ACCELERATING ENCRYPTION/DECRYPTION USING GPU'S FOR AES ALGORITHM

Sanjanaashree P
Department of Information Technology,
Amrita Vishwa Vidyapeetham, Coimbatore.

Abstract— Technology has done a great deal for changing the way we live and do business today. We can see the use of computers from the vegetable shop to large scale businesses. In this fast moving world we need something essential for fast computation. So here comes the Graphics Processing unit for fastest computation through means of its parallel architecture. Along with the popular use of computer, information security has also become one of the problems which need to be solved. Many security issues like the malware authors, information leakage, endangerment and unauthorized exploitation need to be taken into account. To control this, crypto-security is necessary. More Applications started to use Advanced Encryption Standard (AES). However, Since AES on large blocks is computationally intensive and largely byte-parallel. Certain modes of AES are more easily parallelizable and these are ideal candidates for parallelization on GPUs. In this paper, we study the technologies of GPU parallel computing and its optimized design for cryptography. Implementation is done using the CUDA platform. CUDA is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce. The test proves that our approach can accelerate the speed of AES encryption significantly.

Index Terms— Advanced Encryption Standard (AES), Application Programming Interface (API), Compute Unified Device Architecture (CUDA, Data Encryption Standard (DES), Graphics Processing Unit (GPU), and Number of blocks comprising the state (Nb), Operating System (OS), Substitution Box (SBox).

1. INTRODUCTION

Early day's people used paper to record the data. With the evolution of computer this has been changed, we started using computers to store data. Instead of letters we started using email, securing information has also increased with this .Where cryptography has become mandatory .we started using GPUs to accelerate encryption/decryption of the algorithm instead of mutli-core CPUs which is costs high.

A graphics processing unit or GPU (also called visual processing unit or VPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory in such a way so as to accelerate the building of images in a frame buffer intended for output to a display. Years ago, when personal computers found its way into homes, GPU did not exist. All graphics were displayed and manipulated by the CPU.IN 1980, first dedicated GPU was invented. Having dedicated GPU means that all graphics related processing can be offloaded to the graphics card, which can do much better job at rendering graphics. In 1990s with the rise of windows operating system, 2D GPUs capable of high resolution rendering took the market by storm. GPUs have now matured to the point that they can be used for processing other than visual; their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. GPU act as a co-processor with the CPU reducing the overload of CPU by enhancing the processing time.

This paper presents the acceleration of encryption/decryption using GPUs. The algorithm used is AES using CUDA platform.

This paper is organized as follows. Section 2 describes the GPU computing. Section 3 focuses on CUDA Architecture. Section 4 explains AES in brief. The experimental results are given in Section 5. Finally Section 6 ends up with conclusion.

2. GPU COMPUTING

2.1PERFORMANCE OF GPU OVER CPU

Graphics Processing Units are powerful, programmable and highly parallel, tailored for highly parallel operation while a CPU executes programs serially which is where the performance issue comes. GPUs are significantly faster and have more advanced memory interfaces which shift around a lot more data than CPUs. The GPU accelerates applications running on the CPU by offloading some of the compute-intensive and time consuming portions of the code. The rest of the application still runs on the CPU. From a user's perspective, the application runs faster because it is using the massively parallel processing power of the GPU to boost performance. This is known as "heterogeneous" or "hybrid" computing. The application runs its parallel parts on GPU, via kernels. But GPU has instant switching between CPU and GPU. Many threads execute same kernel. GPU. GPU Threads are extremely lightweight when compared to CPU. GPU uses 1000s of threads for efficiency.

2.2 GPU ARCHITECTURE

CPUs and GPUs have fundamentally different design philosophies. The design of the GPUs is forced by the fast growing video game industry that exerts tremendous economic pressure for the ability to perform a massive number of floating-point calculations per video frame in advanced games. The general philosophy for GPU design is to optimize for the execution of massive number of threads.

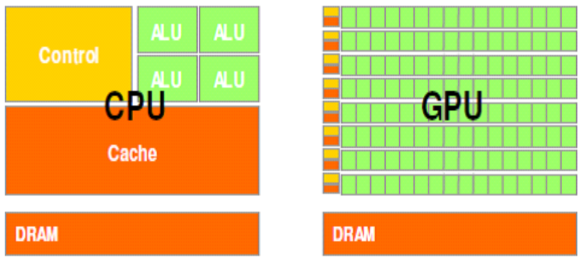

Figure 2.1: CPUs and GPUs different designs

Figure 2.1 shows the architecture of a typical GPU today. It is organized into 16 highly threaded streaming Multiprocessors (SMs). A pair of SMs forms a building block. Each SM has 8 streaming processors (SPs), for a total of 128 (16*8). Each SP has a multiply-add (MAD) unit, and an additional multiply (MUL) unit. Each GPU currently comes with 1.5 megabytes of DRAM. These DRAMs differ from the system memory DIMM DRAMs on the motherboard in that they are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold high-definition video images, and texture information for 3D rendering as in games. But for computing, they function like very high bandwidth off-chip cache, though with somewhat more latency regular cache or system memory. If the chip is programmed properly, the high bandwidth makes up for the large latency.

2.3 GPU COMPUTING WITH COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

CUDA, introduced by NVIDIA in 2006 as new parallel computation architecture, has a new set of instructions and a new parallel programming model. CUDA offers a new software environment which allows the programmers to use C as a programming language for the GPU. The GPU is seen as computational device capable of executing a high number of threads in parallel, when using CUDA. The GPU acts as a coprocessor for the CPU.A major design goal of CUDA is to support heterogeneous computations in a sense that applications are serial parts of an application are executed on the CPU and parallel parts on the GPU. In the computer game industry, in addition to graphics rendering, GPUs are used in calculations (physical effects like debris, smoke, fire, fluids). CUDA provides both a low level API and a higher level API.

3. CUDA ARCHITECTURE

CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).CUDA is a compiler and toolkit for programming NVIDIA GPUs. CUDA API extends the C programming language. It runs on thousands of threads and is a scalable model.

•Support of languages: C, C++, OpenCL.
•Windows, Linux, OS X compatible.

3.1 ADVANTAGES OF CUDA

- Flexibility to the vendor
- hides latency and helps maximize the GPU utilization
- transparent for the programmer
- Automatic thread management
- limited synchronization between threads is provided
- Dead-locks are avoided.

3.2 CUDA PROGRAM STRUCTURE

A CUDA program consists of phases in which the program can be executed either in the host (CPU) or the device (GPU). The NVIDIA C compiler, NVCC separates the two: the phases that don’t exhibit parallelism are implemented in the host code and the phases that have data parallelism are implemented in the device code. The device code is written using C extended with keywords for labeling data-parallel functions, called *kernels.*

The execution starts with host (CPU) execution. When a kernel function is invoked, the execution is moved to the device, where a large number of threads are generated. The threads form a grid. When all the threads of a kernel complete their execution, the corresponding grid terminates, the execution continues on the host until another kernel is invoked.

3.3 THREAD HIERARCHY

Since all threads in a grid execute the same kernel function, they rely on unique coordinates to distinguish themselves from each other and to identify the appropriate portion of the data to process. These threads are organized into a two-level hierarchy using unique coordinates, called block ID and thread ID. The block ID and thread ID appear as built-in variables that are initialized by the runtime system and can be accessed within the kernel functions. When a thread executes the kernel function, references to the block ID and thread ID variables return the appropriate values that form coordinates of the thread.
At the top level of the hierarchy, a grid is organized as a two dimensional array of blocks. The number of blocks in each dimension is specified by the first special parameter given at the kernel launch. For the purpose of our discussions, we will refer to the special parameters that specify the number of blocks in each dimension as a struct variable gridDim, with gridDim.x specifying the number of blocks in the x dimension and gridDim.y the y dimension. The values of gridDim.x and gridDim.y can be anywhere between 1 and 65,536. The values of gridDim.x and gridDim.y can be supplied by runtime

variables at kernel launch time. Once a kernel is launched, its dimensions cannot change in the current CUDA run-time implementation. All threads in a block share the same blockId values. The blockId.x value ranges between 0 and gridDim.x-1 and the blockId.y value between 0 and gridDim.y-1.

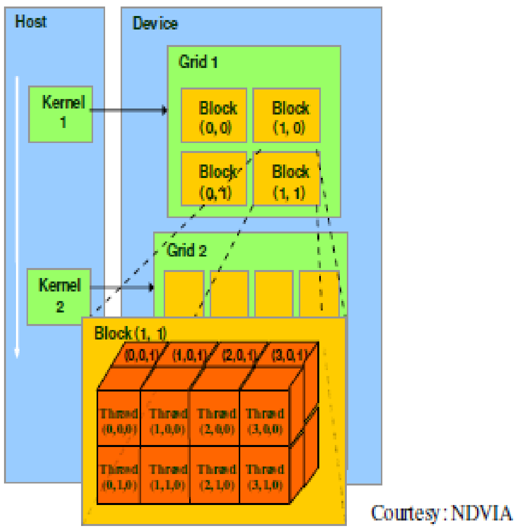

Figure 3.1: Example of CUDA Thread Organization

Figure 3.1 illustrates the organization of threads within a block. Since all blocks within a grid have the same dimensions, we only need to show one of them. In this example, each block is organized into 4*2*2 arrays of threads. Figure 4.1 expands block (1, 1) by showing this organization of all 16 threads in block (1, 1). For example, thread (2, 1, 0) has its threadId.x=2, threadId.y=1, and threadId.z=0. In this example, we have 4 blocks of 16 threads each, with a grand total of 64 threads in the grid. Typical CUDA grids contain thousands to millions of threads.

5. IMPLEMENTATION OF GPU

Since the objective was to fairly compare implementations of AES on GPU and CPU, we ported to GPU the open source CPU implementation. Implementing a cryptographic algorithm to run on a graphic processor is justified because, theoretically, it is cheaper to use a GPU as a co-processor to relieve the CPU from intense computing tasks, than purchasing a dedicated cryptographic co-processor that is more expensive. Adapting an algorithm, that has a computational complexity that consists in simple byte operations (AND, OR, XOR, Shifting, 32 bit adding etc.), on a video card which is designed to run complex operations and floating point operations, is the starting point of our scientific research. The source has two main entries: Encrypt () and Decrypt (). These functions take in a plain text / cipher text 128-bit source block, a 1408-bit expanded key and output an encrypted/decrypted 128-bit block. At a higher level, Encrypt () and Decrypt () functions take in a 128-bit key and a variable length message. They split the message into 128-bit blocks, appropriately pad the block when the message size is a multiple of 128 bits, and pass the blocks to the Encrypt () and Decrypt () functions. This naturally leads to a highly parallel GPU implementation. The GPU threads perform the Encrypt () and Decrypt () functions in parallel. Each thread works on a subset of the data, so there are no dependencies between threads. This assumes that the cipher is used in parallel-friendly modes.

The advantages of using a graphic card that supports CUDA environment, reside in the fact that CUDA can use shifting operations, offers flexibility on memory access, data can be defined directly in the GPU without the need of an extra copy operation from the CPU.

4. AES

Advanced Encryption Standard (AES) is a variant of Rijndael cipher algorithm, a symmetric block cipher which translates the plaintext into cipher text in blocks. This algorithm has the fixed input block size of 128 bits and the key size of 128, 192, 256 bits.

The input – the array of bytes A_0, A_1 ... A_{15} is copied into the state array as shown in figure 4.1.

A0	A4	A8	A12
A1	A5	A9	A13
A2	A6	A10	A14
A3	A7	A11	A15

Figure 4.1 The State array

The algorithm's operations are performed on a two-dimensional array of bytes called the State**.** The State consists of four rows of bytes, each containing nb bytes calculated by taking ratio of the total number of data (input) bits and 32. (128/32=4). The Cipher (encrypt) or Inverse Cipher (decrypt) operations are then conducted on this State array, after which the final value is copied to the output array. The AES algorithm, unlike DES does not form a fiestal network, where encryption and decryption processes are similar. AES processes data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits.

The Encryption and decryption process consists of a number of different transformations applied consecutively over the data block bits, in a fixed number of iterations, called rounds (Nr), which depends on the length of the key used. The Middle round will undergo Nr-1 iterations.

The AES parameters are given in the table 4.1 below

Key Size (Bytes/bits)	16/128	24/192	32/256
Plaintext block size (Bytes/bits)	16/128	16/128	16/128
Number of Rounds	10	12	14
Round key size (Bytes/bits)	16/128	16/128	16/128
Expanded key size (Bytes/bits)	176/1408	208/1664	240/1920

Table 4.1 AES Parameter

The Encryption process consists of 4 phases. They are:

1. Key Expansion
2. Initial Round
   a. AddRoundKey
3. Middle Rounds
   a. Substitute Bytes
   b. Shift Rows
   c. Mix Columns
   d. Add Round Key
4. Final Round
   a. Substitute Bytes
   b. Shift Rows
   c. Add Round Key

The phases of Decryption are (for the particular cipher text):

1. Key expansion
2. Initial round
   a. Add Round Key
3. Middle Round
   a. Inverse Shift Rows
   b. Inverse Substitute Bytes
   c. Add Round Key
   d. Inverse Mix Columns
4. Final Round
   a. Inverse Shift Rows
   b. Inverse Substitute Bytes
   c. Add Round Key

The algorithm for encryption/decryption process is shown in figure 3.1.

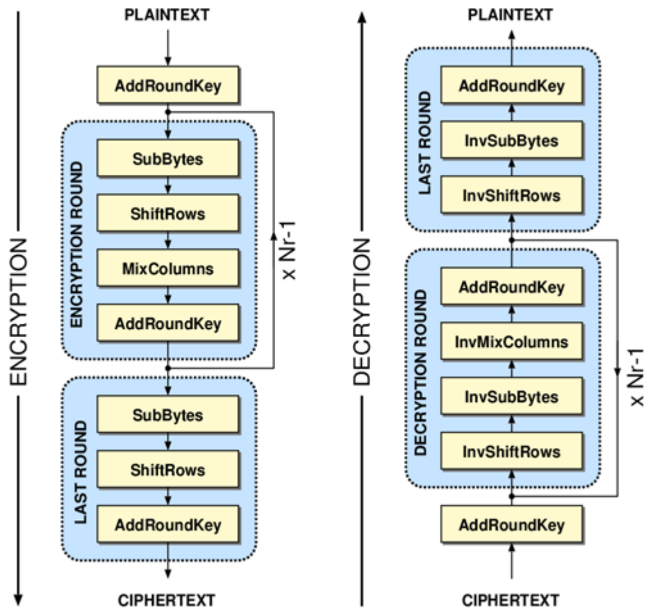

Figure 4.1: AES Algorithm

1. Key Expansion:

   Key expansion takes the input key of 128, 192 or 256 bits and produces an expanded key for use in the subsequent stages. The expanded key's size is related to the number of rounds to be performed. For 128-bit keys, the expanded key size is 352 bits. For 192 and 256 bit keys, the expanded key size is 624 and 960 bits. It is the expanded key that is used in subsequent phases of the algorithm. During each round, a different portion of the expanded key is used in the AddRoundKey step.

2. AddRoundKey:

   During this stage of the algorithm, the message is combined with the state using the appropriate portion of the expanded key.

3. Sub Bytes:

   During this stage, the block is modified by using an 8-bit substitution, or SBox. This is a non-linear transformation used to help avoid attacks based on algebraic manipulation.

4. Shift Rows:

   This stage of the algorithm shifts cyclically shifts the bytes of the block by certain offsets. Blocks of 128 and 192 bits leave the first 32-bits alone, but shift the subsequent 32-bit rows of data by 1, 2 and 3 bytes respectively.

5. Mix Columns:

   This stage takes the four bytes of each column and applies a linear transformation to the data. The column is multiplied by the coefficient polynomial *c(x) = 3x3+x2+x+2 (modulo x4+1).* This step, in conjunction with the *Shift Rows* step, provides diffusion in the original message, spreading out any non-uniform patterns. At the end of the algorithm, the original message is encrypted. To decrypt the cipher text, the algorithm is essentially run in reverse, however, if the key used for *Key Expansion* is not known, a brute-force attack on the cipher could take thousands of years.

The decryption implementation results are similar to the encryption implementation. The key expansion module is modified in the reverse order. In which last round key is treated as the first round and decreasing order follows

5. TESTING AND EVALUATION METHODOLOGY

The first thing to get right is the correctness. To ensure that we did not alter the functionality of the algorithm, both the GPU and the CPU implementations were tested as follows:

- ➢ Make sure that encrypting known plaintext gives back known cipher text.
- ➢ Make sure that decrypting encrypted string gives back the original plaintext.

After the correctness was verified, we evaluated the performance. Since the purpose of any cipher is to quickly encrypt incoming data, the performance metric we picked is the run-time of the algorithm. This includes key generation and moving data between GPU and CPU memories.

6. RESULT

The results were achieved by running a random data set through the encryption and decryption modules 5 times. Since the code path is not data dependent, the data itself

S. No	Input File size (KB)	CPU Encryption time (ms)	CPU Decryption time (ms)	GPU Encryption time (ms)	GPU Decryption time (ms)
1.	23	110.000	32720.000	30.6846	34.628
2.	58	420.00	20500	30.786	34.551
3.	115	770.000	440128	32.1342	36.0249
4.	457	4010.000	6347.000	39.4578	44.113
5.	914	8870.00	10347.000	78.4082	88.131

7. CONCLUSION AND FUTURE WORK

In this paper, an efficient way to enhance the encryption/ decryption using GPU's is proposed. This report presents the most efficient, currently known approaches in encryption and decryption of text with AES on programmable graphics processing units, achieving up to a great speed on a comparable CPU. If the amount of data is large, the encryption/decryption time required is greatly reduced, if it runs on a graphics processing environment.

Future work will include efficient implementations of other common symmetric and asymmetric algorithms. GPU implementations of hashing and public key algorithms may also be implemented, in order to create a complete cryptographic framework accelerated by the GPU.

8. REFERENCES

[1]*Nation Institute of Standards and Technology (NIST), Data Encryption Standard (DES), National Technical Information Service, Sprinfgield, VA 22161, Oct. 1999.*

[2]"*CUDA_C_Programming_Guide_version4.0*", 5/6/2011

[3]*William Stallings, "Cryptography and Network Security"*

[4] Alexander Zibula, "General Purpose Computation on Graphics Processing Units (GPGPU) using CUDA", Winter Term 2009/1010.

[5] http://www.cs.wm.edu

[6] http://www .gpgpu.org

[7] www.nvidia.com

[8] graphics.cs.ucf.edu

[9] http://www.d.umn.edu

[10] en.wikipedia.org

[11]www.ogf.org/OGF25/materials/1605/CUDA_Programming.pdf

[12]Almasi, G.S. and A. Gottlieb (1989). Highly Parallel Computing. Benjamin-Cummings publishers, Redwood City, CA

[13]http://www.s3graphics.com/en/products/index.aspx

[14]http://www.via.com.tw/en/products/graphics

[15]http://www.matrox.com/graphics/en/products/graphics_cards

[16]Bradley Sanford. "Integrated Graphics Solutions for Graphics-Intensive Applications"

[17]http://www.nvnews.net/vbulletin/archive/index.php/t-12714.html