CS378 Concurrency: AES Final Project Writeup

Shishir Jessu
Aneesh Adhikari-Desai
Department of Computer Science
UT Austin

May 3, 2018

1 Project Idea: AES

We chose to implement the Advanced Encryption Standard algorithm (AES), also known as the Rijndael cipher. For this final project, we wanted to explore an idea that did not fall into the category of "classic" concurrency problems, like a concurrent hashmap or an algorithm like K-Means, which is often used as an example of a data-parallelizable algorithm even though it's originally a machine learning algorithm. We decided to work in the area of cryptography, because nearly all events that occur on the internet involve some form of encryption, and optimizing this encryption with parallelism can help speed up many applications in general. Also, our cryptography works primarily because breaking it takes so long that doing so is intractable, and so speeding up cryptography can pose a great security risk in some situations. We weren't aiming to break a cryptography algorithm, but by showing that parallelism at a small scale can result in significant speedups, then if massively parallel methods become extremely efficient they could pose a threat to cryptographic methods even without us finding more clever algorithm or using quantum methods like Shor's algorithm when such computers exist.

We ultimately chose the AES algorithm because it has many characteristics that make it readily parallelizable. The algorithm takes a 16-bit key and an arbitrary length plaintext, and encrypts the plaintext using the key. There are many modes of the algorithm that operate on the plaintext in 128-bit blocks - some of these are less secure than others, but we have chosen a version of the algorithm that is secure but also operates on small blocks independently (we talk more about this choice further down); thus, a clear source of parallelism arises from processing the blocks in parallel.

The algorithm works by applying a scrambling process for a given number of rounds based on the block size chosen (for 128 bits, we run 11 rounds). We follow these steps:

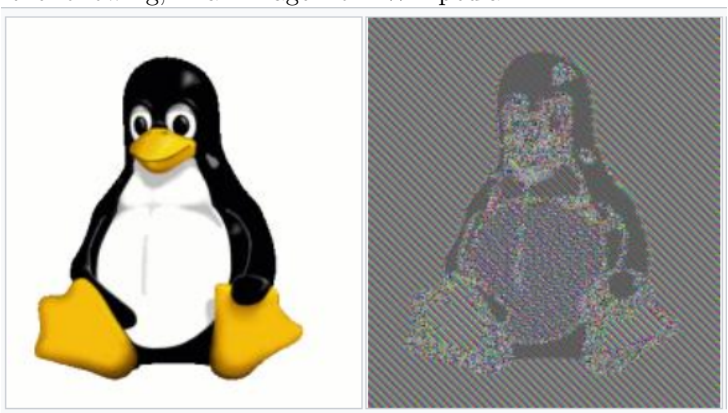
- Key Expansion. To create variety in the result of the encryption between rounds, key expansion takes the original 16 byte key and expands it into a key of size kr, where k is the number of bytes in the key and r is the number of rounds. This ensures that the same key is not applied and thus, there is variety between the rounds. For each of the 11 rounds, we used 16 bytes of the expanded key.

- Key Expansion happens once before the core algorithm begins. After it's done, we run these steps 11 times:

  - Add Round Key. Each byte of the state is combined with a portion of the key corresponding to that round with an x-or operation.
  - Sub bytes: Every byte in the plaintext is substituted with another according to a lookup table.

- Shift rows: the nth row is circularly shifted n bytes to the left. So the top row is not shifted at all, the next row is shifted 1 byte to the left (with the leftmost byte now appearing on the right because of the circular shift), and so on.
- Mix columns: This is the most complicated part of the AES algorithm, and is not performed on the last iteration of the algorithm. It requires multiplying polynomials in Galois fields. Luckily, people have created lookup tables to prevent us from having to implement the functionality ourselves, and just look the results of the multiplication. As Dr. Rossbach would say, this is a Concurrency class, not a MixColumns class, right?

These core steps are common to any version of the algorithm, but there are several "modes" of the algorithm that cater to different security concerns. The most basic of these is the Electronic Codebook Mode (ECB), which encrypts each block separately. The problem, however, is that since AES is deterministic, each 128-bit input will lead to the same 128-bit output. This leads to problems like the following, in an image from Wikipedia:



Original image Encrypted using ECB mode

where even though the image is "encrypted" it is possible to infer the basic structure of the original data. We originally implemented this method, but decided that even though this is a Concurrency class and not an AES class, the security concern is too great to ignore. The most secure method is Cipher Block Chaining, or CBC, in which each block of the plaintext is xored with the previous ciphertext block. Unfortunately, this method cannot be parallelized because each block depends on all the previous blocks. It is different from something like prefix sums, because prefix sums can be computed partially in the middle of an array, but each block depends on all the previous blocks having been encrypted in a particular order.

Luckily, a method that is not much harder led to an equally parallelizable but more secure implementation: Counter Mode. In this mode, each block is associated with a counter that is concatenated to a nonce, *that* is encryped with the AES algorithm, and the encrypted result is xor'd with the plaintext block to create the final ciphertext. This has two advantages: first, it is no longer the case that one 16-byte block of plaintext will always map to the same 16-byte ciphertext given the same key, providing additional security since the counters are private. Second, decryption this way is much simpler than normal decryption. We encrypt the nonce and counter, and then xor with the plaintext, but since the inverse of the xor operation is xor itself, we can decrypt by *encrypting* the nonce and counter concatenation, and simply x-oring it with the ciphertext!

We decided to attempt parallelizing the algorithm in CUDA, C, and Golang. We picked CUDA because we were impressed with the massive speedups it created in the K-Means lab, and its ability to create a massive number of threads seemed well suited to CUDA because each block can be encrypted in parallel even for very large files (on the order of megabytes). Also, we chose Go because Aneesh really likes Go. Seriously though, Aneesh found the semantics, performance, and elegance of Go (especially when compared to C) appealing and wanted to explore the idea of creating many goroutines at once in a thread pool-type approach (and other options discussed below) to parallelize the solution.

2 CUDA

As mentioned above, CUDA is an attractive option for parallelizing AES, because the vast number of threads allows us to encrypt each block in parallel, even for very large file sizes. My sequential baseline to be compared with CUDA was written in C. I had a function called `runAES()`, which does the following setup steps for AES:

- Pads the last block if necessary. We need to encrypt evenly sized 16-byte blocks, but the number of bytes may not be a multiple of 16. Thus, we need to pad it so that it is. But we can't just pad it with 0s, because otherwise the decrypter would not know if the zeros are padding or are actually part of the plaintext. So we use the following rule: if we need to pad with n bytes, we pad with the hex value of n. So if there are 9 bytes left, we pad with 9 $0x9$ values. Then, the decryption function can check to see if there are n copies of value n at the end of the ciphertext, and then remove this many bytes when the decryption is done.
- Runs the key expansion routine, to get a total of 176 bytes of expanded key.

Then I called an `encrypt()` function, which actually performed the rounds of AES as described above. This is a sequential process which unfortunately cannot be parallelized, but it only operates on a small number of bits and is $O(1)$:

```
void encrypt(unsigned char* state, unsigned char* expandedKeys, long cVal) {

   long* counter = (long*) calloc(sizeof(long), 2);
   counter[1] = nonce;
   counter[0] = cVal;

   unsigned char* counterState = (unsigned char*) counter;

   addRoundKey(counterState, expandedKeys);

   for (int x = 1; x < 11; x++) {
      subBytes(counterState);
      shiftRows(counterState);

      if (x != 10) /* no mixCols on last step */
         mixColumns(counterState, 4);

      addRoundKey(counterState, expandedKeys + 16 * x);
   }

   xor(state, counterState);
   free(counterState);
}
```

In the CUDA version, instead of calling `encrypt()` in a loop, I invoked it as a kernel. The main challenges here were choosing an appropriate number of threads per block and dividing up the work based on the thread index. I eventually decided to use 1024 threads per block, the maximum, in order to utilize the streaming multiprocessors as best as possible. Dividing the work between threads was relatively simple, as it just required adding an address to the `state` parameter.

This approach resulted in significant speedup, much more than Go did, as will be described in Section 4.

3 Golang

The Go implementation for this project was quite similar to the raw C and CUDA in that the code was organized into a function call to encrypt for every block in the input, and within each encrypt call the code stepped through the standard steps of AES described above. As such, we developed the sequential solutions for C and Go together, both for the sake of learning AES and to have a flexible model to check against for debugging. As we branched off into seperate implementations in CUDA and parallel Go, there were some specific design decisions we took on the Go implementation that had some interesting bugs and performance impacts.

By nature of AES, a large number of the same operations are performed over and over on various data points through the encryption process. While this translates quite naturally into a GPU implementation, we originally thought that a worker thread pool based solution would also be viable with the intuition being that instead of needing to launch a new go routine or jump to the function, we could just construct our input object and pass it through a channel to the appropriate worker. So, we structured the code so that the encrypt function would pass in inputs to global channels (specifically by reference to avoid copying entire slices unnecessarily), and there would be some pre-created workers that were listening on their respective channels. These channels were left unbuffered since the expectation was that there should always be either a surplus of workers (so some are asleep and ready to pick up a job at any time) or, if the workers are all busy, the hardware is saturated anyway, so it wouldn't matter if the job-producing routines go to sleep. Additionally, some primitive preliminary testing yielded no significant results when adding a buffer to the channels, so we decided against it and the decision became irrelevant with the progress of our development.

When we began timing our solutions and collecting data for the speedup of the Golang parallel solution against the Golang sequential solution, we were shocked to see that the parallel solution was between 5 and 10x slower! Upon further investigation, we discovered that there was a ridiculous amount of memory usage by our parallel solution, on the order of 6 gigabytes when running against our largest 31MB dataset. Upon further investigation, we discovered an issue where the expanded keys were being duplicated for every block (since each block had its own goroutine). The fix for this was quite simple (just pass by reference like we should have originally), but the effects were dramatic. However, ever after this, the speedup was still < 1. After inspecting our design more closely, we realized that attempting to parallelize jobs/encryption steps within a single block could not possibly yield any performance gains since each step depended on the immediately previous step. So, we tried removing the goroutine pools and switched to a simple go routine per block design. To our surprise, this significantly improved our results (which are detailled below).

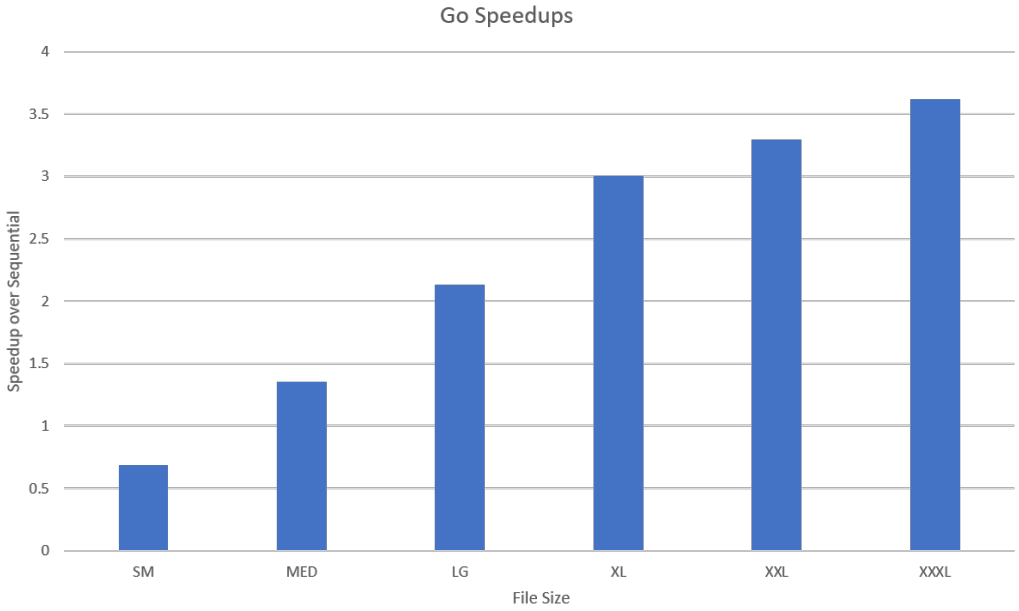
Although we're not certain as to why the thread pool implementation was so slow, we have some suspicions. The aforementioned dependency between encryption steps for a single block is only a cap on why the pools didn't improve performance significantly; it does not account for performance deteriorating. Another possibility is that the combination of context switching between a very large number of routines in the system and implications of switching on cache misses could account for the increased runtime. Furthermore, we suspect that channel communication between two goroutines is possibly worse than simply calling the function normally if there is no parallelism to exploit.
After all this, we are fairly satisfied with the performance achieved by our Go solutions, particularly that of our sequential solution (more on this below).

4 Results

4.1 Implementation and runtime details

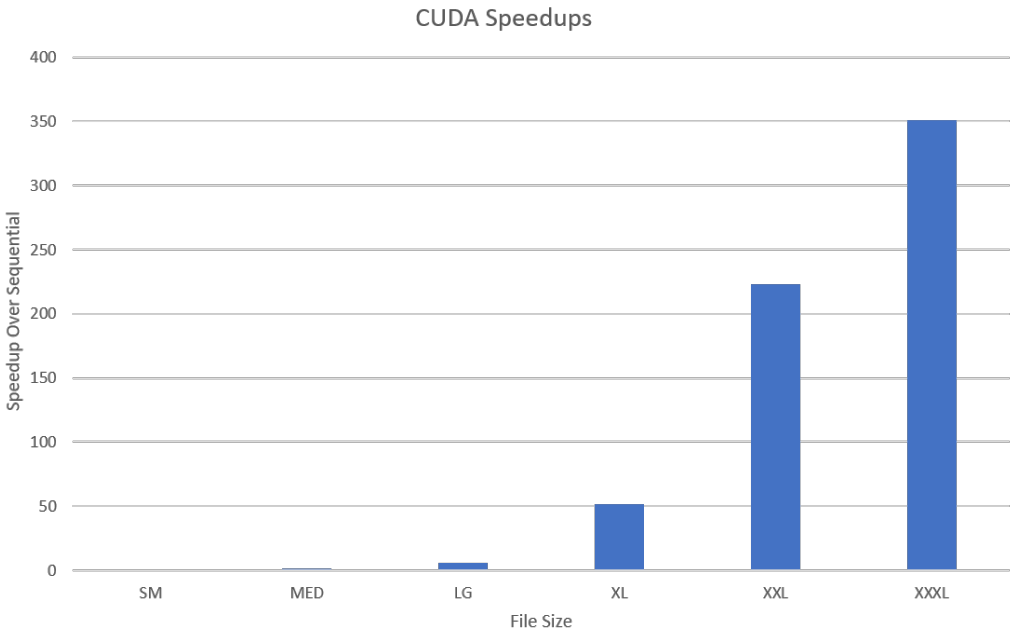
All experiments were performed on eldar-4, go version is 1.9, cuda version is 8.0 running on a GTX 1080, CPU is some 8-core (possibly 4 core hyperthreaded) Intel chip. All data collected is an average over 10 runs, data was run and collected with python, graphs were made in microsoft excel, datasets were randomly produced in python with the lorem library.
For reference, code is available at https://github.com/shishirjessu/aes-concurrency

4.2 Golang speedup



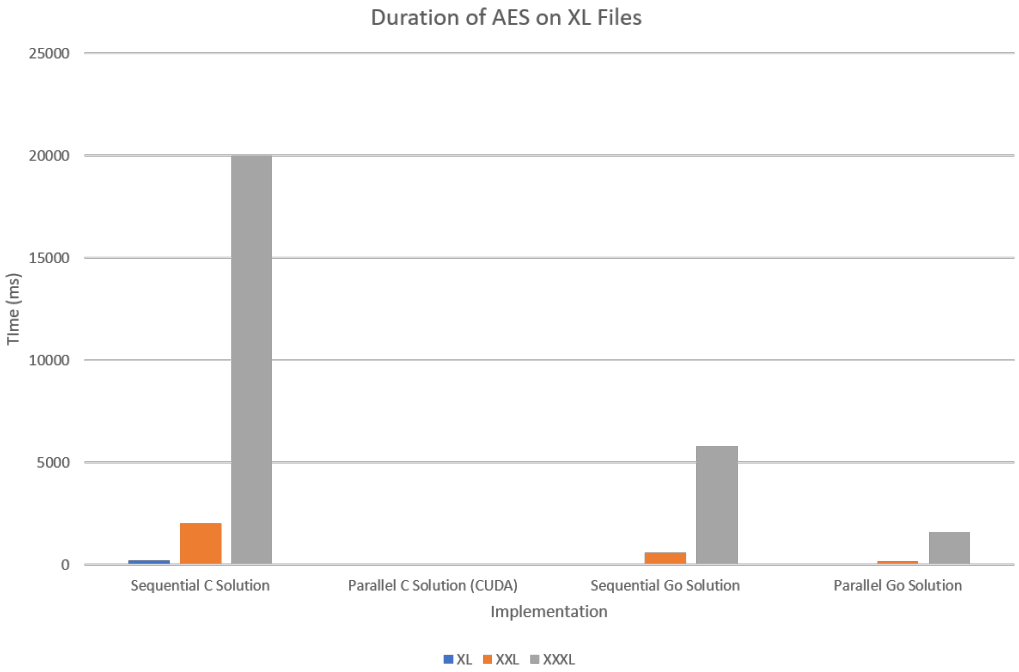
Here, we can see that having a larger dataset yielded better speedup over our sequential. This is to be expected since the parallel solution is able to utilize the hardware much more efficiently. The speedup plateauing as the files get very large suggest that there is some sort of bottleneck in play, but we suspect that if there were more CPU cores available on this machine, the performance would continue to grow more steeply as the code would be able to exploit more parallelism. While the raw speedup numbers don't appear to be that dramatic, the runtime of the naive sequential implementation was impressive, even on very large datasets, refer to the timings comparison graph below. This was kind of unexpected since C is famed for its performance, and we suspect that ~~Shishir just messed up~~ there is some overlooked detail that causes the slowdown, but we think there is still something to be said for the well-designed libraries for Go and the fact that the most obvious, intuitive solution still yields fantastic performance.

4.3 CUDA speedup



The CUDA speedups are significantly larger than the Go speedups. This was expected, because ~~Shishir is a better programmer than Aneesh~~ CUDA is able to encrypt nearly all the blocks in parallel by running thousands of threads at once, and runs on hardware optimized for this use case. Meanwhile, Go's goroutine system can significantly speed up computation, but is limited by CPU hardware and cannot run as many concurrent jobs as CUDA can. Another point of interest is that CUDA saw more of a speedup as the file size grew. Again, this is to be expected because CUDA encrypts all the blocks in parallel, so the raw amount of time for execution (as shown in the next graph) does not grow drastically as the amount of data grows.

4.4 Go vs. CUDA



In truth, this is not a fair graph to judge since we are comparing very different languages from very different times running on very different hardware, but we do clearly see that this particular application benefits greatly from the parallelism of a GPU, that the C sequential solution in its most intuitive state is not very fast, and that the Go sequential solution is impressively fast.

5 References

https://kavaliro.com/wp-content/uploads/2014/03/AES.pdf
https://www.youtube.com/watch?v=K2Xfm0-owS4
https://www.tutorialspoint.com/cryptography/block_cipher_modes_of_operation.htm
https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation
https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#Description_of_the_cipher