Implementing AES on GPU
Final Report

Michael Kipper, Joshua Slavkin, Dmitry Denisenko
University of Toronto
April 20, 2009

Introduction

Encryption and decryption are increasingly important. In order to protect the security of individuals, corporations and even governments, information needs to be secured against potential threats. The basis of encryption's security is its robustness under a brute force attack as the key space of AES-128 is 3.4×10^{38} keys in size. Even at a sustained rate of 1 Tkeys/second, it would take 1019 years to exhaust the key space.

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. Also, using GPU resources as co-processors allows better utilization of the central processing unit. Implementing the AES algorithm on the NVidia GPU has provided a 14.5x speedup over a similar implementation on the CPU.

Related Work

The first paper to deal with cryptography on a graphics card was [1]. It used experimental graphics engine called PixelFlow, consisting of a large number of simple 8-bit processors running at 100 MHz. The authors researched cracking UNIX password cipher and were able to crack most of UNIX passwords in 2-3 days.

Early commercial graphics card were poorly suited for cryptography work due to lack of programmability and integer processing support. That changed in recent years with latest graphics card made by NVidia, and later AMD. A number of recent papers dealt specifically with implementing AES on NVidia's GPUs [2, 3, 4]. All of these papers concentrate on implementing the core algorithm in GPU, leaving key scheduling to the CPU. Also, all the papers concentrate on the parallelizable modes of AES (ECB and CTR). The clearest statement of results is contained in [4]. Using 256-bit keys, the authors achieved 15.7x speed up over the CPU (and 5.4x speed up if memory transfers to and from GPU are included) for OpenSSL library. These speed ups were achieved for the largest measured input size, 8 MB. Smaller file sizes and 128-bit keys saw smaller speed ups. It is not clear if the CPU implementation used SSE instruction set for CPU parallelism.

Description of AES

The AES cipher is the official encryption standard adopted by the US government. The core of the algorithm is based upon the Rijndael cipher developed by Joan Daemen and Vincent Rijmen. The AES algorithm was announced as the new US government standard in November of 2001 and adopted as the standard for US encryption in May of 2002.

The AES cipher comes in 3 key sizes, each working on a fixed block size of 128 bits. The 3 flavours of the cipher are AES-128, AES-192 and AES-256 where the key sizes are 128, 192 and 256-bits respectively.

The AES algorithm consists of 4 phases with the number of rounds to perform being a function of the key size. The phases are:

1. Key Expansion
2. Initial Round
    a. AddRoundKey
3. Middle Rounds
    a. SubBytes
    b. ShiftRows
    c. MixColumns
    d. AddRoundKey
4. Final Round

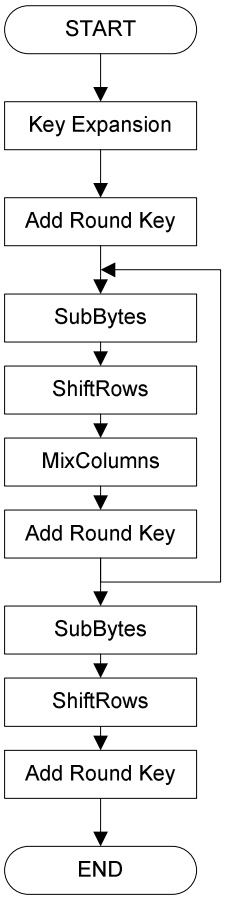

Figure 1 - AES Algorithm

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, there are 10 rounds and the expanded key size is 352 bits. For 192 and 256 bit keys, the number or rounds increases to 12 and 14 rounds respectively with an overall expanded key size of 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.

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

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

ShiftRows: 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.

MixColumns: 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 $c(x) = 3x^3+x^2+x+2$ *(modulo* x^4+1*).* This step, in conjunction with the *ShiftRows* 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 ciphertext, the algorithm is essentially run in reverse, however, if the key used for *KeyExpansion* is not known, a brute-force attack on the cipher could take thousands of years.

Modes of Encryption

AES is a block-based encryption standard and translates the plaintext into ciphertext in blocks. There are different modes under which the encryption can take where some modes are inherently more secure and some lend themselves more to parallelism.

Electronic Codebook (ECB): In this mode, each block is encrypted with an identical key and there is no serial dependence between the blocks. While this leads to extensive parallelism, the large scale structures in the plaintext are preserved.

Cipher-block Chaining (CBC): In this mode, the plaintext block is XOR'ed with the ciphertext produced from the previous block prior to entering the block cipher encryption. This eliminates the large scale structures in the ciphertext but the serial dependence is unsuitable for parallelization.

Propagating Cipher-block Chaining (PBC): In this mode, the plaintext and the ciphertext from the previous block are XOR'ed with the plaintext from the current block. As with CBC, the large scale structures of the plaintext are eliminated from the ciphertext, but again the serial nature of the algorithm does not lend itself to parallelization.

Cipher Feedback (CFB): In this mode, the input to the block cipher encryption is the ciphertext from the previous block. The output of the block cipher encryption is XORed with the plaintext to produce the ciphertext for the block. As with CBC and PCB, this mode is unsuitable for massive parallelism.

Counter (CTR): This mode is very similar to CFB in that the output of the block cipher encryption is XORed with the plaintext to produce the ciphertext. It differs in that the inputs to the block cipher encryption are not dependent on the previous block. Instead of chaining the result from the previous block, a counter value is used as an input along with the key. This mode allows for a high level of parallelization as the inputs to a block are independent of the other blocks. By using a counter, the large scale structure that may have been present in the original plaintext is diminished. It is this mode that we used in our parallelization efforts on the GPU.

Only the more popular modes of encryption were described here. For a more complete list, see the Wikipedia article on this topic [7].

Explanation of GPU Algorithm and implementation

Since the objective was to fairly compare implementations of AES on GPU and CPU, we ported to GPU the open source CPU implementation from [5]. This implementation is byte-oriented which is suitable for a byte-level parallelism on GPU. The source has two main entries: Encrypt() and Decrypt(). These functions take in a plaintext / ciphertext 128-bit source block, a 1408-bit expanded key and output an encrypted/decrypted 128-bit block. At a higher level, the 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 (ECB or CTR).

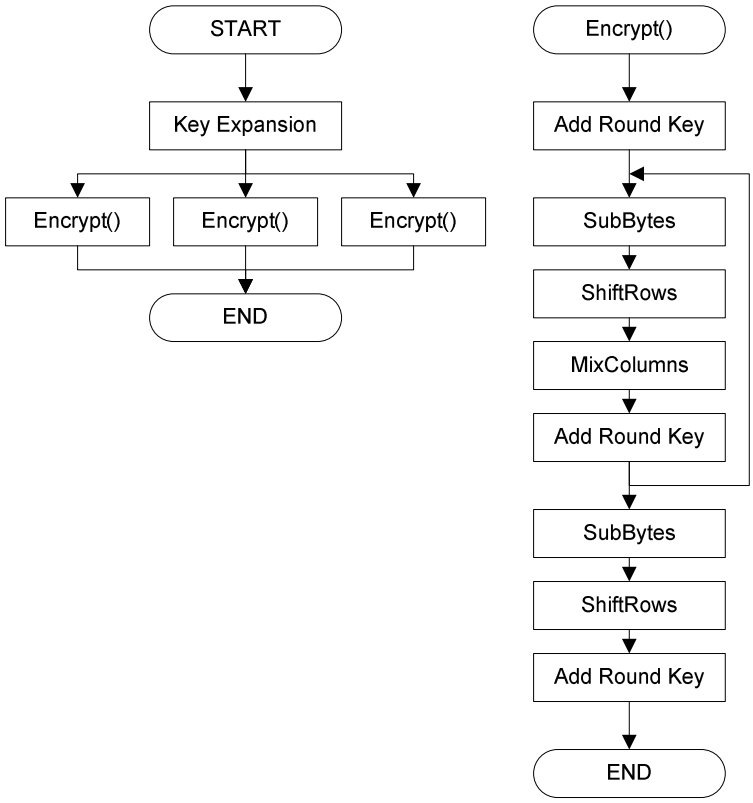

Figure 2 - Parallel AES Algorithm

We perform the Key Expansion on the CPU and pass the expanded key into the GPU. This is because the operation is inherently serial and applies to all threads equally. There is no real reason to duplicate this effort on the GPU.

To optimize global memory access, it is necessary to coalesce data accesses together to allow the GPU to perform wide memory reads. To achieve this, the thread execution is broken into three stages:

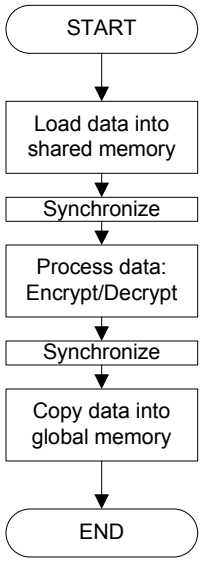

Figure 3 - Thread Execution Stages

Since all the threads access the data from global memory before processing the data, it is easy to order the accesses such that the most efficient usage or the memory bus is achieved. This is done in a data driven way so any changes to the grid or block structure will not require any code updates. Once the data is loaded into shared memory, it can be accessed hundreds of times faster than global memory. Since each thread loads a set of data from global memory that it may not operate on, all threads must synchronize before the shared memory can be used. The same goes for writing the shared memory back to global memory.

Another optimization that is useful is the use of constant memory. The CPU implementation we ported uses several 16x16-byte tables to perform lookups. Since these tables are constant and common between threads, we can load them into the constant memory of the GPU. This memory is cache-backed and the cache can accommodate all of the table data. Once the data is in the cache, there is little penalty in accessing it again.

Note that it would be trivial to apply the same parallelization scheme to the CPU implementation to fairly compare the GPU version with the CPU. However, the CPU cannot come close to the number of active threads the GPU runs concurrently. The CPU does not have to transfer the data from main memory so it likely has an advantage at low message sizes. We did not implement or benchmark the CPU's parallel behaviour.

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. Plaintext / cipher text pair was taken from the "Advanced Encryption Standard" article of the Wikipedia [6].
- 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 overall run-time of the algorithm. This includes key generation and moving data between GPU and CPU memories.

To compare GPU and CPU implementations, we divided the GPU run-time by the CPU run-time. Even though this ratio is good indicator of what benefits the GPU can bring, it should be used with caution when comparing results from different publications. Variations in hardware, relative newness of GPU and CPU chips, CPU implementations, and CPU technologies (e.g. if SSE instruction set is used and which version) will all lead to changes in the GPU / CPU run-time ratio.

Results

The results were achieved by running a random data set through the encryption and decryption modules 200 times. Since the code path is not data dependent, the data itself

does not affect the performance output. The output of the decryption phase was then compared to the input to ensure correctness. This verification time was not included in the elapsed time. The message size begins at 16-bytes, the smallest block unit, and doubles in size until 64MB. This allows us to see a profile of the algorithm as the problem size increases.

Since the CPU implementation runs serially, we expect to see a linear increase in runtime as the message size increases. The parallel GPU runtime should increase linearly as well, however, there may be a step-like behaviour as the message size crosses the boundaries of the maximum work unit.

Table 1 - Results

Message Size (KB)	GPU Time (s)	CPU Time (s)	Speedup
16	0.05	n/a	n/a
32	0.05	n/a	n/a
64	0.06	n/a	n/a
128	0.08	n/a	n/a
256	0.11	n/a	n/a
512	0.14	0.01	0.1
1024	0.14	0.01	0.1
2048	0.16	0.03	0.2
4096	0.30	0.05	0.2
8192	0.31	0.08	0.3
16384	0.32	0.17	0.5
32768	0.36	0.38	1.1
65536	0.37	0.74	2.0
131072	0.56	1.50	2.7
262144	0.74	2.95	4.0
524288	1.14	5.86	5.1
1048576	2.02	11.70	5.8
2097152	3.23	23.50	7.3
4194304	4.88	47.09	9.6
8388608	8.29	94.07	11.3
16777216	14.71	188.23	12.8
33554432	27.35	376.54	13.8
67108864	52.86	753.32	14.3

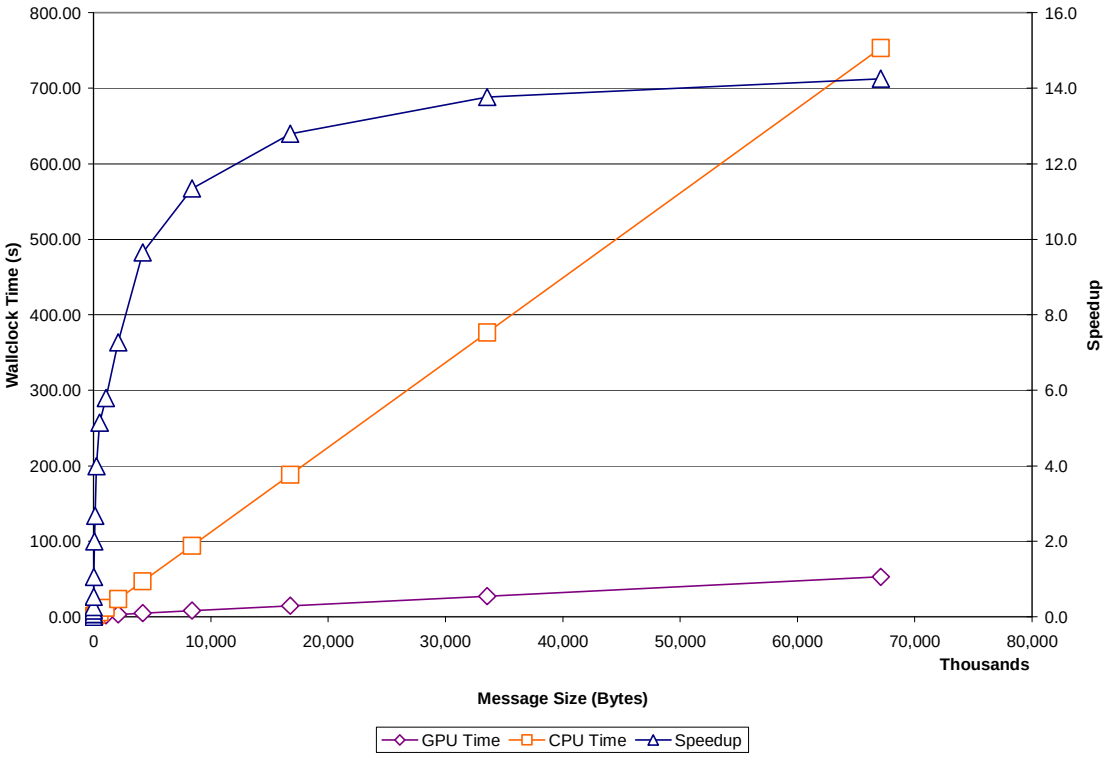

Figure 4 - Results (Linear Scale)

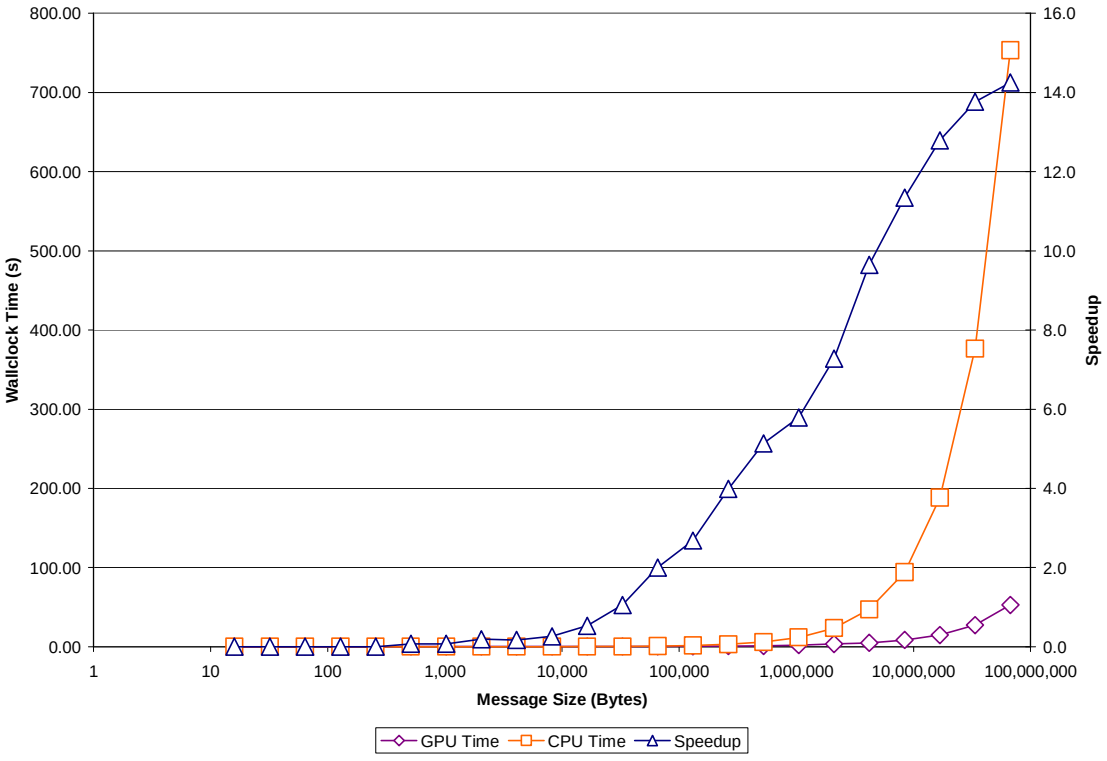

Figure 5 - Results (Log Scale)

From Figure 4 we can see that the parallel GPU implementation achieves massive speedups over the serial CPU implementation. The most effective use of the GPU resources are when the message size is sufficiently large to exploit the parallelism of the architecture and amortize away the memory transfer costs. In Table 1 we see that the break even point does not occur until the message size is 32kB. This means that 2,048 threads are in flight in 8 blocks (256 threads per block).

We also see that the total time required to encrypt and decrypt increases linearly with the message size, but the speedup increases as the CPU time increases faster than the GPU time. Our speedup is capped close to 14.5x as the message size approaches the maximum size.

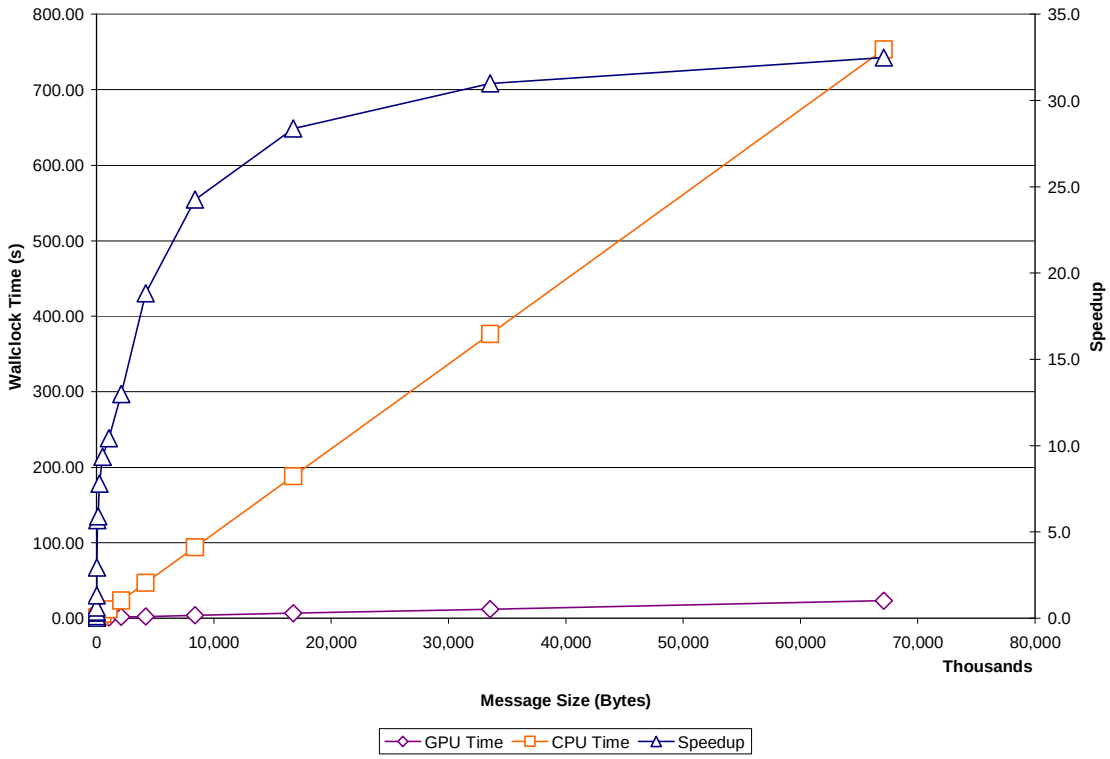

Figure 6 - Results excluding memory transfer time (Linear Scale)

We can see from Figure 6 that when memory transfer to the GPU is excluded, the speedup can approach 33x. This shows that the bottleneck is likely in the GPU itself and the bandwidth between the CPU and GPU is not throttling the performance.

The AES algorithm is very computationally expensive. As such, the bottleneck is not the data bandwidth, rather the processors themselves. For the largest message size, 64MB of data was encrypted and decrypted 200 times in 23.187 seconds. This means that 25.6GB was transferred to and from the GPU, excluding the constant data tables. Overall, this translates to a bandwidth of 9.26Gbps which is well short of the theoretical maximum of 141GBps. Since we are efficiently transferring memory to and from the GPU main memory, we believe this to be a result of the algorithm's computational density as well as its access pattern causing many bank conflicts.

Conclusion

In this report we described our implementation of the AES algorithm in NVidia GPU. Our implementation achieves up to 14.5x speedup over very similar implementation of AES on a comparable CPU. This is comparable to what other authors achieved in their work.

We decided not to implement AES brute-force cracker. Since only one order of magnitude speedup was achieved, brute-force cracking AES on GPU is still infeasible.

References

[1] Gershon Kedem and Yuriko Ishihara. Brute Force Attack on UNIX Passwords with SIMD Computer. In *Proceedings of the 8th USENIX Security Symposium, Aug 1999.*

[2] Owen Harrison and John Waldron. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units.

[3] Camilla Fiorese and Ceren Budak. AES on GPU: a CUDA Implementation. In *CHES, pages* 209–226, 2007.

[4] Svetlin A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *ICSPC 2007.*

[5] 768/1280 Byte Table AES C byte-implementation 03 OCT 06 . http://geocities.com/malbrain/aestable_c.html

[6] Advanced Encryption Standard, Wikipedia, http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[7] Block cipher modes of operation, Wikipedia, http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation