# ECE1782 Final Project
# CUDA Accelerated AES Encryption

Yinfei Li, Sipeng Liang, Lang Sun, Gan Yang

# 1. Overview and Problem Statement

Advanced Encryption Standard (AES) is a widely used encryption standard in computer security, such as encrypting data in communication via the internet and compressing programs. Counter Mode (AES-CTR) is a mode of the AES encryption standard. In CTR mode, an incrementing initialization vector (IV) is generated and assigned to each cipher block. This assigned IV value is then encrypted with the key to produce an intermediate value. Finally, the intermediate value is XORed with the plaintext to generate the encrypted text.

Figure 1 is a diagram of CTR mode encryption. In this graph, the IV is increased by 1 for every block of data. Then this incrementing IV is combined with the key to generate the block cipher for applying encryption on the plaintext.

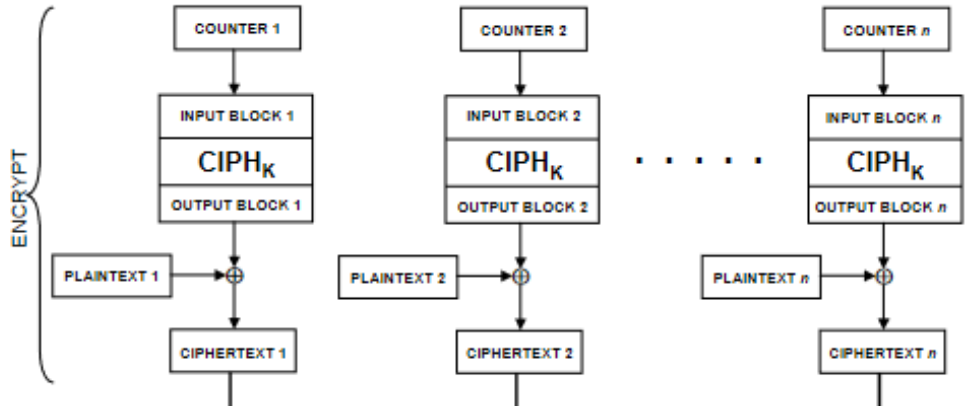



Figure 1. CTR Mode Encryption [1]

Applying AES encryption on large amounts of data such as a novel or video requires a scheme of utilizing the block cipher repeatedly without compromising information security. This process usually takes a lot of time and computational power. Since the block cipher by itself only operates on a fixed length of data, and the repeating nature of applying block ciphers under CTR mode of operation, the encryption is highly parallelizable. This is because, in AES CTR mode, there is no dependency for any input of a block cipher on any results from other block ciphers and each block cipher can execute independently. This will allow multiple ciphers to run at the same time on the GPU to accelerate the process of AES encryption.

# 2. Program Flow

The AES specification supports three different key lengths: 128, 192, and 256 bits. For the purpose of the project, the 128-bit key version will be implemented.

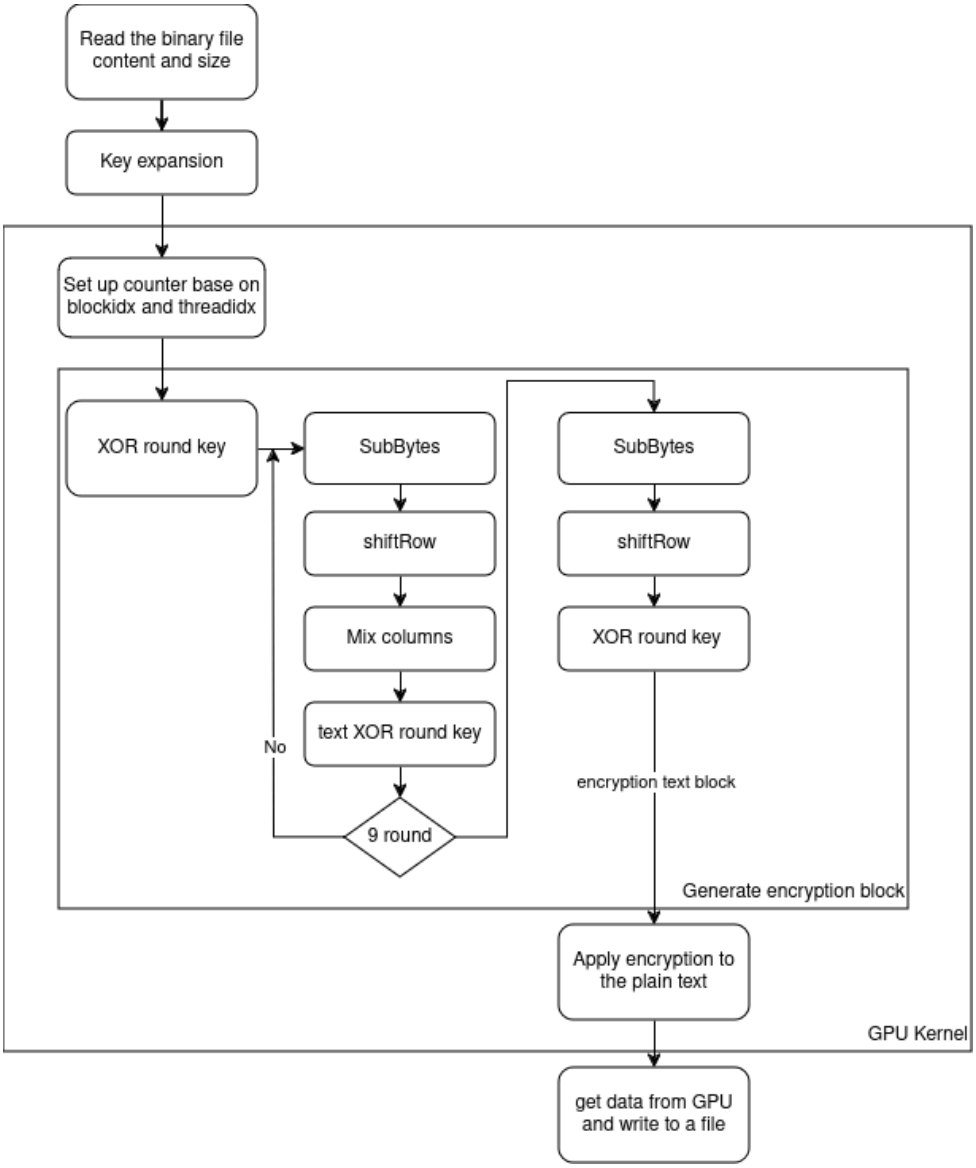



Figure 2. CUDA program flow diagram

Figure 2 is a flow diagram of the basic version code which is implemented from the CPU code and without applying any optimization. Firstly, the CPU reads the binary file from the disk and expands the key stored in the disk from 4 words to 44 words. Each round key contains 4 words. Thus, the total of 44 words allows 11 rounds in total. The input data is divided into small chunks with the size of 128 bits per thread and 256 threads per block. Then the data is passed to the GPU kernel to encrypt independently.

In the GPU kernel, firstly, the IV is calculated based on the counter value which is derived from the block ID and thread ID of the GPU kernel. Then the intermediate value of the encryption block is generated from the IV and the key by going over 10 rounds of encryption. There are 4 steps for each round: substitute Bytes, shift rows, mix columns

and add round keys. The only exception is the last round (10th round) which does not have a mix column step. Finally, the plain text is XORed with the intermediate value to form the output ciphertext.

# 3. Improvements

## 3.1 Improvement V1: Constant Memory, Shared Memory, and Pinned Memory

In the first version of the improvement, the s-box is placed in the constant memory. S-box is a 16-by-16 byte lookup table that is used for substituting bytes. The value is pre-calculated and stored as an initialized global array in the code. Since there is no intention of modifying this array, it is decided to put it into the constant memory region of the GPU. The GPU constant memory region is cached, so this promotes a speed increase when reading from the array in the byte substitution step.

The shared memory is allocated for the IV and expanded key. Both variables are frequently used in the process of encryption. Putting them in the shared memory reduces the memory access time when doing calculations.

Pinned memory is allocated for the plaintext and ciphertext. The pinned memory prevents the memory from being page-swapped out of the main memory, therefore avoids page faults during the memory operation. This allows a faster data transferring time bet en CPU and GPU.

## 3.2 Improvement V2: Coalesced Memory Access

In the previous version, each thread accessed a different block of the plaintext and ciphertext arrays. If the blocks are not contiguous in memory, this could slow down the program. The second version of the program rearranges the data so that the blocks accessed by threads in the same warp are contiguous in memory.

## 3.3 Improvement V3.1: Divergence Avoidance By Padding Data

There was a conditional statement “if (blockId < numBlocks)” in the code. This divergence can be avoided by ensuring that the number of threads is a multiple of the number of blocks, which means padding the data to a multiple of the block size.

### 3.4 Improvement V3.2: Divergence Avoidance By Replacing Conditions

In the mix columns step, byte multiplications under the Galois finite field are performed. In the previous versions, there are two if conditions. In this version, the if conditions are replaced with arithmetic operations. This ensures all threads in a warp take the same execution path, so the divergence is avoided.

### 3.5 Improvement V4: Loop Unrolling and Intrinsic Function

In this version, loop unrolling is added to small loops to eliminate loop control overhead and compute-focused loops to allow for more instruction-level parallelism. Large loops are not unrolled since it would increase the register pressure. The memory-focused loops are also not unrolled since it would lead to instruction cache misses. The fast build-in intrinsic function `__ldg()` is included to load and cache the IV and expanded key.

### 3.6 Improvement V5: Streams

The stream strategy is applied for the kernel in this version. However, due to the host side increment_variable function execution between kernel lunches, the kernel streams were not executing in parallel but in serial. The increment code was used to calculate a unique IV for every stream or data chunk, but since it was a host-side function, which naturally executes in serial, the code blocked the next kernel launch until it was finished. As a result, this version was a negative improvement.

### 3.7 Improvement V5.1: Better Streams and Reduced Register Usage

Based on the findings in V5, Modification was performed to precalculate all necessary variables before launching the kernel, which made every stream truly independent, which means that every stream can execute in parallel without any issue compared to the last version. Therefore, data transfer and kernel execution can be overlapped as intended. In addition, from the Nsight Compute report (Appendix Figure 23. - Figure 25.), it was found that using a limited number of registers could allow the program to achieve max warp occupancy. As a result, the Byte multiplication function was modified and the AES-encrypt-block function was merged with the kernel to reduce local variables and reduce register usage. The register usage per thread was brought down from 249 to 40. This version is able to achieve the best result so far.

# 4. Evaluation

The evaluation was done by comparing the file encryption time taken by the CUDA implementations, a 4-thread OpenSSL implementation, and a naive single-thread CPU implementation. A 1.4GB 4K HDR video was used for large-size tests. A 6.2 MB ebook of Sherlock Holmes was used for medium-size tests. A small section of an ebook was extracted to a 478B file for small-size tests. For small and medium-size tests, encryption is tested 1000 times and averaged the runtime. For large-sized tests, 10 runs are averaged.

As shown in Table 1 and Figure 3, for small-size files, the CUDA implementations did not have any advantage compared to the OpenSSL version, since there was overhead in transferring data between CPU and GPU, as well as the kernel launch and execution overhead. However, all these CUDA versions still outperformed the naive CPU implementation.

| **Implementation** | **Total** | | | **Kernel** | | |
|---|---|---|---|---|---|---|
| | **Runtime (us)** | **Perf Gain (Incremental)** | **Perf Gain (Based on V0)** | **Runtime (us)** | **Perf Gain (Incremental)** | **Perf Gain (Based on V0)** |
| **1-thread CPU** | 2000000 | | | N/A | | |
| **4-thread OpenSSL** | 0.975000 | | | N/A | | |
| **CUDA v0** | 54.409 | baseline | baseline | 11.168 | baseline | baseline |
| **CUDA v1** | 84.251 | -54.85% | -54.85% | 41.76 | -273.93% | -273.93% |
| **CUDA v2** | 61.593 | 26.89% | -13.20% | 19.2 | 54.02% | -71.92% |
| **CUDA v3.1** | 66.6 | -8.13% | -22.41% | 19.2 | 0.00% | -71.92% |
| **CUDA v3.2** | 62.093 | 6.77% | -14.12% | 19.264 | -0.33% | -72.49% |
| **CUDA v4** | 83.727 | -34.84% | -53.88% | 43.552 | -126.08% | -289.97% |
| **CUDA v5** | 1039 | -1140.94% | -1809.61% | 42.08 /stream | 1.73% | -283.24% |
| **CUDA v5.1** | 789.155 | 24.05% | -1350.41% | 62.784 ~ 222.305 /stream | -233.05% | -1176.37% |

Table 1. The benchmark result for the 478B text file

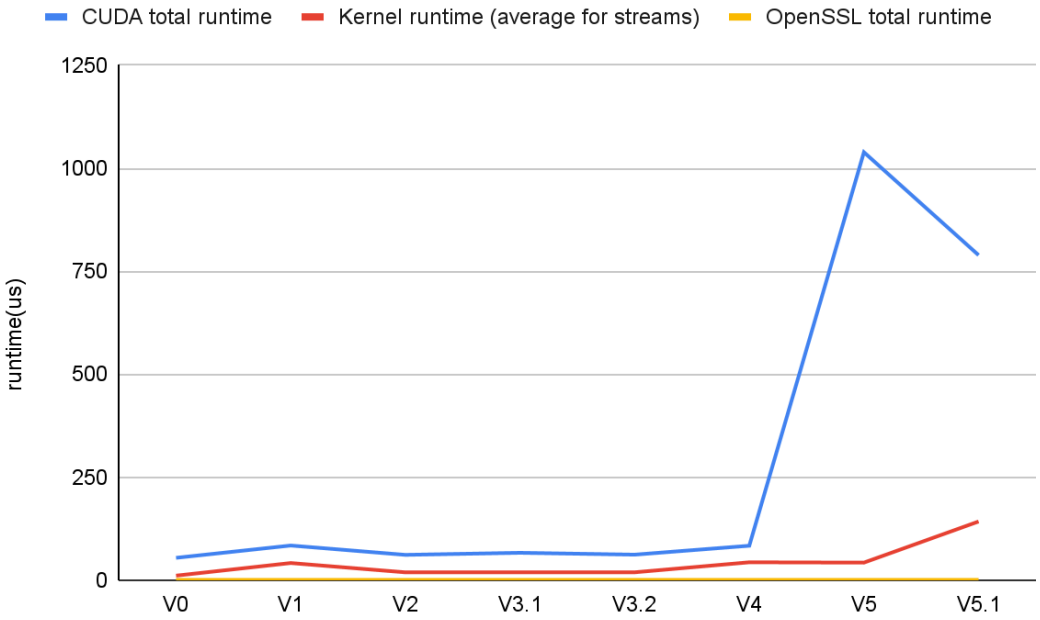



Figure 3. Different versions of CUDA runtime for the 478B text file

As shown in Table 2 and Figure 4, for medium-size files, the kernel was faster than 4-thread OpenSSL. The last version only took 42% time compared to OpenSSL in encryption. If the data transfer time is included, the CUDA version could not match the performance of OpenSSL.

| **Implementation** | **Total** | | | **Kernel** | | |
|---|---|---|---|---|---|---|
| | **Runtime (us)** | **Perf Gain (Incremental)** | **Perf Gain (Based on V0)** | **Runtime (us)** | **Perf Gain (Incremental)** | **Perf Gain (Based on V0)** |
| **1-thread CPU** | 29348000000 | | | N/A | | |
| **4-thread OpenSSL** | 535.221000 | | | N/A | | |
| **CUDA v0** | 2964 | baseline | baseline | 281.217 | baseline | baseline |
| **CUDA v1** | 1246 | 57.96% | 57.96% | 719.137 | -155.72% | -155.72% |
| **CUDA v2** | 1363 | -9.39% | 54.01% | 724.252 | -0.71% | -157.54% |
| **CUDA v3.1** | 1286 | 5.65% | 56.61% | 724.034 | 0.03% | -157.46% |
| **CUDA v3.2** | 1251 | 2.72% | 57.79% | 724.609 | -0.08% | -157.67% |

| | | | | | | |
|---|---|---|---|---|---|---|
| **CUDA v4** | 1394 | -11.43% | 52.97% | 829.826 | -14.52% | -195.08% |
| **CUDA v5** | 2277 | -63.34% | 23.18% | 83.329 / stream | 89.96% | 70.37% |
| **CUDA v5.1** | 766.133 | 66.35% | 74.15% | 111.488 ~ 283.776 / stream | -137.17% | 29.72% |

Table 2. The benchmark result of the 6.2MB text file

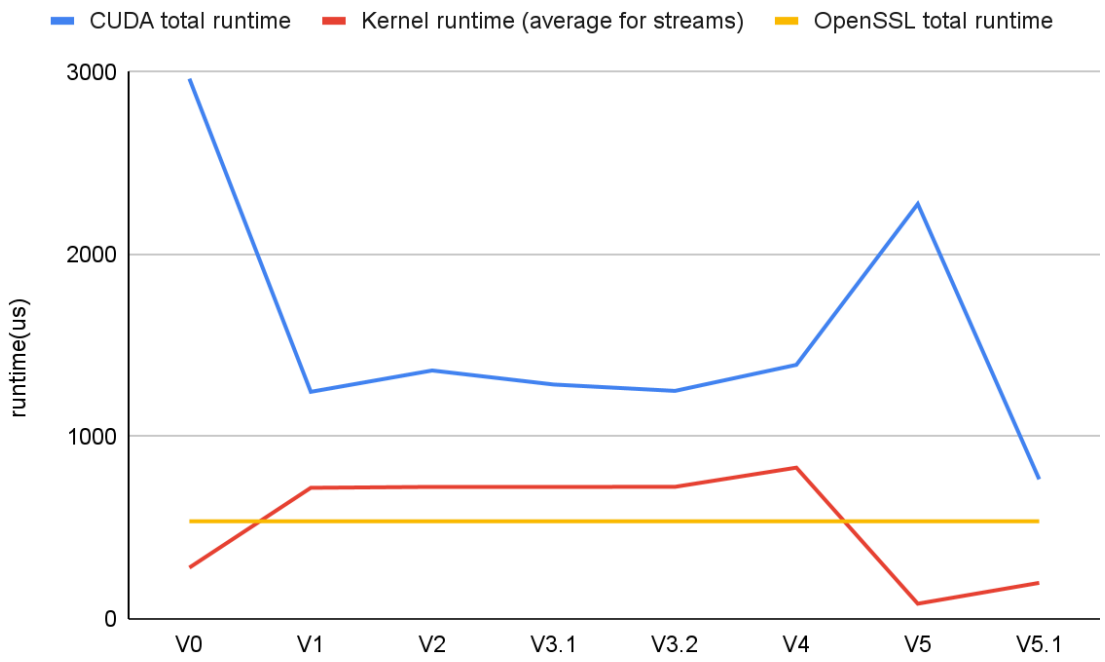



Figure 3. Different versions of CUDA runtime for the 6.2MB text file

As shown in Table 3 and Figure 5, for large-size files, the advantage of the kernel against 4-thread OpenSSL was even larger. V3.2 only took 35% time compared to OpenSSL in encryption. However, the total time including data transfer was still larger than OpenSSL.

| **Implementation** | **Total** | | | **Kernel** | | |
|---|---|---|---|---|---|---|
| | **Runtime (us)** | **Perf Gain (Incremental)** | **Perf Gain (Based on V0)** | **Runtime (us)** | **Perf Gain (Incremental)** | **Perf Gain (Based on V0)** |
| **1-thread CPU** | N/A | | | N/A | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| **4-thread OpenSSL** | 135348.336000 | | | N/A | | |
| **CUDA v0** | 748519 | baseline | baseline | 60801 | baseline | baseline |
| **CUDA v1** | 300097 | 59.91% | 59.91% | 173537 | -185.42% | -185.42% |
| **CUDA v2** | 270727 | 9.79% | 63.83% | 157921 | 9.00% | -159.73% |
| **CUDA v3.1** | 267597 | 1.16% | 64.25% | 151953 | 3.78% | -149.92% |
| **CUDA v3.2** | 270472 | -1.07% | 63.87% | 158116 | -4.06% | -160.05% |
| **CUDA v4** | 300726 | -11.19% | 59.82% | 181801 | -14.98% | -199.01% |
| **CUDA v5** | 290314 | 3.46% | 61.21% | 11220 /stream | 93.83% | 81.55% |
| **CUDA v5.1** | 164477 | 43.35% | 78.03% | 9824 ~ 10002 / stream | 11.65% | 83.70% |

Table 3. The benchmark result of the 1.4GB video file

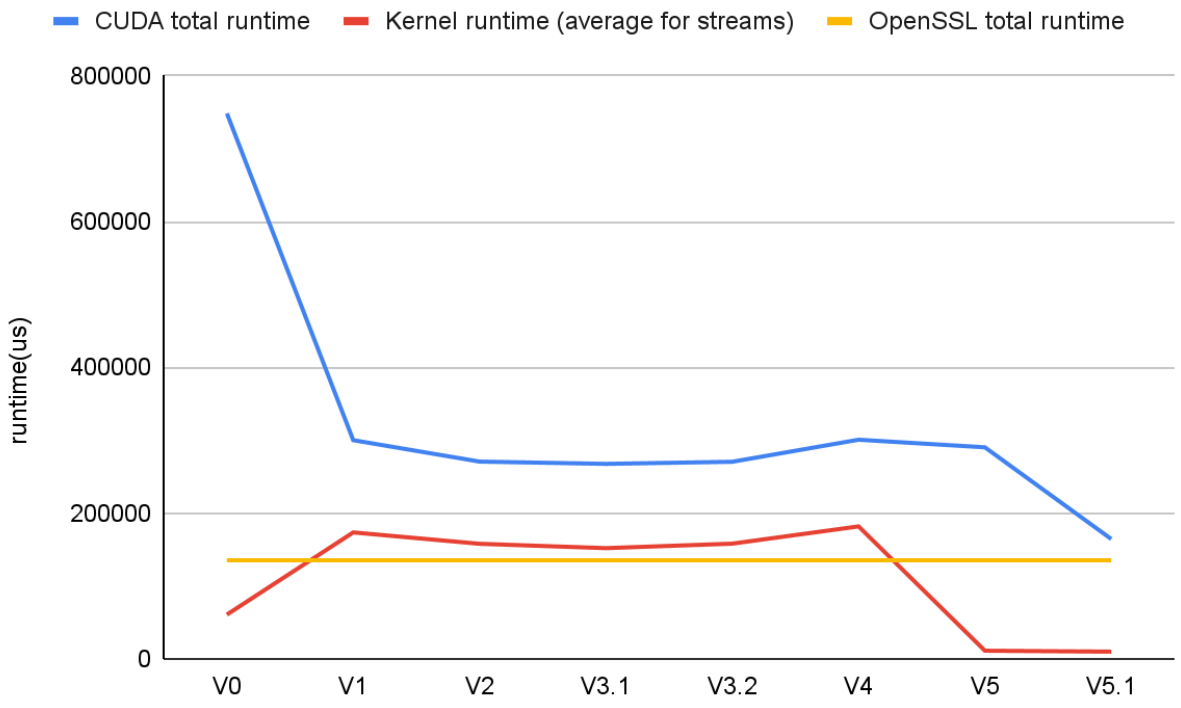



Figure 4. Different versions of CUDA runtime for the 1.4GB video file

## 5. Discussion

This group treats this project very seriously. Significant effort is spent on multiple tool scripts that compare encryption and decryption results to OpenSSL to guarantee the correctness of the implementation. This effort is not directly related to the requirement of the project. But out of the interest of the group to this project topic, such effort is still being made, which is proven worthy later in the project.

In general, small-size files do not illustrate the potential of the CUDA implementation.
The result based on large-size files better demonstrates CUDA implementation's supremacy.

Based on the result shown in the Evaluation section. The GPU version of the code had a great increase in speed compared to the single-thread CPU version of the code. After a few optimizations, the final version of the CUDA total runtime became 22% of the first version. The biggest speed-up was achieved by using the pinned memory from v0 to v1. It reduced 82% data transfer time. From v2 to v3.1, the memory coalescing and data padding problem is addressed and reduced by 4% kernel runtime. In v3.2, the branch divergence from the conditions in the mix column step was replaced by an arithmetic operation. However, it did not provide any actual improvement in the performance. In addition, the loop unrolling and intrinsic function in v4 were proven to be negative optimization. Also, the immature stream implementation in v5 caused the biggest regression in the results. In the end, v5.1 was developed, which solved the stream issue in v5 and achieved the best results. It is shown that the stream strategy is highly dependent on the computation size. Small-size computation with a large number of streams will have a negative impact on performance. Stream is also sensitive to code implementation and data independence. Dependency between data and host-side execution will both cause the stream to execute in serial, which defeats the purpose of using stream to overlap executions.

Although huge improvements are achieved in encrypting speed compared to the single-thread CPU code, the total runtime was still 22% longer than the 4-thread OpenSSL implementation. This was caused by the huge overhead in data transferring time, the kernel launching time, and that OpenSSL utilized the specialized Intel AES Instruction Set in the lab setup which provides SIMD acceleration even on the CPU [2].

There is also an attempted version that utilizes CPU multithreading to increase the speed of capturing data from files and launching the kernel. Eventually, this far-reaching idea is discarded due to the limited time frame of this project and the complexity of combining multithreading and Stream.

Overall, the goal of this project has been achieved. Both the CPU and GPU code are implemented, and the results are verified with OpenSSL and benchmarked all different implementations use different sizes of payloads. Also, the Nsight Compute and Nsight System are used to provide in-depth information on the performance, the execution process, and the hardware infrastructure of the implementations.

## 6. Related work

Back in 2009, Biagio et al. [3] introduced an implementation of AES-CTR using CUDA and proved the huge advantage in performance and cost of GPU compared to CPU in solving this problem. In 2017, the authors of [4] conducted experiments on the effects of different input sizes and threads per block. In 2019, Hajihassani et al. [5] achieved 1478 Gbps AES-CTR encryption throughput on an Nvidia Tesla V100 GPU. Three years later, the authors of [6] got 9% faster than [5] and removed a limitation due to a hardcoded implementation. Outside the academic field, there is a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication, OpenSSL. [2] The OpenSSL project provides open source access to cryptography software tools including CPU side implementation of AES-CTR and more.

## 7. Contributions

1. **Sipeng Liang**
   CPU code, naive single thread implementation, scripts.
2. **Lang Sun (contributed the most)**
   GPU code, CUDA code optimization and analysis, and supplementary scripts.
3. **Gan Yang**
   OpenSSL code, benchmark code, code refactoring, Makefile, and README.
4. **Yinfei Li**
   Paperwork

## 8. Reference

[1] Morris Dworkin (NIST), NIST Special Publication 800-38A, [online], https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38a.pdf

[2] OpenSSL Project Authors, “Cryptography and SSL/TLS Toolkit,” https://openssl.org/

[3] A. D. Biagio, A. Barenghi, G. Agosta and G. Pelosi, "Design of a parallel AES for graphics hardware using the CUDA framework," 2009 IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, 2009, pp. 1-8, doi: 10.1109/IPDPS.2009.5161242.

[4] J. Ma, X. Chen, R. Xu, and J. Shi, "Implementation and Evaluation of Different Parallel Designs of AES Using CUDA," 2017 IEEE Second International Conference on Data Science in Cyberspace (DSC), Shenzhen, China, 2017, pp. 606-614, doi: 10.1109/DSC.2017.19.

[5] O. Hajihassani, S. K. Monfared, S. H. Khasteh, and S. Gorgin, "Fast AES Implementation: A High-Throughput Bitsliced Approach," in IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 10, pp. 2211-2222, 1 Oct. 2019, doi: 10.1109/TPDS.2019.2911278.

[6] W. K. Lee, H. J. Seo, S. C. Seo and S. O. Hwang, "Efficient Implementation of AES-CTR and AES-ECB on GPUs With Applications for High-Speed FrodoKEM and Exhaustive Key Search," in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 69, no. 6, pp. 2962-2966, June 2022, doi: 10.1109/TCSII.2022.3164089.

# 9. Appendix

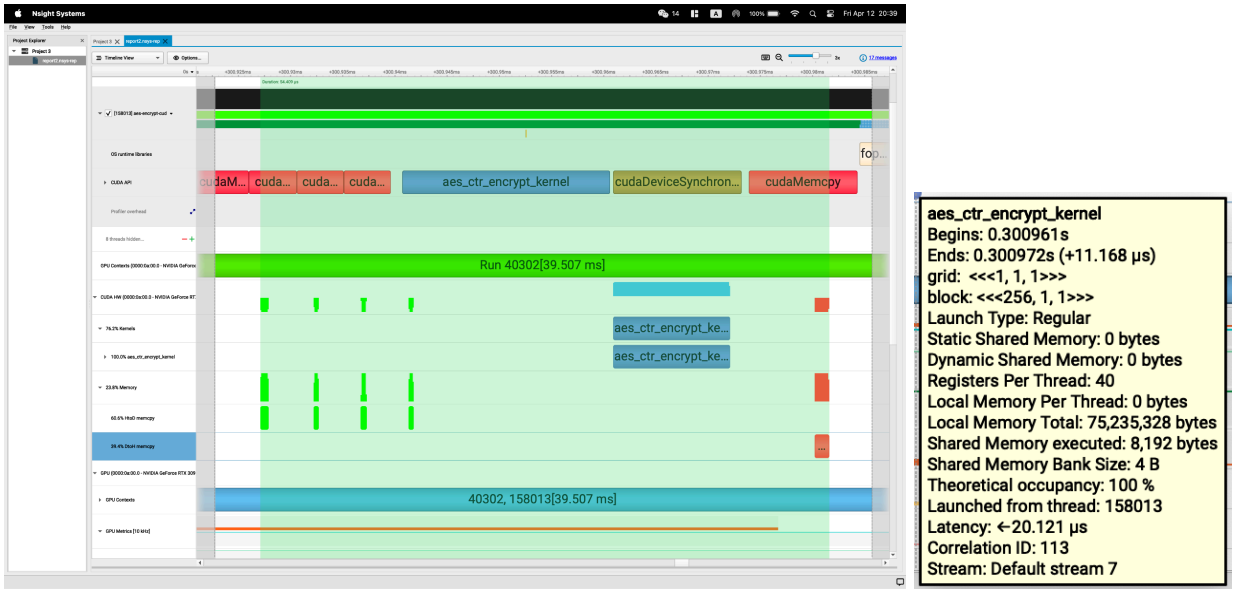



Figure 5. v0 small.txt

Total run time: 54.409us
Kernel run time: 11.168us

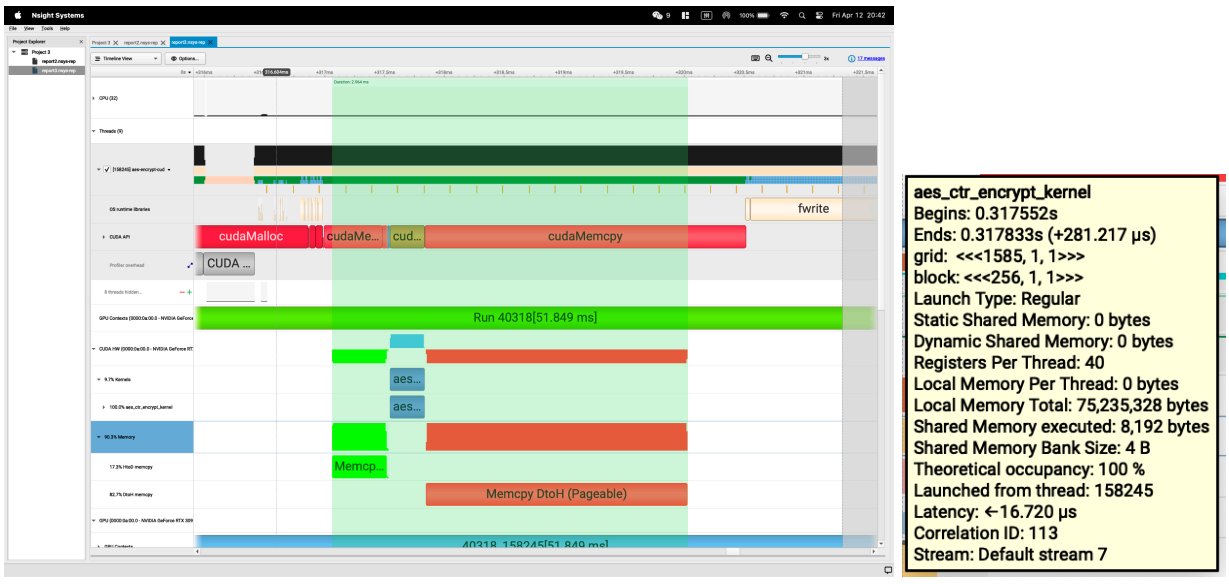



Figure 6. v0 big.txt

Total run time: 2.964ms
Kernel run time: 281.217us

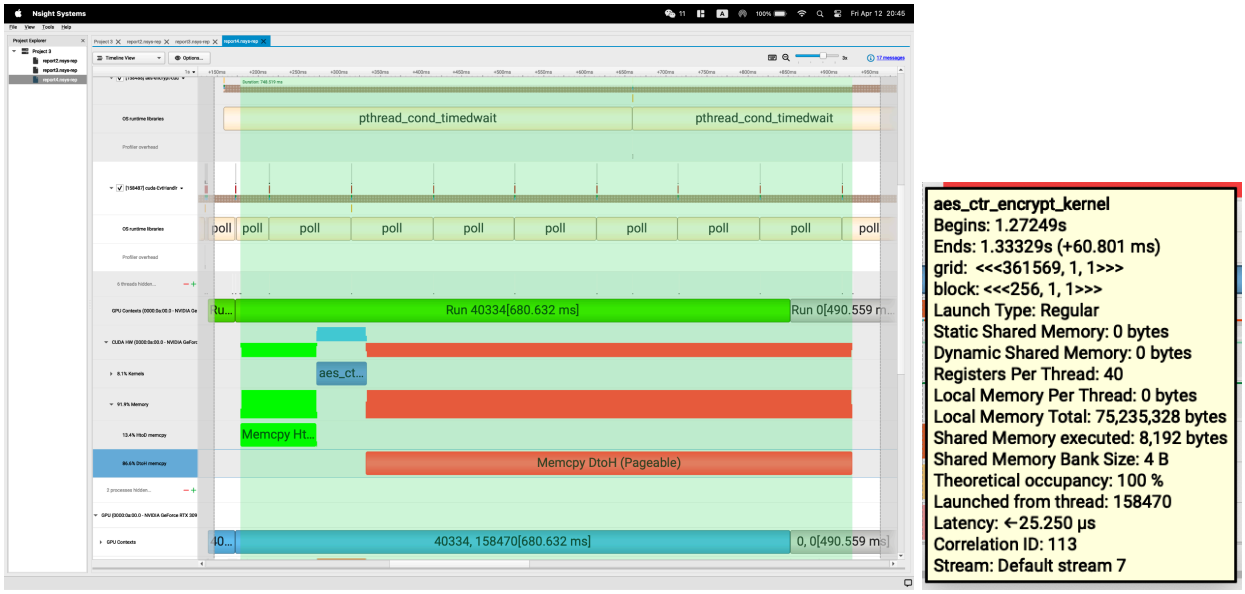



Figure 7. v0 video

Total run time: 748.519ms
Kernel run time: 60.801ms

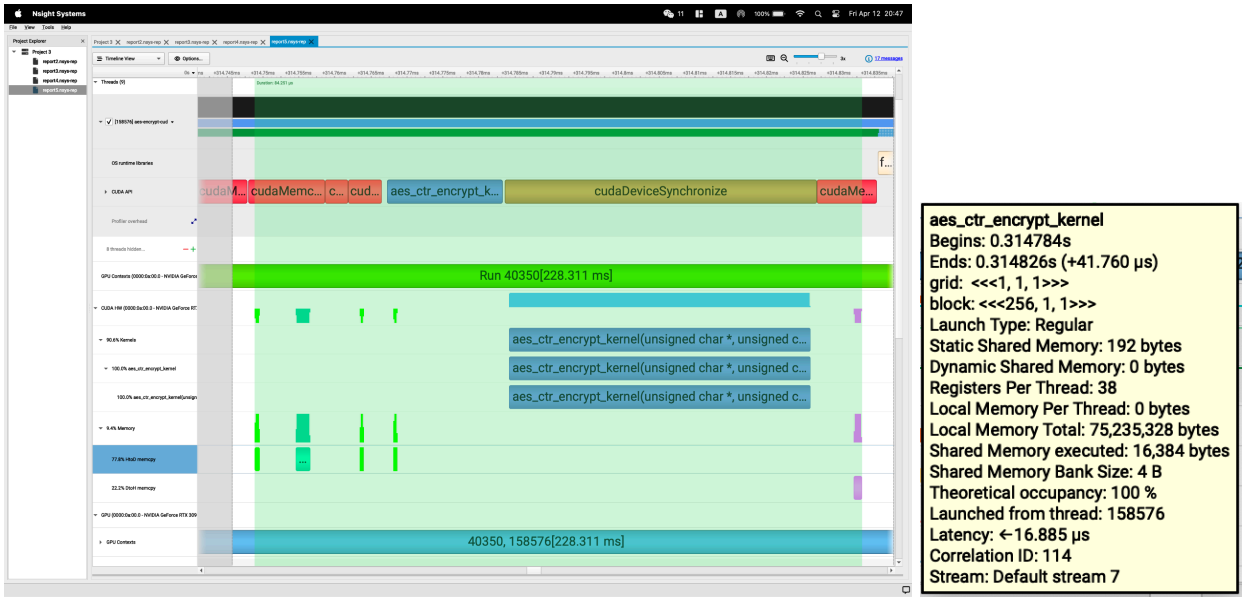



Figure 8. v1 small.txt

Total run time: 84.251us
Kernel run time: 41.760us

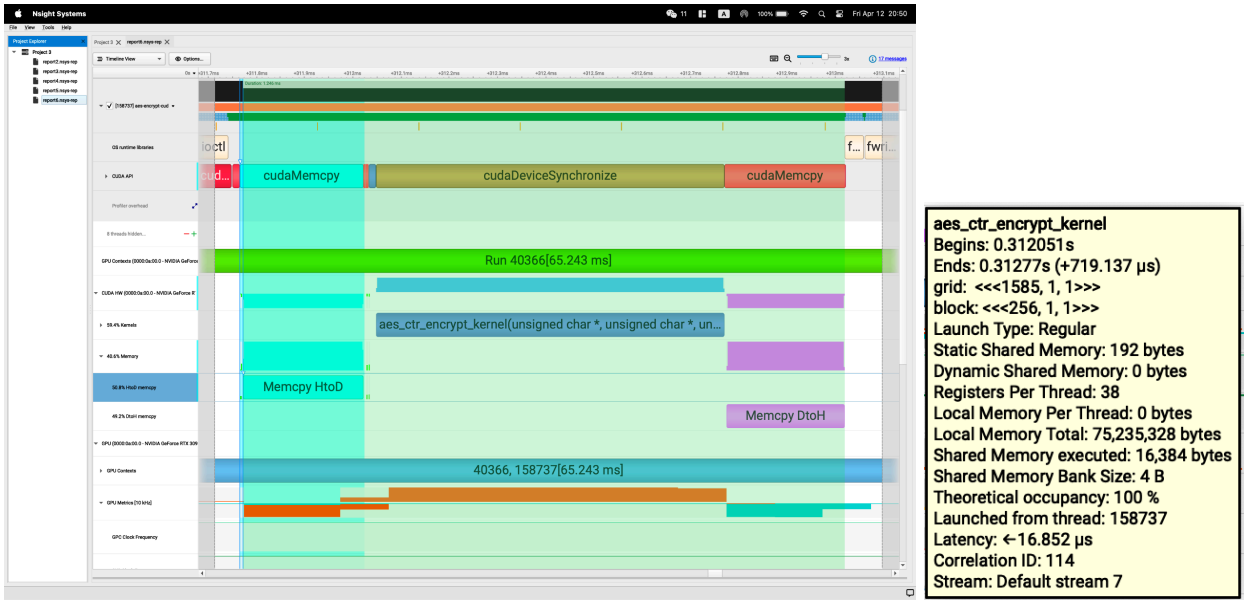



Figure 9. v1 big.txt

Total run time: 1.246ms
Kernel run time: 719.137us

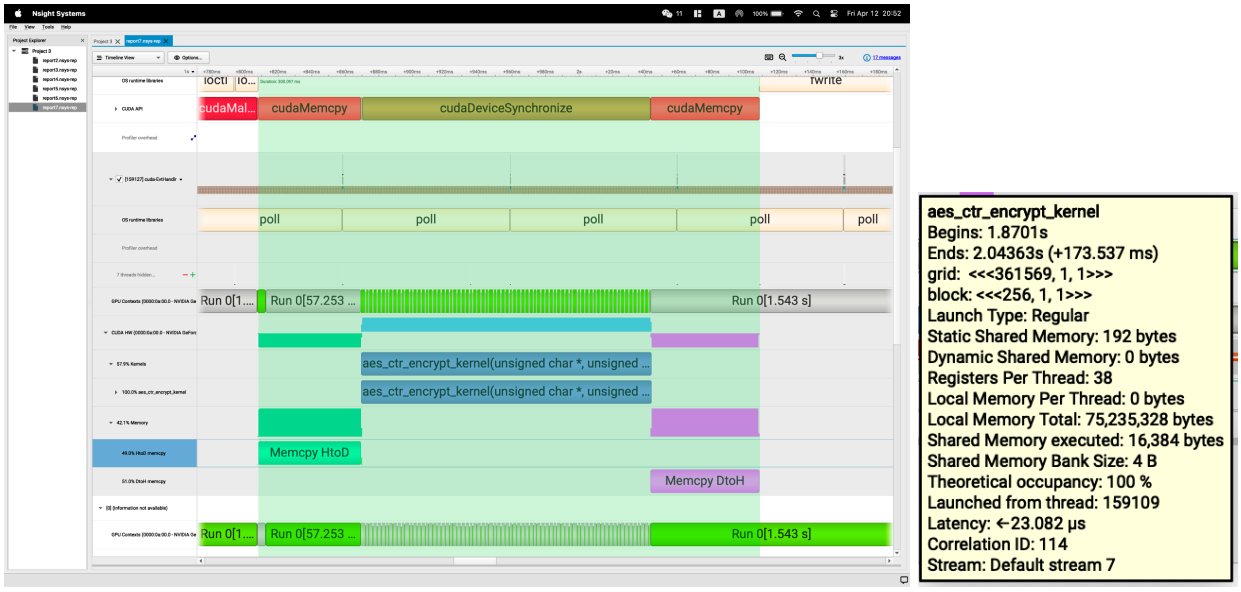



Figure 10. v1 video

Total run time: 300.097ms
Kernel run time: 173.537ms

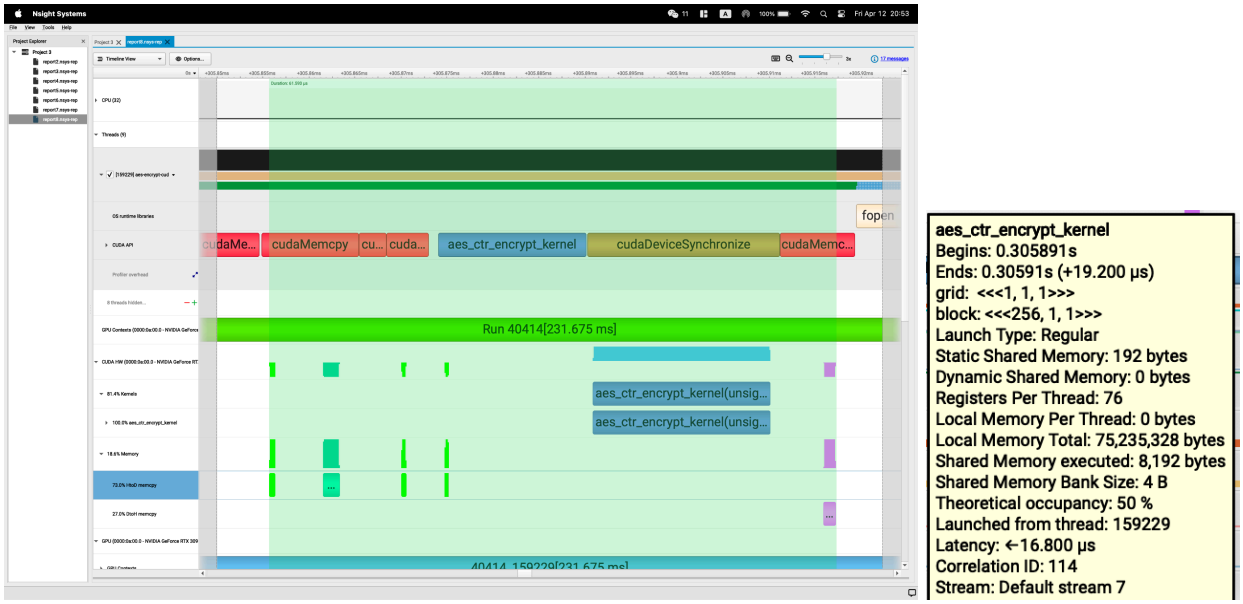



Figure 11. v2 small.txt

Total run time: 61.593us
Kernel run time: 19.200us

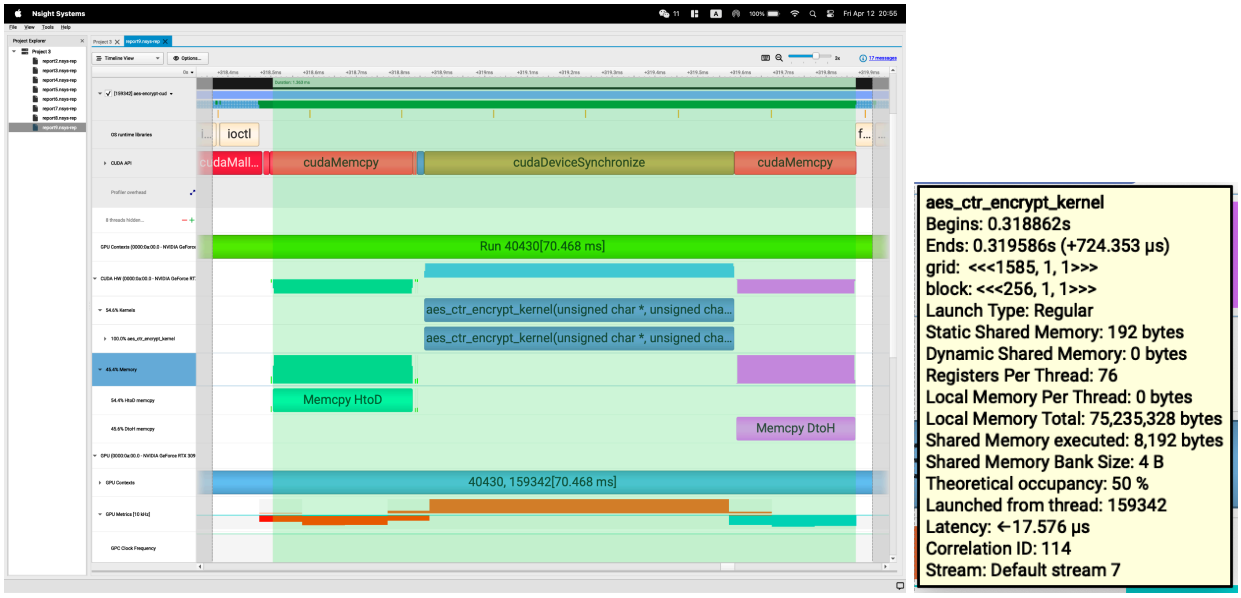



Figure 12. v2 big.txt

Total run time: 1.363ms
Kernel run time: 724.252us

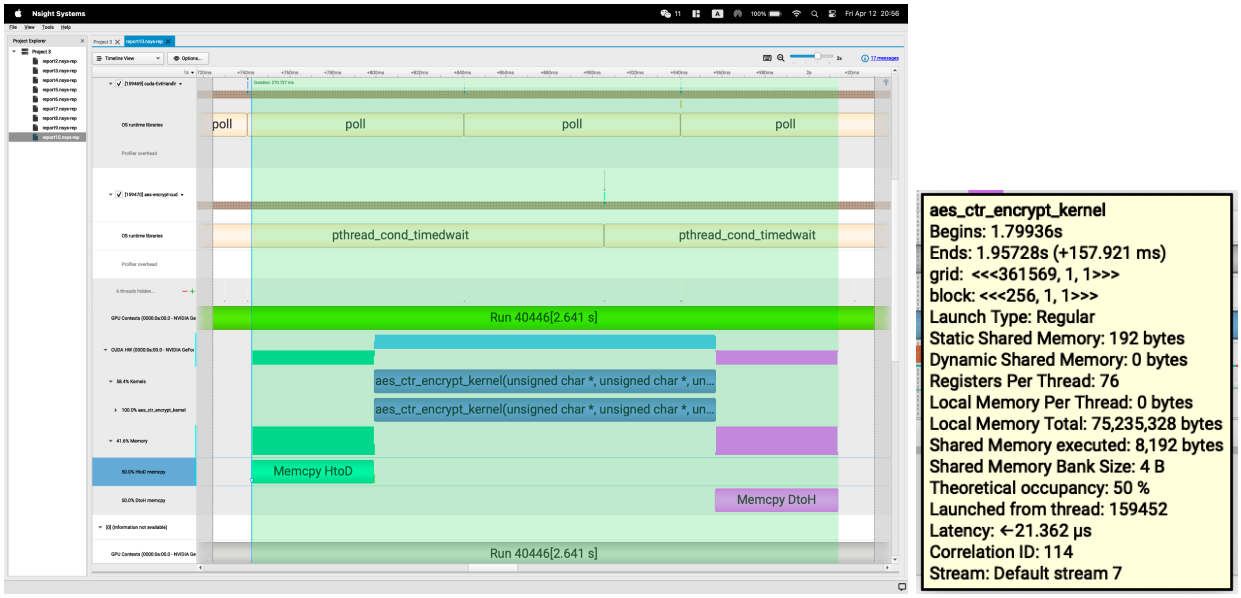



Figure 13. v2 video

Total run time: 270.727ms
Kernel run time: 157.921ms

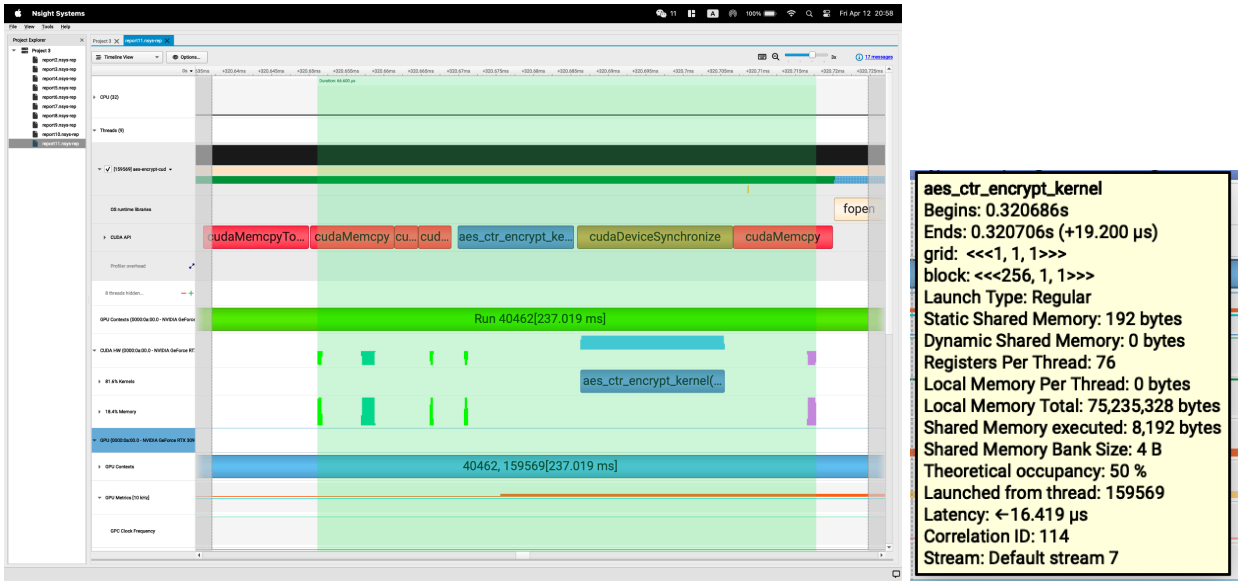



Figure 14. v3.1 small.txt

Total run time: 66.600us
Kernel run time: 19.200us

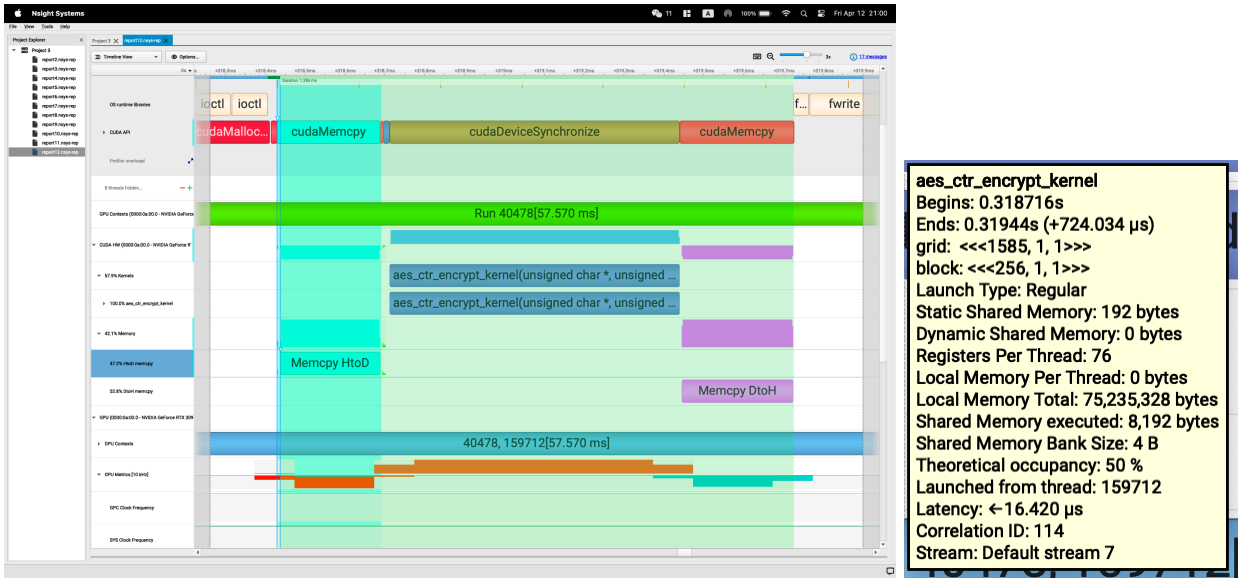



Figure 15. v3.1 big.txt

Total run time: 1.286ms
Kernel run time: 724.034us

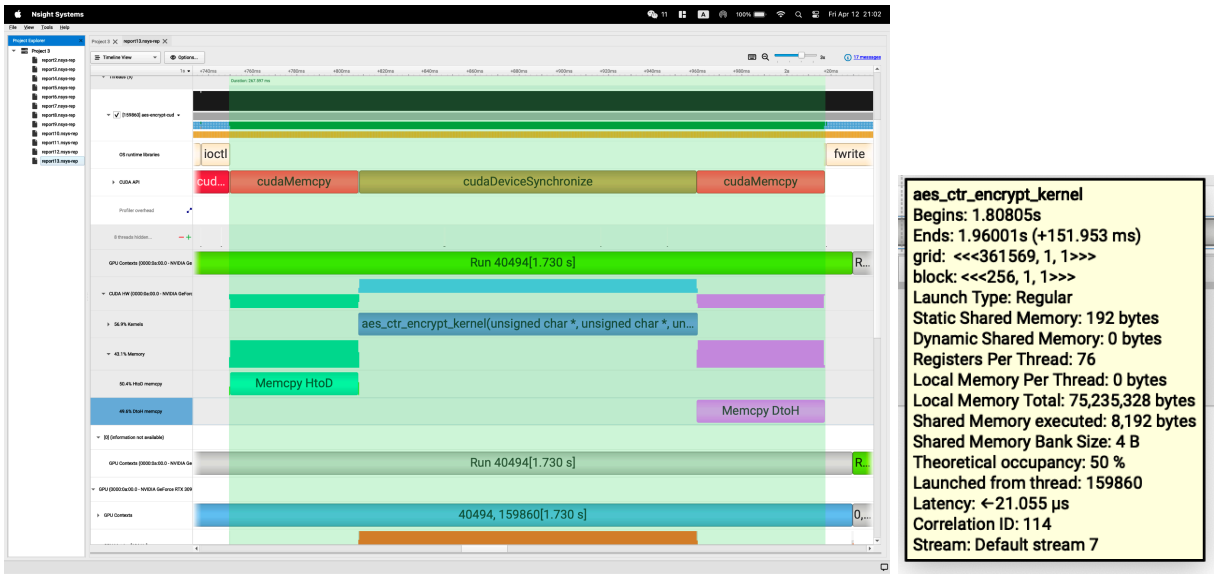



Figure 16. v3.1 video

Total run time: 267.597ms
Kernel run time: 151.953ms

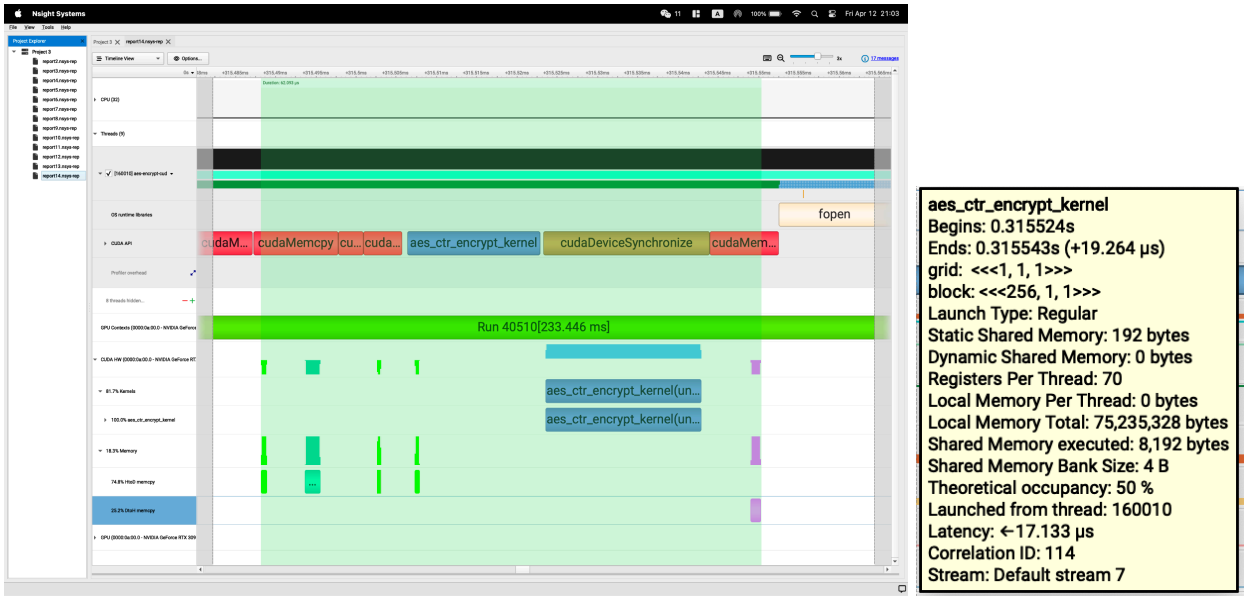



Figure 17. v3.2 small.txt

Total run time: 62.093us
Kernel run time: 19.264us

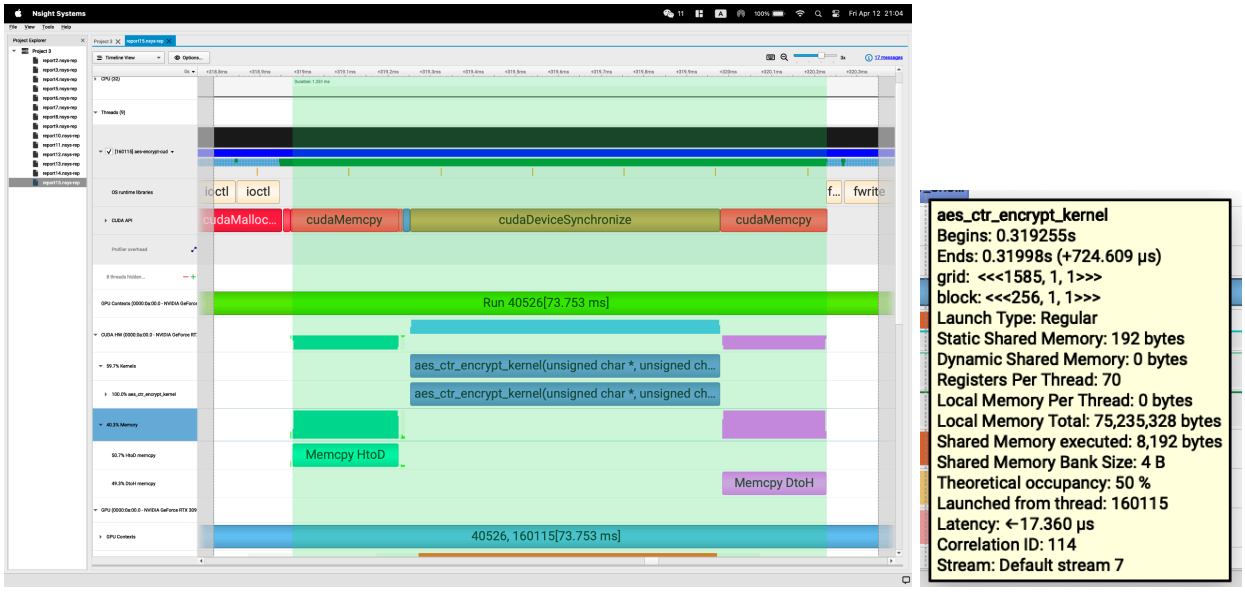



Figure 18. v3.2 big.txt

Total run time: 1.251ms
Kernel run time: 724.609us

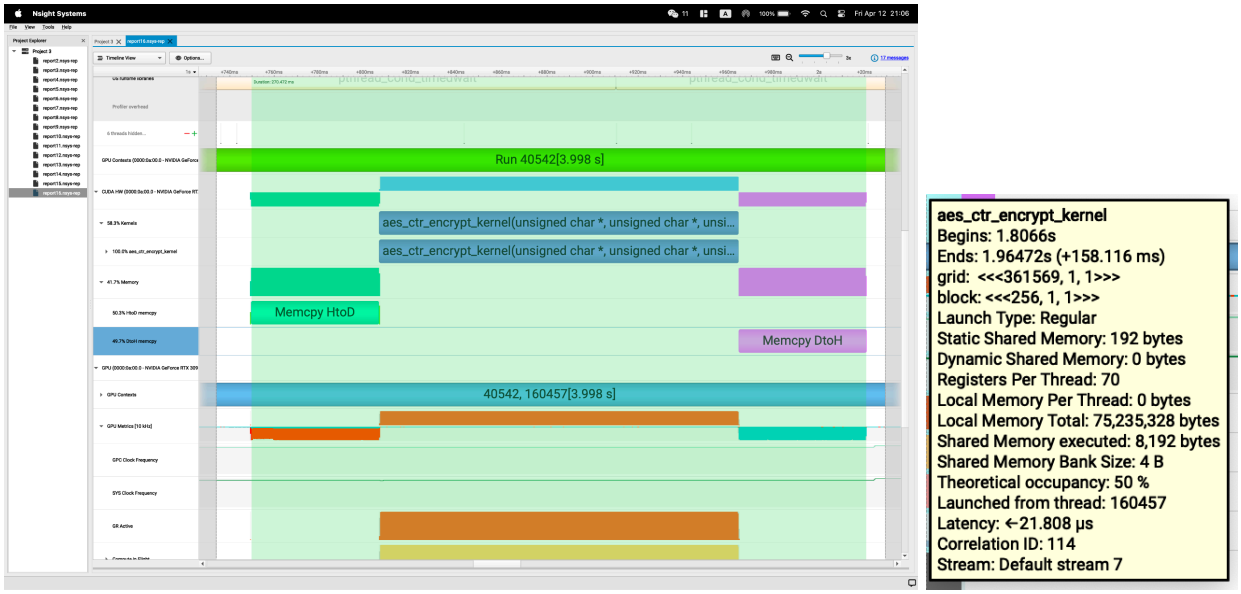



Figure 19. v3.2 video

Total run time: 270.472ms
Kernel run time: 158.116ms

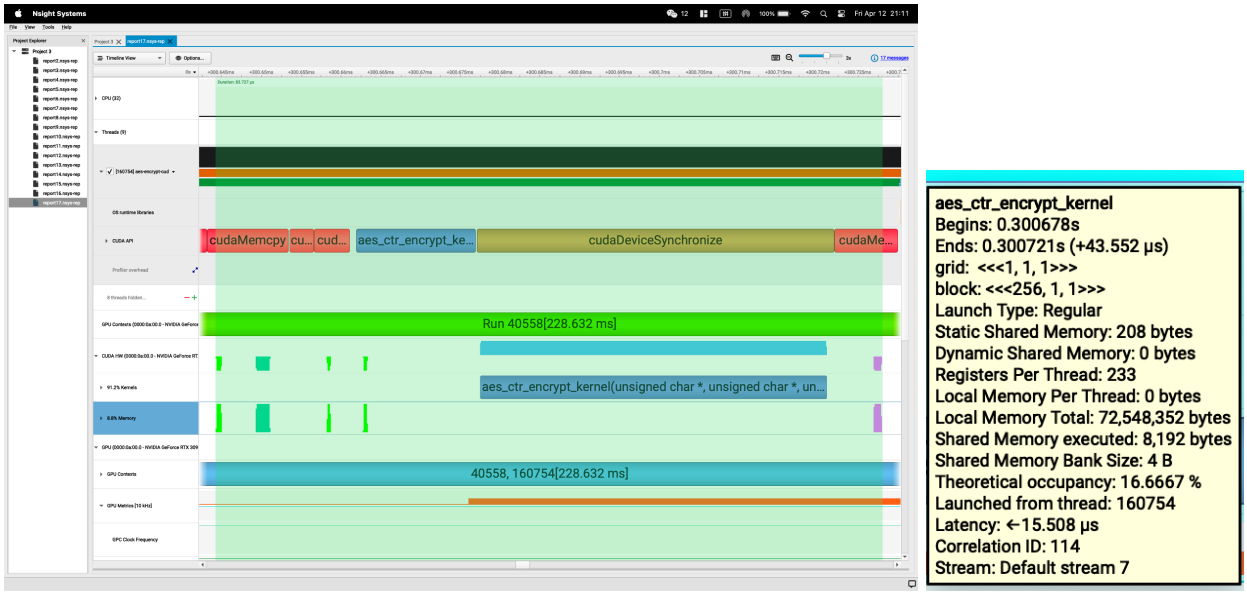



Figure 20. v4 small.txt

Total run time: 83.727us
Kernel run time: 43.552us

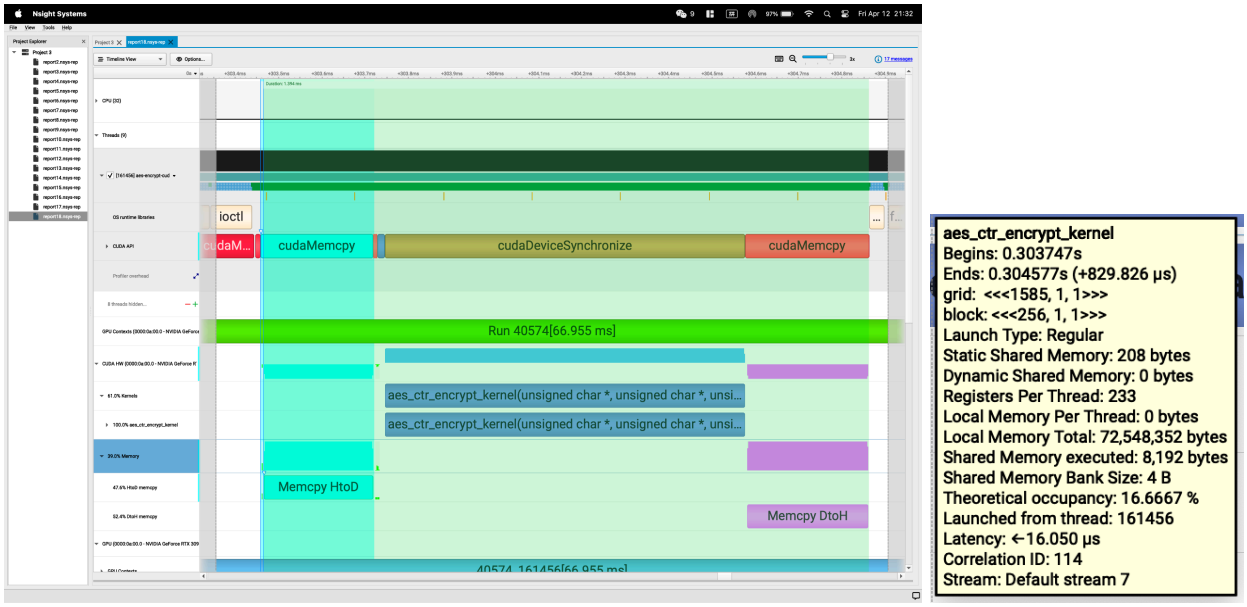



Figure 21. v4 big.txt

Total run time: 1.394ms
Kernel run time: 829.826us

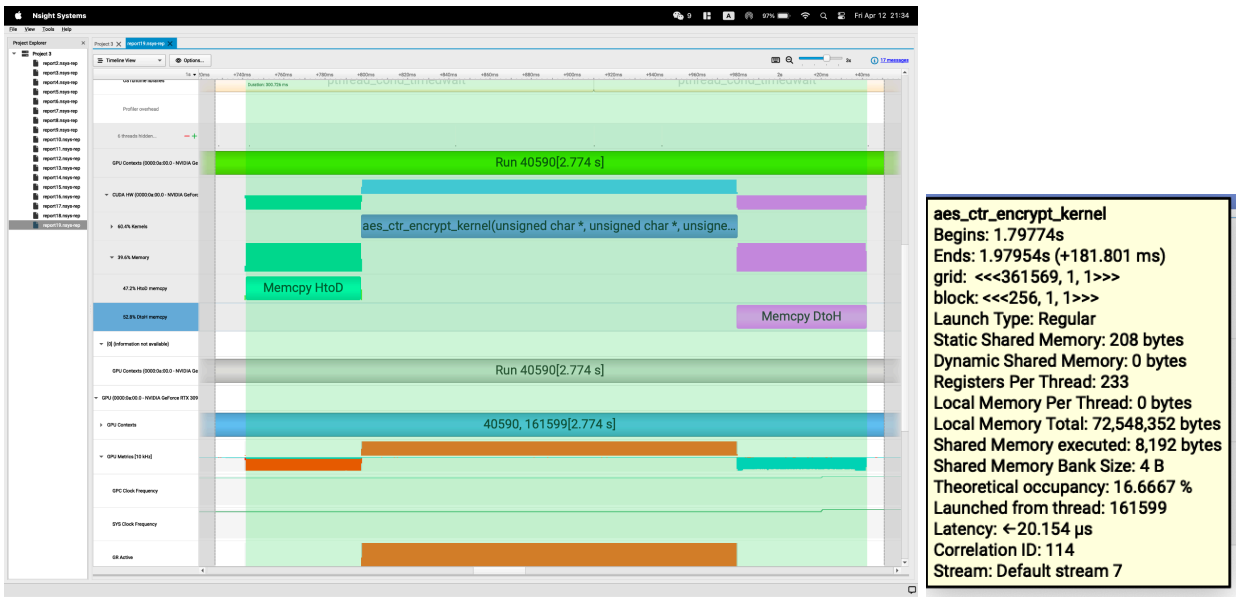



Figure 22. v4 video

Total run time: 300.726ms
Kernel run time: 181.801ms

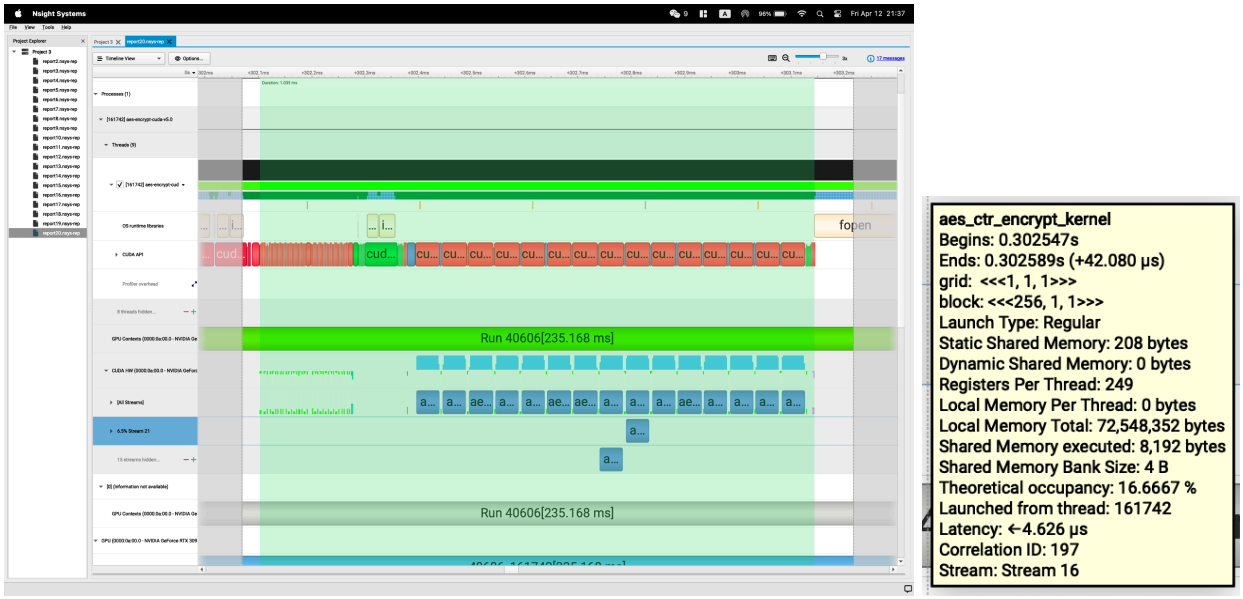



Figure 23. v5 small.txt

Total run time: 1.039ms

Kernel run time: 673.28us (42.080us/stream x 16 streams in serial)

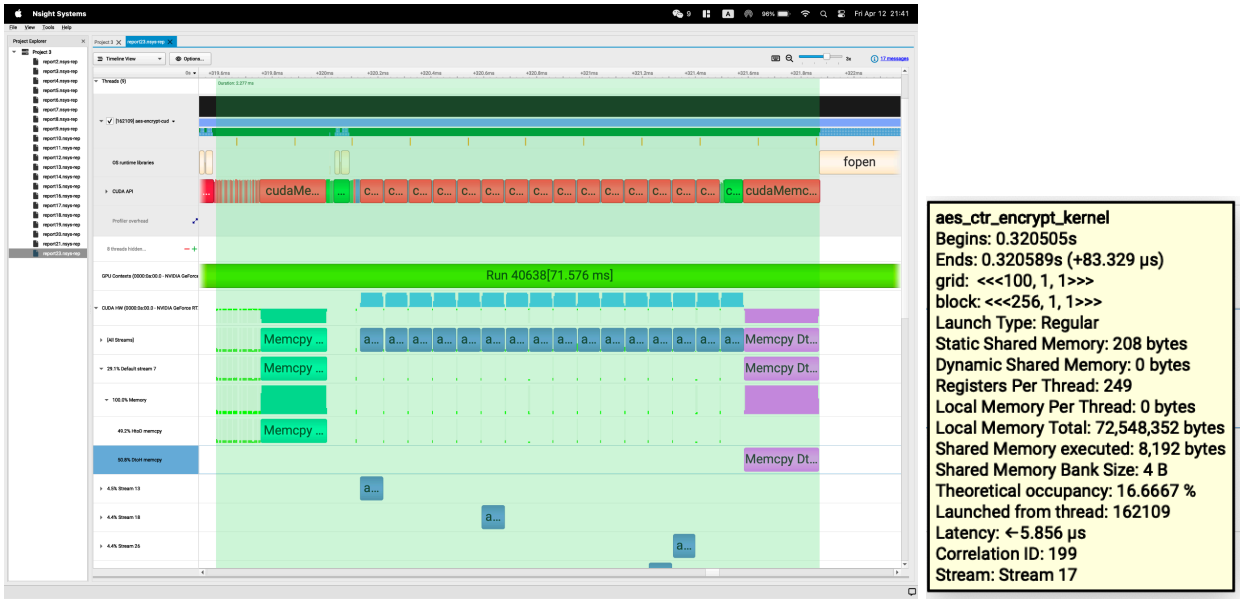



Figure 24. v5 big.txt

Total run time: 2.277ms

Kernel run time: 1.333ms (83.329us/stream x 16 streams in serial)

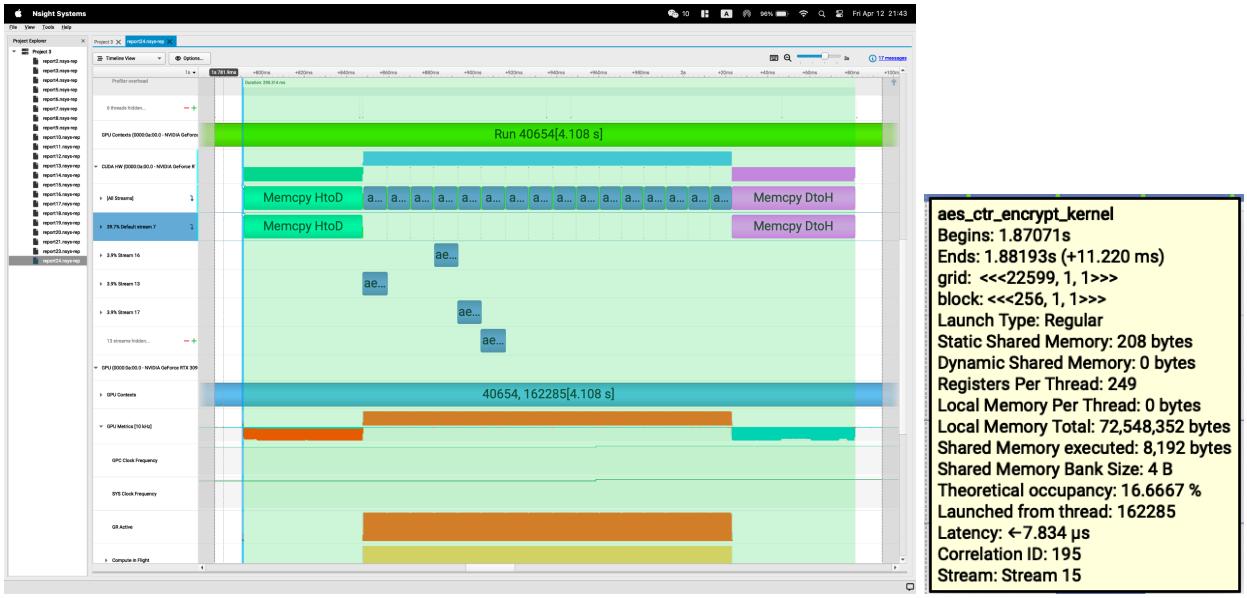
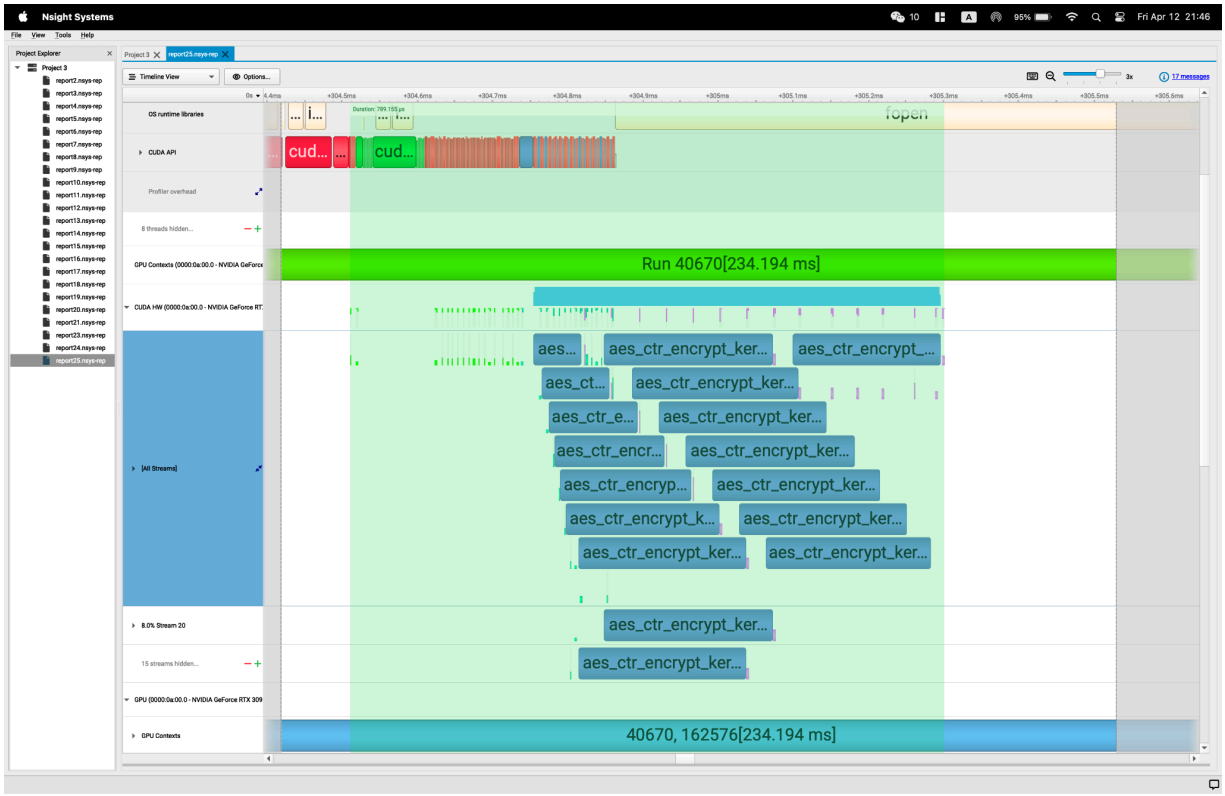



Figure 25. v5 video

Total run time: 290.314ms

Kernel run time: 179.520ms (11.220ms/stream x 16 streams in serial)

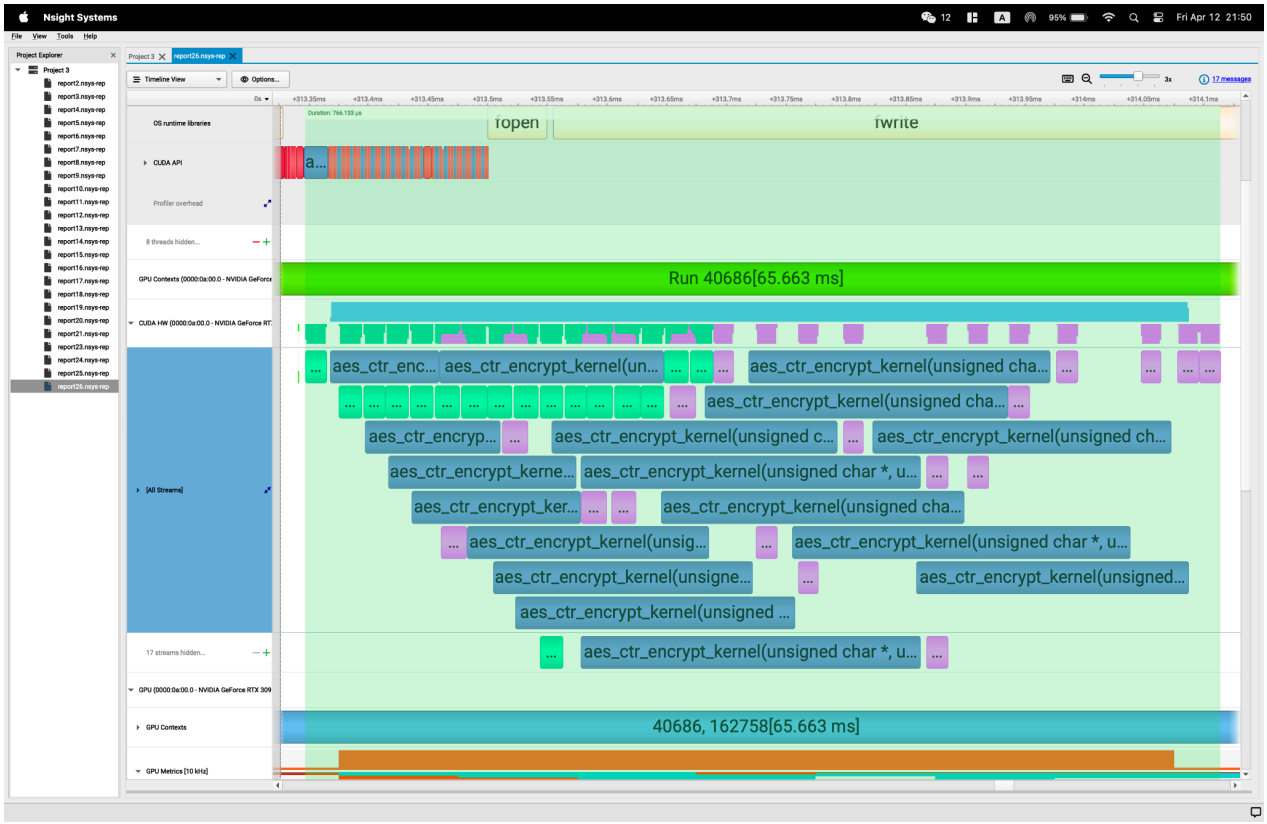



**aes_ctr_encrypt_kernel**
Begins: 0.304756s
Ends: 0.304819s (+62.784 µs)
grid: <<<82, 1, 1>>>
block: <<<256, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 208 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 40
Local Memory Per Thread: 0 bytes
Local Memory Total: 72,548,352 bytes
Shared Memory executed: 16,384 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 100 %
Launched from thread: 162576
Latency: ←17.444 µs
Correlation ID: 190
Stream: Stream 13

**aes_ctr_encrypt_kernel**
Begins: 0.304815s
Ends: 0.305038s (+222.305 µs)
grid: <<<82, 1, 1>>>
block: <<<256, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 208 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 40
Local Memory Per Thread: 0 bytes
Local Memory Total: 72,548,352 bytes
Shared Memory executed: 16,384 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 100 %
Launched from thread: 162576
Latency: ←12.301 µs
Correlation ID: 208
Stream: Stream 19

Figure 26. v5.1 small.txt

Total run time: 789.155us
Kernel run time: min 62.784us, max 222.305us




**aes_ctr_encrypt_kernel**
Begins: 0.313399s
Ends: 0.31351s (+111.488 µs)
grid: <<<100, 1, 1>>>
block: <<<256, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 208 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 40
Local Memory Per Thread: 0 bytes
Local Memory Total: 72,548,352 bytes
Shared Memory executed: 16,384 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 100 %
Launched from thread: 162758
Latency: ←23.633 µs
Correlation ID: 193
Stream: Stream 14

**aes_ctr_encrypt_kernel**
Begins: 0.313579s
Ends: 0.313862s (+283.776 µs)
grid: <<<100, 1, 1>>>
block: <<<256, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 208 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 40
Local Memory Per Thread: 0 bytes
Local Memory Total: 72,548,352 bytes
Shared Memory executed: 16,384 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 100 %
Launched from thread: 162758
Latency: ←90.734 µs
Correlation ID: 232
Stream: Stream 27

Figure 27. v5.1 big.txt

Total run time: 766.133us
Kernel run time: min 111.488us, max 283.776us

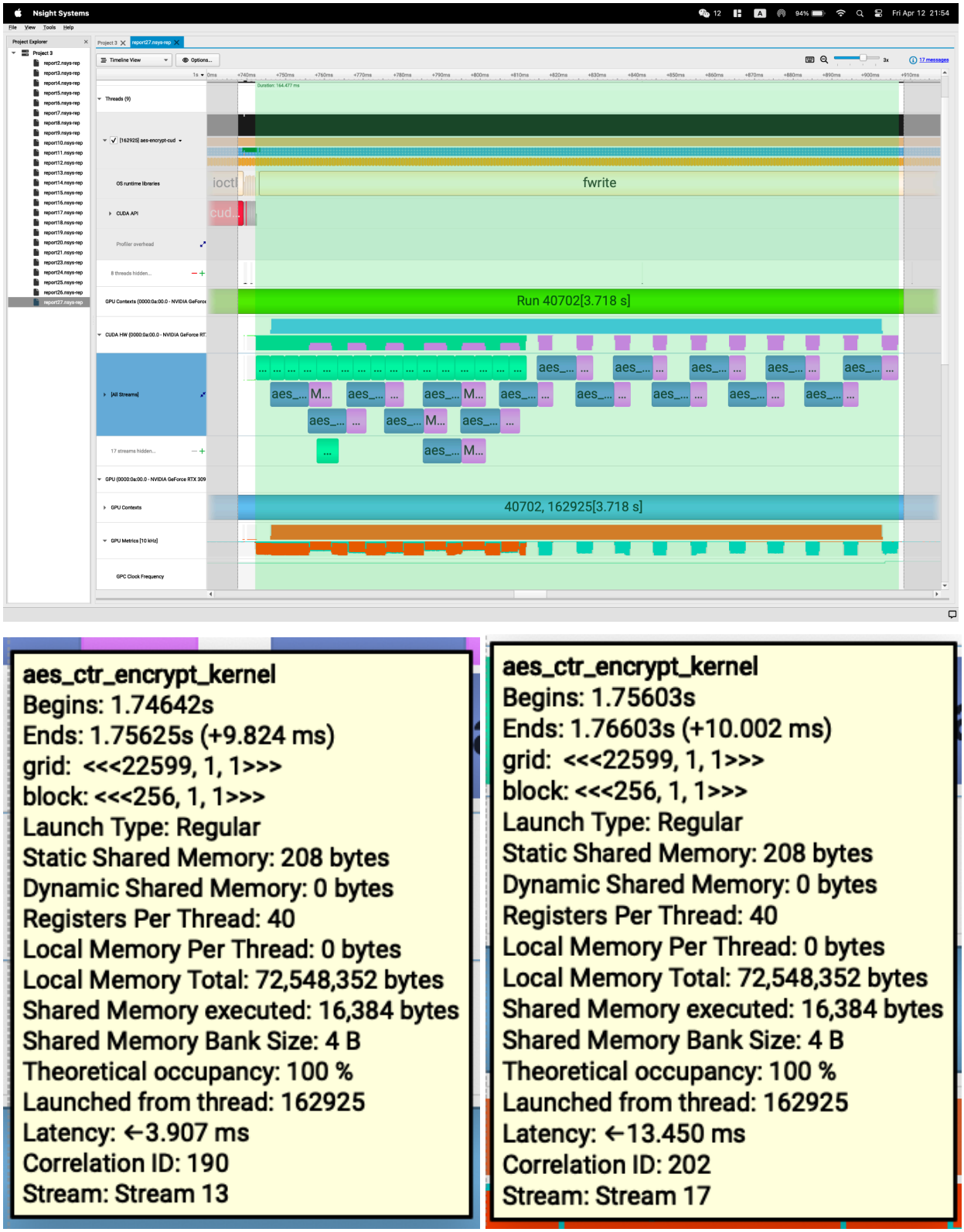



Figure 28. v5.1 video

Total run time: 164.477ms

Kernel run time: min 9.824ms, max 10.002ms