Alfredo Mendez

Professor Mordohai

CS 677 – Parallel Programming for Many-core Processors

May 10, 2023

AES Encryption of Images

For my final project, I decided to attempt a parallel implementation of AES encryption on images. Specifically, my goal was to take the "RGB" values of every pixel, encrypt them, then save the resulting encrypted image as the output. I picked this problem to parallelize because I thought it was a problem that was challenging enough and interesting enough to attempt to parallelize. While I only had little experience working with encryption algorithms, I knew it was a parallelizable problem because of the number of resources I found on the internet. The problem is important to parallelize because it is a standard in encryption, hence, a standard in computing security. Parallelizing encryption means speeding it up as well, which is always important in the field of computing.

First, I will give a quick overview of what AES encryption is and what the encryption process looks like at a low level so that my implementation and decisions are clearer to understand. AES encryption is a block cipher, meaning that it encrypts 16-byte blocks at once. Just like any other encryption algorithm, AES encryption needs an "Encryption Key" to be able to encrypt any data. The length of the key ranges from 128 bits to 256 bits; For this implementation, I will be working with 128-bit keys, which are 16 bytes, the same length as each block that AES encrypts at a time. The tradeoff that is made when choosing a shorter key in this instance is security for simplicity. For this implementation, simplicity is highly desired, and security is much less of a priority. However, using a 128-bit key is still sufficient for most situations.

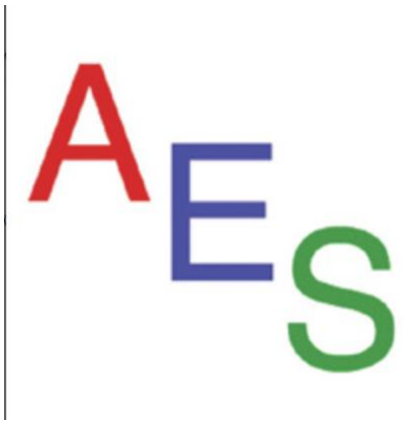
The AES encryption process is made up of four main steps that are repeated for ten rounds, with some small exceptions. One of these steps needs a different encryption key for every round; these keys are called “round keys”. The trick to solving this problem of needing ten additional keys is to generate them from the original encryption key. It is at this point that I encountered the first hard decision that I had to make. Implementing the sequential version of this algorithm, called “key schedule”, that generates new round keys was relatively straightforward, compared to the AES algorithm. However, implementing the key schedule algorithm sequentially made me realize that it would be very difficult, if not impossible, to implement a parallelized version. The main reason for this is that generating a new round key requires the previous round key, making the algorithm inherently sequential. Additionally, the algorithm works with a very small dataset, which means that making it parallelized won’t provide a significant speed-up. The dataset is the 10 keys that are being generated, which is only a total of 160 bytes; Compare this to the tens of thousands of bytes that the AES algorithm works with. For this reason, I decided to exclude the key schedule algorithm from my parallel implementation of AES.

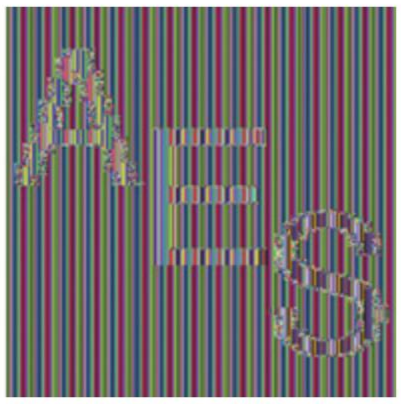
As mentioned before, the AES algorithm is mainly made up of four main steps that are repeated ten times. These steps are the “substitution” step, the “shift rows” step, the “mix columns” step, and the “add round key” step. It is important to remember that each step works with a block of data, which is represented as a four-by-four 16-byte matrix and is the output of the previous step. In the substitution step, each byte in the 16-byte matrix is substituted with another byte using a look-up table called the “substitution box”. The table provides a one-to-one relationship between input and output; hence, the table has a size of 256 bytes. The important thing to remember about the substitution box, or s-box for short, is that the values within it are constant not only throughout all 16-byte matrices that are encrypted but also throughout all standard implementations of AES. The s-box is generated by a clever algorithm that produces an output that optimizes security. This means that calculating a new s-box is not only pointless and work intensive, but also reduces security. There is a lookup for every byte in the 16-

byte matrix and since this step is repeated ten times, there is a total of 160 lookups that happen per encryption of a 16-byte block.

The shift rows step says that the first row does not shift, the second-row shifts left by one, the third shifts left by two, and the fourth shifts left by three (which is the same as shifting right by 1). Implementing this step is quite simple as it only involves moving around values within a matrix. Unlike the shift rows step, the mix columns step is a little difficult to implement. In this step, each column of the 16-byte input matrix is multiplied with the same "set" four-by-four matrix, which is also 16 bytes big. Just like the s-box, this matrix is usually constant through all standard implementations of AES. Keeping this matrix simple is optimal as it increases performance, and making it more complicated or making it hold larger values does not provide a significant increase in security. The hard part about this step is that it uses multiplication in a "Galois Field"/"Finite Field". This type of multiplication is necessary because it limits the range of possible values that a product can be to numbers between 0 and 255, which are the only values that a byte can store. Unfortunately, there is no operator for this operation in C, hence, I had to implement one myself. The last step in the AES encryption process is the "add round key" step which simply means that an XOR operation is performed on the current input with the corresponding round key for the current round.

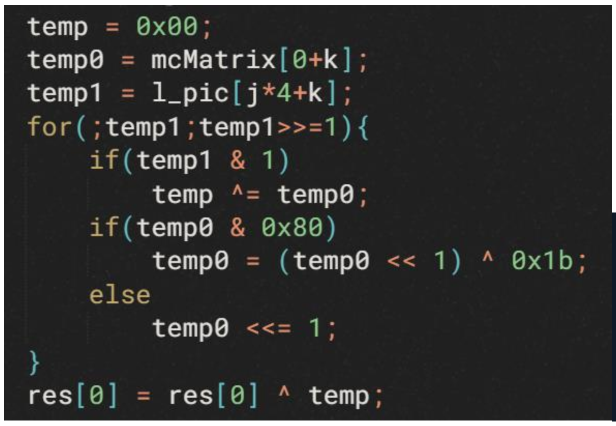
When starting my implementation, the first challenge I ran into was the task of getting the RGB values of the pixels in the input image. Fortunately, we had worked with images earlier in the semester. I simply modified the code that we used so that it collects the individual RGB values into a byte array. This means that after gathering all the RGB values into an array, I had an array full of byte values without a clear way to distinguish between which ones belonged to the 'red' value, 'blue' value, or 'green' value. This situation leads to the next challenge: How should the 3-byte RGB values in the array be fitted into 16-byte matrices? The first possible solution was to assign a pixel RGB value to each matrix. This means that each matrix would hold three bytes of actual data and would need to hold 13 bytes of padding, presumably this padding would just be 0x00 bytes, to make things simple and faster. While this idea would be sufficient and would produce correct results, it is not optimal. First, this approach takes up a lot of space in global memory, most of which is just padding (13/16). Wasting this much memory in global memory means that you would have to do a lot of unnecessary loads and stores. Additionally, this approach wastes computation time that could be used towards completing the encryption process by instead using that computation time to parse the padding in and out. Furthermore, since there's a one-to-one relationship between input and output matrices in AES encryption, assigning one pixel to one matrix, makes it much easier to simply guess what the original image was, as demonstrated by the images below. The second possible solution was to fit as many RGB values into each matrix; this would mean storing a total of 5 RGB values and using only one byte of padding per matrix. While this solution is much better than the first one, it is still using up unnecessary memory and an unnecessary amount of computing power; not to mention that my intuition also told me that this solution was not 'elegant' enough. My final approach was much simpler; instead of seeing the RGB values as a data type, I instead just treated them as individual byte values independent of what color they represented. This means that I could fill up each matrix with actual data and that I would not have to do any parsing or appending to the data. An additional unintentional benefit to this approach is that it increases the level of security by adding randomness to the process. By this, I mean that part of some RGB 3-tuples will be stored in one matrix and the rest will be stored in another matrix.



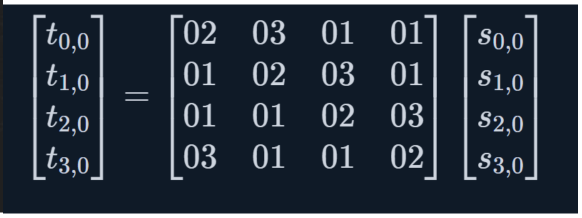


Implementing AES encryption of images was not easy, there were challenges both in implementing the general AES algorithm and in specifically trying to make it work in parallel with pixel RGB values. One of the first things I began to realize when I started to implement AES is that it is a very sequential process. As, stated before, AES is mainly made up of four steps which are repeated ten times. The important thing to note is that each step depends on the output of the previous step and each iteration depends on the previous iteration, obviously meaning that you cannot split up the rounds and steps between different threads. An attribute of AES that made it difficult to implement in a parallel context was that each byte heavily depends on the other bytes in its 16-byte matrix for encryption. This fact is trivial when one considers the “shift rows” and “mix columns” steps.

Implementing AES itself was surprisingly difficult, especially when you try to optimize it for efficiency. One of the toughest challenges was implementing the ‘mix columns’ step. As a reminder, the “mix columns” step is the individual matrix multiplication of each column with a set/constant four-by-four matrix. However, the multiplication is in the finite field of GF(2), which limits values to be between the range of 0 and 255, which are the values that a byte can hold. Implementing this form of multiplication was difficult because you need to start thinking at the bit level and how each operator manipulates a byte’s bits. Debugging code at a level with which one is not very familiar in a language that is not optimized to work at that level can be very difficult. The last challenge that is worth mentioning is that all my matrices were stored as 1D arrays, meaning that the indices of arrays within nested for loops were difficult to work with and prone to small hard-to-find bugs.

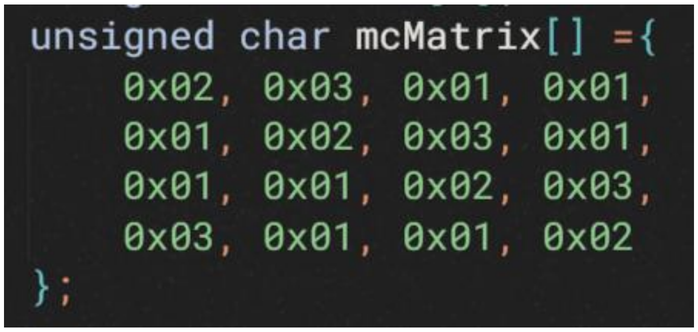


```
temp = 0x00;
temp0 = mcMatrix[0+k];
temp1 = l_pic[j*4+k];
for(;temp1;temp1>>=1){
    if(temp1 & 1)
        temp ^= temp0;
    if(temp0 & 0x80)
        temp0 = (temp0 << 1) ^ 0x1b;
    else
        temp0 <<= 1;
}
res[0] = res[0] ^ temp;
```



$$\begin{bmatrix} t_{0,0} \\ t_{1,0} \\ t_{2,0} \\ t_{3,0} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} \\ s_{1,0} \\ s_{2,0} \\ s_{3,0} \end{bmatrix}$$

I ended up coming up with three main versions of the AES kernel: the naïve version, the shared memory version, and the version that did more work per thread in addition to using shared memory. In the naïve version, almost every piece of data needed was stored in global memory. Specifically, the input RGB array (x dimension X y dimension X 3 bytes), the output encrypted RGB array (x dimension X y dimension X 3 bytes), the encryption keys (11 total keys X 16 bytes each = 176 bytes), and the substitution box (16 X 16 = 256 bytes). The only data that I did not put in global memory was the constant matrix (pictured below) that is used in the “mix columns” step; I put this data directly in registers. I did this because the data is so small (16 bytes) that it is a trivial decision to put it in registers. Also, with the block size that I am using, after doing the math, I had more than enough registers to spare. The math is ~27 registers * 256 threads per block = 6912. In the naïve version, I opted to keep it simple and only assign one 16-byte matrix of input data to each thread. As implied before, each thread has been calculated to use a total of about 27 registers before. These registers are used to store temporary variables, ‘for loop’ variables, and the “mix columns’ matrix. As stated previously, as well, I chose a block size of 256 and a grid size that is appropriate for the size of the image being encrypted and for the number of threads per block. Additionally, it is of significance to state that each thread makes a local copy of the part of the input array that it is assigned to encrypt. This way each thread doesn’t have to load in values from global memory and back into main memory every time a thread needs to manipulate a value in their assigned section of the input array. Some of my decisions make my ‘naïve’ version a little less naïve but I believed that making these decisions for the naïve version would be better for proving that the changes made in my other two versions provide an improvement in efficiency and an overall speedup.



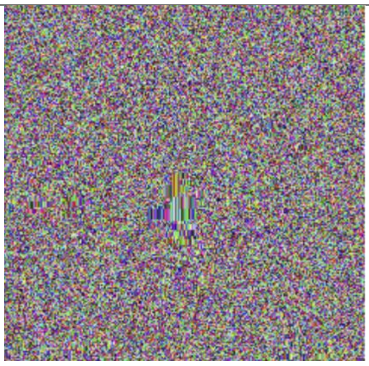
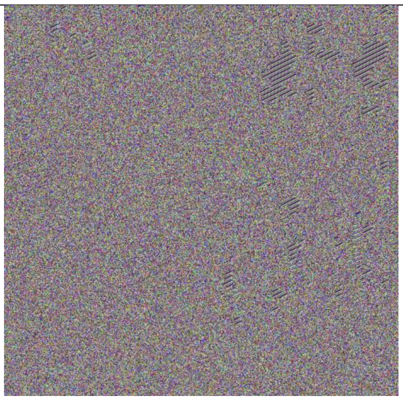
```
unsigned char mcMatrix[] ={
    0x02, 0x03, 0x01, 0x01,
    0x01, 0x02, 0x03, 0x01,
    0x01, 0x01, 0x02, 0x03,
    0x03, 0x01, 0x01, 0x02
};
```

The modifications I made to the second version of the AES kernel were simple. The main encryption key and the ten round keys, which are stored in the same array, are now loaded into shared memory. Additionally, I also decided to put the 16-by-16-byte substitution box into shared memory. The reason why I picked a block size of 256 is now obvious. With a block size of 256, each thread can load in a substitution box byte from the global memory into shared memory. Furthermore, only some bytes, 176 to be precise, have to load in bytes from the global keys array into shared memory. The modification that I made to the third version of the kernel was simple as well: In addition to using shared memory, each thread is now assigned to encrypt two input matrices instead of one input matrix, meaning that each thread now works with 32 bytes instead of 16. This modification does not mean that each thread will take twice the amount of time to complete. This is because when a thread is assigned two matrices to manipulate with the same process, there are optimizations that can be made. For example, one for loop can be used to manipulate both matrices at a time. Another optimization that can be made is that set values that are used for a specific step in AES do not have to be loaded in separately for each matrix; one load can be used when applying that step to both matrices. Additionally, I decided to store both matrices in one array, even if it makes the index values harder to work with.

To measure the difference in performance between the three different kernel versions, I decided to use the ‘CUDA runtime’ library to simply time the time it takes for each kernel to complete. Additionally, I used three different images of three different sizes to further measure how different data size affects the kernels’ performance. The ‘small’ image is of size 240x240, the ‘medium’ image is of size 690x690, and the ‘large’ image is of size 1920x1080. Each kernel was run 10 times for each image, the average was then calculated and rounded to two decimal points; the results are recorded below. The average timing for the sequential implementation of the algorithm has also been recorded for comparison. As displayed below, the time it took for each kernel to complete increased as the image size grew larger. Additionally, with the medium and large image sizes, there seems to be an improvement of about 1000x between the sequential and the naïve implementation of the AES algorithm; the small image showed a smaller improvement, which is expected as parallelism decreases as the number of 16-byte matrices decreases. The improvement from using shared memory to assigning two matrices to each thread in addition to using shared memory was much less significant; there was about a 1.25x improvement between these two versions. Overall, I believe these results show that AES encryption of images was successfully parallelized.

When it comes to correctness, many output blocks were compared to the output blocks of an online tool, given the same input blocks. While I did not have the time to implement encryption before the deadline, I believe that this method is sufficient to demonstrate correctness as every kernel uses the same algorithm to encrypt the matrix/matrices that were assigned to it. Therefore, by testing a few random blocks, and all edge cases (meaning start and end of rows, start and end of the whole array, instances where the matrix is full of 0x00 values, etc.), it is reasonable to conclude that all blocks were encrypted properly, and hence, the whole image was encrypted properly. I do plan to complete the decryption algorithm in the near future; however, I was unable to do so before the project deadline. The images, unencrypted and encrypted, are also provided below.

	Small (240x240)	Medium (690x690)	Large (1920x1080)
Sequential	363.46 ms	3000.35 ms	13810.07 ms
Naïve	.86 ms	3.40 ms	13.74 ms
Shared Memory	.11 ms	.34 ms	1.12 ms
Shared Memory + Two Matrices Per Thread	.09 ms	.25 ms	.97 ms

Before Encryption	Before Encryption	After Encryption
Small (240x240)		
Medium (690x690)		

Large (1920x1080)