Parallel Entropy Calculation in Block Ciphers

David Lochridge

May 13, 2014

1 Abstract

Are entropy measurements in block ciphers meaningful and useful measurements? If they are, what kind of attacks and potential strengths or weaknesses do they reveal in the given cipher? Considering cryptanalysis of other areas of cryptography, this paper hopes to address the application of metrics used in those areas to block ciphers in particular, a kind of cryptography commonly used in varied data protection schemes, as well as recently popular crypto-currencies and many other forms of bulk data.

2 Introduction

The problem that I seek to address in this project is whether or not entropy corresponds with the strength of a given block cipher. Because of the large number of calculations that enciphering requires, calculating each of the ciphers serially is not an option, which is by design.

The problem in parallel terms is a weakly scaling one, and therefore is well suited to parallelization. An ideal solution to this problem is actually embarrassingly parallel, with huge numbers of individual processors completing the work for a small subsection of key/cipher combinations, using each processor to expand the percentage of covered work by a set amount.

Because encryption schemes are primarily math-related, and my approach reused a considerable amount of common data, the CUDA paradigm seemed particularly well-fit to accelerating the experiments I ran. With the common plain text and ciphers, I was able to reduce the amount of data sent to the device, at the same time I was able to generate a large number of ciphertexts for analysis, at a linear cost.

3 Methodology

My approach to the problem itself focused primarily on calculating the resulting ciphertexts from similar keys, and combining those results to calculate entropy per bit change. Each cipher ran the same plaintext through a variety of similar

keys, and the results from those keys were compared within their groups to create an average entropy estimate.

With a given cipher text, iterating over all keys would give the exact same amount of average entropy for a single block, but diverged somewhat when more blocks were processed. Because of this, I opted to calculate individual block entropy based on the original key, as well as cumulative entropy calculated over the entire ciphertext.

Calculation of entropy in this case use two kernels, along with copying the data back to the computer. Kernel A handled actual enciphering of a block, while Kernel B calculated both the entropy for the given block, as well as cumulative entropy for the blocks enciphered so far. Synchronization of the entire process was required between the two kernels, as Kernel A had to complete a block before Kernel B could begin its work.

Two of the primary problems that I ran into with making parallel versions of the selected block ciphers were concurrent modification and other race conditions related to program-global variables. The most modern of the ciphers that I used, threefish, was coded in such a way that prevented these problems, by requiring the creation of a cipher instance for each given key. Because of the additional communication that I would have incurred by creating these instances, I opted to generate these siphers on the device, which ultimately served to increase my speed by no statistically significant amount. On the other hand, I had to refactor a considerable amount of the code for the earlier generations of that cipher type, to allow calculations of multiple keys in blowfish, and preventing modification and replacement of the plaintext in place in twofish.

Parallelization

My approach to parallelizing block cipher encryption was, essentially, an embarrassingly parallel one. From the computer itself, a number of different keys were given to the GPU, along with plain text all of the ciphers ran on. Device code for each of the ciphers was then run, creating a cipher text block for each plain text block, with all of the keys running separately, and at the same time. With this approach, the limit on total runtime was primarily determined by the plaintext size.

Additional speedups could be gained from further parallelizing the work by allowing the two separate kernels to run at the same time, with synchronization happening after each ciphertext block was generated, and parallelizing the actual comparisons between ciphers' entropy measurements, instead of running those serially. Because the amount of data sent to the device was only used for the first block of encryption, IO parallelization could only occur when sending the results from the device to the CPU, meaning the speedup gained from parallelizing IO was limited to returning results.

I was unable to get a noticeable speedup from the version written that did not attempt to parallelize the kernels' runs, giving me the impression that either the separate kernels were not run in parallel, or the kernels were parallelized

automatically in the former version. As the CUDA programming guide states, the kernels must specifically be marked as separate to allow parallel execution, which makes me believe the former possibility is true. I'm not entirely sure why this happened, as the versions installed on both system that I ran the experiments on had a high enough version to support that parallelization.

4 Results

Experiment results

The experiment showed that entropy is not a useful measure of cryptographic strength. I had expected that more secure ciphers would give lower entropy in later blocks, meaning that the blocks were more closely related as the key continued its mutation. In reality, there was no discernible trend between the different ciphers, meaning that, regardless of whether a higher or lower entropy would have been more beneficial, none of the ciphers would have been distinguishable from the others in that respect.

Speedup

As expected, the speedup gained by spreading the ciphers over a huge number of blocks was tremendous, with little change in the overall running time given more keys and resources. In contrast, I found that the serial version of the program changed linearly with the addition of multiple keys, and . Though both implementations of the experiment did have a theoretical linear component, the change in the parallel version was primarily related to moving an extremely small key (On the order of 128-448 bits) as opposed to running a full cipher over a similarly sized key, and longer plaintext.

Synchronization The primary factor that affected the parallelization was the serial nature of the ciphers themselves, and the required synchronization between kernels. With short plain texts, this mattered much less, but when the number of rounds required to complete a given cipher grew, performance began to more closely match the serial version. In the trivial parallelization case, in which the , results were slightly worse than the serial version. Adding only a few keys, however,

Communication For the experiment, there were three primary types of communication: key input, plain text input, and results. Key and plain text input both factored into the size of the results, meaning that a balance between sending data to the device, as well as receiving that data played into my selection of key and overall cipher-text result length. Because the results were mostly independent of each other, results per blocks could be sent back while the kernels were continuing to run, which CUDA automatically handles and attempts to do. Similarly, sending plaintext to the device in blocks helped the performance

somewhat, since the kernels could begin running after a single block had been sent. Unfortunately, keys were not separable among the steps, and all of the keys had to be sent before work the first kernel could complete.

Because of the separation of inputs and outputs, I was able to hide a good amount of IO, and free up some of the more valuable device memory as the program went along. Compared to the speedup between the serial and parallel versions, this was a small benefit, but it did mean that including longer plaintexts would have been less of an issue than it would were the IO pushed to the beginning entirely.

5 Future Work

There are a number of avenues to expand parallelizing cryptography calculations and cryptanalysis, even if entropy is somewhat useless. In particular, verifying that certain attacks are effective in general against a cipher would suit parallel applications well. Entropy calculations for hash functions, and other cryptographic ideas, when the metric is actually useful, would benefit considerably because of their independent design, and inclination towards embarrassingly parallel solutions.

6 Conclusion

Entropy calculation for block ciphers, while it can be interesting in certain cases, is not a useful metric for measuring the strength of a block cipher.

Parallelization, howerever, is well suited to comparing block ciphers with other keys or texts of the same cipher, or other ciphers, particularly due to the fact that block ciphers are a set amount of serial work, and the primary benefit parallelization can give is by expanding on the number of calculated values. Strong scaling, in this case, would require the same number of longer messages, which would increase the amount of required serial work, and remove a large amount of the parallelization from the entire process, with little benefit to actually analyzing the given cipher.