Parallelizing the Advanced Encryption Standard (AES)

Nathan Meyer
PHYS 244
June 13th, 2017

Abstract

In today's world, where cyber-security is becoming more and more of a necessity, the ability to maintain the confidentiality and integrity of data is extremely vital, whether it be for nations or individuals. As the size of data gets larger, the ability to encrypt with great speed becomes very desireable. In this report, I will discuss the Advanced Encryption Standard (AES) and illustrate my parallelism strategy to optimize the encryption speed through OpenMP as well as an attempt to create a CUDA+OpenMP implementation. I hope that by parallelizing AES, one can tremendously speed-up the process of encrypting large files.

What is AES?

AES is today's de-facto standard for encrypting electronic data, file transferring, etcetera and is used in today's most recommended Wi-Fi security standard, WPA2. It is a symmetric-key algorithm, meaning the key used to encrypt is the same key used to encrypt. It is also a block cipher, as opposed to a stream cipher. While a stream cipher encrypts 1 bit at a time, a block cipher encrypts chunks of bits at a time. Symmetric-key algorithms are already very fast compared to asymmetric-key algorithms, but parallelizing them can cut down a significant amount of time. To understand where and how to implement parallelism, it's useful to understand out the algorithm works.

AES takes on 3 key sizes: 128 bit, 192 bit, and 256 bit. This report will focus on a 128 bit sized key. The key size is the determinant of the number of rounds the data goes through when being encrypted. As stated previously, AES is a block cipher. Each block is 16 bytes, and is represented by a 4x4 state-matrix. For each block, the data undergoes a series of transformations that involve a key expansion, and stages called **`AddRoundKey`**, **`SubBytes`**, **`ShiftRows`**, and **`MixColumns`**. Below is the process each 16 byte block goes through:

1. Key Expansion
2. Process Begins
   a. `AddRoundKey`
3. Rounds (Done 9 times for 128 bit key)
   a. `SubBytes`
   b. `ShiftRows`
   c. `MixColumns`
   d. `AddRoundKey`
4. Final Round
   a. `SubBytes`

b. ShiftRows
c. AddRoundKey

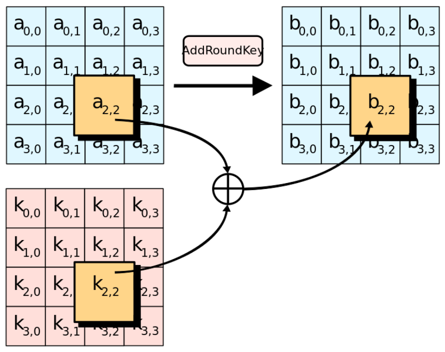
Note that for a 128 bit key, there are a total of 10 rounds. Each step is essentially a way of shuffling the 4x4 state matrix. By understanding the states, you can understand the areas of possible parallelism.

Key Expansion:

The idea behind the key expansion step is to take the provided key and derive a separate 128 bit key that will be used for each round. This is used to protect against using weak encryption keys.

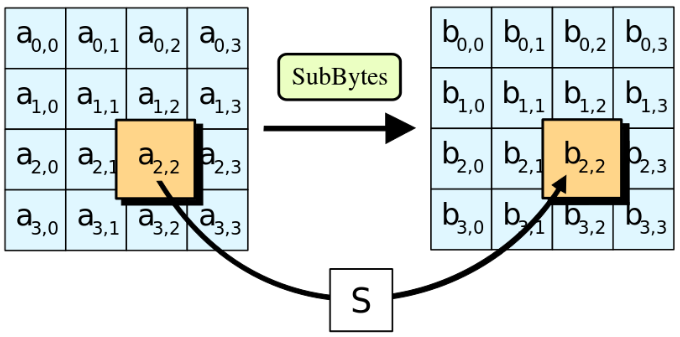
`AddRoundKey:`

In this step, each byte from the state-matrix is combined with a byte from the key generated from the Key Expansion using the XOR operation.



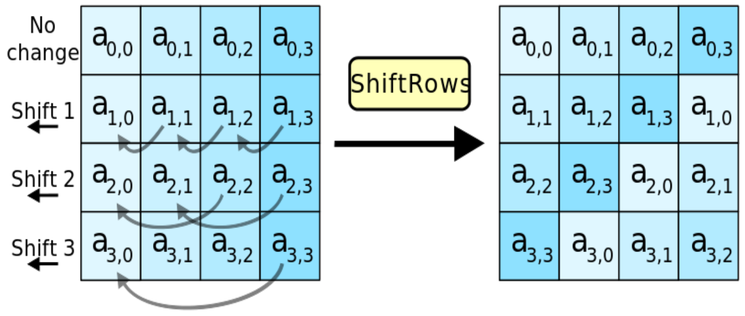
`SubBytes:`

Each byte from the state-matrix is replaced with another byte from a lookup table called an S-Box. This provides non-linearity in the cipher.



<u>ShiftRows:</u>

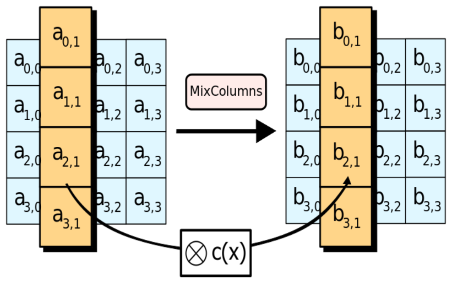
A simple operation on the state-matrix that shifts the elements in the rows like so:



<u>MixColumns:</u>

An operation in which each column is multiplied by this transformation matrix that modifies the state-matrix by columns. The addition from the multiplication is XOR.

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$



In summary, with a 128 bit key, we start with the key expansion, and then begin the rounds. 9 rounds of **`SubBytes`**, **`ShiftRows`**, **`MixColumns`**, and **`AddRoundKey`** are done in that order. For the final round, the same thing as the previous rounds is done but excludes the

`MixColumns` step. The result is the ciphertext, as shown to the right.

Parallelization Strategy

There are two main areas I saw could be parallelized:

1. Applying the cipher to multiple blocks at a time
2. Parallelizing certain substeps of the cipher in addition

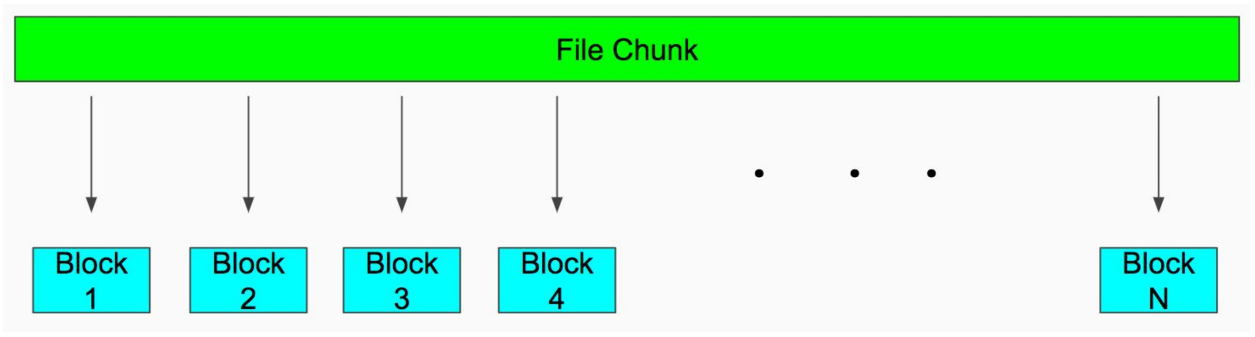
As stated previously, I did an OpenMP implementation and a CUDA+OpenMP implementation.

OpenMP Strategy:

OpenMP is an API designed for shared memory parallel programming. OpenMP utilizes multithreading, dividing work among different threads in a fork-join type system. In the AES problem, work-sharing can definitely be used. As I read in a fixed chunk size from the file, I can designated threads to operate on several blocks at a time within that chunk. This way, rather than encrypting one block at a time, multiple blocks get encrypted simultaneously. However, substeps like **`SubBytes`** and **`ShiftRows`** that are nested in the over-arching parallel for loop also have the potential to be parallelized.

The OpenMP implementation of the cipher can be seen in ‘aes.cpp’ and can be run with the linked executable, ‘paes’, and passing flags like ‘-i’ for input, followed by an input file of your choice to encrypt. The generated output file will contain the ciphertext. Interactively, the program outputs metrics like number of available threads and walltime, depending on the options you supply when running.

Below is a representation of encrypting several blocks at a time:



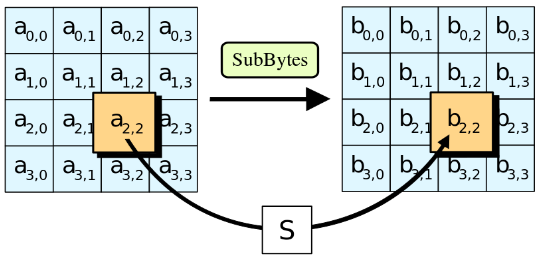
You can see the idea here is to encrypt multiple blocks at a time for each file chunk we read in. This can be done using a for loop that loops through each block in the chunk. Using a `parallel for` construct. I will test out `static` vs `dynamic` scheduling.

In this overarching loop that loops through the 16 byte blocks in each chunk read in, there are possibilities to introduce nested parallelism the sub-routines of the cipher.

- **`SubBytes:`**
  - Because this function is simply mapping a byte in the state-matrix to a byte in the S-Box lookup table, every byte mapping is independent from one another.
- **`ShiftRows:`**
  - The shifts in each row of the state-matrix are independent from each other. Therefore this is possible to parallelize

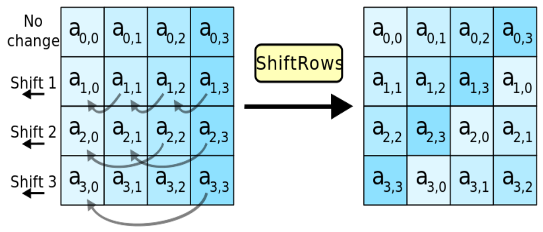
Keep in mind, parallelizing these steps introduces nested parallelism, which can cause problems such as too much overhead.

SubBytes:


$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
SubBytes
S
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$

Parallelization can be done by splitting the work into rows or columns, using sections, or a parallel for that splits the work among rows or columns.

ShiftRows:


No change
Shift 1
Shift 2
Shift 3
ShiftRows
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,0}$
$a_{2,2}$ $a_{2,3}$ $a_{2,0}$ $a_{2,1}$
$a_{3,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$

Since the operations on each row is independent from the others, we can designate a section to each row, ie make a thread responsible for each row.

Challenges:

Nested parallelism naturally creates overhead. By parallelizing the parallelizable parts within the cipher, nested parallel regions are created that are constructed over and over again with each iteration. Even with environment directives to control the nested parallelism like **`SUNW_MP_MAX_POOL_THREADS`** or **`SUNW_MP_MAX_NESTED_LEVELS`**, the amount of overhead still has a noticeable effect on the speed and efficiency of encrypting. I found it was optimal to parallelize the overarching outer loop versus the small functions within that loop.

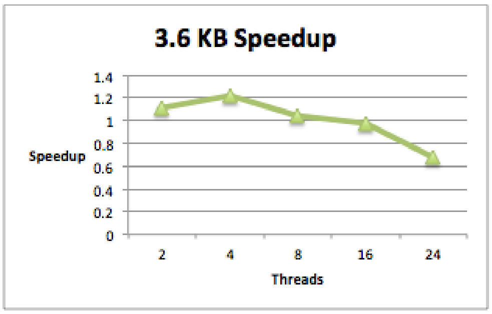
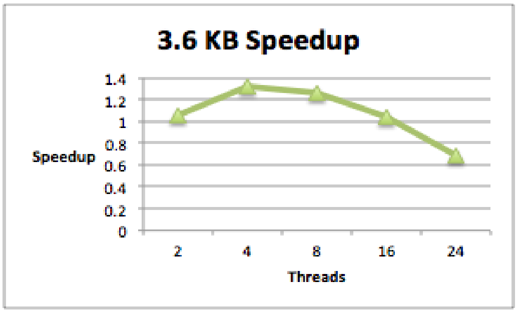
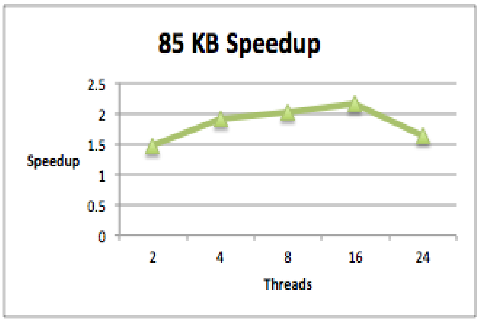
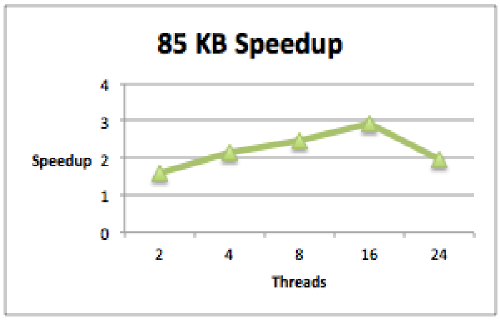
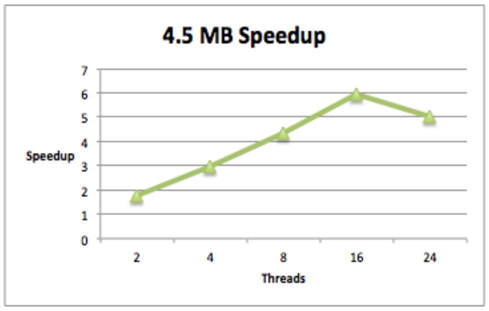
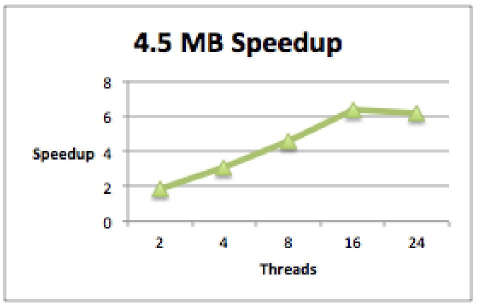
CUDA+OpenMP Strategy:

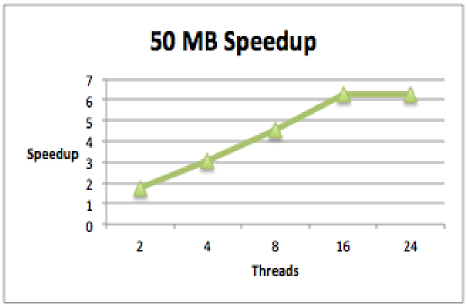
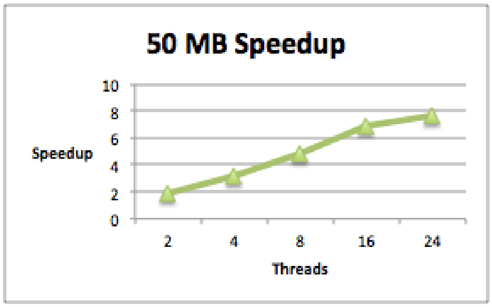
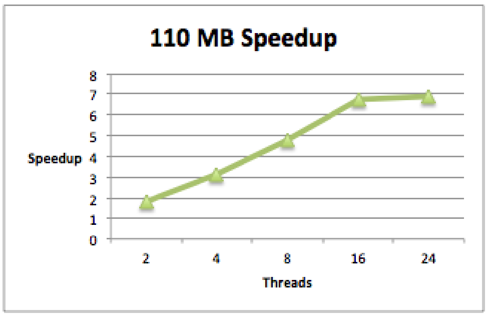
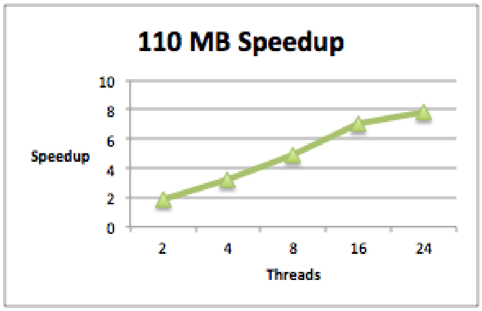
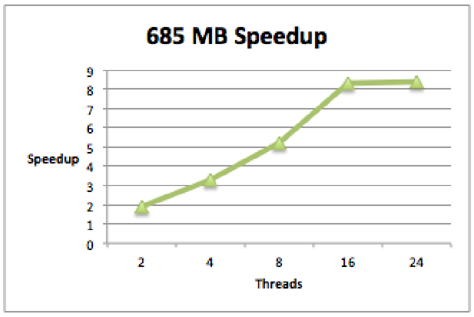
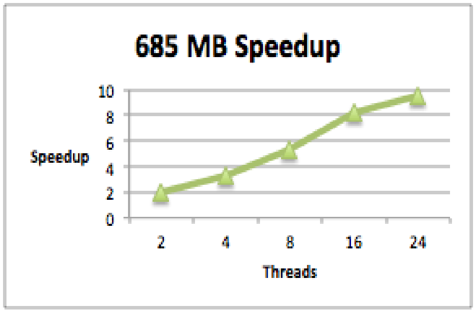
With CUDA+OpenMP, the approach is slightly different. Similar to the OpenMP version I encrypt at the macroscale, by looping over the blocks of the file chunk read in. However, with CUDA, parts of the cipher such as SubBytes and ShiftRows can be parallelized by allocating 16 blocks for `SubBytes` to a grid and 4 blocks to a grid for `ShiftRows`, hence doing these steps in parallel. The problem that can arise is the constant back and forth allocating of memory for each block that gets encrypted when performing the cipher in parallel on 1 block. The larger the file, the more overhead we have because of the constant back and forth movement of each individual 16 byte block when using CUDA on the substeps of the cipher.

Results

Below are the results of the OpenMP implementation from parallelizing the application of the cipher to the 16 byte blocks in each chunk we read in. Here is the comparison in results when using **`dynamic`** vs **`static`** scheduling.

OpenMP Results:

`Dynamic` Scheduling	**`Static` Scheduling**
	
	
	

It can b seen that **`static`** scheduling leads to slightly higher speedup for the larger files. For example, the speedup almost surpasses a value of 10 for the 685 MB file compared to roughly 8 with **`dynamic`** scheduling. **`Dynamic`** scheduling tended to level off at a certain speedup level with the larger files as more threads were added to the environment. Unlike **`dynamic`** scheduling, the speedup with **`static`** scheduling tends to be more linear with larger files. This may be because each chunk we read in is the same size, making it most efficient if the loop that reads in the files is divided into equal chunks. Thus, there is no need to use **`dynamic`** scheduling.

The CUDA+OpenMP implementation results were not as successful, as there was no speedup whatsoever for any file size as seen below:

File Size	Best OpenMP Time(sec)	CUDA+OpenMP Time(sec)
3.6 KB	0.004756084	0.37651
85 KB	0.00588745	1.054444
4.5 MB	0.10032766	38.830496
50 MB	0.927246	Timeout
110 MB	2.145074	Timeout
685 MB	10.332418	Timeout

Conclusion and Areas for Improvement

AES is today's worldwide standard for symmetric encryption of electronic data. When required to securely transfer large data sets, it becomes necessary to use parallelization to encrypt the data faster. My attempts to do this consisted of an OpenMP version and a CUDA+OpenMP hybrid version.

With the OpenMP version I did not parallelize the substeps of the cipher, but rather the application of the cipher to several blocks of data at a time. By doing this, I encrypt at the macroscale and avoid the overhead that would exist from nested parallelism if I had parallelized the substeps of the cipher. This implementation saw great speedup, especially as the file size increased.

For the CUDA+OpenMP hybrid implementation, I tried to combine encrypting several blocks at a time like in the OpenMP version with parallelizing the substeps of the cipher with CUDA. This implementation was not nearly as successful. I believe this is due to the constant movement of memory to and from the device for each 16 byte block whenever the cipher is performed, causing a great deal of overhead, especially for the larger files.

There are a few things I would try to improve if time was not an issue. I would try to use CUDA to not focus on parallelize the substeps of the cipher but more so to focus on encrypting several of the 16-byte blocks at a time, thus improving throughput. In addition, I would try to move arrays of data to and from the device more efficiently, in such a way that it can be accessed quickly. Overall, from this project, it can be seen that there are great gains in speedup by parallelizing AES.