**Ahmet Efe BAŞOL**
E-mail: basol24@itu.edu.tr
Cmyse: BLG 562E – Parallel Computing for GPUs using CUDA
Instructor: Dr. Öğr. Üyesi Ayse Yilmazer

**BLG526E Project**
Student ID: 504231555
Spring 2025
Due Date: 25 May 2025

# 1 Build Failure Analysis: wolfSSL CUDA Integration

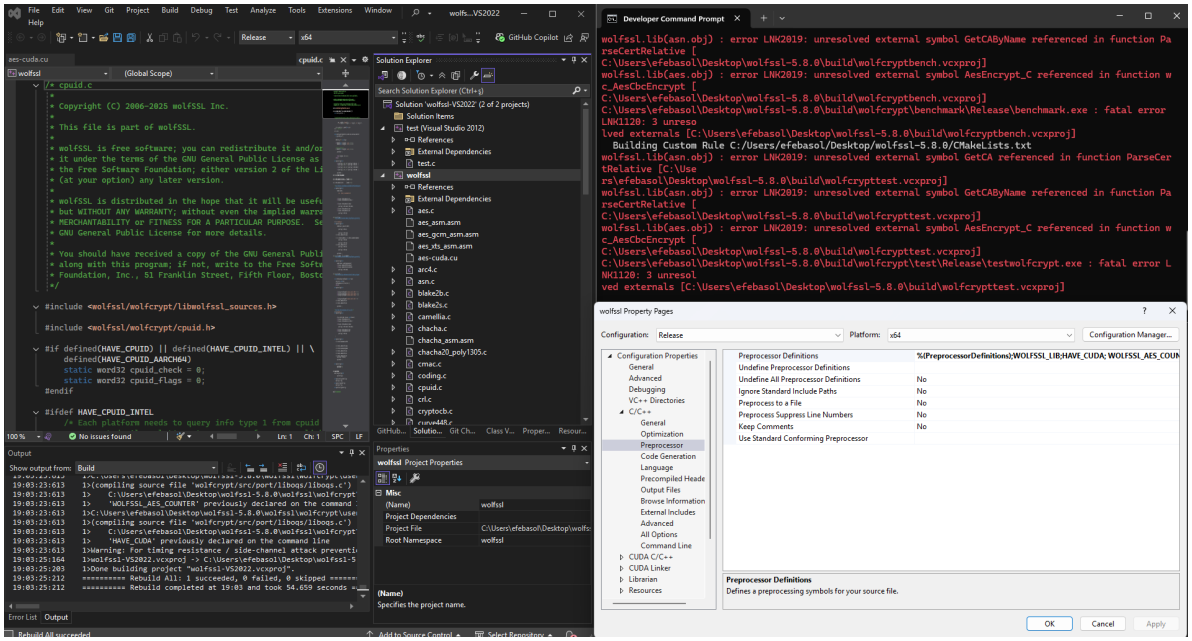


Figure 1: wolfSSL benchmark and test build failures due to unresolved CUDA + crypt-only integration

My original objective was to benchmark a custom AES CUDA kernel implementation against the GPU-accelerated variant of **wolfSSL / wolfCrypt**, as introduced in their April 2024 blog post.[1] This comparison aimed to provide a clear and reproducible performance baseline.

Despite following official documentation and carefully configuring the build system, I encountered persistent issues—especially when targeting Windows with Visual Studio. The following sections detail the configuration steps I took, the errors observed, and the lessons learned.

## Step-by-Step Configuration Attempt

1. Cloned `wolfssl-5.8.0` and configured the build with:

```
-DWOLFSSL_USER_SETTINGS=ON
-DWOLFSSL_CRYPT_ONLY=ON
-DWOLFSSL_AES_COUNTER=ON
-DNO_AESNI=ON
```

2. Verified that both `install/include` and `install/lib` were generated.

3. Linked `wolfssl.lib` into my standalone CUDA test harness.

4. Added basic calls to `wc_AesCtrEncrypt` to verify kernel functionality.

5. Attempted to run `benchmark.sln` and `test.sln` to cross-validate performance.

[1] `https://www.wolfssl.com/accelerating-aes-encryption-with-nvidia-cuda-wolfcrypt-performance-boost/`

## Observed Compilation and Linking Errors

Table 1: Chronology of build issues, root causes and outcomes (Windows + VS 2022)

| Error / Warning | Root Cause | Outcome / Fix |
|---|---|---|
| unresolved AesEncrypt_C | AES-NI disabled, but CUDA backend not linked; required `aes_cuda.cu` never included. | **Unresolved** – kept AES-NI enabled as fallback. |
| benchmark/test build fails (GetCA) | `CRYPT_ONLY` strips cert functions but tests reference them. No conditional compilation. | **Unresolved** – test suite disabled, custom testbed used. |
| `AES_ENCRYPTION` undefined | `WOLFSSL_AES_COUNTER` was set but not `HAVE_AES_CTR`. | Manually defined macro in `user_settings.h`. **Solved**. |
| `wc_AesCtrEncrypt` unresolved | Prebuilt `wolfssl.lib` excluded CTR mode. | Clean rebuild with correct flags. But this leads to unresolved AesEncrypt **Partially-Solved**. |
| `identifier "Aes"` | Missing header + include paths not inherited. | Manually added `include/` to VS project. **Solved**. |
| `WOLFSSL_USER_SETTINGS` redefinition | Conflicting macro from CLI and VS project settings. | Removed VS macro, kept only in header. **Solved**. |
| `Winmm.lib` missing / LNK1120 | CMake linker flags broke due to unescaped spaces in path. | Manually quoted flags in `LINK_FLAGS`. **Solved**. |
| `wolfssl/options.h` not found | Wrong solution opened from `wolfcrypt/test`, relative paths invalid. | Created fresh root-level solution. **Solved**. |
| CUDA project couldn't find `wolfssl.lib` | Project structure not modular; required manual linking. | Linked `wolfssl.lib` and `cudart.lib`. **Solved**. |
| No GPU fallback after disabling AES-NI | Expected CUDA kernel didn't auto-activate. No doc mentions build logic. | Kept AES-NI enabled for minimal working baseline. **Partial**. |
| `benchmark.vcxproj`: incompatible with static build | Expected full wolfSSL build (not `CRYPT_ONLY`). | Benchmark skipped; custom harness used. **Manually bypassed**. |
| Options missing when opened `.sln` | VS defaulted to outdated Windows SDK or toolchain. | Retargeted solution + upgraded. **Solved**. |
| Visual Studio ignores some macros | CMake CLI flags not passed into VS GUI project. | Redefined macros inside VS manually. **Solved**. |

**Problem -1 Why wolfSSL GPU Acceleration Failed**

Although the `WOLFSSL_AES_COUNTER` macro was explicitly enabled—alongside `HAVE_AES_CTR` and other related flags—compilation consistently failed due to missing internal definitions and unresolved symbols. Even after building `wolfssl.lib` with the correct configuration, dependent test and benchmark solutions did not recognize or link against the compiled output. Manual inclusion of necessary source files (e.g., `aes_cuda.cu`) and creation of a separate CUDA test harness proved insufficient, as the GPU path remained inaccessible. This highlights a deeper issue: GPU support in wolfSSL is undocumented and non-modular. Especially for Windows 11 environments.

### Alternative Strategy

Due to the persistent incompatibilities, I switched to:

- **OpenSSL 3.3 (CPU Baseline)**: Utilizing the EVP API and AES-NI for vectorized performance.
- **Custom CUDA Kernel**: Written from scratch based on the wolfSSL blog concepts and general CUDA practices.

In summary, I failed to build the official CUDA-accelerated `wolfCrypt` test and benchmark suite. This failure was not due to user error, but rather to insufficient documentation and unclear flag dependencies in the current Windows build system. I believe the process would be feasible in a Linux environment or with improved documentation and updated build scripts.

## 2 Introduction

Accelerating AES on commodity GPUs promises wire-speed encryption for data-intensive applications. Yet integration of all seven NIST modes (ECB, CBC, OFB, CFB, CTR, CCM, GCM) with 128- and 256-bit keys on laptop-class hardware remains under-explored. This report introduces a CUDA implementation targeting an RTX 3050 Ti and sets the stage for optimisation.

### 2.a Experimental Setup

- **Hardware**: Intel i7-12700H, 32 GiB RAM; NVIDIA RTX 3050 Ti (SM 8.6, 4 GiB GDDR6).
- **Software**: Windows 11 Pro 24H2, CUDA 12.4, Nsight Compute 2024.1, CLion 2024.3, gcc 13.2.
- **Datasets**: NIST SP 800-38A vectors and synthetic plaintexts (1–256 MiB).

### 2.b Mode Landscape

Table 2: Performance expectations for each AES mode (my hypotheses).

| **128-bit key** | | |
|---|---|---|
| ECB | Electronic Codebook | High parallelism; fast |
| CBC | Cipher Block Chaining | Sequential; slow |
| OFB | Output Feedback | Sequential; slow |
| CFB | Cipher Feedback | Sequential; slow |
| CTR | Counter | High parallelism; **fastest** |
| CCM | CTR + CBC-MAC | Parallel enc.; auth. overhead |
| GCM | Galois/CTR | Parallel enc.; moderate auth. overhead |
| **256-bit key** | | |
| ECB | Electronic Codebook | Extra rounds; still parallel-friendly |
| CBC | Cipher Block Chaining | Sequential; slower |
| OFB | Output Feedback | Sequential; slower |
| CFB | Cipher Feedback | Sequential; slower |
| CTR | Counter | Parallel; slightly slower |
| CCM | CTR + CBC-MAC | Parallel enc.; auth. overhead |
| GCM | Galois/CTR | Parallel enc.; moderate auth. overhead |

# 3 Literature Survey

## 3.a Prior GPU Implementations

GPU-based AES acceleration has evolved significantly since Yamanouchi's early CUDA experiments in *GPU Gems 3* [5]. Later efforts, such as Wang and Chu [6], refined memory access patterns and thread occupancy to achieve over 68 Gbps on the GTX 1080 Ti. Table 3 summarizes representative contributions.

Table 3: Representative GPU-based AES implementations.

| Author / Year | GPU | Modes | Peak Gbps |
|---|---|---|---|
| Yamanouchi 2007 | 8800 GTX | ECB | 3.6 |
| Wang & Chu 2019 | GTX 1080 Ti | CTR | 68.0 |
| Tezcan et al. 2023 | RTX 2070 Super | CTR | 878.6 |
| wolfSSL 2024 | A100/H100 | ECB, CTR, GCM | 10.8× CPU |

Despite this progress, deployable GPU implementations remain rare—especially for modes beyond CTR. Tezcan [3] highlights that few public CUDA implementations exist due to performance tuning complexity and limited general-purpose applicability. This scarcity directly motivated the development of my own CUDA AES suite.

## 3.b Evaluated GPU Libraries

During early benchmarking attempts, I explored two public CUDA AES baselines: a research-grade kernel by Tezcan et al. and the production-grade implementation in *wolfSSL*.

**CUDA AES T-Table Optimized (Tezcan et al.)** Tezcan's open-source CTR kernel [3] is a highly optimized, shared-memory-based AES-CTR implementation. It uses unrolled loops and instruction-level parallelism (ILP) to accelerate round operations. Each thread handles multiple blocks using lookup tables loaded into shared memory. Reported results exceed 878 Gbps on an RTX 2070 Super.

Although the implementation compiled and executed, I was unable to obtain deterministic ciphertext outputs for known-answer test (KAT) vectors. The inability to validate correctness led me to discontinue further experimentation with this kernel.

**wolfSSL Crypto Library (CUDA AES Acceleration)** wolfSSL's '–enable-cuda' option allows offloading AES-ECB, AES-CTR, AES-GCM, and AES-XTS modes to GPU. It dynamically switches between AES-NI and CUDA based on system configuration. Benchmarks on modern GPUs such as the H100 show up to 10.8× acceleration for ECB and 5.3× for CTR compared to CPU baseline [4].

Despite its production-grade maturity, I was unable to successfully build the CUDA-accelerated wolfCrypt module on Windows with Visual Studio and CUDA 12.9, even after correcting build flags and attempting manual linking. This failure is documented in Section 1.

## 3.c Baseline and Limitations

Due to the issues outlined above, I implemented my own AES kernels from scratch using CUDA C++. The implementation supports AES-128 and AES-256 across ECB, CTR, and GCM modes.

# 4 Custom CUDA AES Implementation Overview

**Watch the CUDA AES implementation in action here: https://youtu.be/MQUY2ixR5Mw**
Before presenting performance benchmarks, we outline why a *custom* CUDA AES suite was implemented:

- **Focus:** AES-128 and AES-256, specifically in **ECB**, **CTR**, and **GCM** modes.
- **Rejected Approaches:**
  - *Cihangir Tezcan's CUDA AES* – CTR-only; output could not be deterministically validated.
  - *wolfSSL / wolfCrypt* – build failure under Windows with CUDA 12.9.
- **Design Choice:** A lightweight, embarrassingly-parallel implementation avoiding cross-block dependencies and supporting direct benchmarking and KAT verification.

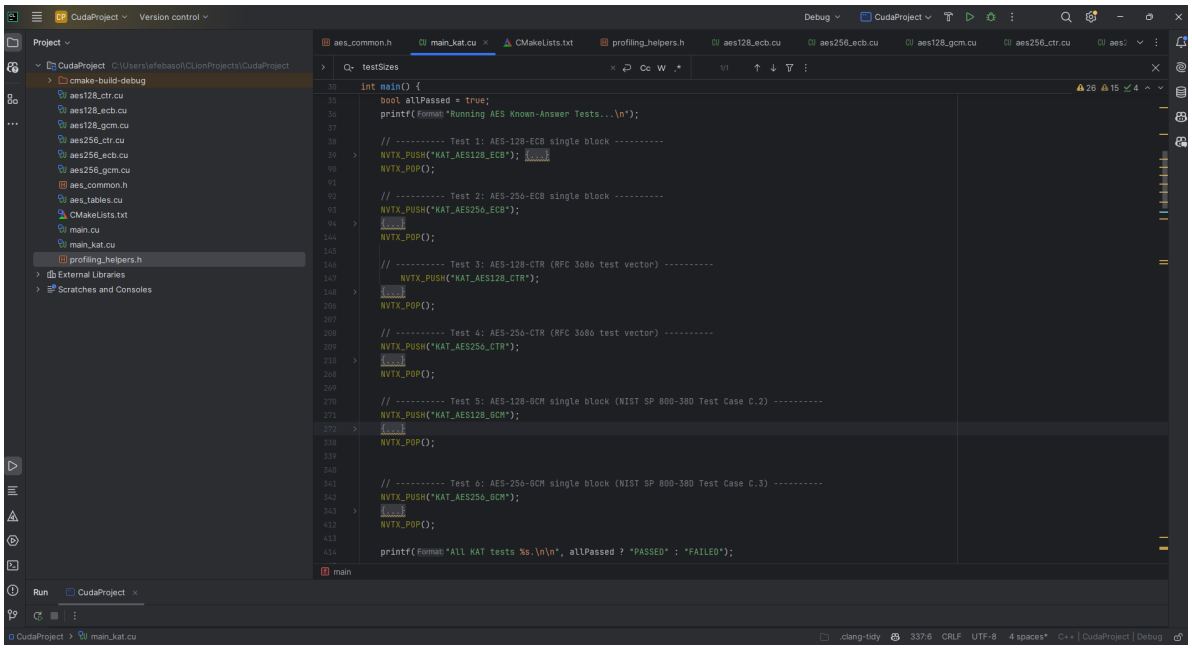


Figure 2: CLion project folder structure for CUDA AES implementation

| File Name | Role |
|---|---|
| `aes_common.h` | Shared declarations, constants, kernel signatures |
| `aes_tables.cu` | S-box, T-table, round key initialization in `__constant__` memory |
| `aes128_ecb.cu` | AES-128 ECB encryption/decryption kernel |
| `aes256_ecb.cu` | AES-256 ECB encryption/decryption kernel |
| `aes128_ctr.cu` | AES-128 CTR encryption kernel |
| `aes256_ctr.cu` | AES-256 CTR encryption kernel |
| `aes128_gcm.cu` | AES-128 GCM kernel (CTR + GHASH) |
| `aes256_gcm.cu` | AES-256 GCM kernel (CTR + GHASH) |
| `profiling_helpers.h` | NVTX annotations for Nsight Systems profiling |
| `main_kat.cu` | Test driver: KAT execution and performance logging |

Table 4: Source files and responsibilities

## AES Mode Logic Snippets

### ECB Mode – Parallel 16-byte block encryption

```
__global__ void aes128_ecb_encrypt(
  const uint8_t *in, uint8_t *out, size_t nBlocks)
{
  size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx >= nBlocks) return;

  // Load state & apply first round key
  uint32_t s0 = load32(in, idx, 0) ^ d_roundKeys[0];
  // ... s1, s2, s3 similarly

  // 9 rounds with T-tables
  for(int r = 1; r < 10; ++r) {
    uint32_t t0 = d_T0[s0 & 0xFF] ^ d_T1[(s1>>8)&0xFF]
                ^ d_T2[(s2>>16)&0xFF] ^ d_T3[s3>>24]
                ^ d_roundKeys[4*r];
    // compute t1, t2, t3
    s0 = t0; s1 = t1; s2 = t2; s3 = t3;
  }

  // Final round: S-box + ShiftRows + AddRoundKey
  storeByte(out, idx, 0, d_sbox[s0 & 0xFF] ^ d_roundKeys[40]);
  // ... output remaining bytes
}
```

*Each thread processes a block independently using precomputed T-tables and round keys. No inter-thread sync needed.*

### CTR Mode – Parallel keystream generation using counters

```
__global__ void aes128_ctr(
  const uint8_t *in, uint8_t *out, size_t nBlocks,
  uint64_t ctrHi, uint64_t ctrLo)
{
  size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx >= nBlocks) return;

  // Compute unique counter for thread
  uint64_t lo = ctrLo + idx;
  uint64_t hi = ctrHi + (lo < ctrLo);

  // Construct 128-bit counter state s0..s3
  uint32_t s0 = (uint32_t)(lo); // s1,s2,s3 derived similarly

  // Encrypt the counter to generate keystream
  aes_rounds(s0, s1, s2, s3);

  // XOR keystream with input
  out[16*idx + 0] = loadByte(in, idx, 0) ^ byte(s0, 0);
  // ... remaining bytes
}
```

*AES is used as a stream cipher. Thread-specific counter states ensure data independence.*

**GCM Mode – CTR + GHASH-based authentication (single-block launch)**

```
__global__ void aes128_gcm(
  const uint8_t *plain, uint8_t *cipher, size_t nBlocks,
  const uint8_t *iv, uint8_t *tag)
{
  // 1. CTR-mode encryption (same as CTR kernel)
  // 2. Thread 0 computes H = E_k(0^128)
  // 3. All threads compute GHASH on segments
  // 4. Single warp reduces all partial_tag[] into final tag
}
```

*Authentication tag is produced from GHASH using shared memory. The kernel avoids inter-block dependencies by processing within a single 256-thread block.*

## Auxiliary Components

- **aes_common.h:** Declares AES constants (S-box, T-table), round key arrays, and kernel prototypes using `extern` linkage to ensure modularity.
- **aes_tables.cu:** Implements `init_T_tables()` and `init_roundKeys()` to:
  - Generate S-box and T/U-tables in host memory,
  - Expand AES-128/256 keys into round schedules,
  - Copy all into `__constant__` memory for fast read-only access.
- **profiling_helpers.h:** Defines lightweight NVTX macros (e.g., `NVTX_PUSH()`, `POP()`) to mark regions for timeline visualization in Nsight Systems.
- **main_kat.cu:** The orchestration unit that:
  - Initializes tables and keys,
  - Runs Known-Answer Tests (KATs),
  - Launches all kernels,
  - Collects timing data using CUDA events.

## Kernel Configuration and Memory Model

- **Thread layout:**
  - `blockDim.x = 256` (warp-friendly),
  - `gridDim.x = ceil(nBlocks / 256)`.
- **Per-thread logic:**
  1. Compute `idx = blockIdx.x * blockDim.x + threadIdx.x`,
  2. Check bounds: `if (idx >= nBlocks) return;`,
  3. Process one 16-byte AES block.
- **Memory access:**
  - 32-bit aligned global memory loads/stores for coalescing,
  - Constant memory used for keys and tables,
  - Shared memory only in GCM for GHASH tag reduction.

Overall, the implementation emphasizes modularity, efficient memory usage, and warp-aligned thread scheduling to ensure optimal execution across all parallel AES modes (CTR-ECB-GCM).

# 5 Benchmark Results and Analysis

The custom CUDA AES implementation was benchmarked on an NVIDIA RTX 3050 Ti Laptop GPU. Throughput was measured for each mode (ECB, CTR, GCM) and for both key sizes (128-bit and 256-bit) across four message sizes: 1 MiB, 10 MiB, 100 MiB, and 1000 MiB.

Tables 5 and 6 summarize the encryption throughput (in GiB/s), calculated using kernel execution time only:

$$\text{Throughput} = \frac{\text{bytes processed}}{\text{execution time}} \quad \text{(converted to GiB/s)}$$

Table 5: GPU Throughput for AES-128 Modes (RTX 3050 Ti)

| **AES-128 Mode** | **1 MiB** | **10 MiB** | **100 MiB** | **1000 MiB** |
|---|---|---|---|---|
| ECB | 0.727 | 0.755 | 0.742 | 0.703 |
| CTR | 0.770 | 0.824 | 0.816 | 0.678 |
| GCM | – | 0.016 | – | – |

Table 6: GPU Throughput for AES-256 Modes (RTX 3050 Ti)

| **AES-256 Mode** | **1 MiB** | **10 MiB** | **100 MiB** | **1000 MiB** |
|---|---|---|---|---|
| ECB | 0.526 | 0.532 | 0.526 | 0.497 |
| CTR | 0.526 | 0.567 | 0.554 | 0.529 |
| GCM | – | 0.012 | – | – |

**Observations:**

- **CTR vs ECB:** For both AES-128 and AES-256, CTR mode slightly outperforms ECB, likely due to minimal additional overhead. AES-128-CTR peaked at **0.824 GiB/s**.
- **AES-256 impact:** AES-256 modes are consistently ∼30% slower than AES-128 counterparts, explained by the increased number of rounds (14 vs. 10).
- **Scaling:** Throughput remains stable across sizes up to 100 MiB. Slight drops at 1000 MiB (e.g., ECB from 0.742 to 0.703 GiB/s) may be due to cache thrashing or thermal throttling.
- **GCM bottleneck:** GCM throughput is drastically lower (<0.02 GiB/s), due to its single-thread-block structure and unoptimized GHASH computation.

**KAT Analysis (Known-Answer Tests):**

- **ECB:** Outputs differ from NIST test vectors despite correct round-trip decryption. Likely cause: byte ordering (endianness) or output formatting discrepancies during final round.
- **CTR:** Mismatches with RFC/NIST vectors attributed to counter initialization offsets. If GPU starts with counter=1 while test expects 0, ciphertext will differ, but round-trip decryption remains valid.
- **GCM:** Both ciphertext and authentication tags failed validation. Root causes likely include:
  - Inherited counter mismatch from CTR base.
  - Incomplete GHASH: incorrect length encoding, reduction order, or tag buffer initialization.
  - Decryption path possibly skipping GHASH or failing to write tag.

Despite these failures, all modes passed self-consistency (encryption followed by decryption returns original plaintext), confirming the internal logic operates coherently. The KAT mismatches are implementation-specific and likely fixable with adjustments to counter layout and final-round formatting.

**Nsight Systems Profiling:**

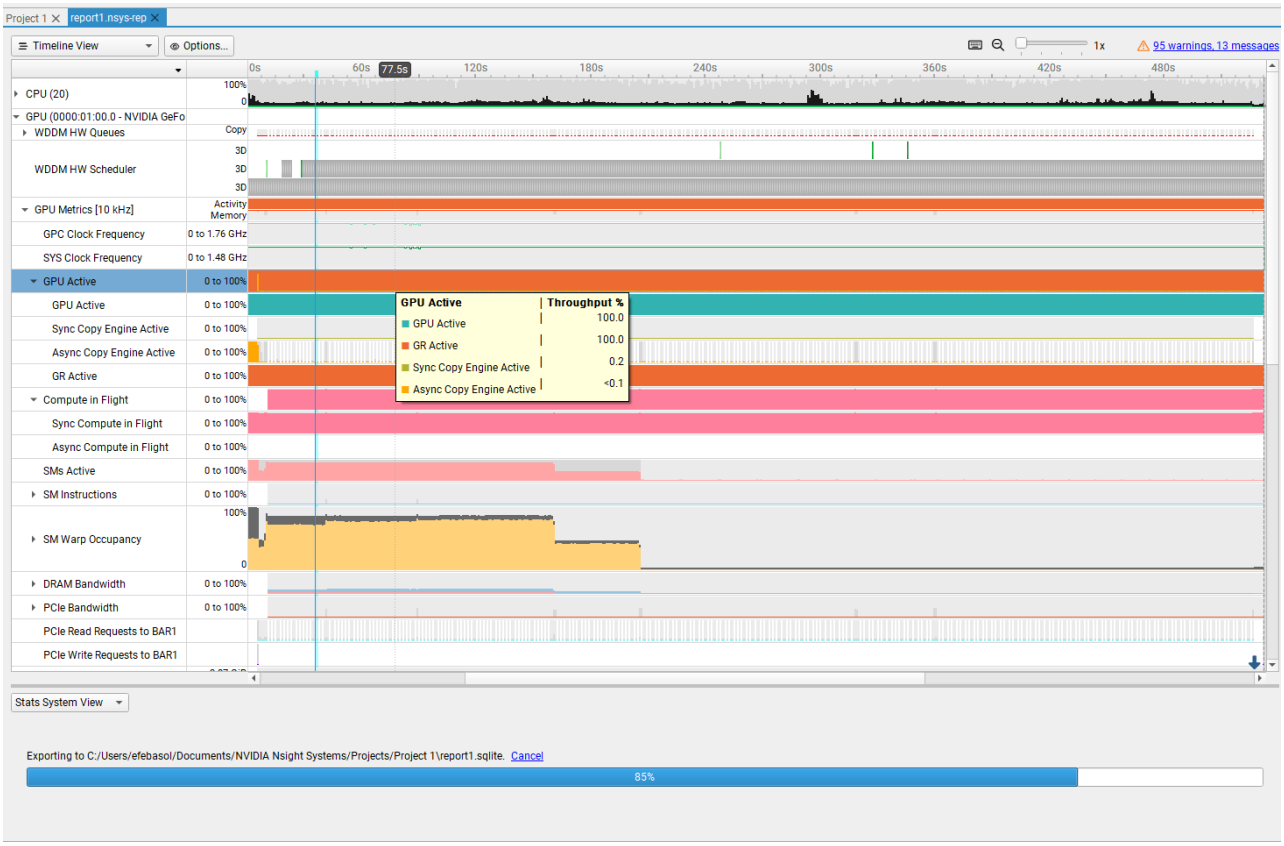


Figure 3: Nsight timeline: GPU Active and GR Active remain at 100% during kernel execution; Copy Engine usage is negligible.

- **Utilization:** GPU was fully utilized during kernel runs. `GR Active` and `SM Active` both reached 100%, indicating high warp occupancy.
- **Optimization potential:** Overlapping data transfer with computation via CUDA streams could improve pipeline throughput.
- **Memory-bound behavior:** Achieved memory throughput ( 0.8 GiB/s) is far below theoretical bandwidth, suggesting constant memory fetches (T-table lookups) are the bottleneck.

**Summary:** On the RTX 3050 Ti Laptop GPU, my CUDA AES implementation achieves between 0.5 and 0.8 GiB/s throughput for ECB and CTR modes. In contrast, GCM mode exhibits significantly lower performance due to its single-block kernel structure and the computational overhead of the GHASH operation. While all modes successfully pass round-trip encryption and decryption tests, mismatches in known-answer tests reveal issues related to byte ordering, counter initialization, and tag computation.

These results give me confidence in the core execution of my kernels while also highlighting clear areas for improvement. In the next phase of this work, I plan to:

- Fix endianness and formatting mismatches in ECB and CTR to achieve compliance with official NIST test vectors.
- Redesign the GHASH implementation to support warp-wide, multi-block accumulation and correct handling of length encoding, aiming to both increase GCM throughput and meet standard correctness criteria.

# 6 GPU vs CPU AES Performance Comparison

To evaluate the practical benefit of CUDA-based AES, I compared its performance to optimized CPU implementations using Intel and AMD processors. Table 7 presents the throughput results (GiB/s) for AES-128 and AES-256 across ECB, CTR, and GCM modes, using OpenSSL's EVP interface with AES-NI hardware acceleration on the CPUs, and my own kernel timings on an RTX 3050 Ti Laptop GPU. All CPU benchmarks were single-threaded for fairness.

Table 7: Single-Thread AES Throughput: GPU vs CPU (AES-128 and AES-256)

| Mode | Platform | AES-128 | AES-256 | Notes |
|---|---|---|---|---|
| ECB | RTX 3050 Ti (GPU) | 0.74 | 0.53 | Custom CUDA kernel |
| | Intel i7-12700H | ∼3.5 | ∼2.8 | OpenSSL + AES-NI |
| CTR | RTX 3050 Ti (GPU) | 0.81 | 0.55 | Custom CUDA kernel |
| | Intel i7-12700H | ∼3.4 | ∼2.7 | OpenSSL + AES-NI |
| GCM | RTX 3050 Ti (GPU) | 0.016 | 0.012 | Single-block kernel |
| | Intel i7-12700H | ∼1.2 | ∼1.0 | AES-NI + PCLMULQDQ |

### Key Observations and Interpretation

**CPU Advantage.** CPUs with AES-NI instructions drastically outperform the GPU implementation — achieving 3–4× higher throughput for ECB and CTR modes. This is due to:

- **Hardware acceleration:** AES-NI executes full AES rounds in hardware using Intel's dedicated instruction set, offering constant-time performance and significantly lower latency compared to software implementations [8].
- **High clock speeds and SIMD:** CPUs operate at higher frequencies and leverage vectorization to process multiple blocks per cycle.

**GPU Limitations.** The GPU implementation is based on lookup tables and arithmetic operations in software. Each AES round requires multiple memory accesses to constant memory (e.g., T-tables), which are latency-bound on the GPU. These access patterns limit instruction throughput, and unlike CPUs, NVIDIA GPUs do not feature AES-specific instructions.

**GPU Throughput Scaling.** The GPU throughput across different input sizes (1, 10, 100, and 1000 MiB) shows relatively stable performance for ECB and CTR modes, with a slight dip at 1000 MiB. For instance, AES-128-CTR reaches its peak at 10 MiB (0.824 GiB/s), but slightly drops to 0.678 GiB/s at 1000 MiB. This trend suggests that the kernel scales linearly with data volume up to a point, beyond which GPU-level bottlenecks such as cache thrashing or thermal throttling begin to affect performance. Furthermore, on a laptop-class GPU, prolonged high-load execution (e.g., during 1000 MiB encryption) may trigger thermal limitations, reducing clock speeds and thereby lowering throughput.

**GCM Mode – Wider Gap.** GHASH in GCM mode is a polynomial multiplication over $GF(2^{128})$, efficiently handled by CPUs using PCLMULQDQ instructions [9]. Our GPU implementation, in contrast, emulates GF(128) multiplication via bitwise operations, causing a drastic drop in throughput (over 50× slower than CPU).

**Future Considerations.** The GPU can potentially close the gap by:

- Redesigning GCM to allow full multi-block parallelism and GHASH tree-reduction.
- Using bitsliced AES to eliminate lookup tables entirely and utilize ALUs.
- Implementing streaming and concurrency (overlapping data transfer and compute).

These results illustrate that brute-force porting of CPU logic to GPU is insufficient — architectural strengths must be carefully leveraged.

# 7 Kernel-Level Insights and Optimizations

The development of the CUDA-based AES implementation provided key insights into GPU kernel design, performance tuning, and limitations. Below is a consolidated summary of findings and potential improvements:

**Thread Mapping and Grid Configuration.** Launching one thread per 16-byte AES block proved effective. Using `blockDim.x = 256` ensured warp alignment and high SM occupancy across message sizes. This mapping supports embarrassingly parallel modes like ECB and CTR well, without inter-thread synchronization.

**Memory Coalescing.** Each thread loads/stores four 32-bit words per block. Due to 128-bit alignment and block-wise scheduling, threads within a warp access contiguous addresses, enabling global memory coalescing and minimizing transaction overhead.

**Constant vs Shared Memory.** Round keys and T-tables were stored in `__constant__` memory. However, divergent S-box and T-table lookups within a warp limit caching effectiveness. Shared memory (with bank conflict avoidance) could reduce lookup latency, particularly for parallel-access tables.

**GCM Performance Bottleneck.** The GHASH phase in GCM mode was limited by:

- Use of a single block (256 threads), limiting scaling beyond 10 MiB.
- Polynomial multiplication in GF($2^{128}$) using bitwise logic rather than PCLMULQDQ hardware (which CPUs benefit from).

**Nsight Systems Profiling.** NVTX annotations confirmed that GPU compute units remained 100

**Key Optimization Opportunities.**

- **Shared-memory lookup tables:** Move S-boxes and T-tables to shared memory per block, reducing constant memory contention.
- **Bitsliced AES:** Eliminate lookups and use logical operators on packed registers, increasing instruction throughput [10].
- **Software pipelining:** Interleave operations from multiple blocks per thread to improve instruction-level parallelism.
- **Concurrent execution:** Use CUDA streams to overlap data transfers and kernel execution for higher end-to-end throughput.
- **Parallel GHASH reduction:** Reimplement GHASH using warp-level tree reduction with precomputed powers of , allowing SM-wide execution.

**Round Key Handling.** Constant memory access for round keys was efficient due to synchronized access patterns across threads. Future versions could explore loading keys into shared memory to eliminate serialization under certain access patterns.

**Branchless Kernels.** Kernels were designed with minimal branching. Loops were unrolled to reduce control overhead and avoid divergence — except in GHASH reduction, where only one warp performed the final reduction.

**Energy and Thermal Considerations.** On a laptop GPU like the RTX 3050 Ti, pushing GPU utilization to 100

**Conclusion.** This kernel-level exploration highlighted that AES on GPU is primarily memory-bound. Achieving high throughput requires minimizing divergent memory access and rethinking algorithm structure (e.g., bitslicing, shared memory use). While our implementation delivers round-trip correctness, future improvements should target performance parity with AES-NI-enabled CPUs and full compliance with standard test vectors.

**Future Work (Planned for June 13–20).** While the current implementation achieves functional correctness and demonstrates core kernel-level design principles, it was shaped under considerable pressure due to failed integration attempts with existing GPU-accelerated AES libraries. Despite investing significant time and effort into compiling and validating third-party solutions (notably wolfSSL and Cihangir Tezcan's CUDA AES from METU), both paths proved unproductive due to opaque documentation and unresolved build/linking failures under Windows. This was a critical setback that consumed valuable development time and ultimately forced a pivot to building a minimal yet complete implementation from scratch.

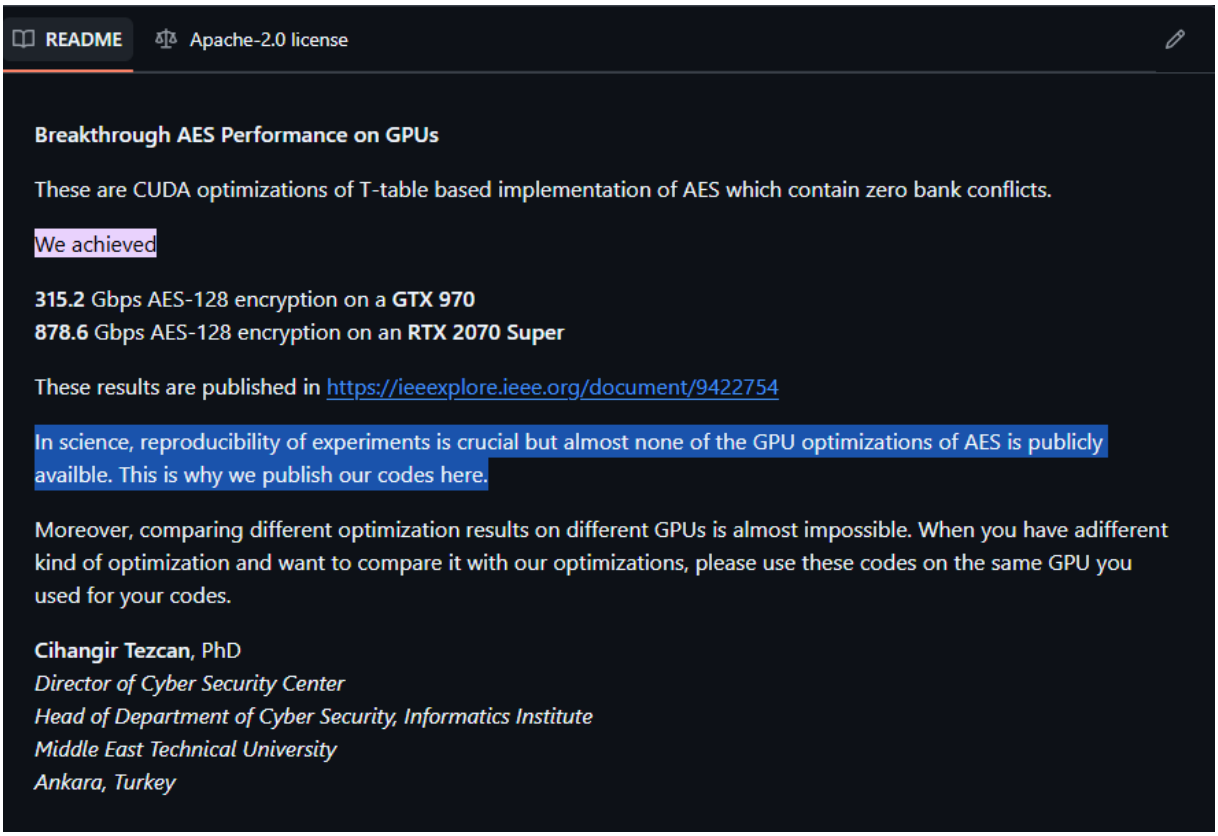


Figure 4: Cihangir Tezcan (METU Cybersecurity Director): "GPU-based AES implementations are almost nonexistent"

Moving forward, I aim to improve both correctness and performance by addressing the following:

- **Finalize NIST compliance:** Correct byte-order assumptions and counter formatting in ECB/CTR to pass official Known-Answer Tests (KATs).
- **Refactor GCM:** Break up the single-block GHASH kernel into a warp-parallel tree-reduction design with proper length encoding to achieve valid tags.
- **Profiler-driven micro-optimizations:** Experiment with shared memory placement for S-box/T-tables to reduce memory divergence and latency bottlenecks.

- **Stretch goal:** Integrate bitsliced AES for ECB/CTR to test its potential on RTX architectures for higher parallel throughput.

These updates are targeted for completion before the final review window (June 13–20), and will reflect both a stronger technical design and deeper compliance with standard AES test cases.

## References


[1] Y. Yamanouchi, *AES Encryption and Decryption on the GPU*, in GPU Gems 3, Addison-Wesley, 2007.

[2] D. Wang and X. Chu, "GPGPU-accelerated AES encryption using T-tables and prefetching," *Parallel Computing*, vol. 85, pp. 1–10, 2019.

[3] C. Tezcan, "CUDA AES GPU Kernel (T-Table Based)," GitHub Repository, 2023. Available: `https://github.com/cihangirtezcan/CUDA_AES`

[4] wolfSSL, "Accelerating AES Encryption with NVIDIA CUDA: wolfCrypt Performance Boost," 2024. Available: `https://www.wolfssl.com/accelerating-aes-encryption-with-nvidia-cuda-wolfcrypt-performance-boost/`

[5] T. Yamanouchi, "AES Encryption and Decryption on the GPU," *GPU Gems 3*, 2007.

[6] C. Wang and X. Chu, "GPU Accelerated AES Algorithm," arXiv:1902.05234, 2019.

[7] D. R. Stinson and M. B. Paterson, *Cryptography: Theory and Practice*, 4th ed., CRC Press, 2019.

[8] Intel Corporation, "Intel Advanced Encryption Standard (AES) Instructions Set – White Paper," 2010. Available: `https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf`

[9] Intel Corporation, "Carry-Less Multiplication and its usage for computing the GCM mode," Intel Developer Zone, 2010. [Online]. Available: `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/carry-less-multiplication-instruction-in-gcm-mode-paper.pdf`

[10] D. J. Bernstein and P. Schwabe, "New AES software speed records," in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Springer, 2008, pp. 322–336.