# cs378: CUDACha20

Phuc Dang
Department of Computer Science
UT Austin

December 5, 2023

## 1 Introduction

The ChaCha20 cipher is very easy to think about in terms of parallelizing it. The keystream is generated in 64-byte blocks, and generating the keystream is done via a counter-mode-esque construction of the stream cipher. This means that the generation is not dependent on any past state (i.e. you can manually set the counter for a block, then consume that state to get a block of the keystream). This is trivially parallelized where every thread can generate a specified chunk of the keystream (containing many blocks) and individually encrypt chunks of the message.

The security assumption for this cipher is that it acts like a pseudorandom function (PRF) where it only evaluates one way and is hard to invert. Assuming properties of a PRF, changing the counter of the state by just one bit thus result in an entirely different block output that looks indistinguishable from a randomly sampled byte string. When the block output is xor'd with the message, what you get is also indistinguishable from a randomly sampled byte string. To be actually secure, this cipher would need to be paired with its corresponding message authenticator, namely Poly1305. However, for purposes of performance testing a cipher, I forego this extra step. Some additional unsafe things included in the code is a hard-coded key and nonce. This is for convenience and ease of testing and totally unsafe in practice.

In this project, I derive my serial implementation directly from the specifications in IETF RFC 8439 [1] and try to corroborate the experimental data found by this paper [2]. One big difference is that I will try to optimize each approach (single, multi, GPU) rather than normalizing every parameter across the board. Multi-threading is provided by the UNIX pthreads library, and GPU is provided by the CUDA C++ library.

## 2 Specifications

- CPU: AMD Ryzen 7 5800x 8 Cores 16 Threads @ 4.8 GHz
- GPU: NVIDIA Geforce RTX 3060 Ti
- NVCC Flags: -O3, –use-fast-math (mainly for optimized GPU computations since floats are not needed)
- All tests are ran 5 times and averaged.

## 3 Serial Implementation

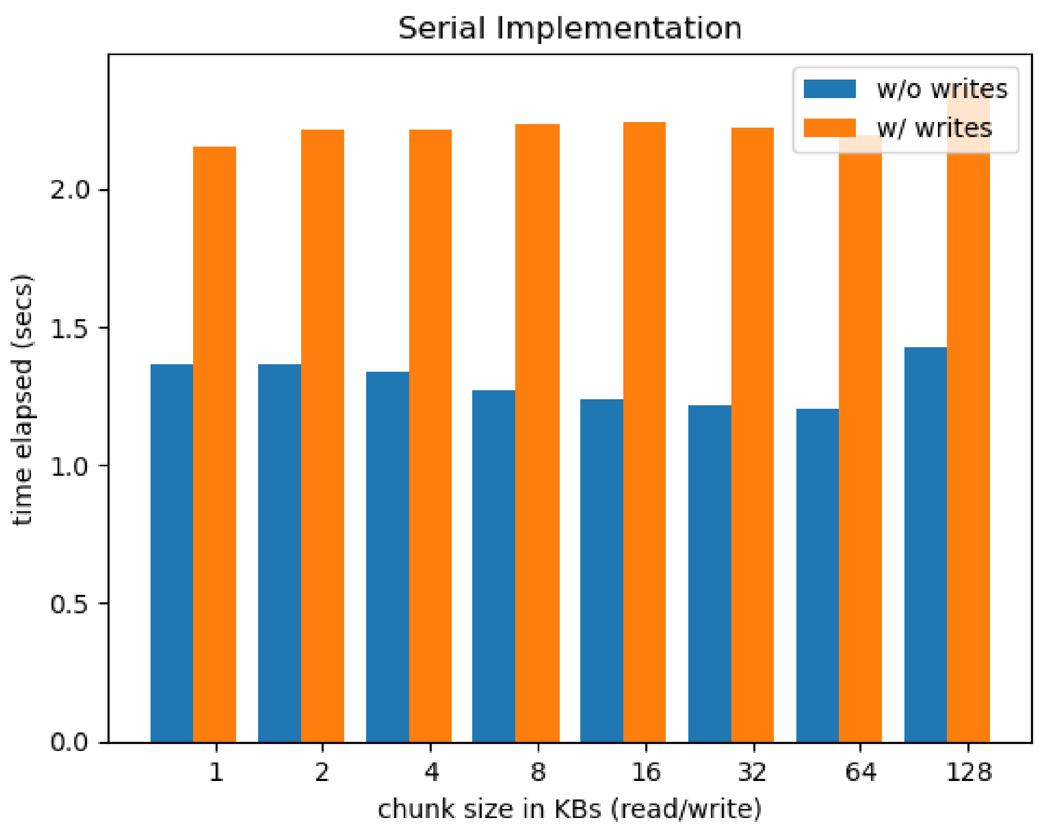


Figure 1: Multi-thread implementation comparison reading/writing in 64 KB chunks.

Logically, reading and writing in larger chunks means that the program spends less time accessing the disk, which would speed up the program. At a certain point though, the performance gains seem to stop. It caps at 64 KB at about 1.2 seconds without writing and 2.2 seconds with writing. It doesn't seem to really make any sense though. How can it be the case that a larger chunk size causes such a spike? I speculate that the process might have been descheduled as it takes too long on a chunk size of 128 KB, so that temporary state of sleeping would cause it to begin adding overhead.

With the clock rate of my CPU being around 4.8 GHz as this was running, I am able to achieve about 5.7 cycles/byte encrypted for the version without writes. This is pretty good for a base implementation that does not utilize any specialized SIMD CPU instructions.

In any case, I will now use a 64 KB chunk size for testing the multi-threaded implementation since it seems to be most optimal here.

## 4 Multi-Threaded Implementation

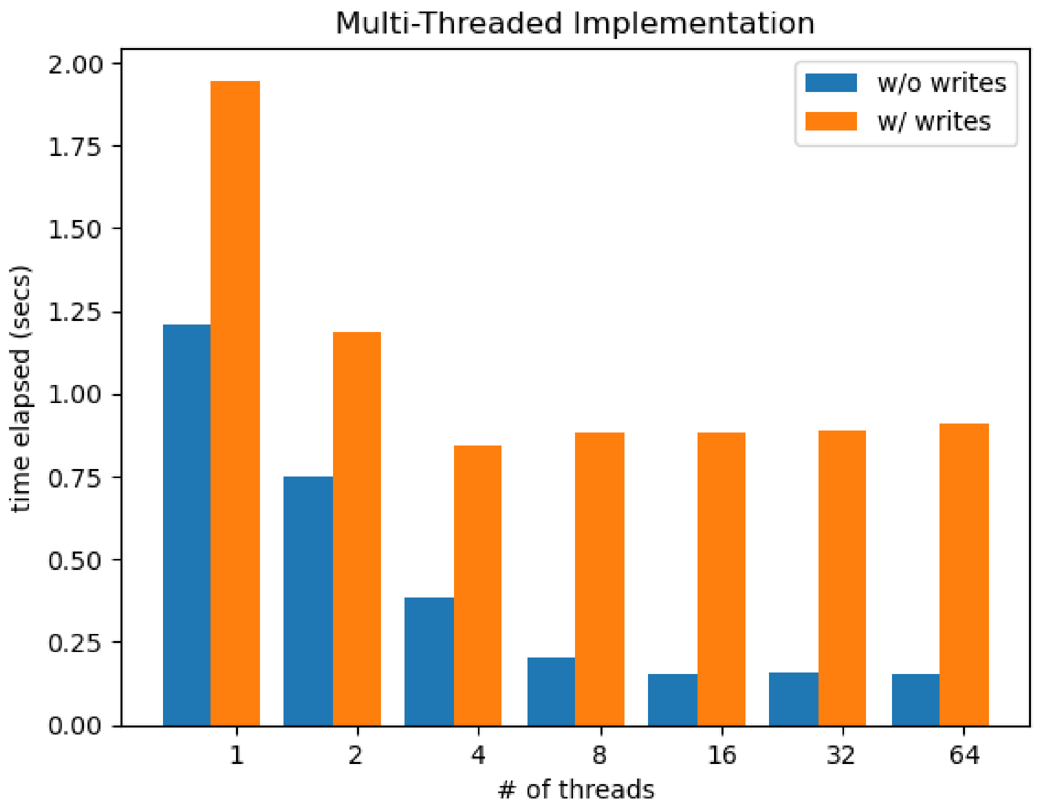


Figure 2: Comparison across number of threads between read-only and read/write.

Both reading and writing had their own respective input and output. These streams are shared between all the threads which would necessitate using a synchronization primitive. I chose a mutex lock: one for reading, one for writing.

Given the triviality of parallelizing the cipher, we can very clearly see Amdahl's law in action. The majority of the work that is done by this cipher can be parallelized, showing on the graph as a strongly scaling program with respect to the number of threads active. The speed is only limited by the number of threads and cores on my CPU.

But given that I had opted for a mutex surrounding the critical sections, higher thread counts did see a decrease in performance due to higher contention on the lock. The sweet spot for both reading and writing is exactly at 16 threads.

## 5 GPU Implementation

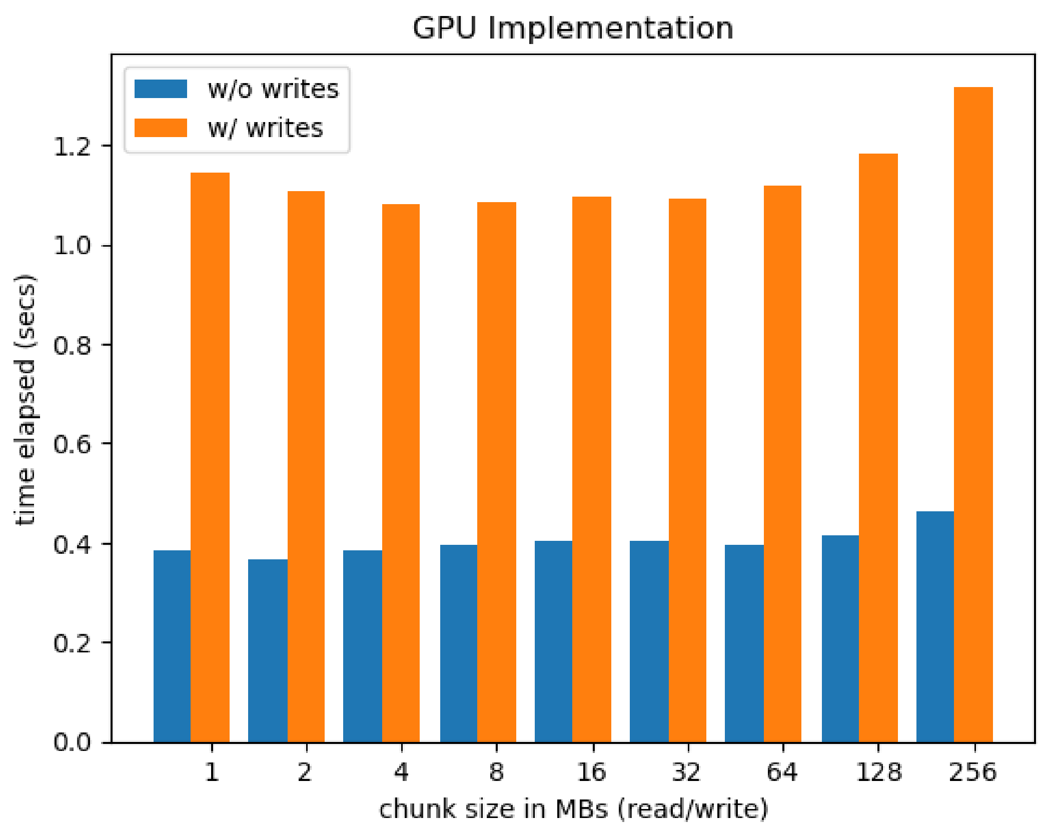


Figure 3: Comparison across different chunk sizes for reading and writing.

This is where a lot of interesting things happen. The current implementation spawns two threads, one for asynchronously reading and one for asynchronously writing. The keystream generation and encryption all happens on the GPU. Each GPU thread is responsible for generating a single block of the stream (which is just 64 bytes) and encrypting 64 bytes of the message. Thus, the chunk size determines how many GPU threads the program should spawn. For example, with 128 MBs, the program would spawn a total of 2097152 GPU threads. I have found that 128 threads per block generally performs the best, resulting in 16384 blocks. Within the kernel, there is a shared memory array of unsigned int$[16 + (\text{THREADS_PER_BLOCK} * 16)]$. This is to speed up local access for each GPU thread performing the ChaCha quarter rounds.

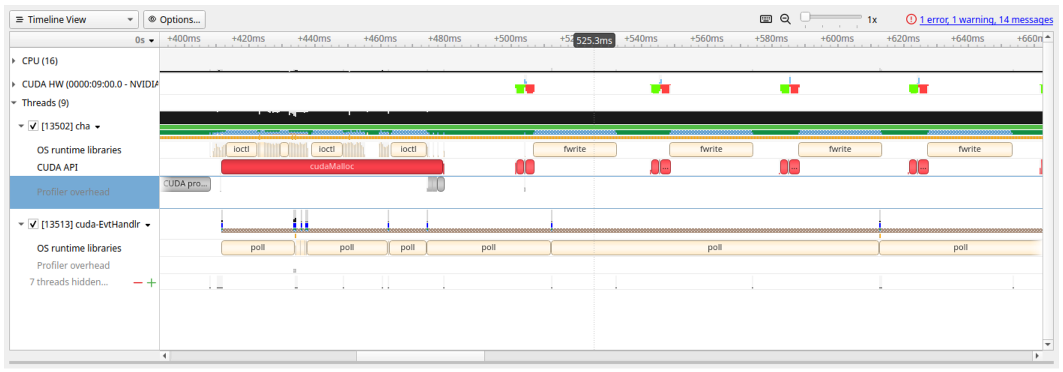


Figure 4: The function fwrite (and fread though not shown) vs the actual time spent on the GPU.

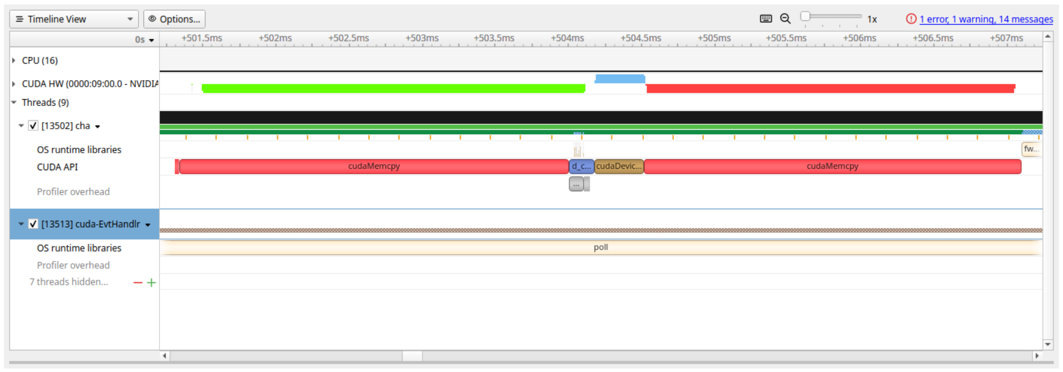


Figure 5: Speed of the kernel with respect to the amount of time spent copying. This is prior to fwrite being asynchronous.

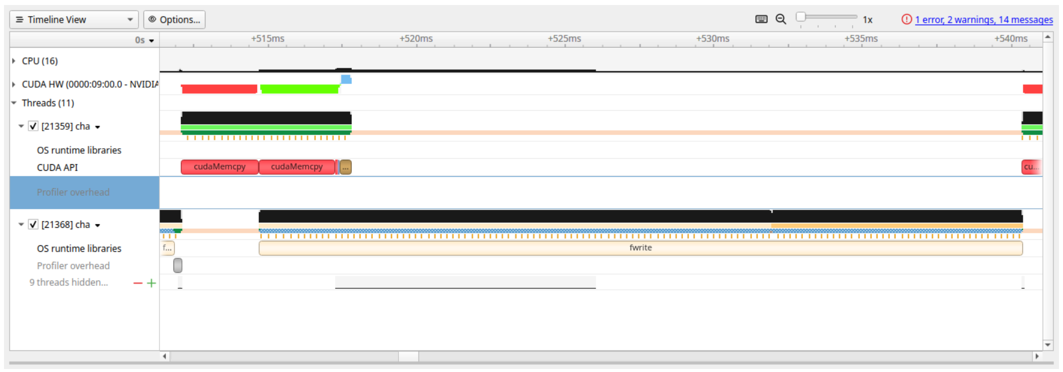


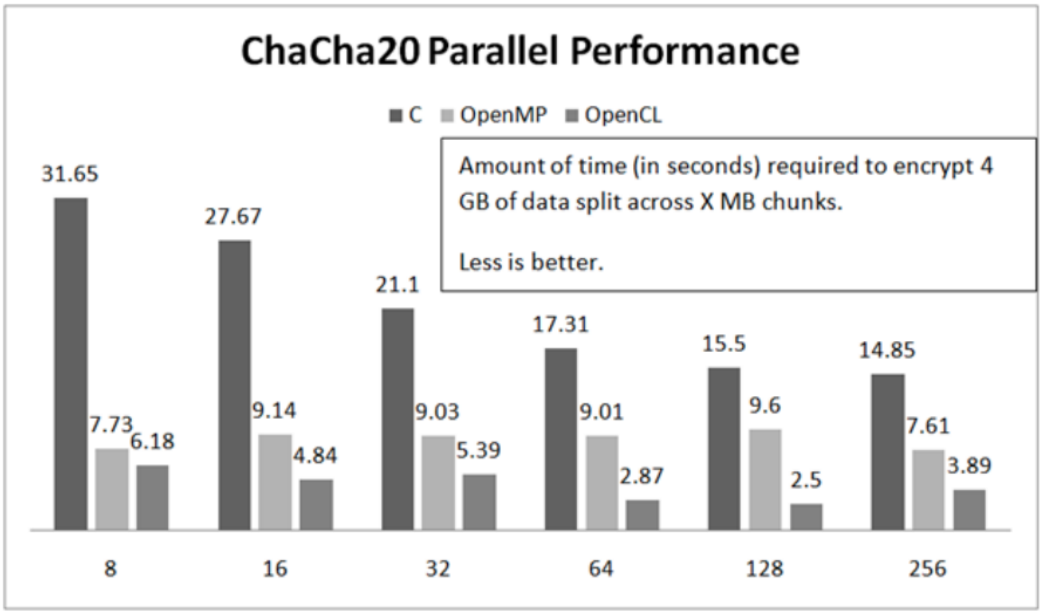
Figure 6: Comparison across different chunk sizes for reading and writing.

Before the current implementation, however, I had an implementation that was fully serial for both reading and writing. Figure 4 and 5 shows this clearly as there are gaps in between CUDA API calls. Reads and writes to the disk are costly. The speed of this program is capped by the speed at which the disk can respond to these massive requests. The way I had optimized the GPU kernel certainly helped increase the program's speed when I had not yet implemented asynchronous reading and writing, but this optimization of the GPU kernel was ultimately fruitless for outperforming the multi-threaded version. Figure 6 shows why.

Even with constant execution of the code (fwrite continuing in the background while the GPU takes care of the new buffer), it is still bottlenecked by fwrite. Until fwrite is finished, the program cannot continue with the new result from the GPU until the old has been fully written into the disk. The entire program is serialized by parameters that are mostly out of my control. The only thing that worked for marginally improving performance was tweaking the chunk sizes that the GPU implementation would deal with. In this case, 2 MBs seem to be the best option according to figure 3.

This is, more or less, the ultimate and final form of Amdahl's law. I have, to the best of my abilities, parallelized all there was to parallelize. Of course, if I were to uncap memory limits, I could perhaps buffer both reads and writes so that the GPU could just continue. I could spawn more writers in that case. But within the program limits, it seems that all that is left are unavoidable serial sections.

## 6 Analysis



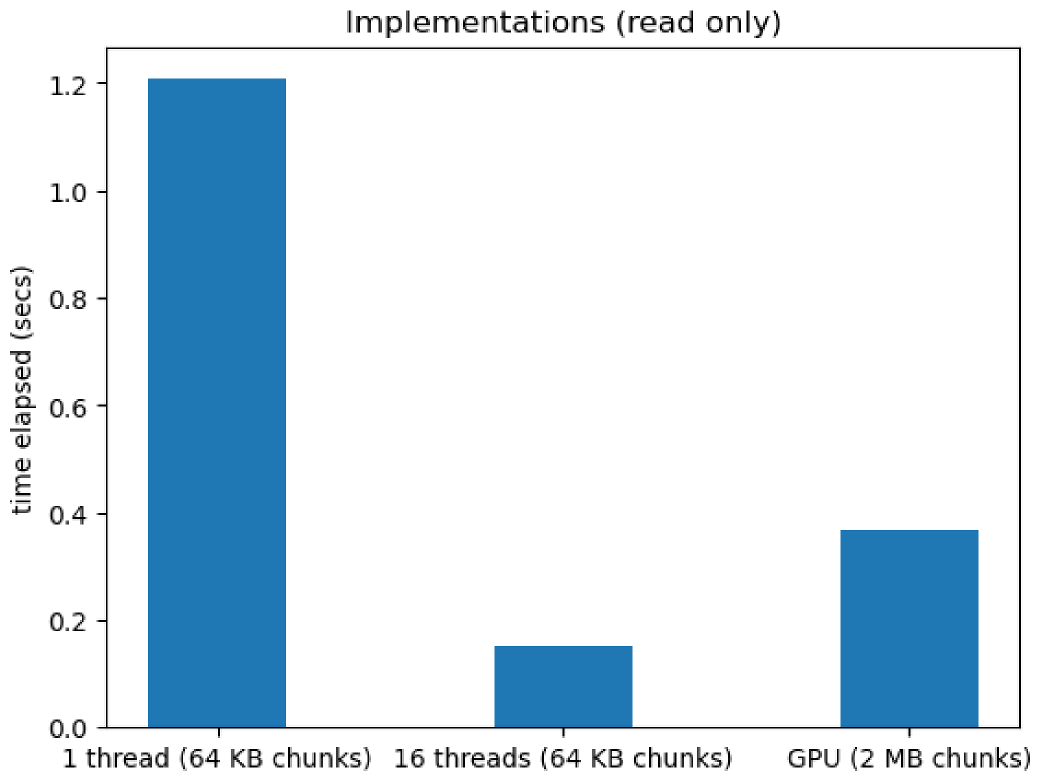


Figure 7: Comparison between the paper's results [2] and the best of my results.

Although the paper did encrypt 4 GB of data, I can, without loss of generality, compare it to what I have encrypting just 1 GB of data. They ran their experiment on older hardware so it makes sense that my results are faster. Troublingly, I could not get my GPU results faster than my multi-threaded version. The GPU version runs two times slower than using 16 CPU threads. Analyzing the program in NVIDIA Nsight Systems tells me that managing memory took up a lot of time relative to just running the kernels. This is definitely a part of the reason why the GPU runs slower than 16 CPU threads.

Another discrepancy is how significantly affected their runtime was as the chunk sizes increase. The general trend for the paper is that the larger chunk sizes run faster whereas it was the opposite for me. I could maybe explain this through the fact that the paper could have been running the experiment on a hard drive instead of an SSD. Hard drives incur heavy impacts on performance if the program constantly requests operations from it; hence, why it seems like increasing the chunk size greatly improved performance for them. I am not too sure what the analogous situation is for SSDs but keeping my reads/writes relatively small (compared to the paper) improves performance.

I cannot help but feel like a lot of what I observed in this project could be massively improved if the GPU did not need to have its own copy of the data. If it was possible to just have the device directly operate on memory already malloc'd on the system, I imagine that it would improve performance quite a bit. All the overhead of copying from host to device and vice versa would disappear, leaving only the performance of the disk as the final arbiter of performance.

## 7 Conclusion

Realistically, the GPU probably is not the best choice when it comes to cryptography performance, especially when it is for a cipher that is not that intensive on CPU computing. Furthermore, a lot of cryptography relies on operations done over integers, something that a CPU can handle fairly well by itself. A lot of crypto also just is not parallelizable, due to so many of the schemes requiring some kind of past state to operate on.

And also realistically, the viability of parallelizing the ChaCha20 cipher is heavily dependent on the context of which the cipher is used in. It is firstly a stream cipher, meaning that it is typically used in applications that stream data over some network. We could indeed parallelize the cipher but at the cost of having to implement a system that buffers the cipher results such that packets are sent/received in the correct order. This could prove quite difficult and tricky. In a server setting, where it talks to $n$ clients using $n$ threads, the best implementation of this ChaCha20 cipher would likely be the serial version merely due to its simplicity. The serial version is quite fast after all, only getting faster when utilizing specialized CPU instructions.

I do plan to revisit this project sometime later. Maybe I could polish up and tweak some other parameters that I have yet to discover for CUDA. Maybe then it might run just as fast or faster than my multi-threaded implementation.

## References

[1] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018.

[2] Radu Velea, Florina Gurzău, Laurenţiu Mărgărit, Ion Bica, and Victor-Valeriu Patriciu. Performance of parallel chacha20 stream cipher. In *2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 391–396, 2016.