GPU and Hardware-Accelerated AES Encryption

Dominic DiPuma

I. Introduction

In recent years, encryption has become ubiquitous. Smartphones and personal computers are often fully encrypted by default, and cryptography is being applied in new ways, such as for cryptocurrencies. As a result, GPU, FPGA, and ASIC-accelerated cryptographic implementations are in focus. On modern x86 and ARM processors, there are instructions to accelerate the advanced encryption standard (AES) algorithm. This paper will describe the AES algorithm in enough detail to implement it, and then study three implementations of the algorithm: a naive software implementation written in C; a C implementation using compiler intrinsics for the Intel AES instructions; and an Open Computer Library (OpenCL) implementation targeted at GPUs. Each of the implementations will be profiled for their performance.

II. Technical Background

AES is an encryption standard that was approved and published by the US government in the early 2000's. Since it a government-backed standard, it has been thoroughly researched and documented, and hardware implementations are even available in consumer electronics. As such, it makes for an interesting case study in hardware acceleration.

From an implementation perspective, the AES algorithm is relatively simple. At a high level, the algorithm has only a few elements:

- A cryptographic key (either 128 bits, 192 bits, or 256 bits) is expanded several times (varies by key length), so that a different key is used at each "round" of encryption. This expansion process is standardized, and is essentially a matrix multiplication followed by a byte substitution and then an XOR with a constant.
- The plaintext to be encrypted is broken up into 128-bit blocks, and each block is processed through several rounds. Each round consists of a byte substitution, a shift operation, a matrix multiplication, and an XOR with that round's key.

How the 128-bit blocks are handled can vary depending on the specific cipher mode being used. Popular modes for AES include electronic codebook (ECB), cipher block chaining (CBC), and counter (CTR)[1]. ECB is the simplest of the group, and encrypts each block of plaintext on its own. While ECB is the easiest mode to implement, the ciphertext produced can reveal repetition in the plaintext, so it is insecure. CBC XORs the current block's plaintext with the previous block's ciphertext in order to remedy the security flaws of ECB. Although this method is far more secure than ECB, it cannot be parallelized, since each block depends on its immediate predecessor. This makes CBC a bad candidate for GPU acceleration. The CTR mode XORs each block with a deterministic number. The number can be produced by a simple counter or a pseudorandom sequence. CTR can be parallelized, provided that the counter is parallel-safe.

This paper will implement the ECB mode, since it is the easiest, and it can be verified against the official AES specification published by the National Institute of Standards and Technologies (NIST). The specification includes a thorough example of the encryption of a single block (which is effectively ECB mode).

Additionally, the AES specification supports three different key lengths: 128-bit, 192-bit, and 256-bit. Each key length uses a different number of encryption rounds per the specification (10, 12, and 14 rounds, respectively). For this project, only the 128-bit version will be implemented.

A. *AES Encryption Algorithm*

Given a 128-bit block and the fully expanded key schedule (key expansion is described below), the AES algorithm consists of the following[2]:

Algorithm 1 AES-128 Encryption

```
1: function AesEncrypt128($input$, $output$, $key$)
2:     $state \leftarrow$ AddRoundKey($input$, $key$, 0)
3:     $round \leftarrow 1$
4:     while $round \leq 9$ do
5:         $state \leftarrow$ SubBytes($state$)
6:         $state \leftarrow$ ShiftRows($state$)
7:         $state \leftarrow$ MixColumns($state$)
8:         $state \leftarrow$ AddRoundKey($state$, $key$, $round$)
9:         $round \leftarrow round + 1$
10:    end while
11:    $state \leftarrow$ SubBytes($state$)
12:    $state \leftarrow$ ShiftRows($state$)
13:    $state \leftarrow$ AddRoundKey($state$, $key$, 10)
14:    return $state$
15: end function
```

The initial block can be viewed as a 4-by-4 matrix, where A is a 16-byte array as in 1.

$$\begin{bmatrix} A[0] & A[4] & A[8] & A[12] \\ A[1] & A[5] & A[9] & A[13] \\ A[2] & A[6] & A[10] & A[14] \\ A[3] & A[7] & A[11] & A[15] \end{bmatrix} \tag{1}$$

Moreover, treating the current round key as a 16-byte array named K, the key is also a 4x4 matrix as in 2.

$$\begin{bmatrix} K[0] & K[4] & K[8] & K[12] \\ K[1] & K[5] & K[9] & K[13] \\ K[2] & K[6] & K[10] & K[14] \\ K[3] & K[7] & K[11] & K[15] \end{bmatrix} \tag{2}$$

1) AddRoundKey: The AddRoundKey step is trivial. The round key from the expanded key schedule is XOR'd with the block to form an updated block as shown in 3.

2) SubBytes: The SubBytes sub-function replaces every byte of the block with a unique byte from a standardized lookup table, denoted by the byte array s in 4. In the literature, this table is referred to as the "s-box".

3) ShiftRows: ShiftRows rotates each row of the block cyclically. The "top" row is left alone, the next row is shifted by 1 byte, then 2 bytes, and the "bottom" row is shifted by 3 bytes as shown in 5.

4) MixColumns: MixColumns involves a matrix multiplication of the block with a standardized matrix as shown in 6.

This is not a "typical" matrix multiplication on 8-bit integers. Instead, this is performed in in the Galois Field GF(2^8) modulo 0x1b. As far as computation goes, this means that multiplications are performed without carries, addition is replaced with XOR, and there is a modulus[3].

Computing the new A is shown in 7.

$$A[0] = 2 \cdot A[0] \oplus 3 \cdot A[1] \oplus A[2] \oplus A[3] \tag{7}$$
$$A[1] = A[0] \oplus 2 \cdot A[1] \oplus 3 \cdot A[2] \oplus A[3] \tag{8}$$
$$\vdots$$

Note that $\cdot$ is used to represent carryless multiplication with a modulus of 0x1b. A similar expression can be found for each of the 16 bytes of A.

5) Key Expansion: Key expansion produces unique round keys for each round of encryption from the original 128-bit key. These 11 round keys form the key schedule which is used throughout the entire encryption process (and therefore only needs to be calculated one time per key).

The first round key in the schedule is the original key. From there, each round key is produced by manipulating the previous round key. For the purposes of key expansion, the key is considered as an array of 4 32-bit words.

The first word in each round key is produced by taking the last word of the previous round key and then bit-manipulating it. This word is rotated cyclically by one byte, and each of its bytes are substituted using the s-box from SubBytes. Then, the word is XOR'd with a "round constant". (Round constants are unique for each round of encryption. The lower 3 bytes of each round constant are 0s, but the upper bytes are 0, then powers of 2 modulo 0x1b. That is, the round constants are 0x00000000, 0x01000000, 0x02000000, ..., 0x1b000000, 0x36000000.) Finally, this word is XOR'd with the first word of the previous round key,

The second, third, and fourth words of each round key are much simpler to calculate. These are produced by XOR'ing the previous word with the previous round key's Nth word. For example, the second round key's second word is the second round key's first word XOR the first round key's second word; the second round key's third word is the second round key's second word XOR the first round key's third word; and so on.

B. AES Decryption

AES decryption is closely related to encryption since it is just inverting every operation of encryption in reverse order. Several changes are made to invert the operations of encryption:

- The decryption key schedule can be obtained by performing MixColumns on the encryption key schedule.
- The s-box of SubBytes is replaced by an inverse s-box.
- The direction of the shift in ShiftRows is reversed.
- The coefficient matrix used in MixColumns is changed.

These changes do not meaningfully affect the computational complexity of the algorithm, so implementing decryption would not provide meaningful results that cannot be obtained from the encryption implementations.

C. OpenCL

OpenCL is a standard[4] which is designed to enable parallel computing across GPUs, CPUs, and FPGAs. Often, it is used to perform general purpose GPU computation.

An OpenCL program is made of a main executable and a "kernel".

The main executable is written in standard C or C++. This program's purpose is essentially to set up and launch the OpenCL kernel. It makes API calls that query for OpenCL-compatible devices, creates an OpenCL context and a command queue on that device (or devices), compiles and loads a kernel onto the device(s), performs memory transfers inputs to the GPU, enqueues kernel commands, and transfers outputs from the GPU. OpenCL will automatically determine how to distribute the enqueued kernel command across the available compute resources, and executes the kernel in parallel across many inputs.

The kernel is a function written in OpenCL C and designed to execute in parallel. In OpenCL C, a function can be made into a kernel by using the "__kernel" qualifier before the function prototype. Generally, the inputs and outputs to a kernel are arrays which are accessible to many kernel calls at once. As a result, each kernel will generally work on a subset of array indices, which can be determined using the "get_global_id" function. For example, the 0th kernel ID may operate on array index 0, or on array indices 0 through 9; and the 1st kernel ID may operate on array index 1, or on array indices 10 through 19. The decision about how to divide the work across kernels is left to the programmer.

Within kernels, OpenCL has more complex memory management than standard C does, and often requires variables to have explicitly declared address spaces. There is "__global" memory, which is accessible from the entire GPU, but is the most costly to access. Alongside global memory is "__constant" memory, which has the added restriction of being

$$\begin{bmatrix} A[0] & A[4] & A[8] & A[12] \\ A[1] & A[5] & A[9] & A[13] \\ A[2] & A[6] & A[10] & A[14] \\ A[3] & A[7] & A[11] & A[15] \end{bmatrix} = \begin{bmatrix} A[0] & A[4] & A[8] & A[12] \\ A[1] & A[5] & A[9] & A[13] \\ A[2] & A[6] & A[10] & A[14] \\ A[3] & A[7] & A[11] & A[15] \end{bmatrix} \oplus \begin{bmatrix} K[0] & K[4] & K[8] & K[12] \\ K[1] & K[5] & K[9] & K[13] \\ K[2] & K[6] & K[10] & K[14] \\ K[3] & K[7] & K[11] & K[15] \end{bmatrix} \tag{3}$$

$$\begin{bmatrix} A[0] & A[4] & A[8] & A[12] \\ A[1] & A[5] & A[9] & A[13] \\ A[2] & A[6] & A[10] & A[14] \\ A[3] & A[7] & A[11] & A[15] \end{bmatrix} = \begin{bmatrix} s[A[0]] & s[A[4]] & s[A[8]] & s[A[12]] \\ s[A[1]] & s[A[5]] & s[A[9]] & s[A[13]] \\ s[A[2]] & s[A[6]] & s[A[10]] & s[A[14]] \\ s[A[3]] & s[A[7]] & s[A[11]] & s[A[15]] \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} A[0] & A[4] & A[8] & A[12] \\ A[1] & A[5] & A[9] & A[13] \\ A[2] & A[6] & A[10] & A[14] \\ A[3] & A[7] & A[11] & A[15] \end{bmatrix} = \begin{bmatrix} A[0] & A[4] & A[8] & A[12] \\ A[5] & A[9] & A[13] & A[1] \\ A[10] & A[14] & A[2] & A[6] \\ A[15] & A[3] & A[7] & A[11] \end{bmatrix} \tag{5}$$

$$\begin{bmatrix} A[0] & A[4] & A[8] & A[12] \\ A[1] & A[5] & A[9] & A[13] \\ A[2] & A[6] & A[10] & A[14] \\ A[3] & A[7] & A[11] & A[15] \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} A[0] & A[4] & A[8] & A[12] \\ A[1] & A[5] & A[9] & A[13] \\ A[2] & A[6] & A[10] & A[14] \\ A[3] & A[7] & A[11] & A[15] \end{bmatrix} \tag{6}$$

read-only. “__local” variables can be accessed from anything on the same execution unit, and this form of memory is faster and less abundant than global memory. The fastest but most sparse is “__private” memory, which can only be accessed within a single function call. “__private” is the default for variables which are not explicitly labeled.

D. Intel AES-NI

Intel’s instruction set extensions for AES include 6 instruction[1]. They perform the following tasks:

- A single round of encryption
- The final round of encryption
- A single round of decryption
- The final round of decryption
- A single round of key expansion
- Invert the mix columns step (used to convert the encryption key schedule to the decryption key schedule)

Using these instructions, encrypting one block only requires an XOR, several calls to the encryption round instruction, and a final encryption round call. Similarly, decrypting a block requires an XOR, several calls to the decryption round instruction, and a final decryption round instruction call.

Although requiring multiple calls to the round encryption functions sounds sub-optimal, it is important to note that each AES key length uses a different number of encryption rounds. Not only would a one-instruction encryption command be difficult to implement in hardware, but it would make the instruction set more complicated. Additionally, the instruction-per-round approach makes this instruction set flexible enough that it could be used on similar cryptographic algorithms.

Key expansion is a slightly more complicated operation. The key expansion instruction that Intel provides only does part of the work. Some byte-shifting and shuffling needs to be done separately, and the specific operations that are needed are described in Intel’s whitepaper on AES-NI. The inverse mix columns instruction allows for a key schedule to be converted into a decryption key schedule easily.

Since key expansion is only performed once per key, it is not necessary to optimize this operation as heavily as encryption and decryption rounds are optimized. This is part of why manual byte-shifting and shuffling is considered acceptable. The manual byte-manipulation steps are also useful because they enable the key expansion instruction to work with 128, 192, and 256-bit keys (as well as larger key lengths if AES grows in the future).

III. Setup and Implementation

A. System Setup

This was developed for a ThinkPad T450s laptop with an Intel Core i5 (Broadwell) 5200U dual core CPU. This CPU includes an integrated Intel HD 5000 GPU and supports AES-NI instruction set extensions, as well as SSE 4.2 and AVX 2 vector instruction set extensions. Hyperthreading is enabled to allow 4 simultaneous processor threads, and 128MB of RAM is dedicated to the integrated GPU in BIOS.

In terms of system software, the system is running Arch Linux which is up to date as of November 2020. This includes Linux kernel 5.9.9, GNU Compiler Collection 10.2.0, and glibc 2.32.

The un-accelerated implementation should compile and run on any modern system with a C compiler and support for POSIX threads (pthreads). The CPU-accelerated implementation requires Intel compiler intrinsics and hardware support for the AES-NI extension set, which is available on many modern Intel and AMD CPUs. Finally, the GPU-accelerated implementation requires OpenCL 2.2 support in software and hardware. However, only the configuration discussed above has been tested.

B. Implementation

1) Software-Only Implementation: The software-only implementation is largely a translation of the specification's pseudocode into C. Each major step in the AES algorithm is written as its own function, with an AesCipher128 function to encrypt an entire block. Some meaningful optimizations are included in this implementation.

The Galois field multiplication is replaced by two 256-byte lookup tables (one for multiplication by 2 and another for multiplication by 3). This replaces one of the most computationally complex operations in the algorithm with a simple lookup.

Additionally, the ShiftRows step is performed using the "shuffle" GCC builtin. This is faster than the naive approach of using temporary variables and then re-assigning bytes in the block as it takes advantage of an x86 shuffle SIMD instruction. SIMD instructions are also used in AddRoundKey, which performs its XOR operation on the entire block using one 128-bit variable.

After disassembling an earlier iteration of the compiled code, it was found that most loops in the AES encryption function were not unrolled (even when using GCC's "-funroll-all-loops" option). Loops were manually unrolled to achieve better performance.

Since the CPU implementation handles the blocks keys in a number of different ways (as a single 128-bit integer, as an array of bytes, and as a GCC vector type), these types are defined C unions. MixColumns can access the individual bytes of each block, ShiftRows can access the block as a vector, and the block behaves as a 128-bit integer in AddRoundKey.

2) OpenCL Implementation: The OpenCL version of the algorithm was derived from the software-only version. The implementation of AES is almost identical, with a few notable exceptions.

The OpenCL code is essentially the CPU header file, with AesCipher128 implemented as a kernel, and its inputs declared as __global. Additionally, any instances of the C "const" qualifier are replaced by OpenCL's equivalent "__constant" flag. There are some minor variable type changes to support OpenCL's types, as OpenCL has vector types in the standard but excludes types such as "uint8_t" in favor of types such as "uchar". However, the optimizations described for the CPU implementation still apply to the OpenCL version.

One additional optimization made to the OpenCL code is in the main executable. Instead of compiling the OpenCL immediately before executing it, the compilation is done in a separate executable, and the binary is saved. The AES encryption program loads this binary from the hard disk to reduce overhead.

3) AES-NI Accelerated Implementation: The instructions greatly simplify the code needed to implement AES. Aside from using the built-ins and implementing key expansion as described by Intel, the only optimization is manual loop unrolling.

IV. Results

After executing all of the implementations several times on a range of file sizes, the average runtime for each implementation/size combination was plotted, as in IV. Note that software-only and hardware-accelerated implementations were run using 4 simultaneous CPU threads.

In general, all implementations have linear runtime growth. The algorithm for encrypting a single block is not influenced at all by the number of blocks, the data in the blocks, or the key. Therefore, the algorithm's execution time should grow linearly with the number of blocks.

At small file sizes, the execution time does not change significantly as the file size increases. This is because there are overhead costs to running these benchmarks. All of these programs open/mmap two files (and later munmap/close them) and they derive the AES key schedule at runtime. The pure C and AES-NI accelerated C both spawn pthreads, and the OpenCL program makes calls to gain access to the GPU and loads the AES algorithm onto it. These operations should take the same amount of time to execute regardless of the file size, and with short execution times, the constant overhead dominates the expected linear runtime growth.

With this hardware/software setup, AES-NI is clearly the best performing at large file sizes, and the GPU-accelerated code is slightly faster than the software-only version. This is in spite of the relatively weak integrated GPU used for this analysis. Using a dedicated GPU would allow the OpenCL software to outperform the software-only implementation by a larger margin.

The runtime decrease from 1KB to 2KB is unexpected, and occurred consistently across executions and implementations. This may be a quirk of handling 1KB files on the system being studied.

V. Code Description

Full source code is available on GitHub at https://github.com/DDiPuma/Accelerated-AES-Benchmark.

A. Test Driver Software

Each implementation was tested against known-accurate input-output combinations.

The AES-NI implementation was tested using a full block encryption using the NIST specification test input. The AES-NI encryption code was modified slightly to become a decryption program. Any encrypted file should decrypt back to its original plaintext.

The software-only implementation was spot-checked by evaluating the output of each function against the NIST-provided test inputs for the first encryption round. Then, a full block encryption was performed to verify that the encryption works end-to-end on a single block. Finally, a large file was encrypted, and its output was decrypted using the previously discussed AES-NI decryption program. This procedure made it easier to identify which functions introduced bugs in the C implementation (it uncovered several byte-ordering bugs).

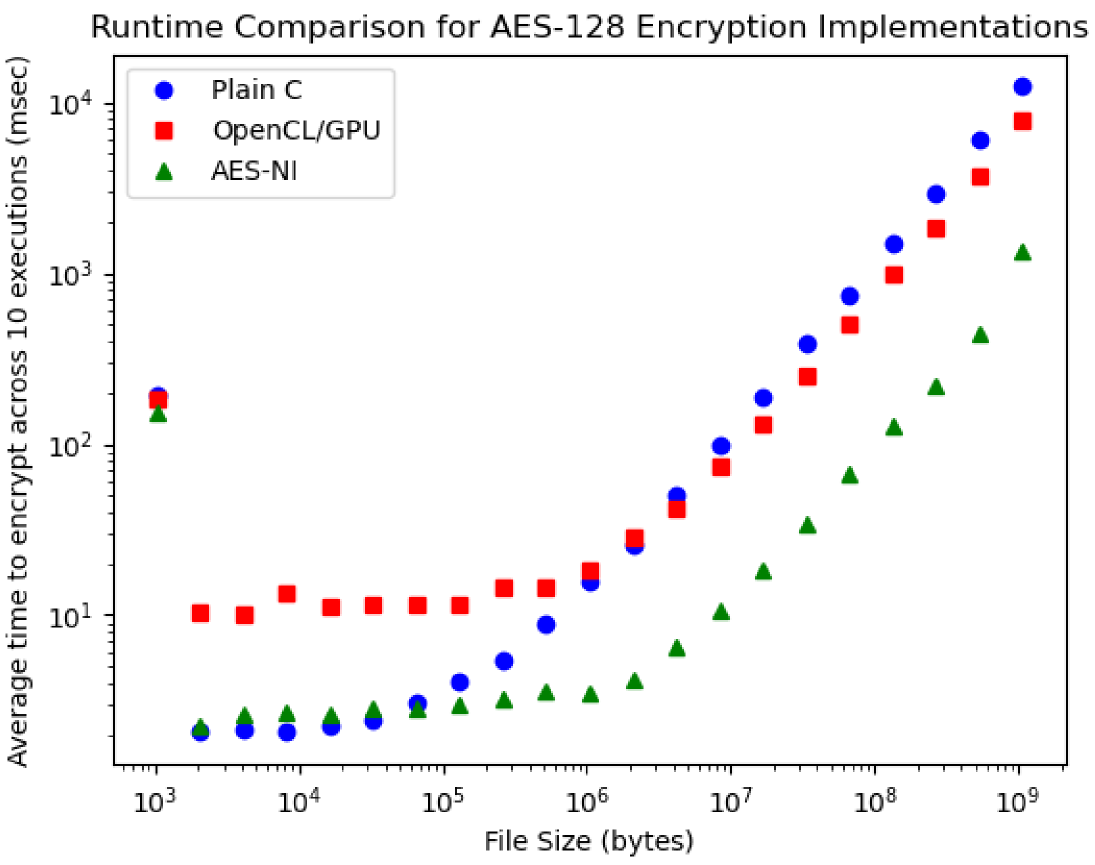

Fig. 1. Runtime growth for all AES implementations

The OpenCL implementation was tested by encrypting a full file and then decrypting it to ensure that the decrypted text matched the plaintext. Since this implementation used functions derived from the plain C code, it was not necessary to debug function-by-function.

B. Benchmark Software

Benchmark executables were developed to run each AES implementation. These programs use a hard-coded key to eliminate the need to provide one as an input. Memory-mapped input (i.e. plaintext) and output (i.e. ciphertext) files are used. By using mmap() instead of read() and write(), the number of system calls is reduced substantially, which improves performance.

The software-only C and AES-NI accelerated C programs use POSIX threads (i.e. pthreads) to achieve parallelism. Each thread runs a for-loop that encrypts one block per iteration. Instead of using mutexes or any other synchronization method, the input and output files are broken up along cache line boundaries. Since each cache line is accessed from only one thread, there are no data hazards, and synchronization is completely unnecessary. Another benefit of this approach is that each thread is assigned a continuous span of memory, which improves data locality and therefore should improve the cache performance of the benchmark program.

The OpenCL version of the code performs some "boilerplate" OpenCL setup, including querying the system for devices and creating a context. Afterwards, it loads a pre-compiled program onto the GPU, sets it up as a kernel, copies data from the input file to the GPU. The encryption is placed on the GPU's queue, and once it completes execution, the outputs are copied back to the CPU to be stored. OpenCL does not require any special considerations to be made to prevent data hazards.

C. Scripts to Reproduce Output

The source code includes several scripts to set up the system to run benchmarks. After cloning and cd'ing into the the git repository, executing "scripts/build.sh" will compile all of the programs into the "bin" directory. Running "scripts/make_inputs.sh" will generate pseudorandom files ranging from 1 kilobyte to 1 gigabyte in the "input" directory and will create an "output" directory.

Additionally, the scripts which execute the benchmarks and generate the results plot are provided. Executing "scripts/run_bench.py" will execute all three benchmarks 10 times per input file. It then outputs the average runtime for each implementation and file size combination. Copying this output into the "scripts/plot.py" program and then executing it provides the final plot.

D. Other Utility Programs

There is another program in "bin", which calculates complete lookup tables for each byte multiplied by 0x2 and 0x3 in GF(2^8) modulo 0x1b. This output is used in the source code as an optimization for the software-only and GPU implementations.

E. Future Work

There are several aspects of this project which can be studied further.

The possibility of encrypting multiple blocks in one function call should be explored, instead of encrypting each block independently in a for-loop. This may optimize performance by removing some branch instructions and even allowing the compiler to identify optimizations using SIMD instructions.

In regards to the GPU version of the program, it should be studied on a dedicated GPU rather than the integrated GPU that was used in this project. Such a GPU would have more memory and more cores, so it would better demonstrate the massively parallel nature of GPUs. It would be interesting to find out whether a GPU can outperform the hardware-accelerated software by encrypting hundreds of blocks simultaneously. One cost-effective way to test this is to rent an instance of a cloud server with a GPU.

The OpenCL/GPU code might have opportunities for further optimization. Experimenting with kernel size and memory synchronization options may allow for better performance than relying on the defaults.

Additionally, the implementations presented here are insecure. It is vulnerable to side-channel attacks which may be mitigated using more advanced software techniques. Additionally, the ECB mode is cryptographically vulnerable, as the ciphertext reveals the presence and location of repeated 16-byte blocks in the plaintext. Replacing ECB mode with CTR mode would mitigate this risk while preserving the parallel nature of this implementation.

REFERENCES

[1] S. Gueron, "Intel advanced encryption standard (aes) new instructions set," tech. rep., Intel, May 2010.

[2] "Advanced encryption standard (aes)," tech. rep., National Institute of Standards and Technology (NIST), Nov. 2001.

[3] N. R. Wagner, "The laws of cryptography: The finite field gf(2^8)," 2001.

[4] "The opencl specification," tech. rep., Khronos OpenCL Working Group, July 2019.