PARALLELIZING AES ON MULTICORES AND GPUS

Julián Ortega[1], Helmuth Trefftz[1], and Christian Trefftz[2]
[1]Universidad EAFIT, Medellín, Colombia
[2]Grand Valley State University, Allendale, Michigan, USA

Abstract - The AES block cipher cryptographic algorithm is widely used and it is resource intensive. An existing sequential open source implementation of the algorithm was parallelized on multi-core microprocessors and GPUs. Performance results are presented.

Index Terms - AES, cryptography, OpenMP, CUDA

I.INTRODUCTION

The Advanced Encryption Standard (AES) [1] is a widely used symmetric key cryptographic algorithm. It is used in OpenSSL for secure web applications and it is used to encrypt files as well. AES in computationally demanding and hence efforts have been made to parallelize its execution on different platforms.

There are two kinds of parallel platforms that are becoming widely available at affordable prices: Microprocessors with multiple cores and Graphical Processing Units (GPUs). Given the computational demands of the AES algorithm, it is relevant to explore these parallel platforms to parallelize AES and the performance gains that can be obtained on them.

GPUs have been explored previously as platforms for parallelizing AES. Manavski [2] parallelized an AES implementation based on OpenSSL using an NVIDIA GeForce 8800 GTX. He used CUDA [3], a programming language based on C with extensions to program GPUs. Di Biagio et al. [4] performed a set of similar experiments obtaining even better performance. Neither paper compares the results with a parallelization that uses microprocessors with multiple cores.

Microprocessors with multiple cores have been used previously by other researchers to parallelize AES as well. Bielecki [5] reports a parallelization that uses OpenMP and Coens and Zhou [6] performed a study in which they compared three different styles of parallelism: thread level parallelism with OpenMP, instruction level parallelism with SSE3 and data level parallelism using CUDA. Their implementation is based on the sequential open source associated with XySSL. The results reported here use different GPUs and CPUs and a different sequential version. The emphasis of the work presented here is on the compression of files, including the I/O operations.

The rest of this paper is structured as follows: a brief description of AES, OpenMP, GPUs and CUDA is included in section two. The results obtained with CUDA and OpenMP are presented and compared in section three. Conclusions and future work are in section four.

II.AES, OPENMP, GPUS AND CUDA

A. AES

The AES, or Advanced Encryption Standard, is based on the Rijndael algorithm developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen. It accepts a fixed block size of 128 bits and a key size of 128, 192 or 256 bits. The implementation described in this paper used the AES algorithm and only a 128-bit key, thus 10 rounds in all the AES implementations.

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 as input, and produces a corresponding n-bit block of cipher-text. The exact transformation is controlled using a second input, the key. Figure 1 shows an overview of the process.

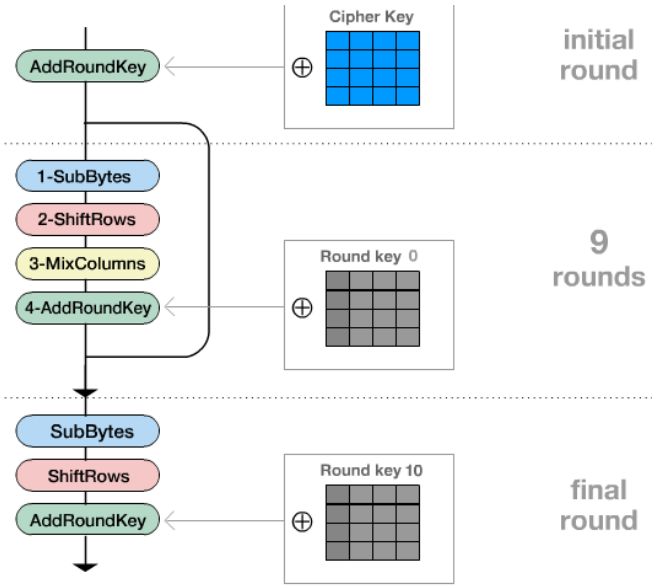

Figure 1. Overview of AES[1].

[1]Source:www.cs.bc.edu/~**straubin**/cs38105/blockciphers/rijndael_ingles2004.swf

B. OpenMP

Microprocessor manufacturers have been selling microprocessors with several cores for several years. The maximum number of cores available in microprocessors keeps increasing at a steady rate. A program can spawn multiple threads and every thread can execute on a separate core, allowing a single program to take advantage of the presence of several cores in a microprocessor.

OpenMP is "an Application Program Interface (API) that can be used to explicitly direct multi-threaded, shared memory parallelism"[2] It has three main components: Compiler Directives, Runtime Library Routines and Environment Variables.

OpenMP makes it easy to take advantage of the presence of several cores in a microprocessor. One inserts compiler directives in existing source code in C/C++ or Fortran. A compiler that supports OpenMP then generates code that spawns threads as indicated by the directives and according to the values of the environment variables. Each thread can execute on a separate core, taking advantage of the presence of multiple cores in a processor.

Typically, applications spend a significant amount of time in loops. If the iterations in the loop do not have data dependencies, in other words, if the results of one iteration do not depend on the results of a previous iteration, then it is possible to execute the different iterations of the loop in parallel. The iterations are divided among the available cores.

C. GRAPHICAL PROCESSING UNITS AND CUDA

Graphical Processing Units (GPUs) were originally designed as specialized processors to accelerate graphic processing. They were designed to process numerous pixels, vertices and/or polygons simultaneously. As the operations on those pixels, vertices or polygons were inherently independent, the designers of GPUs created chips with large numbers of specialized processors that would execute the same transformation on different vertices/polygons in parallel. The number of these specialized processors, now referred to as "stream processors" has grown substantially over time and recent GPUs contain hundreds of these processors. Recently, researchers started using those GPUs for non graphical tasks. Initially "shaders", short programs meant to perform graphical transformations were used to program GPUs for other purposes. Eventually, data parallel programming languages were created for using GPUs for General Purpose computing (GP-GPU).

CUDA or Compute Unified Device Architecture is a co-designed hardware and software system built to harness the computational horsepower of NVIDIA GPUs for GP-GPU computing. 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. GPU are very efficient for data parallelism, given that they have multiple ALUs, fast onboard memory and a high throughput on parallel tasks. GPU threads are extremely lightweight. Thread creation and context switching are very fast and the GPU expects 1000s of threads for full utilization. The stream processors are grouped into multiprocessors that share registers and a local memory.

When programming GPUs, one should take into account that the GPU is a co-processor with an independent memory space; hence it is necessary to copy data from the main memory of the computer to the GPU memory. Figure 2 illustrates the process.

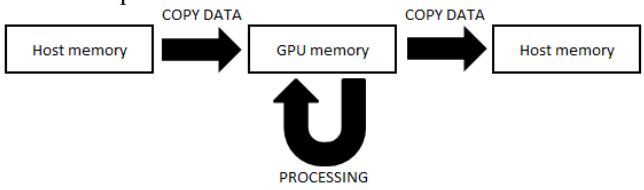

Figure 2. Memory transfers to and from a GPU card.

The code that is executed on the stream processors in the GPU card is written as functions called "kernels". Kernels are, frequently, written to work on a particular entry (or entries) in an array. Predefined variables are available in CUDA that allow the program to find a unique id that serves to access the entry (or entries) in the array that this kernel works on.

When calling a "kernel" function it is necessary to specify a grid size and a block size. The block size indicates the amount of threads that will be assigned for execution to one of the multiprocessors in the GPU. The grid size indicates the amount of threads blocks that will be active.

III.PARALLELIZATION

The base source code used for this implementation was an open source implementation made available by PK Niyaz [7]. This particular implementation was chosen because it was simple and easy to modify.

Changes were made to the code, adding file reading and writing capabilities. The code was further improved to work with manageable chunks of the input file, keeping in mind that large files might not fit in RAM at once. The chunk size was established at 256 KBs, meaning that the code will be able to encrypt any file whose size is a multiple of 256 KBs. Each chunk is further subdivided into block of 16 bytes. Each block of 16 bytes is processed by the AES function, producing 16 bytes of output.

Parallelizing with OpenMP was straightforward: The main loop that processes all blocks was identified as a parallel loop using OpenMP directives.

In the CUDA implementation, the code is executed

[2] https://computing.llnl.gov/tutorials/openMP/#Introduction

using a chunk size of 256KBs, a block size of 256 and a computed grid size that is calculated by dividing the chunk size by the number of threads per block, 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.

Two platforms were used for the experiments:

Machine 1:

• Processor: Intel Core i5 M 450 @ 2.4 GHz. This processor has 2 physical cores and can execute 4 threads simultaneously.

• GPU: GeForce GT 330M. Number of multiprocessors: 6. Number of CUDA cores: 48.

Machine 2:

• Processor: Intel Core i7 960 @ 3.2 GHz. This processor has 4 physical cores and can execute 8 threads simultaneously.

• GPU: Quadro FX 1800. Number of multiprocessors: 8. Number of CUDA cores: 64.

The averages for the total execution of the program on machine 1 are shown in Table 1. There are four columns: One for the file size, one for the sequential execution time, one for the execution time with 4 threads (the code was compiled using OpenMP) and the last column contains the total execution time using the GPU.

File Size	I5 Sequential	I5 4Threads	GT 330M GPU
256 KB	833 ms	519 ms	142 ms
512 KB	1639 ms	1015 ms	217 ms
1 MB	3285 ms	2069 ms	335 ms
2 MB	6535 ms	4123 ms	570 ms
4 MB	13123ms	7998 ms	1051 ms
8 MB	26063 ms	15359 ms	2067 ms
16 MB	52367 ms	32138 ms	4033 ms
32 MB	105507 ms	61524 ms	7818 ms
64 MB	208730 ms	125044 ms	16387 ms
128 MB	416993 ms	245823 ms	31088 ms
256 MB	834216 ms	495718 ms	63391 ms

TABLE II. Total execution times for the first platform.

Table 2 contains the times for the same platform for the encryption time only. The reported times were the times for the execution of the portion of the program that carried out the encryption. These times exclude the time required for I/O operations.

File Size	I5 Sequential	I5 4Threads	GT 330M GPU
256 KB	663 ms	342 ms	34 ms
512 KB	1298 ms	671 ms	69 ms
1 MB	2578 ms	1360 ms	138 ms
2 MB	5146 ms	2742 ms	279 ms
4 MB	10342ms	5266 ms	553 ms
8 MB	20358 ms	10219 ms	1111 ms
16 MB	41327 ms	20714 ms	2218 ms
32 MB	83193 ms	41006 ms	4447 ms
64 MB	164309 ms	82095 ms	8872 ms
128 MB	328694 ms	163641 ms	17732 ms
256 MB	657278 ms	327999 ms	35477 ms

TABLE II. Compression times for the first platform

It can be observed that the total execution time is reduced to 60-75% of the original sequential time using four threads on the microprocessor with two cores and to 10-25% using the GPU. When the encryption times are examined, the observed times are around 50% of the original execution time for the OpenMP version and around 6% for the GPU version. The greatest reductions in execution times are obtained with the largest files. The speedup metric (sequential execution time / parallel execution time) is presented in Table 3.

File Size	Speedup OpenMP Total	Speedup GPU Total	Speedup OpenMP Encrypt.	Speedup GPU Encrypt.
256 KB	1.60	5.86	1.93	19.33
512 KB	1.61	7.55	1.93	18.63
1 MB	1.58	9.79	1.89	18.68
2 MB	1.58	11.46	1.87	18.42
4 MB	1.64	12.48	1.96	18.70
8 MB	1.69	12.65	2.00	18.48
16 MB	1.62	12.98	1.99	18.62
32 MB	1.71	13.49	2.02	18.70
64 MB	1.66	12.73	2.00	18.51
128MB	1.69	13.41	2.00	18.53
256MB	1.68	13.15	2.00	18.52

TABLE III. Speedups for the first platform.

It can be observed in Table 3 that the speedups for the encryption portion of the code are very similar across the different file sizes and for the total execution time using OpenMP; whereas, as the file size increases, the speedups improve slightly for the total execution time when using the GPU.

The experiments were repeated on the second platform and the results are shown in tables 4, 5 and 6.

File Size	I7 Sequential	I7 8Threads	Quadro FX 1800 GPU
256 KB	546 ms	350 ms	168 ms
512 KB	1082 ms	664 ms	196 ms
1 MB	2169 ms	1317 ms	267 ms
2 MB	4255 ms	2623 ms	404 ms
4 MB	8386 ms	5216 ms	674 ms
8 MB	17488 ms	10420 ms	1236 ms
16 MB	34393 ms	20964 ms	2358 ms
32 MB	68466 ms	42113 ms	4554 ms
64 MB	136931 ms	83933 ms	9007 ms
128 MB	270202 ms	167718 ms	18347 ms
256 MB	555078 ms	338793 ms	37759 ms

TABLE IV. Total execution times for the second platform.

File Size	I7 Sequential	I7 8Threads	Quadro FX 1800 GPU
256 KB	382 ms	147 ms	21 ms
512 KB	761 ms	259 ms	42 ms
1 MB	1530 ms	508 ms	84 ms
2 MB	2983 ms	1015 ms	168 ms
4 MB	5827 ms	2008 ms	336 ms
8 MB	12309 ms	4030 ms	672 ms
16 MB	24030 ms	8127 ms	1344 ms
32 MB	47842 ms	16450 ms	2688 ms
64 MB	95685 ms	32401 ms	5376 ms
128 MB	187683 ms	64993 ms	10752 ms
256 MB	389140 ms	131613 ms	21511 ms

TABLE V. Compression times for the second platform.

File Size	Speedup OpenMP Total	Speedup GPU Total	Speedup OpenMP Encryption	Speedup GPU Encryption
256 KB	1.55	2.94	2.60	18.22
512 KB	1.63	4.39	2.93	18.13
1 MB	1.64	6.23	3.00	18.21
2 MB	1.62	7.77	2.93	17.75
4 MB	1.60	8.61	2.90	17.34
8 MB	1.67	9.64	3.05	18.31
16 MB	1.64	9.82	2.95	17.87
32 MB	1.62	9.74	2.90	17.79
64 MB	1.63	9.76	2.95	17.79
128 MB	1.61	9.64	2.88	17.45
256 MB	1.63	9.85	2.95	18.09

TABLE VI. Speedups for the second platform.

Several observations can be made: A performance gain can be made by replacing the microprocessor. In the sequential version, the total execution time on the I7 is about 65% of the total execution time on the I5. For the compression portion, the time on the I7 is about 60% of the execution time on the I5. The difference in execution times appears to be proportional to the different in the clock speeds: 2.4 GHz for the I5 and 3.2 GHz for the I7.

The speedups obtained using OpenMP and the GPUs are lower for the total execution times than for the encryption times. This is reasonable as the total execution times include I/O times and I/O operations which are inherently sequential. Thus, it is to be expected that the speedups should be better for the encryption times as encryption is a CPU (or GPU) intensive activity.

The maximum speedups obtained using OpenMP for the total execution times are similar for both platforms. When one examines the speedups obtained for encryption, the speedups top at 2 on the I5 and almost at 3 for the I7. The I5 has two physical cores, so a speedup of two is very good. Finding the reasons why the maximum speedup on the I7 is close to 3 requires further investigation.

The maximum speedups obtained for the total execution time using the GPU are better on the I5 platform than on the I7 platform, even with a lower number of stream processors on the GPU that was used with the I5 platform. For the encryption times, the speedups are similar for both platforms. Recall that is it necessary to copy the data to and from the GPU card and that the speed at which the GPU stream processors are working is slower than the speed of the microprocessor. These are (partial) explanations for the sub-linear speedups, regarding the number of stream processors, on the GPUs.

IV.CONCLUSIONS AND FUTURE WORK

The results suggest that using a GPU to encrypt files with AES is a cost-effective alternative and that the resulting code will run faster than sequential code or code that has been parallelized using OpenMP. If a GPU is not available, using OpenMP is a good alternative that reduces execution times.

It will be interesting to repeat these experiments on other platforms with several microprocessors, microprocessors with larger number of cores and with GPUs with larger numbers of stream processors. The architectural trends in computer systems point in the direction of larger numbers of cores on microprocessors and larger numbers of stream processors on GPUs and it is worth exploring the behavior of this code on these platforms with the help of tools that shed further light in the bottlenecks of the execution of this code on different platforms. Another challenge is to write a version of this code that can take advantage of several GPU cards simultaneously.

REFERENCES

[1] J. Daemen and V. Rijmen, "The Design of Rijndael: AES – The Advanced Encryption Standard", Springer, 2002.

[2] S. Manavski, "CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography", in *Proceedings of the 2007 IEEE International Conference on Signal Processing and communications ICSPC*, Dubai United Arab Emirates , November 24-27, 2007, pp. 65-68.

[3] NVIDIA Corporation, "CUDA Technology", http: //www.nvidia.com/object/cuda_home_new.html. December 2010.

[4] A. De Biagio, A. Barnghi and G. Agosta, "Design of a Parallel AES for Graphics Hardware using the CUDA framework*", in Proceedings of IPDPS 2009* Rome, Italy, May 2009.

[5] W. Bielecki and D. Burak, "Parallelization of the AES Algorithm", in Proceedings of the 4th. WSEAS International Conference on Information Security, Tenerife, Spain, December 16-18, 2005.

[6] J. Coens, Z. Zhou and Y. Li, "Parallelizing AES Encryption", http://www.Journalogy.org/Paper/13610187.aspx

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