computational tasks
- Limits to serial computing: Both physical and practical reasons pose significant constraints to build even faster serial computers.

3.1.2 CUDA

CUDA or Compute Unified Device Architecture is a co-designed hardware and software to expose the computational horsepower of NVIDIA GPUs for GPU computing [Cor08].

A CUDA capable GPU can be thought of as a massively-threaded co-processor, for which you write "kernel" functions that execute on the device – processing multiple data elements in parallel. GPUs are great for data parallelism, given that they have multiple ALUs, fast onboard memory and a high throughput on parallel tasks [TW08]. GPU threads are extremely lightweight. Thread creation and context switching are essentially free and the GPU expects 1000s of threads for full utilization [Cor08].

Sections 3.2.1 and 3.2.2 describe CPU applications that do the following: Allocate memory that will be used for the computation (variable declaration and allocation), read the data that we will compute on (input), specify the computation that will be performed , write to the appropriate device the results (output).

Since the GPU does not have any input/output devices, it is necessary to copy the input data from the memory in the host computer into the memory in the GPU, using previously allocated variables. Then, the code to be executed is specified and the results are copied back to the memory in the host computer. Figure 1 illustrates the process.

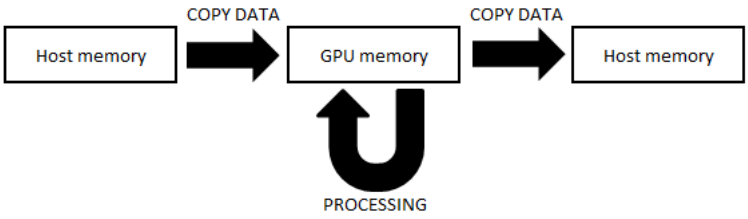

Figure 1: Overview of executing code in the GPU

3.1.3 AES

A block cipher is a symmetric key cipher operating on fixed-length groups of bits, called blocks, with an unvarying transformation. A block cipher encryption algorithm takes n-bit block of plaintext or unaltered text as input, and outputs a corresponding n-bit block of ciphertext or encrypted text. The exact transformation is controlled using a second input — the key, as shown in Figure 2.

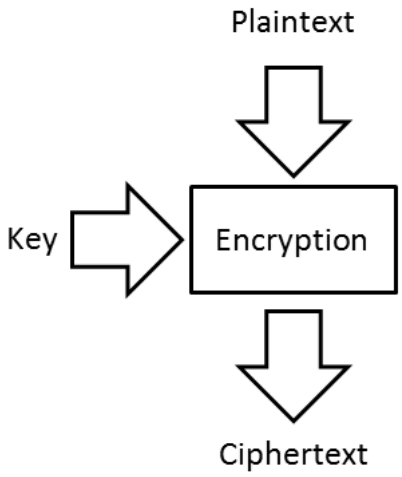

Figure 2: Overview of the AES - Taken from [EZ04]

The AES, or Advanced Encryption Standard, is based on the Rijndael algorithm developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, and it accepts a fixed block size of 128 bits or 16 bytes, usually organized in a 4 by 4 matrix of bytes, and a key size of 128, 192 or 256 bits. The type of key selected, determines the number of rounds to be performed by the algorithm. Each round performs certain transformations on the input data. 10, 12 and 14 rounds will be performed for a 128, 192 and 256 bit key, respectively. As stated in section 1, this project will use the AES algorithm and only a 128-bit key, thus 10 rounds in all the AES implementations.

There are 4 individual transformations - SubBytes, ShiftRows, MixColumns, and AddRoundKey, that are essential to the algorithm. AddRoundKey is XOR addition of a scheduled round key for mixing the state using the key). SubBytes is a byte substitution using an S-box which is a lookup table. ShiftRows is cyclical shift of bytes in each row. MixColumns is linear transformation which mixes column state data. Figure 3 shows an algebraic representation of the transformations. For detailed reference, visit [NIS01].

SubBytes	$b_{i,j} = S[a_{i,j}]$
ShiftRows	$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j-1} \\ b_{2,j-2} \\ b_{3,j-3} \end{bmatrix}$
MixColumns	$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}$
AddRoundKeys	$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$

Figure 3: AES transformations. Taken from [MJJ10]

The overall AES process is depicted in Figure 4. The input or plaintext 4 by 4 matrix of bytes, a total of 16 bytes of data. This block is copied into a state matrix, which is modified in each round of the algorithm. The first 9 rounds are composed by the same steps: SubBytes, ShiftRows, MixColumns and AddRoundkey. The MixColumns is not performed in the last round. After all the 10 rounds have been performed, the resulting state matrix is the ciphertext or encrypted data.

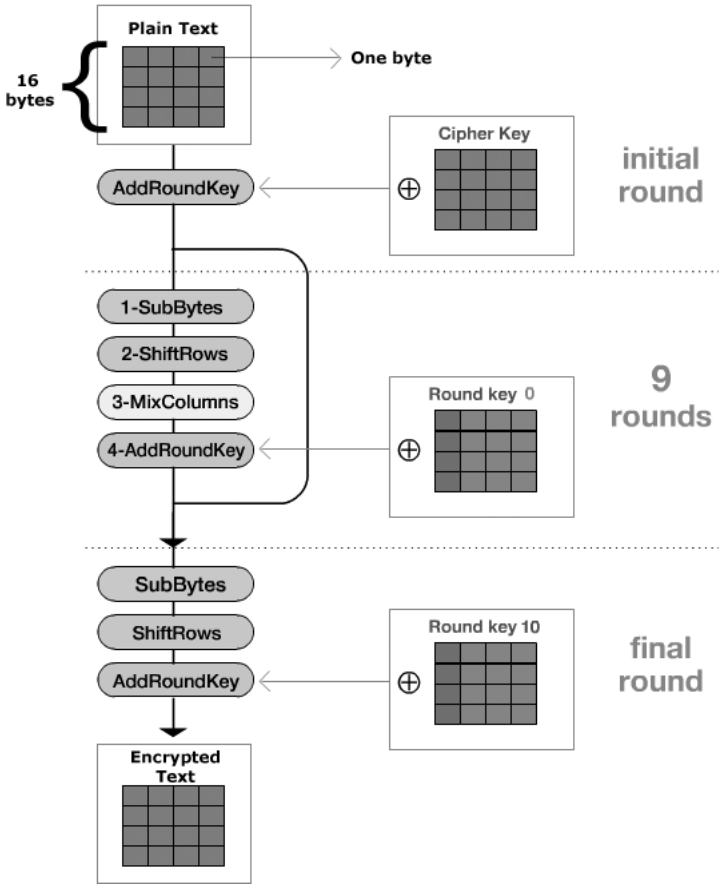

Figure 4: AES - Encryption process. Taken from [EZ04]

3.2 Implementation

This section goes through the process of the various implementations of the AES algotithm:

- Implementation for single threaded execution using the CPU
- Implementation for multi threaded execution using the CPU
- Implementation for CUDA using the GPU

3.2.1 Sequential execution using the CPU

This stage was rather simple, since no parallelization was required for the implementation of the algorithm. A basic implementation provided by [PK09], was used as base for this stage, since the code they provide focuses on simplicity and it was easy to modify and improve.

Major changes were made to the code, adding file reading and writing capabilities for the tasks such as the acquisition of the data to be encrypted from a file rather than from the user through the command line.

The code was further improved to work with manageable chunks of the input file, keeping in mind that files with large size might not fit into the RAM all at once. The chunk size was established at 256 KBs, meaning that the code will be able to encrypt any file which size is a multiple of 256 KBs. Table 1 shows how a 512KBs file is divided for processing. Here, the numbers indicate the position of the input file from which each byte came from.

0	1	2	. . .	262143	262144	262145	. . .	524288
Part 1					Part 2			

Table 1: File content distribution for processing

The first part is taken through the AES encryption process (See: Figure 4), one block at a time and then placed in the output file. The process is then repeated with the second part. Each block to be processed contains 16 bytes, so to cover the 256kBs, 4096 state blocks or matrixes are required. Figure 5 shows how part 1 from the example on table 1 is processed.

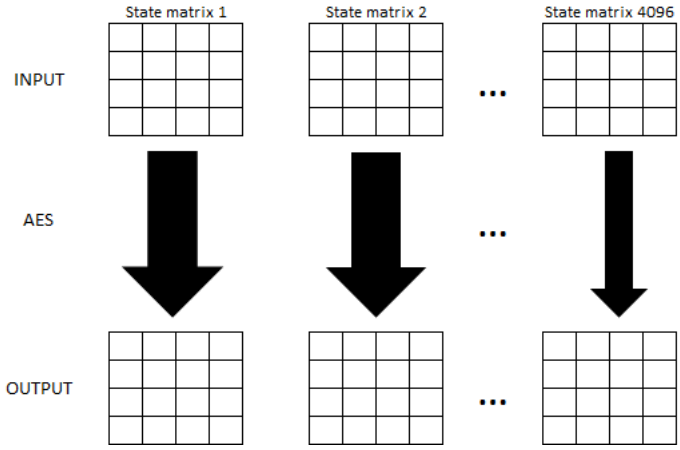

Figure 5: AES processing of 256kBs of data distributed in 4096 state matrixes.

3.2.2 Multi threaded execution using the CPU

For this stage, the code produced in section 3.2.1 was used as a base. The task of parallelizing was approached by making use of OpenMP.

OpenMP is an implementation of multithreading, in which a master thread subsequently creates slave threads and assigns different tasks for them to perform. The OpenMP API relies mostly on processor directives that indicate which sections of the code should be executed in parallel.

In the case of the AES implementation, the focus was on distributing the iterations of loops amongst the number of processors available in the system. It is important to mention that even if the API lets developers set as many threads as they want, the optimal amount of threads would be the one that is equal to the amount of processors in the system, one thread per processor, to avoid wasting valuable processing cycles while the processors switch between contexts and threads.

For instance, assuming a system that possesses 4 processors and a loop that iterates from 1 to 100, the distribution would be like the one shown in Figure 6.

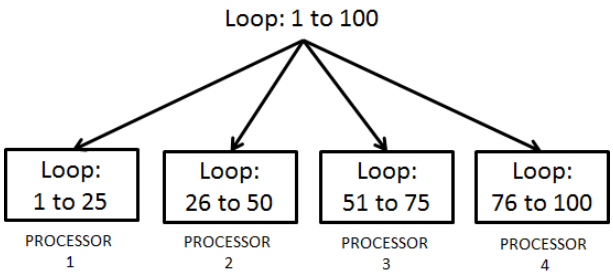

Figure 6: Loop distribution among 4 threads

OpenMP allows the distribution of iterations in a number of different ways: static, dynamic, guided, at runtime, and automatic. For the AES code in this stage, the option -auto- was chosen, so the amount of threads was decided by the compiler according to the resources available in the system.

The idea behind this, is to be able to process, as shown in Figure 5 in Section 3.2.1, any given chunk, faster, by distributing the task among the available processors. In the case of the example from Figure 6, 4 different threads could process one chunk at the same time, as displayed in Figure 7

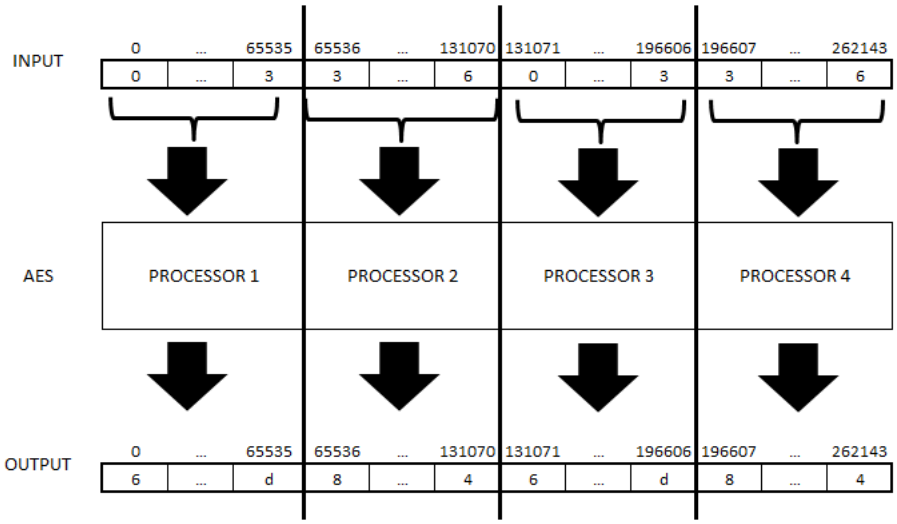

Figure 7: Distributing input chunks among 4 threads

3.2.3 CUDA using the GPU

When calling a "kernel" function (See Section 3.1.2) it is necessary to specify a grid size and a thread block size. The grid size indicates the amount of threads blocks to be assigned to the "kernel" the block size indicates the amount of threads for each block as displayed in Figure 8. Each thread runs the same code and has an ID that it uses to compute memory addresses and make control decisions as seen in Figure 9.

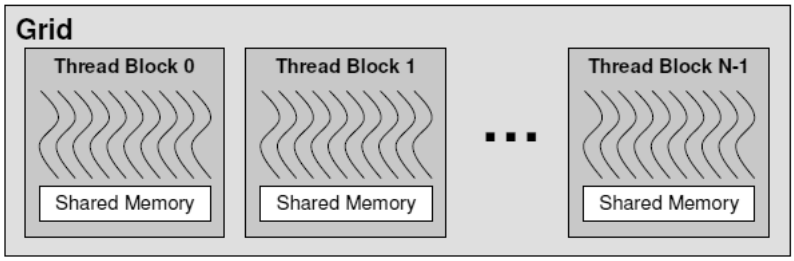

Figure 8: CUDA Grid. Taken from [Cor08]

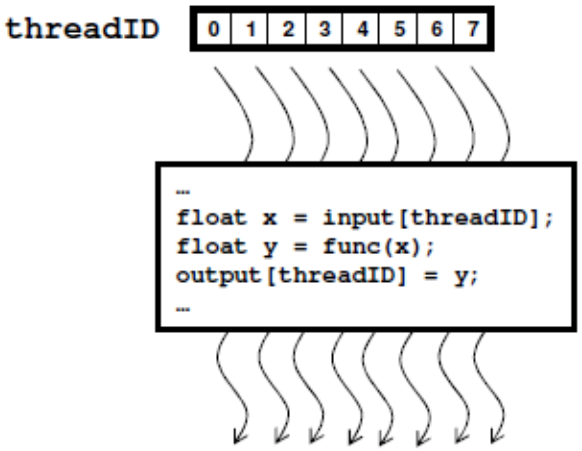

Figure 9: Threads in the GPU. Taken from [Cor08]

In a similar fashion to Section 3.2.1, this implementation also used the code produced in Section 3.2.2 as base. The code is executed using a chunk size of 256kBs, a block size of 256 and a computed grid size that is calculated by diving the chunk size by the number of threads per block, in this case 262144/256, which is equal to 1024 blocks in the grid. Each chunk from input file is copied into the host computer's memory and from there is moved to the GPU's memory, where is it processed. The result is copied back to the host computer's memory and from there it is stored in the output file.

When using CUDA it is often desired that each thread in the kernel to access a different element of an array. Each thread has access to three special variables:

- threadIdx.x - thread ID within block
- blockIdx.x - block ID within grid
- blockDim.x - number of threads per block

The special variables are often used in the manner displayed by Figure 10. In this project they are used to go through the chunks of the input file, performing the AES operations.

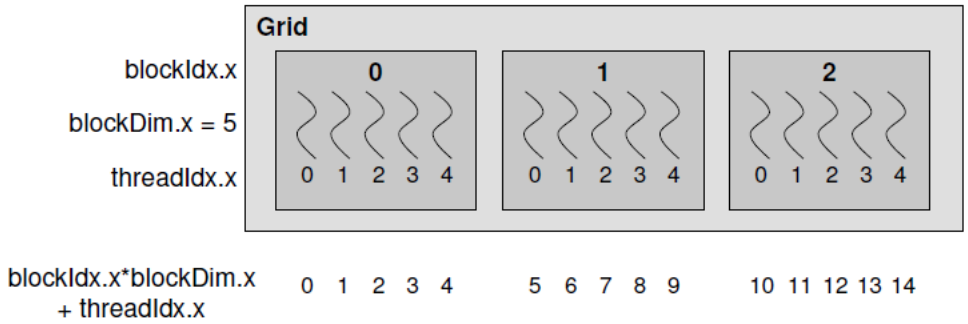

Figure 10: Data decomposition in the GPU. Taken from [Cor08]

4 Results

Input test files were generated with random content in sizes of 256kBs, 512kBs, 1MB, 2MBs, 4MBs, 8MBs, 16MBs, 32MBs, 64MBs, 128MBs and 256MBs. For each file, each implementation was run 3 times on each

machine. Each run outputs a total execution time and an encryption time. Total time is the amount of time the codes takes to run, that includes, reading from a file and writing to a file, copying to memory, performing encryption. The encryption time only takes into account the time it took the AES algorithm to run, with no regard for other actions. Notice that the `LARGE_INTEGER Union` (See [Cor10b]), which "represents a 64-bit signed integer value" was used to calculate time executions.

Tests were made on two machine configurations:

Machine 1:

- Processor: Intel Core i5 M 450 @ 2.4 ghz [Cor10a]. This processor has 2 physical cores and 4 threads.
- GPU: Geforce GT 330M[Cor10e]. Number of multiprocessors: 6. Number of CUDA cores: 48
- Software: Windows 7 Professional 64 bit and Visual Studio 2008

Machine 2:

- Processor: Intel Core i7 960 @ 3.2 ghz [Cor09a]. This processor has 4 physical cores and 8 threads.
- GPU: Quadro FX 1800 [Cor09b]. Number of multiprocessors: 8. Number of CUDA cores: 64
- Software: Windows 7 Professional 64 bit and Visual Studio 2008

To ensure that the AES executed appropriately, the resulting encrypted files produced by the sequential implementations were decrypted and the original files were obtained, which indicates that both, the encryption and decryption were correct. Furthermore, the resulting files from the parallel and CUDA implementations were compared to the files from the sequential implementations to establish correct execution of both implementations.

This section displays the results obtained from running the implementations from Section 3.2 for each machine. First, sequential execution on the CPU. Second, parallel execution on the CPU. Third, parallel execution on the GPU. Each subsequent subsection displays the computed average of times of each of 3 test runs, and a graphic chart for the average times. Furthermore, speedup in execution is shown for each machine. Finally, some special considerations are presented.

4.1 Machine 1

4.1.1 Comparison

Total execution time			
File Size	i5	i5 - 4 threads	Geforce GT 330M
256 kB	833.33 ms	519.00 ms	205.33 ms
512 kB	1,639.00 ms	1,015.33 ms	272.33 ms
1 MB	3,285.00 ms	2,069.00 ms	403.33 ms
2 MB	6,535.33 ms	4,123.33 ms	738.67 ms
4 MB	13,123.33 ms	7,998.67 ms	1,389.00 ms
8 MB	26,063.67 ms	15,359.67 ms	2,551.00 ms
16 MB	52,367.33 ms	32,138.67 ms	5,008.00 ms
32 MB	105,507.00 ms	61,625.67 ms	9,856.00 ms
64 MB	208,730.67 ms	125,044.67 ms	20,380.00 ms
128 MB	416,993.67 ms	245,823.33 ms	40,913.00 ms
256 MB	834,216.67 ms	495,718.00 ms	80,293.33 ms

Table 2: Total time comparison on machine 1

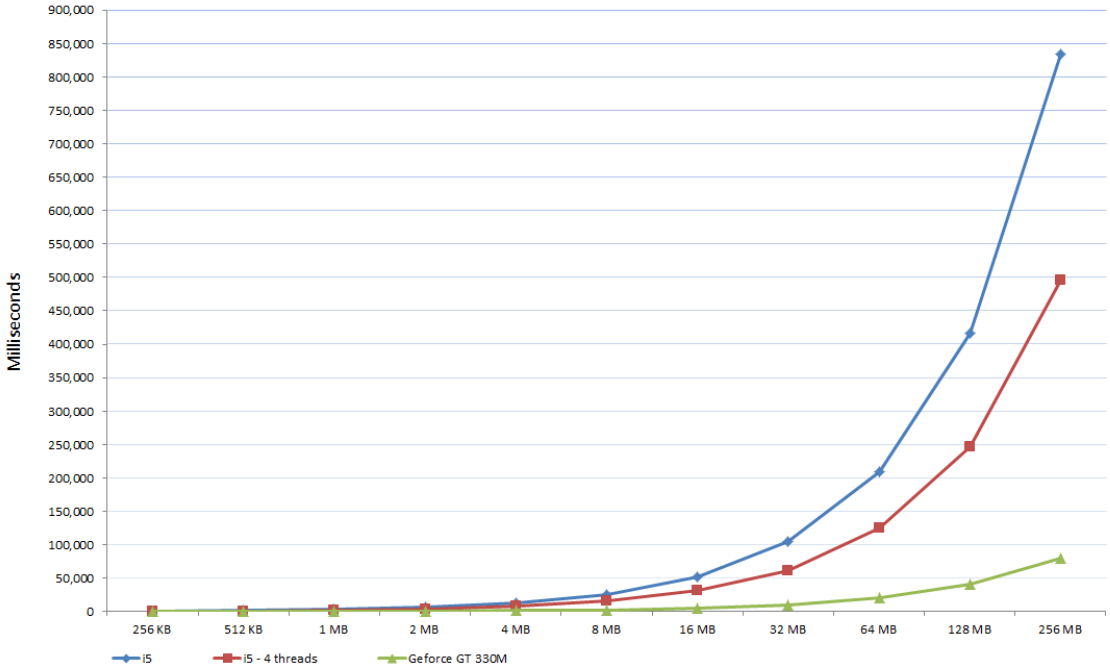

Figure 11: Total time comparison on machine 1

Encryption execution time			
File Size	i5	i5 - 4 threads	Geforce GT 330M
256 KB	663.67 ms	342.33 ms	36.00 ms
512 KB	1,298.00 ms	671.67 ms	68.00 ms
1 MB	2,578.33 ms	1,360.33 ms	140.00 ms
2 MB	5,146.33 ms	2,742.00 ms	284.00 ms
4 MB	10,342.33 ms	5,266.67 ms	554.67 ms
8 MB	20,538.33 ms	10,219.00 ms	1,127.33 ms
16 MB	41,327.33 ms	20,714.33 ms	2,251.33 ms
32 MB	83,193.33 ms	41,006.33 ms	4,500.00 ms
64 MB	164,309.67 ms	82,095.33 ms	9,003.00 ms
128 MB	328,694.33 ms	163,641.33 ms	18,006.67 ms
256 MB	657,278.33 ms	327,999.33 ms	36,016.67 ms

Table 3: Encryption time comparison on machine 1

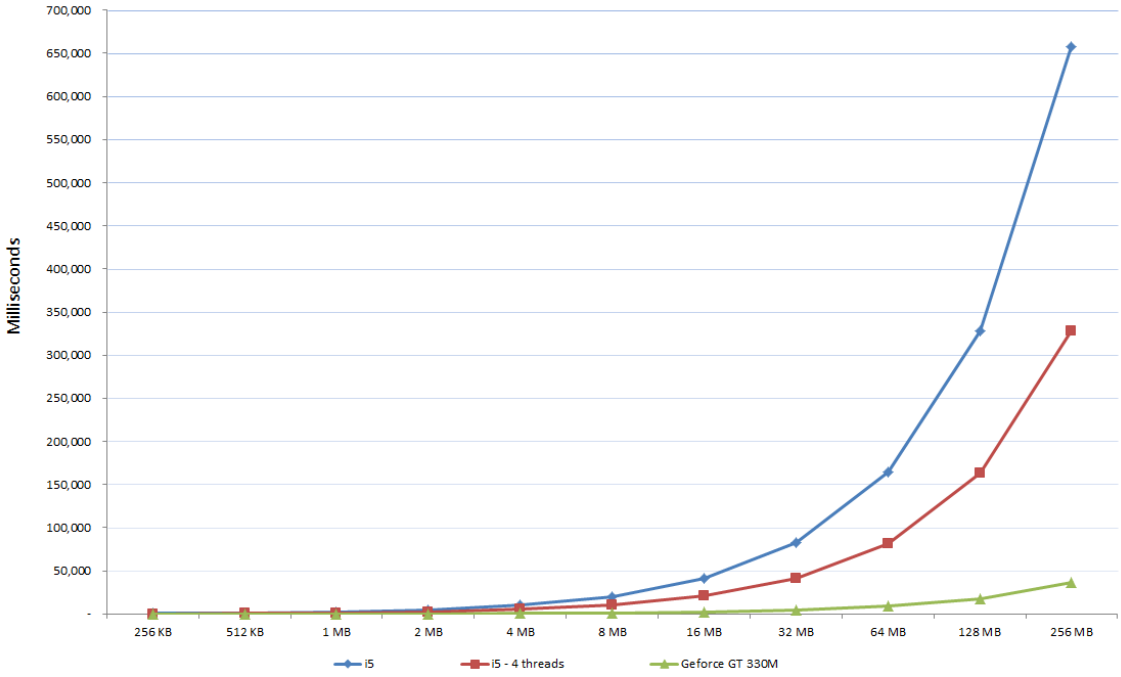

Figure 12: Encryption time comparison on machine 1

4.1.2 Speedup

The speedup was calculated for both, total time and encryption time, by taking the slowest execution times, in this case, the ones from the i5 processor on Machine 1, and diving it by the times obtained in other tests runs.

Table 4 and Figure 13 show the speedup achieved for the encryption execution time.

Total execution time		
File Size	i5 - 4 threads	Geforce GT 330M
256 kB	1.60565x	4.05844x
512 kB	1.61425x	6.01836x
1 MB	1.58772x	8.14463x
2 MB	1.58496x	8.84747x
4 MB	1.64069x	9.44804x
8 MB	1.69689x	10.21704x
16 MB	1.62942x	10.45674x
32 MB	1.71206x	10.70485x
64 MB	1.66925x	10.24194x
128 MB	1.69631x	10.19220x
256 MB	1.68285x	10.38961x

Table 4: Speedup, on total time, over sequential execution on machine 1

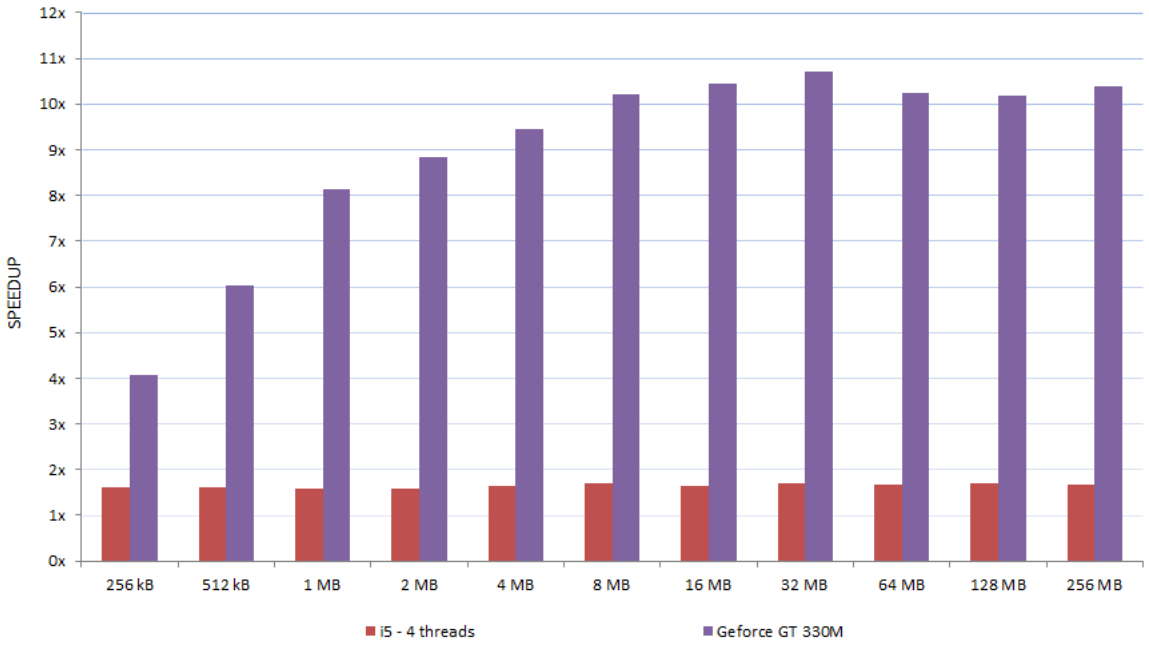

Figure 13: Speedup, on total time, over sequential execution on machine 1

Table 5 and Figure 14 show the speedup achieved for the encryption execution time.

Encryption execution time		
File Size	i5 - 4 threads	Geforce GT 330M
256 KB	1.93866x	18.43519x
512 KB	1.93251x	19.08824x
1 MB	1.89537x	18.41667x
2 MB	1.87685x	18.12089x
4 MB	1.96373x	18.64603x
8 MB	2.00982x	18.21851x
16 MB	1.99511x	18.35683x
32 MB	2.02879x	18.48741x
64 MB	2.00145x	18.25055x
128 MB	2.00863x	18.25404x
256 MB	2.00390x	18.24928x

Table 5: Speedup, on encryption time, over sequential execution on machine 1

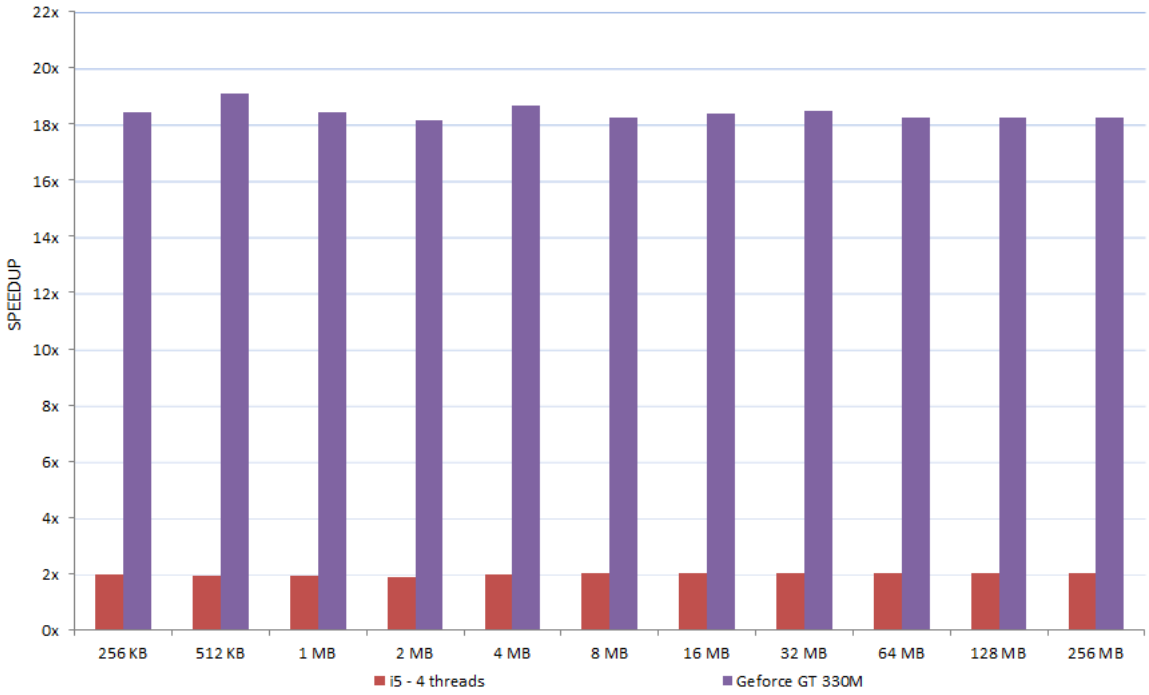

Figure 14: Speedup, on encryption time, over sequential execution on machine 1

4.2 Machine 2

4.2.1 Comparison

Total execution time			
File Size	i7	i7 - 8 threads	Quadro FX 1800
256 KB	546.00 ms	350.67 ms	185.67 ms
512 KB	1,082.33 ms	664.00 ms	246.00 ms
1 MB	2,169.00 ms	1,317.67 ms	347.67 ms
2 MB	4,255.00 ms	2,623.00 ms	547.33 ms
4 MB	8,386.33 ms	5,216.67 ms	973.00 ms
8 MB	17,488.33 ms	10,420.00 ms	1,812.33 ms
16 MB	34,393.67 ms	20,964.33 ms	3,501.00 ms
32 MB	68,466.33 ms	42,113.33 ms	7,025.67 ms
64 MB	136,931.00 ms	83,933.00 ms	14,019.67 ms
128 MB	270,202.00 ms	167,718.00 ms	28,015.00 ms
256 MB	555,078.00 ms	338,793.00 ms	56,336.00 ms

Table 6: Total time comparison on machine 2

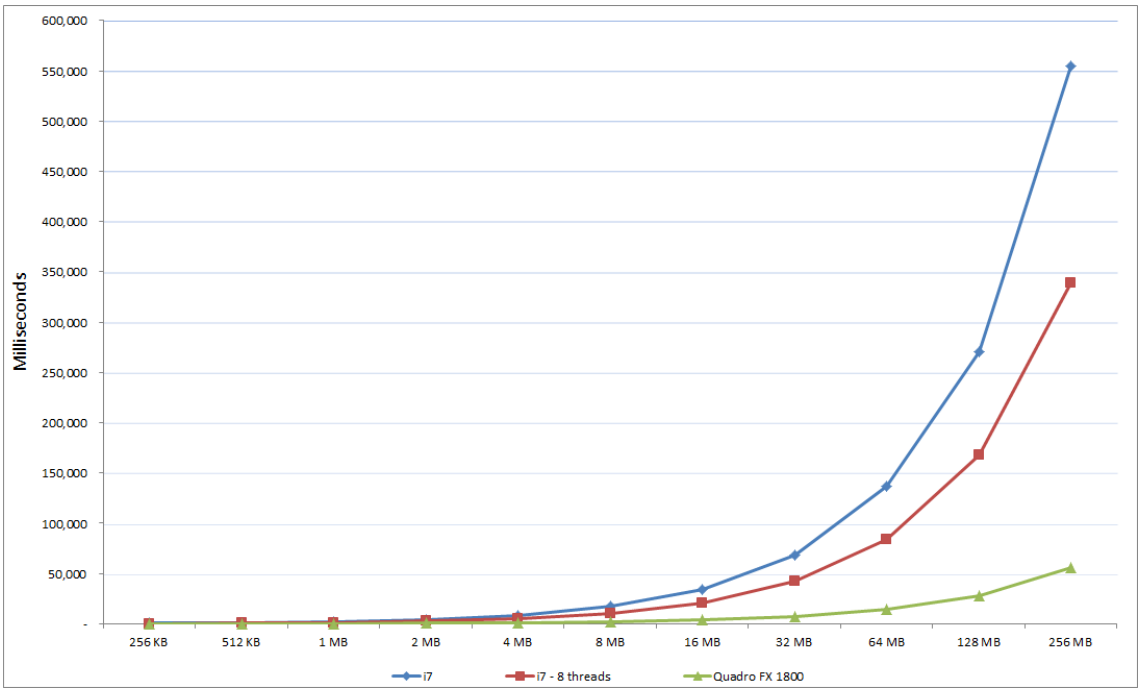

Figure 15: Total time comparison on machine 2

File Size	i7	i7 - 8 threads	Quadro FX 1800
256 kB	382.67 ms	147.00 ms	21.00 ms
512 kB	761.67 ms	259.33 ms	42.00 ms
1 MB	1,530.00 ms	508.33 ms	84.00 ms
2 MB	2,983.67 ms	1,015.67 ms	168.00 ms
4 MB	5,827.00 ms	2,008.00 ms	336.00 ms
8 MB	12,309.00 ms	4,030.33 ms	672.00 ms
16 MB	24,030.00 ms	8,127.00 ms	1,344.00 ms
32 MB	47,842.67 ms	16,450.00 ms	2,688.00 ms
64 MB	95,685.33 ms	32,401.67 ms	5,376.00 ms
128 MB	187,683.67 ms	64,993.33 ms	10,752.00 ms
256 MB	389,140.00 ms	131,613.67 ms	21,511.33 ms

Table 7: Encryption time comparison on machine 2

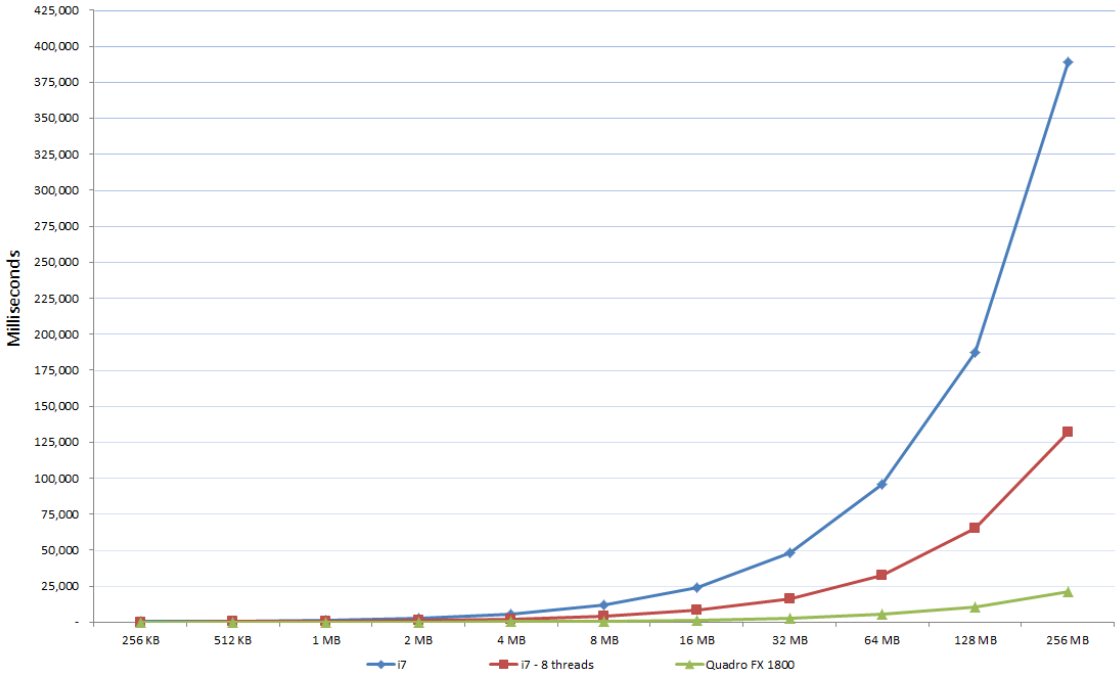

Figure 16: Encryption time comparison on machine 2

4.2.2 Speedup

The speedup was calculated for both, total time and execution time, by taking the slowest execution times, in this case, the ones from the i7 processor on Machine 2, and diving it by the times obtained in other tests runs.

Table 8 and Figure 17 show the speedup achieved for the total execution time.

Total execution time		
File Size	i7 - 8 threads	Quadro FX 1800
256 kB	1.55703x	2.94075x
512 kB	1.63002x	4.39973x
1 MB	1.64609x	6.23873x
2 MB	1.62219x	7.77406x
4 MB	1.60760x	8.61905x
8 MB	1.67834x	9.64962x
16 MB	1.64058x	9.82396x
32 MB	1.62576x	9.74517x
64 MB	1.63143x	9.76707x
128 MB	1.61105x	9.64490x
256 MB	1.63840x	9.85299x

Table 8: Speedup, on total time, over sequential execution on machine 2

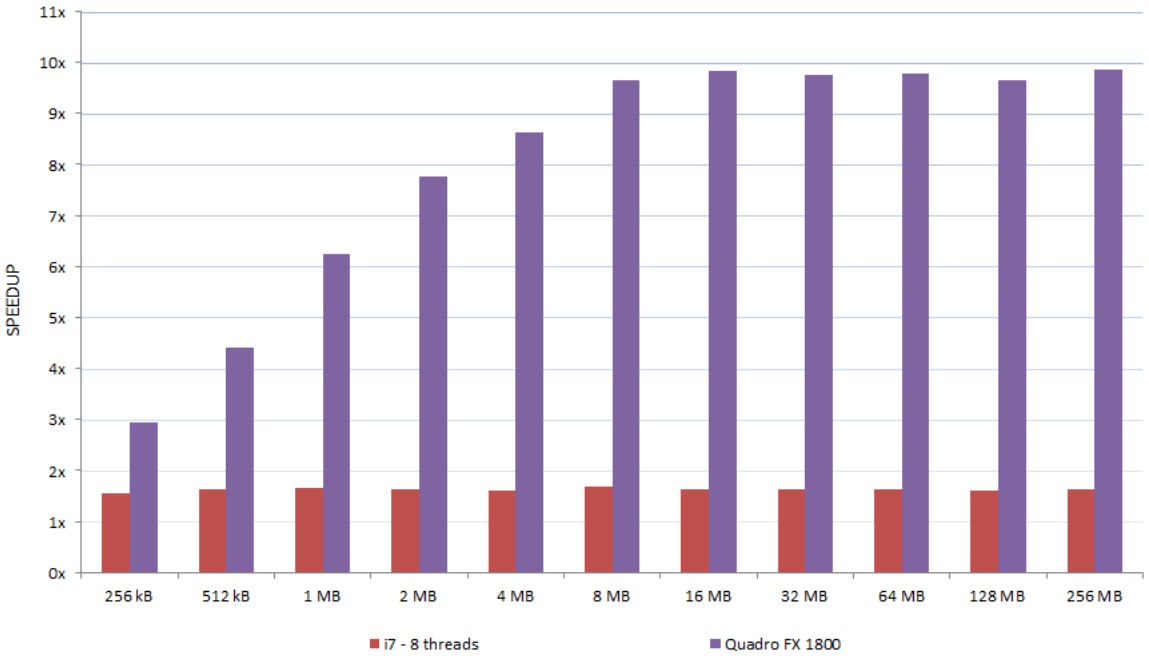

Figure 17: Speedup, on total time, over sequential execution on machine 2

Table 9 and Figure 18 show the speedup achieved for the encryption execution time.

Encryption execution time		
File Size	i7 - 8 threads	Quadro FX 1800
256 KB	2.60317x	18.22222x
512 KB	2.93702x	18.13492x
1 MB	3.00984x	18.21429x
2 MB	2.93764x	17.75992x
4 MB	2.90189x	17.34226x
8 MB	3.05409x	18.31696x
16 MB	2.95681x	17.87946x
32 MB	2.90837x	17.79861x
64 MB	2.95310x	17.79861x
128 MB	2.88774x	17.45570x
256 MB	2.95668x	18.09000x

Table 9: Speedup, on encryption time, over sequential execution on machine 2

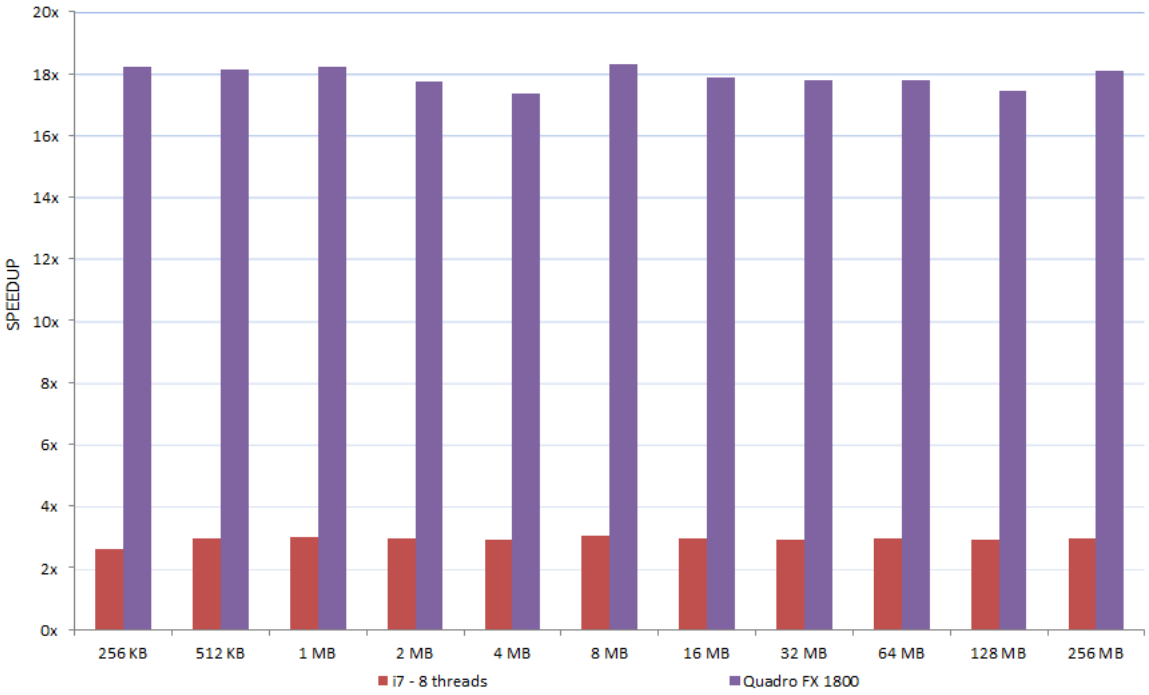

Figure 18: Speedup, on encryption time, over sequential execution on machine 2

4.3 Special considerations

This section, contains and points out, information and behaviours in the results that are regarded as important. First, a quick look at the amount of time, from the total execution time, that the the encryption time is responsible for, expressed percentually. After that, the asymptotic behaviour of speedups on the GPUs, when plotted on a chart. Then, the effects it has on performance while using the GPUs, the usage of a small chunk size for processing.

4.3.1 Execution time percentage

Figure 19 shows the percentage from the total time that the encryption time accounts for. It is worth noting that remaining percentage corresponds to input/output and data copying activities which increases significantly for the 8 threaded execution and the GPUs' executions. This indicates that the AES encryption is highly parallelizable and there is quite a bottleneck in execution times caused by in input/output and data copying.

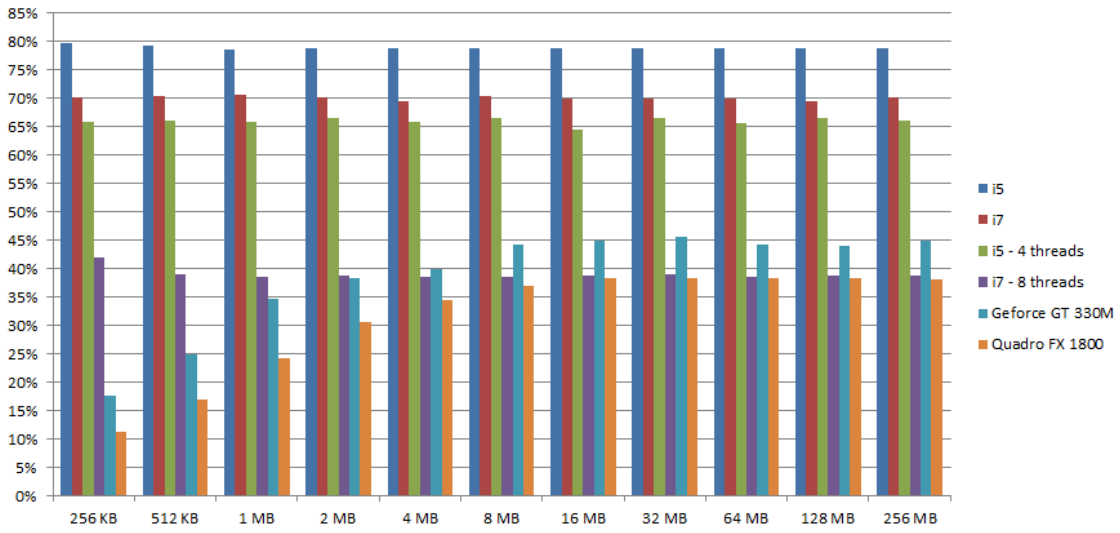

Figure 19: Encryption time / Total time * 100%

4.3.2 Speedup asymptote

It is not clear why Figures 13 and 17 on Sections 4.1.2 and 4.2.2, respectively, which depict the speedups in total execution time on machine 1 and machine 2, behave asymptotically for the GPUs. More tests could be done to evaluate if the tendency continues. However, to perform this speedup calculations, sequential execution would be needed and for files bigger than 1GB, it took hours to run the code, thus it is quite unpractical to carry out the required tests under the scope of this project. Nonetheless, executions times for files of 512 MBs, 1 GB and 2 GBs were forecasted using the available data. Said forecast was done using Microsoft Excel's tools.

Figures 20, and 21 and Tables 10, and 11 show the forecasted data for machine 1 and 2, respectively.

File Size	i5	i5 - 4 threads	Geforce GT 330M
512 MB	1,664,077.07 ms	971,591.27 ms	150,976.10 ms
1 GB	3,323,673.53 ms	1,927,800.93 ms	272,091.19 ms
2 GB	6,638,397.91 ms	3,825,082.13 ms	470,147.33 ms

Table 10: Forecasted values for files of 512 MBs, 1 GB and 2 GBs for machine 1

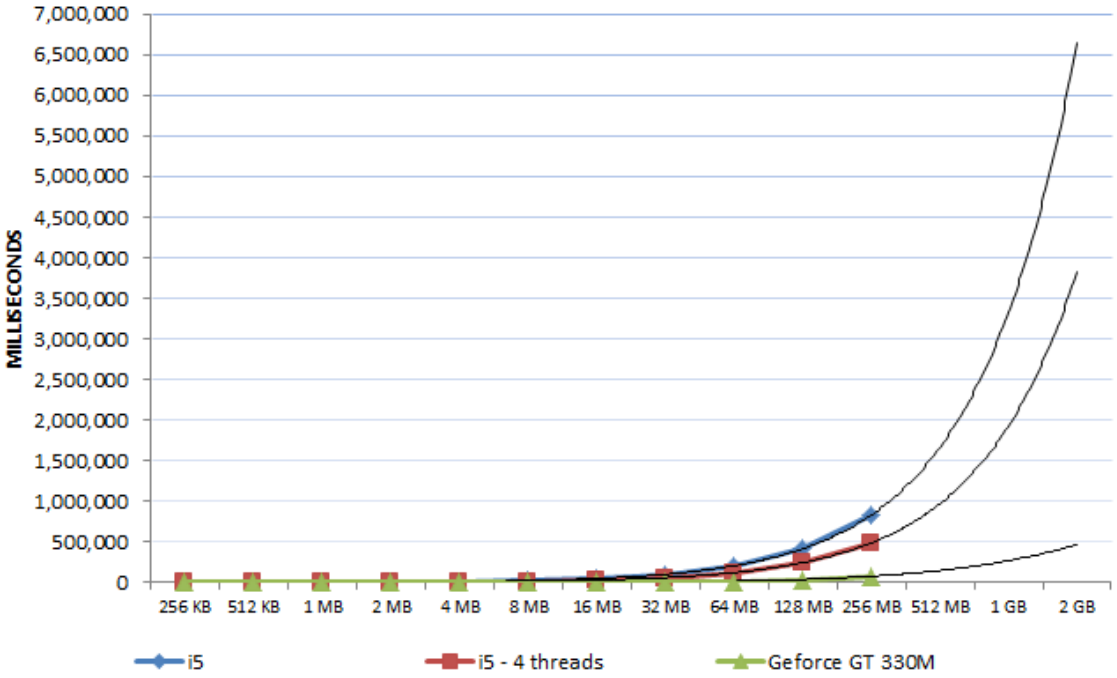

Figure 20: Forecasted values for files of 512 MBs, 1 GB and 2 GBs for machine 1

File Size	i7	i7 - 8 threads	Quadro FX 1800
512 MB	1,092,074.36 ms	663,926.24 ms	110,449.07 ms
1 GB	2,181,426.38 ms	1,323,680.06 ms	208,959.45 ms
2 GB	4,357,414.89 ms	2,639,041.49 ms	379,222.35 ms

Table 11: Forecasted values for files of 512 MBs, 1 GB and 2 GBs for machine 2

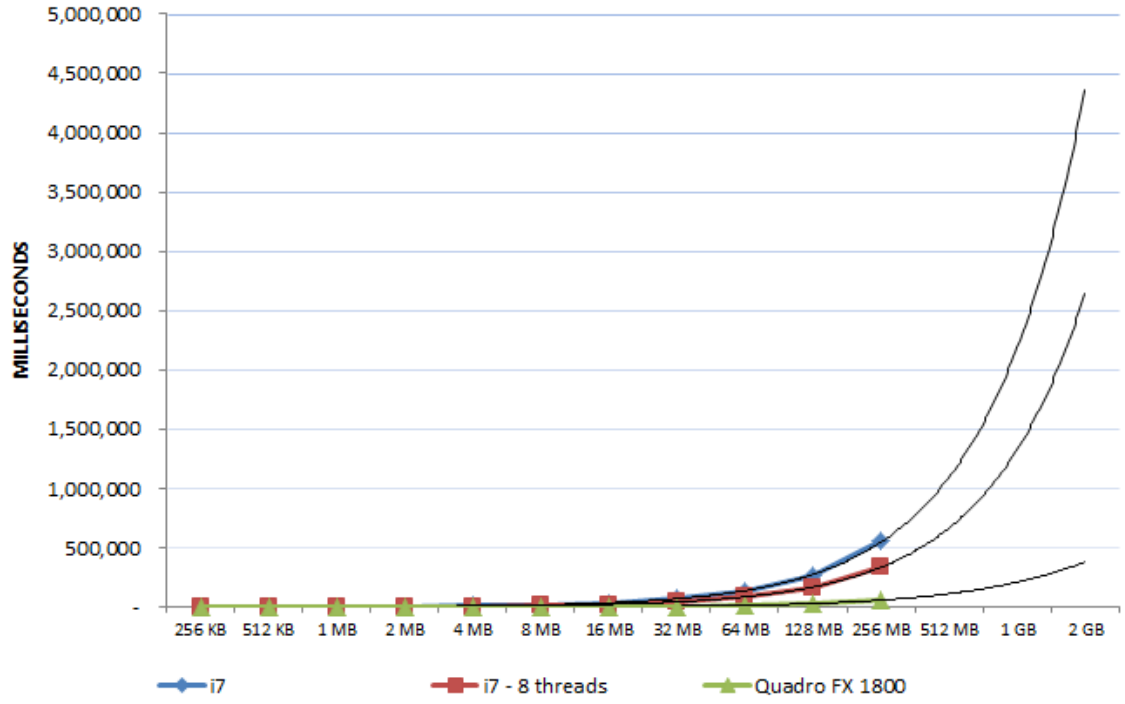

Figure 21: Forecasted values for files of 512 MBs, 1 GB and 2 GBs for machine 2

The forecasted data was used among the data that was already available to make a graphic of the speedup trendline, as seen in Figure 22 for machine 1 and Figure 23 for machine 2.

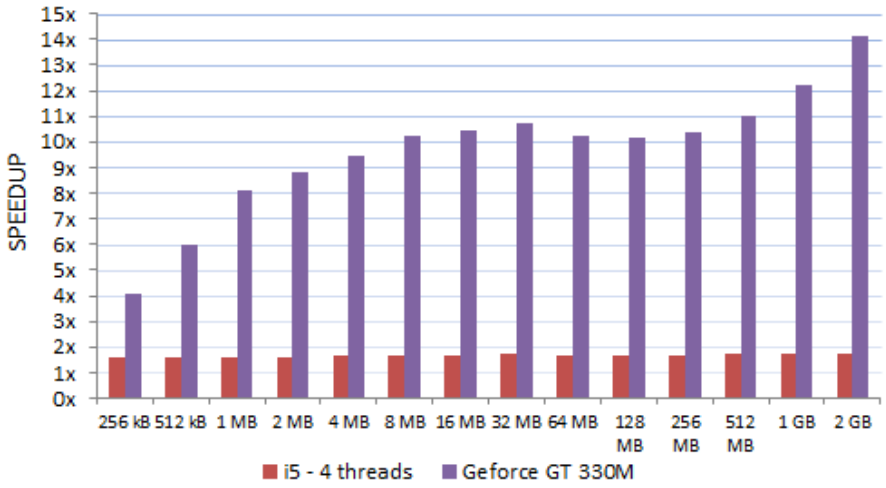

Figure 22: Speedup trendline on machine 1

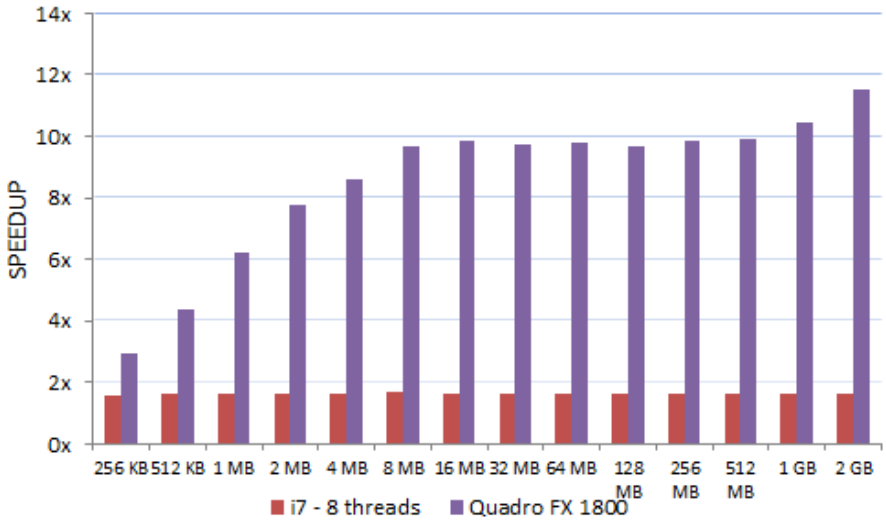

Figure 23: Speedup trendline on machine 2

The tendencies from the speedup figures, suggest that there might be an increase in performance when the CUDA implementation on larger files, nevertheless, real tests remain to be performed to better understand its behavior.

4.3.3 Behavior on GPUs

For the implementation of the AES algorithm using the GPU as described in Section 3.2.3 a chunk size of 256kBs was used. However, while running different tests on the GPU on machine 1 and lowering the value of the chunk size, the total execution time started to increase, which could be explained by the increment in the amount of data that has to be copied to and from the GPU, causing a great overhead. Nevertheless, the values of the encryption time were extremely low as shown in Tables 12, 13 and Figure 24 for machine 1. Said times in those tables and charts correspond to executions of the code using a chunk size of 1kB, instead of the 256kBs

value used before. Just as before, the information displays the time of each of 3 test runs, the computed average based on them and a graphic chart for those average times.

File Size	Total time	Encryption time	Total time	Encryption time	Total time	Encryption time
256 kB	283 ms	1 ms	320 ms	3 ms	267 ms	0 ms
512 kB	481 ms	5 ms	463 ms	0 ms	583 ms	4 ms
1 MB	840 ms	2 ms	837 ms	0 ms	975 ms	3 ms
2 MB	1588 ms	0 ms	2058 ms	6 ms	1765 ms	5 ms
4 MB	3293 ms	8 ms	3348 ms	15 ms	3498 ms	19 ms
8 MB	6725 ms	26 ms	6840 ms	17 ms	6626 ms	17 ms
16 MB	12985 ms	60 ms	13379 ms	52 ms	13459 ms	50 ms
32 MB	27843 ms	115 ms	27125 ms	174 ms	27329 ms	151 ms
64 MB	54082 ms	211 ms	55723 ms	320 ms	55550 ms	238 ms
128 MB	107988 ms	385 ms	111484 ms	474 ms	109257 ms	406 ms
256 MB	218153 ms	809 ms	221275 ms	1149 ms	221946 ms	1960 ms

Table 12: Test runs for parallel execution using Geforce GT 330M and a chunk size of 1kB

File Size	Total time	Encryption time
256 kB	290.0 ms	1.3 ms
512 kB	509.0 ms	3.0 ms
1 MB	884.0 ms	1.7 ms
2 MB	1803.7 ms	3.7 ms
4 MB	3379.7 ms	14.0 ms
8 MB	6730.3 ms	20.0 ms
16 MB	13274.3 ms	54.0 ms
32 MB	27432.3 ms	146.7 ms
64 MB	55118.3 ms	256.3 ms
128 MB	109576.3 ms	421.7 ms
256 MB	220458.0 ms	1306.0 ms

Table 13: Average times for sequential execution using Geforce GT 330M and a chunk size of 1kB

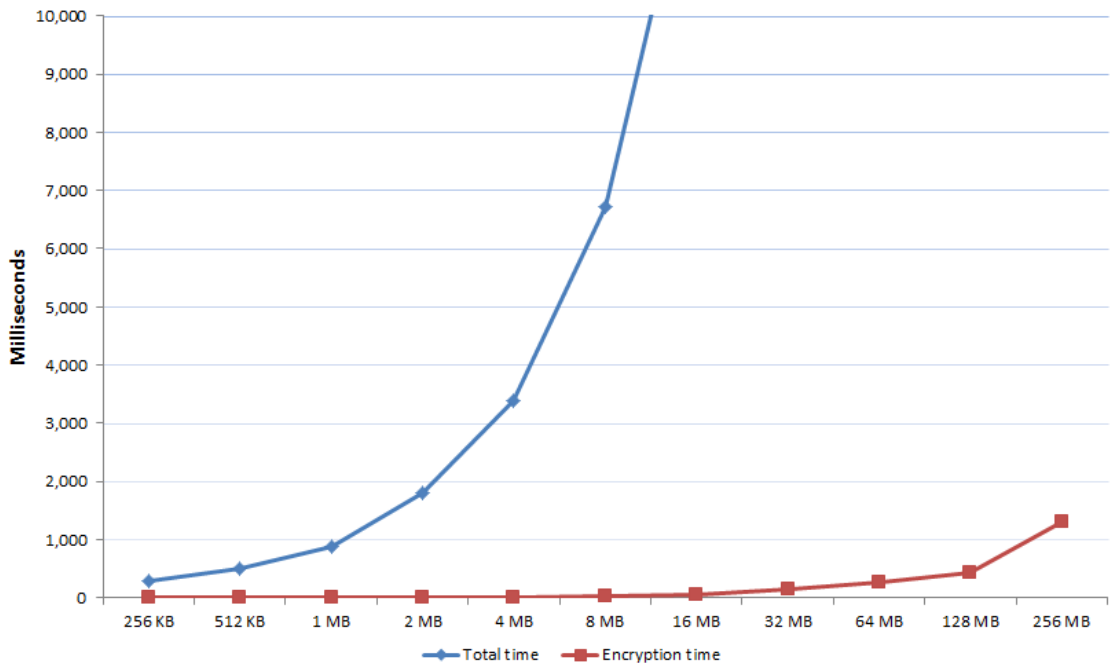

Figure 24: Average times for sequential execution using Geforce GT 330M and a chunk size of 1kB

Calculating the speedup, in a similar fashion to Section 4.1.2, over the sequential execution of the i5 processor as seen in Figure 25, the speedup for the encryption times are impressive, leaving the question of whether the code is running properly or not, so to be certain, the resulting encrypted files were compared to the files obtained before. The comparisons showed that the files were identical, which ensures that the code was running properly, yet this leaves behind the matter of whether the AES does run that fast or the actions taken to measure execution times are limited and measurements cannot be properly carried out.

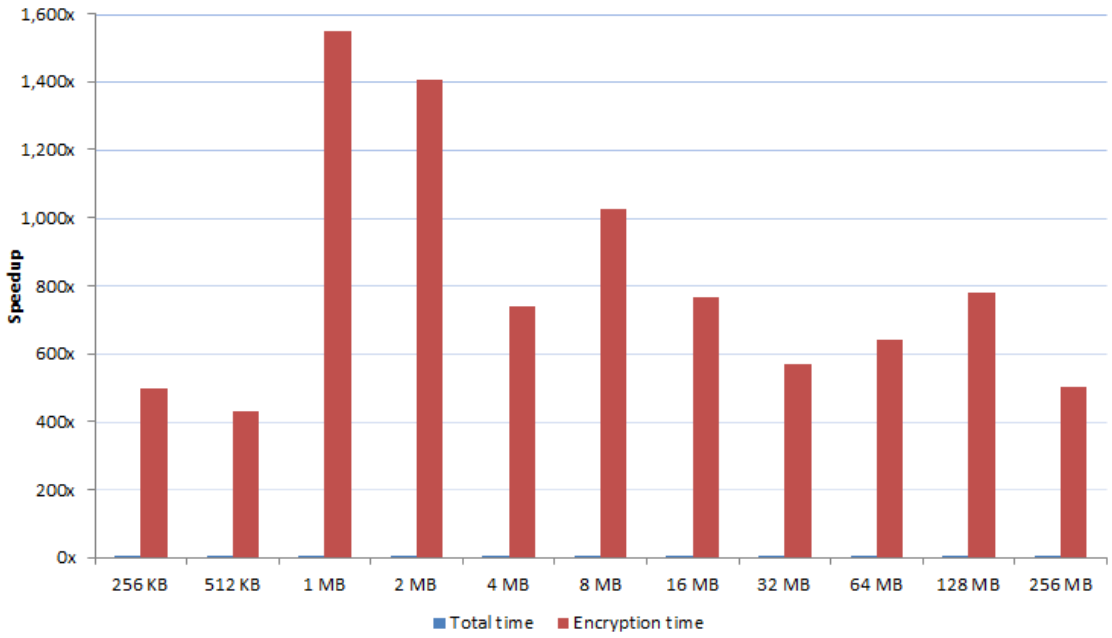

Figure 25: Speedup over sequential execution on machine 1 using Geforce GT 330M and a chunk size of 1kB

5 Conclusions

Within this project, several implementations have been presented for the AES block cipher encryption, sequential execution on the CPU, parallel execution on the CPU and parallel execution on the GPU. Execution times have been compared for each approach's implementation. The developed implementations showed a maximum speedup on the GPUs, over the sequential executions on the CPUs, of `10.70485x` on machine 1 and a maximum speedup of `9.85299x` on machine 2, were achieved for total execution times. Regarding encryption times, maximum speedups on the GPUs, over sequential execution on the CPUs, of `19.08824x` and `18.31696x` were achieved, for machine 1 and 2, respectively.

In todays digital world, cryptography plays a vital role to protect information stored all around the globe. Encryption allows to keep data secret, especially when sending sensitive data over a network, making sure no one reads the confidential data, like financial transactions or user information.

CUDA GPU implementations can be a inexpensive alternative to CPU implementations that perform equivalently. The GPU installed base, worldwide, is considerable, given that there are quite a low-cost variety to choose from, which gives developers a wide audience of users for their CUDA applications. Considering that a large amount of computer users have access to dedicated GPUs, that are quite underutilized, this opens the playground to develop applications or create certain ways, to alleviate the central processor of some time consuming tasks that could, potentially, be taken care of by the GPU, like the AES encryption, as demonstrated in this project, with focus on large amounts of data to really notice a difference.

Not only can the end users benefit at home, every kind of company that requires protection to the sensitive data they store, like email providers, social networks, banks, and universities, might implement a low-cost CUDA solution to greatly benefit from its features and accessible price. This type of co-processing can be performed in a different number of fields such as research, medicine, finance, astrophysics, images and video [Cor10d], like the Berkeley Open Infrastructure for Network Computing [UoC10], making it possible for researchers to tap into the enormous processing power of personal computers around the world.

6 Future work

A lot of desired work remains to be carried out. Implementing the AES encrypting algorithm and analysing its performance using 192 and 256 bit keys. In addition, and in a similar manner, working on the decryption of ciphered files and studying possible speedups in execution. As of now, only files with a size multiple of 256kBs can be processed by any of the implementations product of this project, however, changes to the code can, and should, be made to allow it to encrypt a file of any size, choosing an adequate chunk size to subdivide the encryption task. Moreover, optimizations to the code could be explored to further improve the execution times and achieve an even greater speedup, both in the CPU and the GPU. Furthermore, there is also a lot of interest in appreciating how the application would perform, distributing the task among different CPUs and GPUs in computers connected through a network or even distributed across the world using the internet to communicate. The usage of computer grids with several GPUs could be explored to carry out analyses of the AES performance and hardware co-processing efficiency.

All of the future work stated can also be done exploring other encryption algorithms, or even, compression algorithm such as the BZIP or one of its variants (bzip2, pbzip2, mpibzip2, lbzip2), given that is a block compression algorithm and each block would be compressed separatedly.

References

[And08] M. Andrecut, *Fast GPU Implementation of Sparse Signal Recovery from Random Projections*, Institute for Biocomplexity and Informatics, University of Calgary (2008), Calgary, Alberta, Canada.

[Bar10] Blaise Barney, *Introduction to parallel computing*, https://computing.llnl.gov/tutorials/parallel_comp/, 2010.

[BBAP09] Andrea Di Biagio, Alessandro Barenghi, Giovanni Agosta, and Gerardo Pelosi, *Design of a parallel aes for graphics hardware using the cuda framework*, Parallel and Distributed Processing Symposium, International **0** (2009), 1–8.

[BS10] Peter Bakkum and Kevin Skadron, *Accelerating SQL Database Operations on a GPU with CUDA*, Department of Computer Science, University of Virginia (2010), Charlottesville, VA.

[CBM+08] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron, *A performance study of general-purpose applications on graphics processors using CUDA*, Journal of parallel and distributed computing, Elsevier (2008), University of Virginia, Department of Computer Science, Charlottesville, VA, USA.

[CC10] Victor León Higuita Cardona and Juan Diego Toro Cano, *Simulation of collisions with parallel deformation.*, EAFIT University (2010), Medellin - Colombia.

[Cor08] Nvidia Corporation, *CUDA Technical Training. Volume I: Introduction to CUDA Programming*, http://www.nvidia.com/docs/IO/47904/VolumeI.pdf, 2008.

[Cor09a] ©Intel Corporation, *Intel® Core™ i7-960 Processor*, http://ark.intel.com/Product.aspx?id=37151, 2009.

[Cor09b] Nvidia Corporation, *Quadro FX 1800*, http://www.nvidia.com/object/product_quadro_fx_1800_us.html, 2009.

[Cor10a] ©Intel Corporation, *Intel® Core™ i5-450M Processor*, http://ark.intel.com/Product.aspx?id=49022, 2010.

[Cor10b] Microsoft Corporation, *LARGE_INTEGER Union*, http://msdn.microsoft.com/en-us/library/aa383713(VS.85).aspx, 2010.

[Cor10c] Nvidia Corporation, *CUDA GPU Occupancy Calculator*, http://www.nvidia.com/object/cuda_get.html, 2010.

[Cor10d] ———, *CUDA in Action*, http://www.nvidia.com/object/cuda_in_action.html, 2010.

[Cor10e] ———, *Geforce GT 330M*, http://www.nvidia.com/object/product_geforce_gt_330m_us.html, 2010.

[Cor10f] ———, *What is CUDA?*, http://www.nvidia.com/object/what_is_cuda_new.html, 2010.

[EZ04] Universidad ORT Montevideo Uruguay Enrique Zabala, *Rjindael cipher*, http://www.cs.bc.edu/~straubin/cs381-05/blockciphers/rijndael_ingles2004.swf, 2004.

[GDNB10] Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud, *K-nearest neighbor search: fast GPU-based implementations and application to high-dimensional feature matching*, IEEE International Conference on Image Processing (ICIP) (Hong Kong, China), September 2010.

[gpg10] gpgpu.org, *General-purpose computation on graphics processing units*, http://gpgpu.org/about, 2010.

[GTA+10] Matthew A. Goodrum, Michael J. Trotter, Alla Aksel, Scott T. Acton, and Kevin Skadron, *Parallelization of particle filter algorithms*, Department of Electrical and Computer Engineering, University of Virginia (2010), Charlottesville, VA.

[HN07] Pawan Harish and P. J. Narayanan, *Accelerating large graph algorithms on the GPU using CUDA*, Center for Visual Information Technology (2007), International Institute of Information Technology Hyderabad, INDIA.

[HW07] Owen Harrison and John Waldron, *AES encryption implementation and analysis on commodity graphics processing units*, Computer Architecture Group, Trinity College (2007), Dublin - Ireland.

[HW08] _____, *Practical symmetric key cryptography on modern graphics hardware*, Computer Architecture Group, Trinity College (2008), Dublin - Ireland.

[KG10] Sarnath Kannan and Raghavendra Ganji, *Porting Autodock to CUDA*, IEEE World Congress on Computational Intelligence (2010), CCIB, Barcelona, Spain.

[LEBG10] Christian Andres Diaz Leon, Steven Eliuk, Pierre Boulanger, and Helmuth Trefftz Gomez, *Simulating soft tissues using a GPU approach of the mass-spring model*, EAFIT University (2010), Medellin - Colombia.

[LR09] Lukasz Ligowski and Witold Rudnicki, *An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases*, Interdisciplinary Centre for Mathematical and Computational Modelling, University of Warsaw (2009), Warsaw, Poland.

[LSM09] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell, *Parallel Reconstruction of NeighborJoining Trees for Large Multiple Sequence Alignments using CUDA*, School of Computer Engineering, Nanyang Technological University (2009), Singapore.

[Man07] Svetlin A. Manavski, *CUDA compatible GPU as an efficient hardware accelerator for AES cryptography*, ITCIS (2007), Sofia - Bulgaria.

[MJJ10] Chonglei Mei, Hai Jiang, and J. Jenness, *Cuda-based aes parallelization with fine-tuned gpu memory utilization*, Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, 2010, pp. 1 –7.

[NHA09] A.M. Nazlee, F.A. Hussin, and N.B.Z. Ali, *Serpent encryption algorithm implementation on compute unified device architecture (cuda)*, Research and Development (SCOReD), 2009 IEEE Student Conference on, 2009, pp. 164 –167.

[NIS01] NIST, *FIPS 197: Advanced Encryption Standard (AES)*, Federal Information Processing Standards (2001).

[PK09] Niyaz PK, *Advanced Encryption Standard (AES) implementation in c/c++*, http://www.hoozi.com/Articles/AESEncryption.htm, 2009.

[RL09] Guillaume Rizk and Dominique Lavenier, *GPU Accelerated RNA Folding Algorithm*, IRISA - Symbiose Campus universitaire de Beaulieu (2009), Cedex, France.

[TW08] Christian Trefftz and Greg Wolffe, *CUDA: Introduction*, Grand Valley State University, 2008.

[UoC10] Berkeley University of California, *Berkeley open infrastructure for network computing*, http://boinc.berkeley.edu/, 2010.