A CANONICAL IMPLEMENTATION OF THE ADVANCED ENCRYPTION STANDARD ON THE GRAPHICS PROCESSING UNIT

Nick Pilkington[1], Barry Irwin[2]

Rhodes University
Department of Computer Science
South Africa

[1]n.pilkington@ru.ac.za, [2]b.irwin@ru.ac.za

ABSTRACT

This paper will present an implementation of the Advanced Encryption Standard (AES) on the graphics processing unit (GPU). It investigates the ease of implementation from first principles and the difficulties encountered. It also presents a performance analysis to evaluate if the GPU is a viable option for a cryptographics platform. The AES implementation is found to yield orders of maginitude increased performance when compared to CPU based implementations. Although the implementation introduces complications, these are quickly becoming mitigated by the growing accessibility provided by general programming on graphics processing units (GPGPU) frameworks like NVIDIA's Compute Uniform Device Architechture (CUDA) and AMD/ATI's Close to Metal (CTM).

KEY WORDS

Cryptography, AES, GPU, GPGPU, Offload, Rijndael, OpenGL, CG

A CANONICAL IMPLEMENTATION OF THE ADVANCED ENCRYPTION STANDARD ON THE GRAPHICS PROCESSING UNIT

1 INTRODUCTION

General programming on graphics processing units (GPGPU) refers to non-graphics related programming operations being performed on the graphics processing unit (GPU) rather that on the CPU. This programming paradign opens up many possibilities for increased performance by utilising the specialised processing nature of the GPU. There are currently two frameworks available to program GPUs, namely Compute Uniform Device Architechture (CUDA) [3] from NVIDIA and Close to Metal (CTM) [2] from AMD/ATI. These frameworks currently only support their native GPU architechture and as a result are not portable across all hardware. GPGPU can, however, be acheived from first principles in a general way that allows the code to be executed on a wide range of different hardware configurations. This method will be explained in section **??**. This approach requires that manufacturer specfic caveats and optimizations cannot be taken advantage of, since a canonical implementation is being presented general applicability is more important that specifically optimised performance. There are a number of popular cryptographic algorithms in use in computing today including AES [6], Triple DES [9], and Blowfish [12]. Rijndael (AES) was selected for sample implemetation as it is the FIPS accepted Advanced Encyption Standard [6]. This paper seek to investigate, in detail, the implementation of AES on a GPU, more specifically it will be concerned purely with the encryption process as the decryption process is similar. It also presents a performance analysis of the implementation in comparison to CPU based implementations and discussion to substantiate the results. Finally sample applications and proposed future derivative works are suggested.

2 AES ENCRYPTION

Advanced Encryption Standard (AES) is a symmetric key cryptographic algorithm also known as Rijndael designed by Vincent Rijmen and Joan Daemen in 1998, it was subsequently adopted as the Advanced Encryption Standard in 2002. AES is a block cipher which means that is encrypts data in

Table 1: Key-Block-Round Combinations

Type	Key Length	Block Size	Number of Rounds
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Algorithm 1 AES Encryption Pseudocode

```
KeyExpansion
Initial Round
AddRoundKey

for N = 1 to Rounds-1
  SubBytes
  ShiftRows
  MixColumns
  AddRoundKey

SubBytes
ShiftRows
AddRoundKey
```

finite blocks as opposed to operating on a stream like Trivium [7]. The block of data to be encrypted is termed the state. In AES the state is a 4x4 matrix of bytes (figure 1). The state paired with an encryption key of a certain length form the inputs for the AES algorithm. AES is comprised of four different stages, which together represent a single round. Each stage performs some operations on the current state. The number of rounds varies with different implementations of AES (table 1). This paper implements AES-128. Algorithm 1 gives the pseudocode for the AES Encryption process and a depiction of an encryption stage is shown in figure 2. It should be noted that the final round of the encryption process varies from the rest as the mix columns stage is ommitted.

Figure 1: AES Encryption State

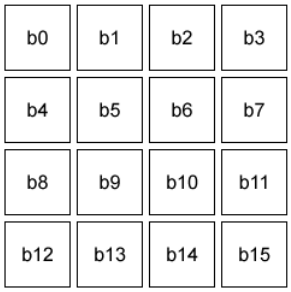
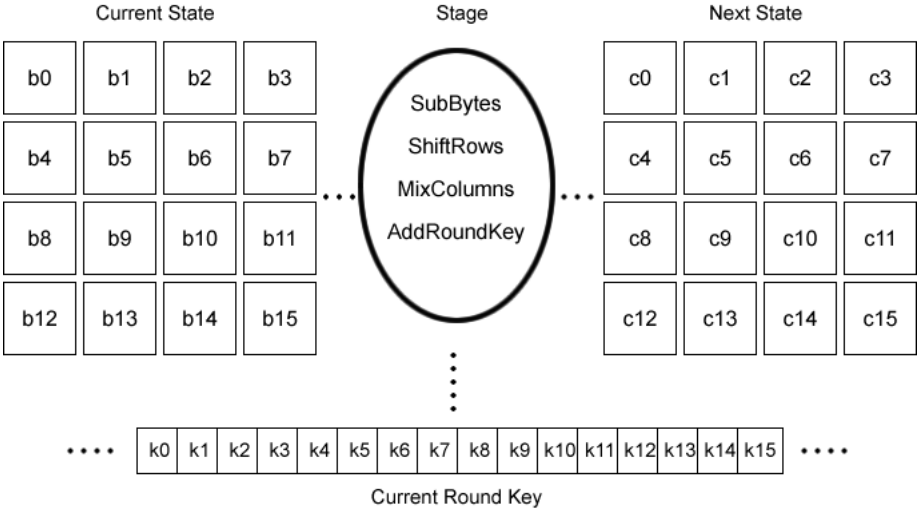


Figure 2: AES Encryption Round



3 GENERAL PROGRAMMING ON GRAPHICS PROCESSING UNITS

Until the third generation of GPUs was released in 2001, GPUs were not programmable and were merely configurable to a limited degree. The advent of this generation exposed areas of the graphics pipeline to programmers allowing them execute custom code on the GPU. This is achieved through pixel shaders which execute once on each rendered pixel in the viewport. Colour, vertex and normal data does not need to be interpreted geometrically but are in fact just arrays of numbers. Rendering an NxN quad onto the screen call the execution of any mapped pixel shaders on each of the N^2 elements of the quad and their output value overwrites the value currently at that position in the quad. Once a problem has been formulated in terms of shaders and rendering it can be mapped and solved on the GPU. A thorough treatise of the basics of GPGPU and how it can be achieved from first principles is given in [10].

4 APPROACH

At the time of writing there are three different shader languages available for programmable shaders: HLSL [4], GLSlang [11] and Cg [1]. Cg and the OpenGL API were used for this implementation. The basis for the AES encryption algorithm is rooted deeply in algebra and the technical specifics of the algorithm [6] have been omitted from this paper. This section will present a high level view of each of the four stages of the encryption process and how each was modelled on the GPU. Each stage of the encryption process was implemented in a separate shader. The four shaders were each executed in order ten times on the initial state to encrypt the data.

4.1 Encryption Stages

Key Expansion

The first stage in the AES encryption process is to expand the key to ten times its original size such that there is a key for each round of the algorithm [6]. This is a pre-process to the encryption process and as such the expanded key was precomputed.

Substitute Bytes

The substitute bytes step of the algorithm replaces each byte in the current state with a corresponding byte using an 8-bit Sbox [6]. The Sbox represents a non-linear transformation. This transformation can be represented in matrix form as:

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

The actual transformations to generate the Sbox do not need to be computed explicitly. As the values are constant for a given initial b vector. There are 256 different different b vectors and as a result 256 corresponding Sbox transformed values. The operation can be viewed as a table look up. Thus the resulting look up values for all 2^8 initial b vectors can be computed and stored in a 16x16 texture. The GPU indexes into this texture using the current byte in the state and received the Sbox transformed value, which is then written into its place.

Shift Rows

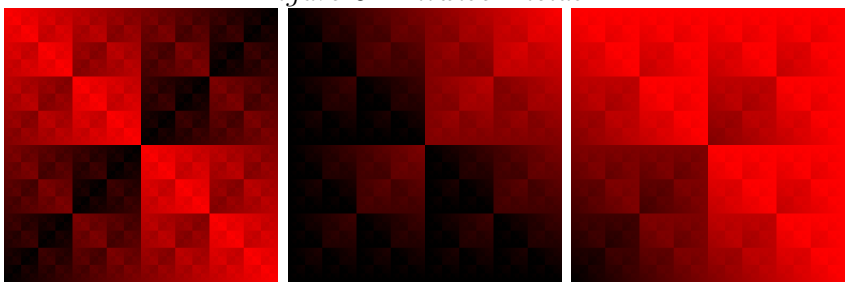
The shift rows operation cycles the bytes in each row cyclically left. The first row is not shifted, the second row is shifted one position, the third row two positions and finally the forth row three positions [6]. The shift operation is performed on the GPU by offsetting the current fragment shaders texture coordinates based on its row and performing a single texture look up on its own texture.

Mix Columns

The mix columns stage operates on each of the four columns of the state. Each column of the state is representative of a four-term polynomial over the Galois field GF(2^8) [6], this polynomial is multiplied modulo $x^4 + 1$ with the fixed polynomial $a(x)$, given by:

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x^1 + \{02\}$$

Figure 3: Bitwise Fields



(a) XOR. (b) AND. (c) OR.

This can be written as a logical bitwise matrix multiplication in the form $s(x)' = a(x) \oplus s(x)$. Shader languages like Cg do not have support for logical operations [8]. Although reservation has been made for the corresponding symbols &, | and ^ [8], they had not been implemented at time of writing. This makes a seemingly trivial task like a logical XOR impossible to perform without some other mechanism in place. In order to provide this functionality, a look up table of values was precomputed and stored in a texture. A 256x256 texture was used and its red, green and blue colour channels corresponded to the XOR, OR and AND operations respectively. These are all binary operations and the x and y indices of the texture correspond to the operands and the values stored in each channel to the resulting binary operations value. When a bitwise operation needed to be performed the two operands were scaled to the texture coordinate range of $[0.0 \ldots 1.0]$ and a dependent texture look up was performed on the texture. The resulting colour channel could then be read to give the XOR, OR or AND of the operands respectively. Figure 3 depicts the red, green and blue channels respectively. The limitation of this implementation is the range of values of the operands. A single 256x256 texture was used and since AES operates within this range of values, these constraints were not problematic.

Add Round Key

The add round key stage XORs the current key with the state. With bitwise operations the XOR operation can be implemented as the XOR between the current byte of the state and the corresponding byte of the key.

4.2 Algorithm Execution

With each of the operations of AES implemented, the whole encryption process can be achieve by encoding the initial state and expanded round key into textures. A 4x4 pixel quad was then rendered to the screen with the initial sub bytes fragment shader bound. This produced the output for the first stage of the AES encryption. The contents of the frame buffer were then copied back into the texture using a render to texture feedback mechanism after which the shift rows fragment shader was loaded and another 4x4 pixel quad rendered. This process was replicated for the mix columns and add round key stages to yield one iteration of the AES encryption. Since ten iterations were required, the whole process is preformed 10 times giving the encrypted state. Care was taken to treat the final iteration correctly, since the add round key operation does not take place here [6].

4.3 State Tiling

GPU shader operations take place in parallel [10]. Since the only data dependence in AES encryption is that the stages of the encryption take place in order there is no reason to limit processing to a single state per rendering if more than one can be represented. If a single 4x4 texture were used to represent the current state it would utilise less than 0.0016% of a 1024x1024 view space. For this reason $65,536$ state were tiled across the view port to enable complete utilisation of the view space as in figure 4.

5 TESTING CONFIGURATION

The GPU implementation was benchmarked for speed and accuracy. Its speed was measured by how much data it could encrypt per second. This amount was measured as the average amount of data encrypted per second over a 60 second run. All runs were executed on the machine specification detailed in table 2. The encryptions were performed on deterministically random data. The results of each stage of the AES encryption process were cross validated against OpenSSL's AES implementation [5] to ensure correctness.

Figure 4: State Tiling in the View port

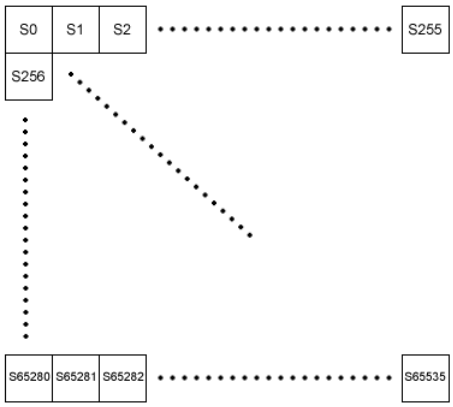


Table 2: Test Platform Configuration

Category	Details
Processor	Intel Core 2 Duo (1.86Ghz)
Memory	2048MB DDR2 (400Mhz)
Graphics	NVIDIA GeForce 7900 GT (256MB)
Mainboard	Intel Corporation Q965
Hard drive	80GB SATA
Operating System	Windows XP Service Pack 2

Table 3: AES Encryption Rate

Type	Encryptions per Second (16-byte state)
CPU	7254.25
GPU	25449.65

Table 4: Maximum Encryption Rate

Type	Encryption Rate (Mb/s)
CPU	2.32
GPU	12.00

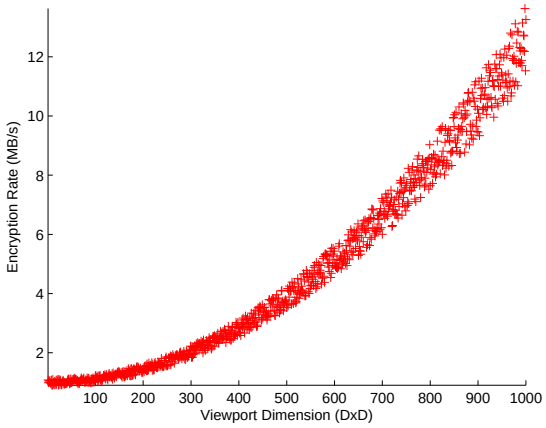
6 RESULTS

Tables 3 and 4 show the average number of complete AES encryptions performed on both the CPU and GPU and the average encryption rate.

7 PERFORMANCE ANALYSIS

Considering the results of both the CPU and GPU implementations in table 4 the GPU outperforms the CPU by 5.17 times. It is important to gain a deeper understanding of what causes this vast performance increase. In general GPUs are slower than CPUs on the clock speed basis the performance gain is not due to this. Figure 5 shows the results of the GPU implementation of AES encryption with increasing numbers of states tiled into the the view port. It may seem retrogressive to look at performance results with smaller tiling, but it is instructive in understanding how the results in table 3 are achieved. It can be seen from figure 5 that data volume is not bottle-necking the encryption process, as when more data is tiled into the view port the encryption rate increases. GPUs perform well on large streams on uniform data and this statement is mirrored by the graph. Using a view port of 1024x1024 and tiling the states, as detailed in subsection 4.3, allowed all of them to be encrypted in parallel. This allowed for far more data to be encrypted per rendering. A CPU cannot do this, thus gains nothing from being passed more concurrent data as it all needs to be processed sequentially anyway. Figure 5 implies that more blocks will yield even higher encryption rates. There is a limit to the size of the renderable surface while maintaining a 1:1 aspect ratio. This problem can be circumvented in a number of ways

Figure 5: AES Encryption Rate



but these methods are not general and as a result is one of the advantages of GPGPU frameworks like CUDA and CTM.

8 CONCLUSION

The results in section 6 show that high performance encryption is possible on the GPU. The unoptimized implementation used exhibited large performance increases over the CPU implementation. The results show that this performance increase is due to the parallel processing nature of the GPU and its ability to operate on more than one data item concurrently. By tiling more that one state into the view port the GPU is able to take advantage of the per-stage parallelism of AES and yield large performance gains. As mentioned on the outset the implementation was restricted to a canonical method such that it could illustrate a proof of concept that is invariable across different hardware configurations. In recent months a large emphasis has been placed on the computing power of GPUs and as a result general computing framework have been released from both NVIDIA and ATI. These allow for more fine grained thread control of the execution of the code which is beyond the OpenGL implementation presented here. The advantage of the implementation presented here is that is that it achieves the same ends as an implementation on CUDA or CTM would but without the abstraction layer that masks the finer implementation details. The developments in CUDA and CTM have largely eclipsed this kind of GPGPU development,

however it still remains important as a foundation of understanding. This implementation paves the way for implementing further mainstream cryptographic algorithms like 3DES and Blowfish on the GPU and making similar performance analyses.

References

[1] The cg language. Tech. rep., NVIDIA Corporation (Available Online: http://developer.nvidia.com/).

[2] Close to the metal project, amd/ati. Available Online: http://sourceforge.net/projects/amdctm/.

[3] Cuda, nvidia corporation. Available Online: http://www.nvidia.com/.

[4] High level shading language, microsoft corporation. Available Online: http://msdn.microsoft.com/.

[5] The openssl project. Available Online: http://www.openssl.org/.

[6] *Federal Information Processing Standards Publication 197, ADVANCED ENCRYPTION STANDARD (AES).* 2001.

[7] Christophe De Cannière, B. P. Trivium specifications. Available Online: http://www.ecrypt.eu.org/stream/ciphers/trivium/trivium.pdf.

[8] Fernando, R., and Kilgard, M. *The Cg Tutorial*. Addison-Wesley Professional, 2003.

[9] Kammer, R. G. *Federal Information Processing Standards Publication, DATA ENCRYPTION STANDARD (DES).* U.S. Department of Commerce/National Institute of Standards and Technology, 1999.

[10] Pilkington, N. An investigation into general processing on graphics processing units [unpublished]. Department of Computer Science, Rhodes University, South Africa.

[11] Rost, R. The opengl shading language. Available Online: http://www.opengl.org/.

[12] Schneider, B. The blowfish encryption algorithm. http://www.schneier.com/blowfish.html.