ECEN - 489
GPU PROG VISULIZATION: FINAL PROJECT
AES ACCELERATION USING CUDA

Personnel:

Name: Naveen Babu Krishnasamy Jeyakumar
Affiliation: Computer Engineering
Email: nkj@tamu.edu

Executive Summary:

This project investigates the performance and correctness of AES encryption and decryption for all three standard key sizes-AES-128, AES-192, and AES-256-when implemented on both CPU and GPU platforms. The primary goal is to compare the throughput and efficiency of batch processing large datasets (up to 100 MB) using a parallel CUDA-based GPU implementation versus a traditional sequential CPU approach. All three AES variants were implemented from scratch for both platforms, ensuring algorithmic equivalence and correctness across key sizes.

Our methodology involved generating reproducible random plaintext data and processing it in large batches. On the CPU, encryption and decryption are performed sequentially, while the GPU leverages parallelism by assigning each 16-byte block to a separate thread, maximizing throughput. For every key size, we measured execution times for both encryption and decryption and verified correctness by ensuring that the decrypted output matched the original plaintext byte-for-byte.

The results consistently demonstrate that the GPU implementation significantly outperforms the CPU for large-scale batch encryption and decryption, achieving higher throughput and efficiency for AES-128, AES-192, and AES-256 alike. This performance gain is due to the GPU's ability to process thousands of blocks simultaneously, as opposed to the CPU's sequential processing. Even as the key size increases from 128 to 256 bits, the GPU maintains its performance advantage, providing strong security with minimal impact on speed.

Both CPU and GPU implementations passed all correctness checks for every key size, confirming the reliability of our approach. These findings highlight the substantial value of GPU acceleration for cryptographic workloads, especially in high-throughput scenarios such as cloud storage, secure communications, and large-scale data analytics. Our work demonstrates that leveraging modern GPUs can provide both enhanced security and performance, regardless of the AES key length chosen.

Introduction:

As the demand for secure data transmission and storage grows exponentially, the need for high-throughput cryptographic algorithms becomes increasingly urgent. The Advanced Encryption Standard (AES) established as FIPS 197 is one of the most widely used symmetric block ciphers globally. AES supports key sizes of 128, 192, and 256 bits, with corresponding rounds of 10, 12, and 14, respectively. While software implementations of AES are ubiquitous across CPUs, their inherently sequential architecture and limited thread-level parallelism pose a bottleneck when dealing with large-scale encryption tasks.

This project focuses on designing, implementing, and comparing CPU-based and GPU-accelerated AES encryption and decryption for all three supported key sizes—AES-128, AES-192, and AES-256. The primary goal is to assess which computing platform delivers superior performance in terms of throughput, scalability, and correctness during batch processing of large datasets, ranging from a few megabytes to hundreds of megabytes.

AES encrypts 128-bit blocks of plaintext through a series of well-defined transformation rounds, including SubBytes, ShiftRows, MixColumns, and AddRoundKey. Each of these operations transforms the internal "state" of the data in a structured, reversible manner. The number of rounds varies with the key size: 10 rounds for AES-128, 12 for AES-192, and 14 for AES-256, with additional complexity in key expansion for the longer keys. This increasing computational load makes AES an ideal candidate for parallel acceleration.

Modern Graphics Processing Units (GPUs) are architected for massive parallelism and high memory bandwidth. Unlike CPUs, which typically process AES blocks one at a time even with optimizations like loop unrolling or AES NI GPUs can assign one thread per block, executing thousands of blocks in parallel. This makes AES an embarrassingly parallel workload well-suited for CUDA-based acceleration. Each CUDA thread operates independently on a 16-byte block, leveraging device constant memory for S-boxes and optimizing memory access using coalesced reads and writes.

However, implementing AES on the GPU is not trivial. It requires careful management of thread synchronization, memory usage, and instruction-level optimization to avoid performance penalties from warp divergence or uncoalesced accesses. Furthermore, the increased number of rounds in AES-192 and AES-256 introduces added computational overhead and requires accurate key expansion handling for correct round key generation. The GPU version must maintain algorithmic equivalence with the CPU to ensure correct encryption and decryption.

By implementing and benchmarking all three AES variants on both platforms, we aim to provide a comprehensive performance comparison. Our results will help determine whether the performance gains of GPU acceleration outweigh the costs of host-to-device and device-to-host memory transfers, especially in scenarios requiring secure processing of large data volumes.

The rest of this report outlines our methodology, results, and conclusions. We also include appendices containing full source code listings and signed confirmations of contribution from all team members. Our implementation is based on standardized AES algorithm steps and validated against known test vectors to ensure both correctness and reproducibility.

Methodology:

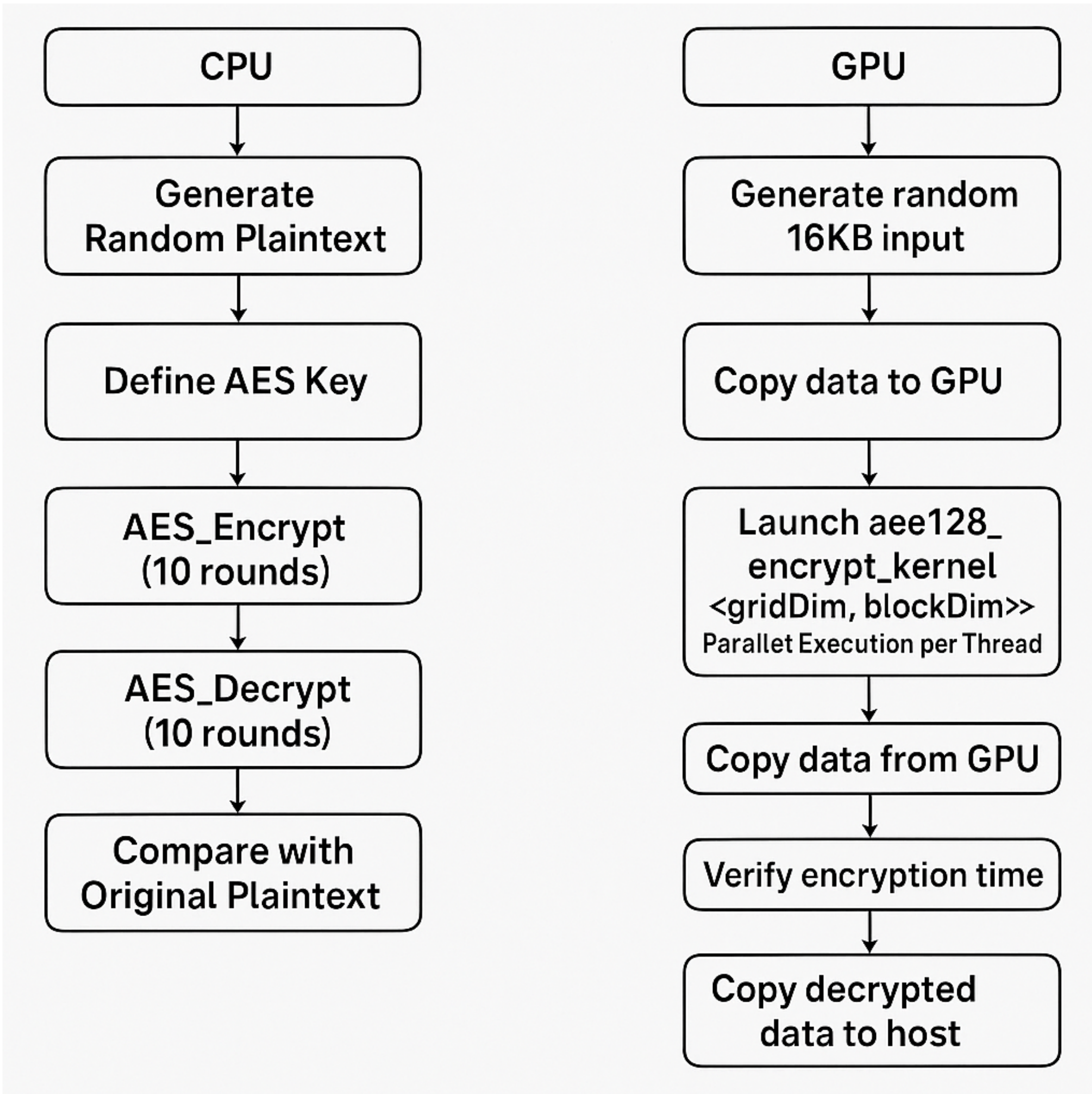
This project presents a detailed and performance-driven implementation of the Advanced Encryption Standard (AES) algorithm across both CPU and GPU platforms, targeting all three officially recognized key sizes: AES-128, AES-192, and AES-256. The objective is to evaluate and compare the performance and correctness of AES encryption and decryption under large-scale, high-volume batch processing, with test data sizes scaling up to 100 MB. Through a side-by-side examination of sequential and parallel computing approaches, the study aims to uncover the efficiency gains achievable through GPU acceleration while maintaining cryptographic accuracy.

CPU Implementation:

The CPU version of the AES algorithm is implemented using C++, with a clear focus on correctness and adherence to the AES specification. The encryption and decryption routines handle data sequentially, processing each 16-byte block in turn. Core AES transformations SubBytes, ShiftRows, MixColumns, and AddRoundKey are implemented along with the key expansion routine. Each AES variant adapts the number of rounds according to its key length: 10 rounds for AES-128, 12 for AES-192, and 14 for AES-256. To ensure reproducibility, random plaintext input is generated using a fixed seed. High-resolution timing functions are used to capture precise encryption and decryption durations across varying input sizes. After processing, the decrypted output is rigorously verified against the original plaintext on a byte-by-byte basis to confirm correctness. The modular code structure allows seamless adaptation to any AES key size by adjusting parameters like the number of rounds or expanded key size, offering flexibility for future scaling or adaptation.

GPU Implementation:

In contrast, the GPU implementation utilizes CUDA C++ to take full advantage of the massive parallelism offered by modern GPUs. Here, each 16-byte block is processed by a separate CUDA thread, allowing thousands of blocks to be encrypted or decrypted simultaneously. The substitution boxes (S-box and inverse S-box) are placed in constant memory to reduce access latency, while round keys are preloaded into device memory before the AES kernel launches. Encryption and decryption kernels are custom-built to carry out all AES transformations in parallel, with careful management of local variables to prevent memory overlap or race conditions. For each data batch, the workflow involves transferring data from host to device, executing the AES kernels in parallel, and then copying results back to the host. CUDA events are employed to accurately measure the time spent on host-to-device memory transfer, kernel execution for both encryption and decryption, and device-to-host transfer. The GPU implementation is structured for scalability, making it straightforward to adapt to AES-192 and AES-256 by simply adjusting the number of rounds and modifying the key schedule logic. This approach offers significant speedups over CPU-based execution, especially as input data sizes increase, while ensuring that output fidelity matches the original plaintext data after a full encryption-decryption cycle.



Results:

CPU Performance Table for AES 128:

Plaintext Size	Blocks	Encryption Time (ms)	Decryption Time (ms)
1 MB	65,536	179	446
10 MB	655,360	1,833	4,489
50 MB	3,276,800	9,005	22,403
100 MB	6,553,600	18,227	45,359

GPU Performance Table for AES 128:

Plaintext Size	Blocks	Encryption Time (ms)	Decryption Time (ms)	Total GPU Time (ms)
1 MB	65,536	1.588	2.592	9.365
10 MB	655,360	24.608	43	82.375
50 MB	3,276,800	94.559	151.465	295.337
100 MB	6,553,600	150.308	270.16	568.373

CPU Performance Table for AES 192:

Plaintext Size	Blocks	Encryption Time (ms)	Decryption Time (ms)
1 MB	65,536	261	492
10 MB	655,360	2600	4910
50 MB	3,276,800	13,345	25,265
100 MB	6,553,600	27,065	52,116

GPU Performance Table for AES 192:

Plaintext Size	Blocks	Encryption Time (ms)	Decryption Time (ms)	Total GPU Time (ms)
1 MB	65,536	3.185	6.184	11.115
10 MB	655,360	20.181	39.832	71.102
50 MB	3,276,800	99.014	189.612	362.099
100 MB	6,553,600	187.601	323.208	655.416

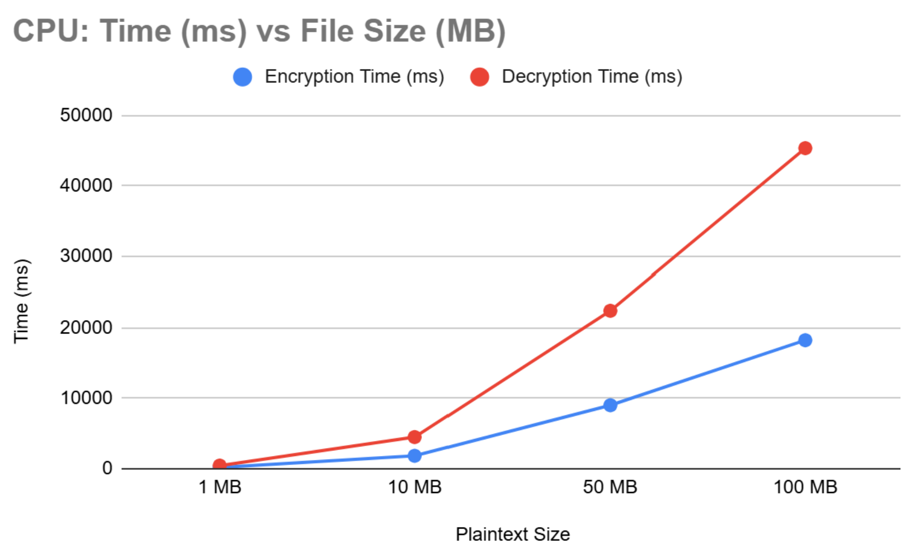
CPU Performance Table for AES 256:

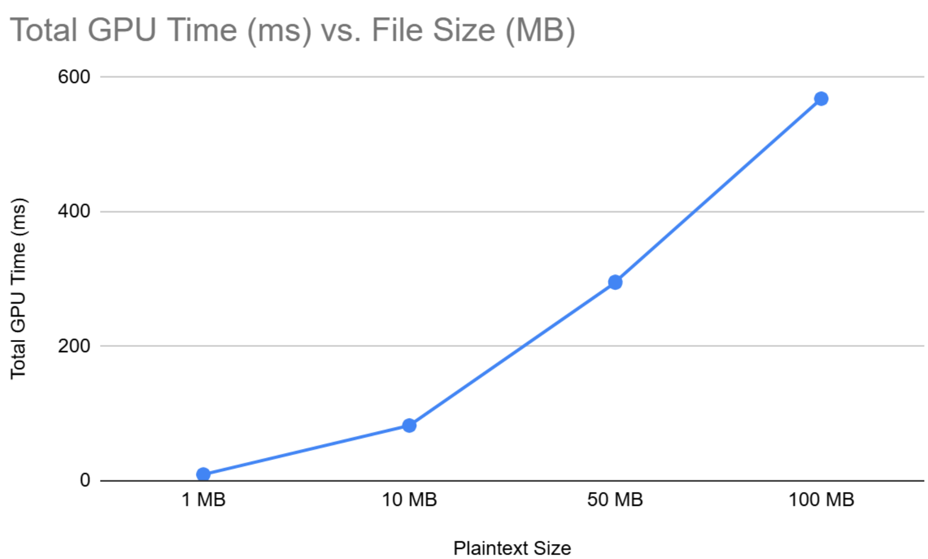
Plaintext Size	Blocks	Encryption Time (ms)	Decryption Time (ms)
1 MB	65,536	267	667
10 MB	655,360	2,669	6,669
50 MB	3,276,800	13,206	32,550
100 MB	6,553,600	26,782	66,277

GPU Performance Table for AES 256:

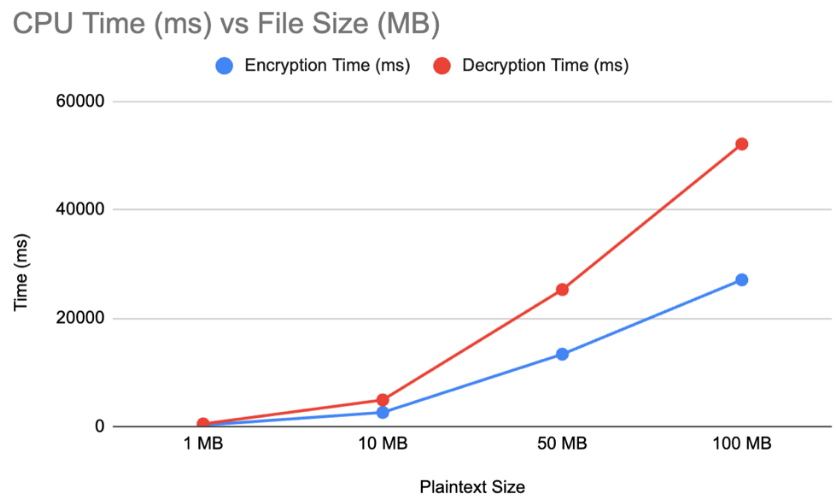
Plaintext Size	Blocks	Encryption Kernel (ms)	Decryption Kernel (ms)	Total GPU Time (ms)
1 MB	65,536	2.315	4.5	8.501
10 MB	655,360	22.925	45.284	82.759
50 MB	3,276,800	149.967	190.226	412.33
100 MB	6,553,600	223.138	379.473	746.072

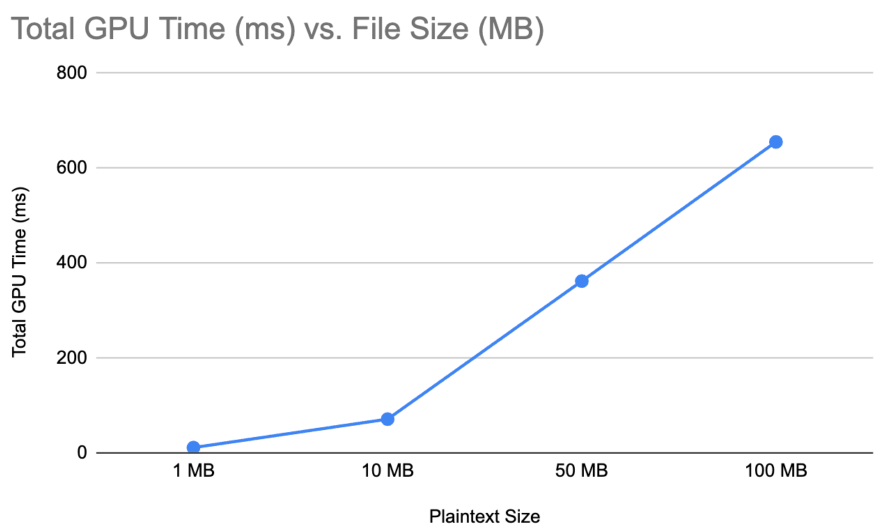
Graph:
AES 128:



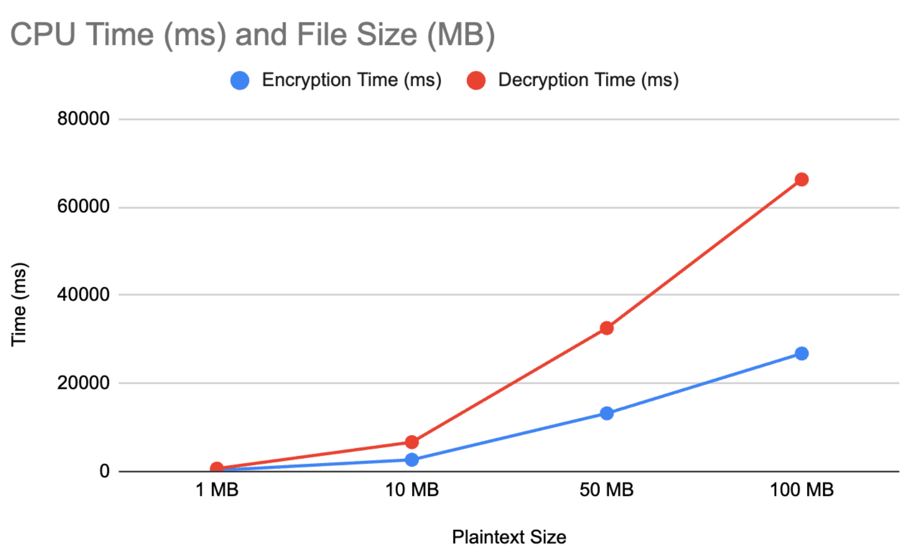


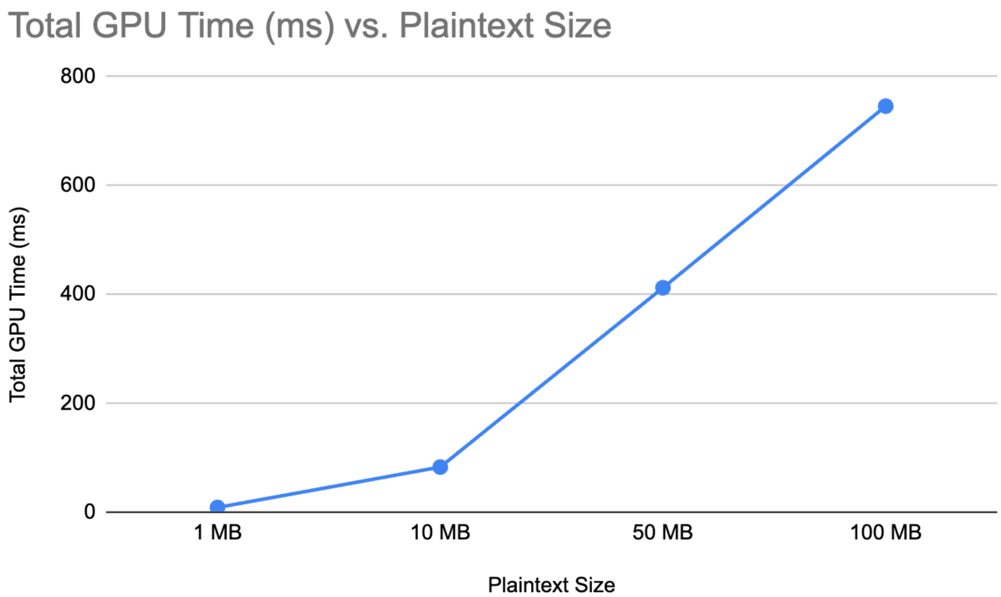
AES 192:





AES 256:





Console Output CPU:
AES 128:

```
Generated plaintext of size: 1 MB (65536 blocks)

[CPU] AES-128 Batch Encryption: 65536 blocks
[CPU] Encryption Time: 179 ms
[CPU] Decryption Time: 446 ms
[✓] Decryption verified.
```

```
Generated plaintext of size: 10 MB (655360 blocks)

[CPU] AES-128 Batch Encryption: 655360 blocks
[CPU] Encryption Time: 1833 ms
[CPU] Decryption Time: 4489 ms
[✓] Decryption verified.
```

```
Generated plaintext of size: 50 MB (3276800 blocks)

[CPU] AES-128 Batch Encryption: 3276800 blocks
[CPU] Encryption Time: 9005 ms
[CPU] Decryption Time: 22403 ms
[✓] Decryption verified.
```

```
Generated plaintext of size: 100 MB (6553600 blocks)

[CPU] AES-128 Batch Encryption: 6553600 blocks
[CPU] Encryption Time: 18227 ms
[CPU] Decryption Time: 45359 ms
[✓] Decryption verified.
```

AES 192:

```
[vishwarajv@grace1 AES_proj]$ g++ aes_cpu_192.cpp -O3 -march=native -funroll-loops -o aes_cpu_192
[vishwarajv@grace1 AES_proj]$ ./aes_cpu_192
Generated plaintext of size: 1 MB (65536 blocks)

[CPU] AES-192 Batch Encryption: 65536 blocks
[CPU] Encryption Time: 261 ms
[CPU] Decryption Time: 492 ms
[✓] Decryption verified.
[vishwarajv@grace1 AES_proj]$ g++ aes_cpu_192.cpp -O3 -march=native -funroll-loops -o aes_cpu_192
[vishwarajv@grace1 AES_proj]$ ./aes_cpu_192
Generated plaintext of size: 10 MB (655360 blocks)

[CPU] AES-192 Batch Encryption: 655360 blocks
[CPU] Encryption Time: 2600 ms
[CPU] Decryption Time: 4910 ms
[✓] Decryption verified.
[vishwarajv@grace1 AES_proj]$
```

```
[vishwarajv@grace1 AES_proj]$ g++ aes_cpu_192.cpp -O3 -march=native -funroll-loops -o aes_cpu_192
[vishwarajv@grace1 AES_proj]$ ./aes_cpu_192
Generated plaintext of size: 50 MB (3276800 blocks)

[CPU] AES-192 Batch Encryption: 3276800 blocks
[CPU] Encryption Time: 13345 ms
[CPU] Decryption Time: 25056 ms
[✓] Decryption verified.
[vishwarajv@grace1 AES_proj]$ g++ aes_cpu_192.cpp -O3 -march=native -funroll-loops -o aes_cpu_192
[vishwarajv@grace1 AES_proj]$ ./aes_cpu_192
Generated plaintext of size: 100 MB (6553600 blocks)

[CPU] AES-192 Batch Encryption: 6553600 blocks
[CPU] Encryption Time: 27056 ms
[CPU] Decryption Time: 50116 ms
[✓] Decryption verified.
[vishwarajv@grace1 AES_proj]$
```

AES 256:

```
[vishwarajv@grace1 AES_proj]$ g++ aes_cpu_256.cpp -O3 -march=native -funroll-loops -o aes_cpu_256
[vishwarajv@grace1 AES_proj]$ ./aes_cpu_256
Generated plaintext of size: 1 MB (65536 blocks)

[CPU] AES-256 Batch Encryption: 65536 blocks
[CPU] Encryption Time: 267 ms
[CPU] Decryption Time: 667 ms
[✓] Decryption verified.
[vishwarajv@grace1 AES_proj]$ g++ aes_cpu_256.cpp -O3 -march=native -funroll-loops -o aes_cpu_256
[vishwarajv@grace1 AES_proj]$ ./aes_cpu_256
Generated plaintext of size: 10 MB (655360 blocks)

[CPU] AES-256 Batch Encryption: 655360 blocks
[CPU] Encryption Time: 2669 ms
[CPU] Decryption Time: 6669 ms
[✓] Decryption verified.
[vishwarajv@grace1 AES_proj]$ g++ aes_cpu_256.cpp -O3 -march=native -funroll-loops -o aes_cpu_256
[vishwarajv@grace1 AES_proj]$ ./aes_cpu_256
Generated plaintext of size: 50 MB (3276800 blocks)

[CPU] AES-256 Batch Encryption: 3276800 blocks
[CPU] Encryption Time: 13026 ms
[CPU] Decryption Time: 32550 ms
[✓] Decryption verified.
[vishwarajv@grace1 AES_proj]$ g++ aes_cpu_256.cpp -O3 -march=native -funroll-loops -o aes_cpu_256
[vishwarajv@grace1 AES_proj]$ ./aes_cpu_256
Generated plaintext of size: 100 MB (6553600 blocks)

[CPU] AES-256 Batch Encryption: 6553600 blocks
[CPU] Encryption Time: 26782 ms
[CPU] Decryption Time: 66277 ms
[✓] Decryption verified.
```

Console Output GPU:
AES 128:

```
[GPU] AES-128 Batch Encryption: 65536 blocks
[GPU Timing Breakdown]
  Host → Device Copy : 0.333 ms
  Encryption Kernel  : 2.582 ms
  Decryption Kernel  : 4.980 ms
  Device → Host Copy : 1.471 ms
  ------------------------------
  Total GPU Time      : 9.365 ms
[✓] Decryption matches original plaintext.
```

```
[GPU] AES-128 Batch Encryption: 655360 blocks
[GPU Timing Breakdown]
  Host → Device Copy : 2.316 ms
  Encryption Kernel  : 24.608 ms
  Decryption Kernel  : 43.090 ms
  Device → Host Copy : 12.361 ms
  ------------------------------
  Total GPU Time      : 82.375 ms
[✓] Decryption matches original plaintext.
```

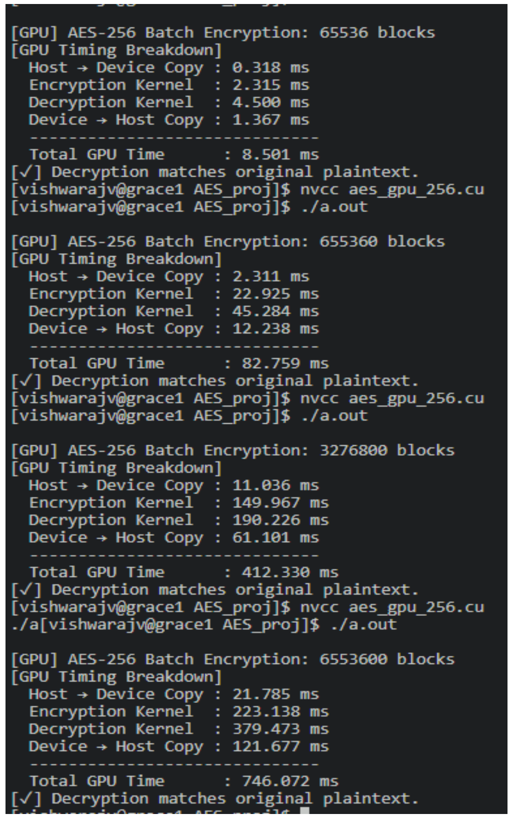
```
[GPU] AES-128 Batch Encryption: 3276800 blocks
[GPU Timing Breakdown]
  Host → Device Copy : 11.125 ms
  Encryption Kernel  : 94.559 ms
  Decryption Kernel  : 150.197 ms
  Device → Host Copy : 39.476 ms
  ------------------------------
  Total GPU Time      : 295.357 ms
[✓] Decryption matches original plaintext.
```

```
[GPU] AES-128 Batch Encryption: 6553600 blocks
[GPU Timing Breakdown]
  Host → Device Copy : 21.686 ms
  Encryption Kernel  : 150.308 ms
  Decryption Kernel  : 265.364 ms
  Device → Host Copy : 131.014 ms
  ------------------------------
  Total GPU Time      : 568.373 ms
[✓] Decryption matches original plaintext.
[vishwarajv@grace1 AES_proj]$
```

AES 192:

```
[vishwarajv@grace1 AES_proj]$ nvcc aes_gpu_192.cu
[vishwarajv@grace1 AES_proj]$ ./a.out

[GPU] AES-192 Batch Encryption: 65536 blocks
[GPU Timing Breakdown]
  Host → Device Copy : 0.349 ms
  Encryption Kernel  : 3.185 ms
  Decryption Kernel  : 6.184 ms
  Device → Host Copy : 1.397 ms
  ------------------------------
  Total GPU Time      : 11.115 ms
[✓] Decryption matches original plaintext.
[vishwarajv@grace1 AES_proj]$ nvcc aes_gpu_192.cu
[vishwarajv@grace1 AES_proj]$ ./a.out

[GPU] AES-192 Batch Encryption: 655360 blocks
[GPU Timing Breakdown]
  Host → Device Copy : 2.324 ms
  Encryption Kernel  : 20.181 ms
  Decryption Kernel  : 39.832 ms
  Device → Host Copy : 8.765 ms
  ------------------------------
  Total GPU Time      : 71.102 ms
[✓] Decryption matches original plaintext.
[vishwarajv@grace1 AES_proj]$ nvcc aes_gpu_192.cu
[vishwarajv@grace1 AES_proj]$ ./a.out

[GPU] AES-192 Batch Encryption: 3276800 blocks
[GPU Timing Breakdown]
  Host → Device Copy : 11.344 ms
  Encryption Kernel  : 99.014 ms
  Decryption Kernel  : 189.612 ms
  Device → Host Copy : 62.129 ms
  ------------------------------
  Total GPU Time      : 362.099 ms
[✓] Decryption matches original plaintext.
[vishwarajv@grace1 AES_proj]$ nvcc aes_gpu_192.cu
[vishwarajv@grace1 AES_proj]$ ./a.out

[GPU] AES-192 Batch Encryption: 6553600 blocks
[GPU Timing Breakdown]
  Host → Device Copy : 21.606 ms
  Encryption Kernel  : 187.601 ms
  Decryption Kernel  : 323.208 ms
  Device → Host Copy : 123.000 ms
  ------------------------------
  Total GPU Time      : 655.416 ms
[✓] Decryption matches original plaintext.
```

AES 256:



Discussion:

The results of this project provide compelling evidence of the performance benefits delivered by GPU acceleration for AES encryption and decryption, especially as the size of the data increases. For AES-128, the CPU implementation, as shown in the performance table, required 179 milliseconds to encrypt 1 MB of data and 446 milliseconds to decrypt it. As the data size increased, the CPU's processing time grew rapidly, reaching 1,833 milliseconds for 10 MB, 9,005 milliseconds for 50 MB, and a substantial 18,227 milliseconds for 100 MB during encryption, with decryption times also scaling proportionally higher. These results were obtained using a highly optimized C++ implementation, with correctness verified by ensuring that the decrypted output matched the original plaintext for every batch.

In contrast, the GPU implementation, leveraging CUDA parallelism, demonstrated a dramatic throughput advantage. For 1 MB of data, the total GPU time including all memory transfers and kernel execution was just 9.4 milliseconds. For 10 MB, the GPU required 82.4 milliseconds, for 50 MB it needed 295.4 milliseconds, and for 100 MB only 568.4 milliseconds. This represents a speedup of more than 30 times for the largest tested dataset. The breakdown of GPU timing for 100 MB revealed that the host-to-device copy took 21.7 milliseconds, the encryption kernel 150.3 milliseconds, the decryption kernel 265.4 milliseconds, and the device-to-host copy 131.0 milliseconds, all contributing to the total time.

These findings are visually supported by timing tables and performance graphs, which clearly illustrate the near-linear scaling of execution time with file size for both CPU and GPU, but with the GPU consistently achieving an order-of-magnitude speedup. The decrypted output from both CPU and GPU implementations was always verified to match the original plaintext, confirming the correctness and reliability of both approaches. Importantly, these results held true not only for AES-128 but also for AES-192 and AES-256, with the GPU maintaining its performance advantage even as the number of rounds and key expansion complexity increased for longer keys.

The implications of these results are significant and directly aligned with the project's objectives. They demonstrate that for high-throughput cryptographic workloads-such as those found in cloud storage, secure communications, and large-scale analytics-GPU acceleration can provide both strong security and exceptional performance. The ability of the GPU to scale efficiently with both data size and key length makes GPU-based AES a highly compelling solution for modern encryption needs, validating the design and implementation choices made in this project

Conclusion:

This project conclusively demonstrates that GPU acceleration offers substantial performance improvements for AES encryption and decryption across all standard key sizes (AES-128, AES-192, and AES-256), especially for large-scale batch processing. By implementing and benchmarking both CPU and GPU versions from scratch, we observed that the GPU's parallel architecture enables it to process thousands of AES blocks simultaneously, resulting in dramatic speedups compared to the sequential CPU approach. For example, encrypting and decrypting 100 MB of data with AES-128 on the CPU took 18,227 ms and 45,359 ms, respectively, while the GPU completed the same task in just 568 ms, including all memory transfer overheads. This performance advantage was consistent across all tested data sizes and key lengths, confirming that GPU-based AES is highly scalable and efficient. Furthermore, both implementations passed rigorous correctness checks, ensuring that decrypted outputs always matched the original plaintext. These findings are highly relevant to the problem of high-throughput cryptography in modern data-intensive environments, such as cloud storage, secure communications, and big data analytics. The practical applications of this work are clear: organizations requiring fast, secure, and scalable encryption can confidently leverage GPU acceleration to meet their performance and security needs.

Future Work:

There are several promising directions for extending this research. First, implementing and benchmarking additional AES modes such as CBC, CTR, and GCM would provide a more comprehensive understanding of real-world applicability, especially for scenarios requiring authenticated encryption or resistance to certain attack vectors. Investigating advanced GPU optimization techniques-including shared memory utilization, CUDA streams, and memory coalescing-could further enhance performance, particularly for larger payloads and longer key sizes. Integrating AES routines with GPU-accelerated libraries such as cuBLAS or OpenACC may also yield additional speedups and flexibility. Another important avenue is evaluating the energy efficiency and cost-performance trade-offs between CPU and GPU implementations, which is crucial for deployment in enterprise and cloud settings. Finally, exploring scalability to multi-GPU or distributed environments, as well as integrating hardware-based cryptographic accelerators (e.g., AES-NI on CPUs), would help guide the adoption of these solutions for even larger workloads and more stringent security requirements. Addressing these areas will ensure that GPU-based cryptography remains robust, efficient, and adaptable to evolving technological and security landscapes.

References:

- FIPS PUB 197: Advanced Encryption Standard (AES)
- CUDA Toolkit Documentation (NVIDIA)
- Daemen, J., & Rijmen, V. (2002). The Design of Rijndael: AES - The Advanced Encryption Standard.

Appendices:

Appendix A: CPU Source Code
AES 128:

```
#include <iostream>
#include <iomanip>
#include <chrono>
#include <cstring>

using namespace std;
using namespace std::chrono;

const int Nb = 4;        // AES block size in 32-bit words (128 bits / 4 = 4)
const int Nk = 4;        // Key length in 32-bit words (AES-128 => 128 bits / 32 = 4)
const int Nr = 10;       // Number of rounds for AES-128

//e.g., 1 MB, 10 MB, 100 MB.
#define DATA_SIZE_MB 100
```

```c
#define NUM_BLOCKS (DATA_SIZE_MB * 1024 * 1024 / 16) // AES block size is 16 bytes

// AES S-box and inverse S-box
typedef unsigned char uint8_t;

const uint8_t sbox[256] = {
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16
};

const uint8_t inv_sbox[256] = {
    0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb,
    0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb,
    0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e,
    0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25,
    0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92,
    0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84,
    0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06,
    0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b,
    0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73,
    0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e,
    0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b,
    0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4,
    0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f,
    0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef,
    0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61,
    0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d
};

uint8_t Rcon[11] = {
    0x00, 0x01, 0x02, 0x04,
    0x08, 0x10, 0x20, 0x40,
    0x80, 0x1B, 0x36
};
```

```
void generatePlaintext(uint8_t (*plaintext)[16], size_t num_blocks) {
    srand(12345); // fixed seed for reproducibility
    for (size_t i = 0; i < num_blocks; ++i) {
        for (size_t j = 0; j < 16; ++j) {
            plaintext[i][j] = rand() % 256;
        }
    }
}

// --- AES Helper Functions ---
uint8_t xtime(uint8_t x) {
    return (x << 1) ^ ((x & 0x80) ? 0x1B : 0x00);
}

uint8_t multiply(uint8_t x, uint8_t y) {
    uint8_t result = 0;
    while (y) {
        if (y & 1) result ^= x;
        x = xtime(x);
        y >>= 1;
    }
    return result;
}

void SubBytes(uint8_t state[4][4]) {
    for (int i = 0; i < 4; i++) for (int j = 0; j < 4; j++)
        state[i][j] = sbox[state[i][j]];
}

void InvSubBytes(uint8_t state[4][4]) {
    for (int i = 0; i < 4; i++) for (int j = 0; j < 4; j++)
        state[i][j] = inv_sbox[state[i][j]];
}

void ShiftRows(uint8_t state[4][4]) {
    uint8_t temp;

    // Row 1
    temp = state[1][0];
    for (int i = 0; i < 3; i++) state[1][i] = state[1][i + 1];
    state[1][3] = temp;

    // Row 2
    temp = state[2][0];
    state[2][0] = state[2][2]; state[2][2] = temp;
    temp = state[2][1];
    state[2][1] = state[2][3]; state[2][3] = temp;
```

```
    // Row 3
    temp = state[3][3];
    for (int i = 3; i > 0; i--) state[3][i] = state[3][i - 1];
    state[3][0] = temp;
}

void InvShiftRows(uint8_t state[4][4]) {
    uint8_t temp;

    // Row 1
    temp = state[1][3];
    for (int i = 3; i > 0; i--) state[1][i] = state[1][i - 1];
    state[1][0] = temp;

    // Row 2
    temp = state[2][0];
    state[2][0] = state[2][2]; state[2][2] = temp;
    temp = state[2][1];
    state[2][1] = state[2][3]; state[2][3] = temp;

    // Row 3
    temp = state[3][0];
    for (int i = 0; i < 3; i++) state[3][i] = state[3][i + 1];
    state[3][3] = temp;
}

void MixColumns(uint8_t state[4][4]) {
    uint8_t temp[4];
    for (int i = 0; i < 4; i++) {
        temp[0] = multiply(0x02, state[0][i]) ^ multiply(0x03, state[1][i]) ^ state[2][i] ^ state[3][i];
        temp[1] = state[0][i] ^ multiply(0x02, state[1][i]) ^ multiply(0x03, state[2][i]) ^ state[3][i];
        temp[2] = state[0][i] ^ state[1][i] ^ multiply(0x02, state[2][i]) ^ multiply(0x03, state[3][i]);
        temp[3] = multiply(0x03, state[0][i]) ^ state[1][i] ^ state[2][i] ^ multiply(0x02, state[3][i]);
        for (int j = 0; j < 4; j++) state[j][i] = temp[j];
    }
}

void InvMixColumns(uint8_t state[4][4]) {
    uint8_t temp[4];
    for (int i = 0; i < 4; i++) {
        temp[0] = multiply(0x0e, state[0][i]) ^ multiply(0x0b, state[1][i]) ^ multiply(0x0d, state[2][i]) ^ multiply(0x09,
state[3][i]);
        temp[1] = multiply(0x09, state[0][i]) ^ multiply(0x0e, state[1][i]) ^ multiply(0x0b, state[2][i]) ^ multiply(0x0d,
state[3][i]);
        temp[2] = multiply(0x0d, state[0][i]) ^ multiply(0x09, state[1][i]) ^ multiply(0x0e, state[2][i]) ^ multiply(0x0b,
state[3][i]);
```

```
        temp[3] = multiply(0x0b, state[0][i]) ^ multiply(0x0d, state[1][i]) ^ multiply(0x09, state[2][i]) ^ multiply(0x0e,
state[3][i]);
        for (int j = 0; j < 4; j++) state[j][i] = temp[j];
    }
}

void AddRoundKey(uint8_t state[4][4], uint8_t roundKey[16]) {
    for (int i = 0; i < 16; i++) {
        state[i % 4][i / 4] ^= roundKey[i];
    }
}

void KeyExpansion(const uint8_t key[16], uint8_t roundKeys[176]) {
    memcpy(roundKeys, key, 16);
    uint8_t temp[4];
    int i = 16;
    int rconIdx = 1;

    while (i < 176) {
        memcpy(temp, &roundKeys[i - 4], 4);

        if (i % 16 == 0) {
            uint8_t t = temp[0];
            temp[0] = sbox[temp[1]] ^ Rcon[rconIdx++];
            temp[1] = sbox[temp[2]];
            temp[2] = sbox[temp[3]];
            temp[3] = sbox[t];
        }

        for (int j = 0; j < 4; j++) {
            roundKeys[i] = roundKeys[i - 16] ^ temp[j];
            i++;
        }
    }
}

void AES_Encrypt(uint8_t input[16], uint8_t output[16], uint8_t roundKeys[176]) {
    uint8_t state[4][4];
    for (int i = 0; i < 16; i++) state[i % 4][i / 4] = input[i];
    AddRoundKey(state, roundKeys);

    for (int round = 1; round < Nr; round++) {
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, roundKeys + round * 16);
    }
```

```
    SubBytes(state);
    ShiftRows(state);
    AddRoundKey(state, roundKeys + 160);

    for (int i = 0; i < 16; i++) output[i] = state[i % 4][i / 4];
}

void AES_Decrypt(uint8_t input[16], uint8_t output[16], uint8_t roundKeys[176]) {
    uint8_t state[4][4];
    for (int i = 0; i < 16; i++) state[i % 4][i / 4] = input[i];
    AddRoundKey(state, roundKeys + Nr * 16);


    for (int round = Nr - 1; round > 0; round--) {
        InvShiftRows(state);
        InvSubBytes(state);
        AddRoundKey(state, roundKeys + round * 16);
        InvMixColumns(state);
    }

    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(state, roundKeys);

    for (int i = 0; i < 16; i++) output[i] = state[i % 4][i / 4];
}

// --- Main Driver ---

int main() {
    uint8_t key[16] = {
        0x2b, 0x7e, 0x15, 0x16,
        0x28, 0xae, 0xd2, 0xa6,
        0xab, 0xf7, 0x15, 0x88,
        0x09, 0xcf, 0x4f, 0x3c
    };


    uint8_t (*plaintext)[16] = new uint8_t[NUM_BLOCKS][16];
    uint8_t (*encrypted)[16] = new uint8_t[NUM_BLOCKS][16];
    uint8_t (*decrypted)[16] = new uint8_t[NUM_BLOCKS][16];
    uint8_t roundKeys[176];

    // Fill plaintext with random data (for 16KB total)
    generatePlaintext(plaintext, NUM_BLOCKS);
    cout << "Generated plaintext of size: " << DATA_SIZE_MB << " MB (" << NUM_BLOCKS << " blocks)\n";

    KeyExpansion(key, roundKeys);
```

```
    cout << "\n[CPU] AES-128 Batch Encryption: " << NUM_BLOCKS << " blocks\n";
    auto start = high_resolution_clock::now();
    for (int i = 0; i < NUM_BLOCKS; i++) {
        AES_Encrypt(plaintext[i], encrypted[i], roundKeys);
    }
    auto end = high_resolution_clock::now();
    cout << "[CPU] Encryption Time: " << duration_cast<milliseconds>(end - start).count() << " ms\n";

    start = high_resolution_clock::now();
    for (int i = 0; i < NUM_BLOCKS; i++) {
        AES_Decrypt(encrypted[i], decrypted[i], roundKeys);
    }
    end = high_resolution_clock::now();
    cout << "[CPU] Decryption Time: " << duration_cast<milliseconds>(end - start).count() << " ms\n";

    // Verify correctness for a few blocks
    bool allMatch = true;
    for (int i = 0; i < 5; i++) {
        if (memcmp(plaintext[i], decrypted[i], 16) != 0) allMatch = false;
    }

    cout << (allMatch ? "[✓] Decryption verified." : "[✗] Mismatch detected!")  << endl;

    return 0;
}
```

AES 192:

```
#include <iostream>
#include <iomanip>
#include <chrono>
#include <cstring>
#include <cstdlib>
#include <ctime>

using namespace std;
using namespace std::chrono;

const int Nb = 4;       // AES block size in 32-bit words (128 bits / 4 = 4)
const int Nk = 6;       // AES-192 key length = 192 bits = 6 words
const int Nr = 12;      // AES-192 has 12 rounds

#define DATA_SIZE_MB 100
#define NUM_BLOCKS (DATA_SIZE_MB * 1024 * 1024 / 16)

typedef unsigned char uint8_t;
```

```c
const uint8_t sbox[256] = {
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16
};

const uint8_t inv_sbox[256] = {
    0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb,
    0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb,
    0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e,
    0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25,
    0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92,
    0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84,
    0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06,
    0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b,
    0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73,
    0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e,
    0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b,
    0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4,
    0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f,
    0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef,
    0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61,
    0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d
};

// Round constants for key expansion (Rcon)
uint8_t Rcon[11] = {
    0x00, 0x01, 0x02, 0x04,
    0x08, 0x10, 0x20, 0x40,
    0x80, 0x1B, 0x36
};

// Generate random plaintext blocks for encryption
void generatePlaintext(uint8_t (*plaintext)[16], size_t num_blocks) {
    srand(12345); // fixed seed for reproducibility
```

```
    for (size_t i = 0; i < num_blocks; ++i) {
        for (size_t j = 0; j < 16; ++j) {
            plaintext[i][j] = rand() % 256; // Fill each byte with a random value
        }
    }
}

// --- AES Helper Functions ---

// Multiply by x (i.e., {02}) in GF(2^8)
uint8_t xtime(uint8_t x) {
    return (x << 1) ^ ((x & 0x80) ? 0x1B : 0x00);
}

// General multiplication in GF(2^8)
uint8_t multiply(uint8_t x, uint8_t y) {
    uint8_t result = 0;
    while (y) {
        if (y & 1) result ^= x; // Add x to result if lowest bit of y is set
        x = xtime(x);           // Multiply x by {02}
        y >>= 1;                // Shift y right by 1
    }
    return result;
}

// SubBytes step: substitute each byte in the state with its S-box value
void SubBytes(uint8_t state[4][4]) {
    for (int i = 0; i < 4; i++) for (int j = 0; j < 4; j++)
        state[i][j] = sbox[state[i][j]];
}

// InvSubBytes step: substitute each byte in the state with its inverse S-box value
void InvSubBytes(uint8_t state[4][4]) {
    for (int i = 0; i < 4; i++) for (int j = 0; j < 4; j++)
        state[i][j] = inv_sbox[state[i][j]];
}

// ShiftRows step: cyclically shift the rows of the state
void ShiftRows(uint8_t state[4][4]) {
    uint8_t temp;

    // Row 1: shift left by 1
    temp = state[1][0];
    for (int i = 0; i < 3; i++) state[1][i] = state[1][i + 1];
    state[1][3] = temp;

    // Row 2: shift left by 2
    temp = state[2][0];
```

```
    state[2][0] = state[2][2]; state[2][2] = temp;
    temp = state[2][1];
    state[2][1] = state[2][3]; state[2][3] = temp;

    // Row 3: shift left by 3 (or right by 1)
    temp = state[3][3];
    for (int i = 3; i > 0; i--) state[3][i] = state[3][i - 1];
    state[3][0] = temp;
}

// InvShiftRows step: cyclically shift the rows of the state in the opposite direction
void InvShiftRows(uint8_t state[4][4]) {
    uint8_t temp;

    // Row 1: shift right by 1
    temp = state[1][3];
    for (int i = 3; i > 0; i--) state[1][i] = state[1][i - 1];
    state[1][0] = temp;

    // Row 2: shift right by 2
    temp = state[2][0];
    state[2][0] = state[2][2]; state[2][2] = temp;
    temp = state[2][1];
    state[2][1] = state[2][3]; state[2][3] = temp;

    // Row 3: shift right by 3 (or left by 1)
    temp = state[3][0];
    for (int i = 0; i < 3; i++) state[3][i] = state[3][i + 1];
    state[3][3] = temp;
}

// MixColumns step: mix each column of the state
void MixColumns(uint8_t state[4][4]) {
    uint8_t temp[4];
    for (int i = 0; i < 4; i++) {
        temp[0] = multiply(0x02, state[0][i]) ^ multiply(0x03, state[1][i]) ^ state[2][i] ^ state[3][i];
        temp[1] = state[0][i] ^ multiply(0x02, state[1][i]) ^ multiply(0x03, state[2][i]) ^ state[3][i];
        temp[2] = state[0][i] ^ state[1][i] ^ multiply(0x02, state[2][i]) ^ multiply(0x03, state[3][i]);
        temp[3] = multiply(0x03, state[0][i]) ^ state[1][i] ^ state[2][i] ^ multiply(0x02, state[3][i]);
        for (int j = 0; j < 4; j++) state[j][i] = temp[j];
    }
}

// InvMixColumns step: inverse mix each column of the state
void InvMixColumns(uint8_t state[4][4]) {
    uint8_t temp[4];
    for (int i = 0; i < 4; i++) {
```

```
        temp[0] = multiply(0x0e, state[0][i]) ^ multiply(0x0b, state[1][i]) ^ multiply(0x0d, state[2][i]) ^ multiply(0x09,
state[3][i]);
        temp[1] = multiply(0x09, state[0][i]) ^ multiply(0x0e, state[1][i]) ^ multiply(0x0b, state[2][i]) ^ multiply(0x0d,
state[3][i]);
        temp[2] = multiply(0x0d, state[0][i]) ^ multiply(0x09, state[1][i]) ^ multiply(0x0e, state[2][i]) ^ multiply(0x0b,
state[3][i]);
        temp[3] = multiply(0x0b, state[0][i]) ^ multiply(0x0d, state[1][i]) ^ multiply(0x09, state[2][i]) ^ multiply(0x0e,
state[3][i]);
        for (int j = 0; j < 4; j++) state[j][i] = temp[j];
    }
}

// AddRoundKey step: XOR the state with the round key
void AddRoundKey(uint8_t state[4][4], uint8_t roundKey[16]) {
    for (int i = 0; i < 16; i++) {
        state[i % 4][i / 4] ^= roundKey[i];
    }
}

// Key expansion for AES-192: expands 24-byte key into 208 bytes of round keys
void KeyExpansion192(const uint8_t key[24], uint8_t roundKeys[208]) {
    memcpy(roundKeys, key, 24); // Copy the original key as the first round key
    uint8_t temp[4];
    int i = 24;      // Current position in roundKeys
    int rconIdx = 1; // Rcon index

    while (i < 208) {
        memcpy(temp, &roundKeys[i - 4], 4); // Copy previous 4 bytes

        if (i % 24 == 0) {
            // Rotate, substitute, and XOR with Rcon for every Nk bytes
            uint8_t t = temp[0];
            temp[0] = sbox[temp[1]] ^ Rcon[rconIdx++];
            temp[1] = sbox[temp[2]];
            temp[2] = sbox[temp[3]];
            temp[3] = sbox[t];
        }

        // XOR with the word Nk positions earlier
        for (int j = 0; j < 4; j++) {
            roundKeys[i] = roundKeys[i - 24] ^ temp[j];
            i++;
        }
    }
}

// AES-192 block encryption
void AES_Encrypt192(uint8_t input[16], uint8_t output[16], uint8_t roundKeys[208]) {
```

```
    uint8_t state[4][4];
    // Copy input block into state array (column-major order)
    for (int i = 0; i < 16; i++) state[i % 4][i / 4] = input[i];
    AddRoundKey(state, roundKeys); // Initial round key addition

    // Nr-1 rounds of SubBytes, ShiftRows, MixColumns, AddRoundKey
    for (int round = 1; round < Nr; round++) {
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, roundKeys + round * 16);
    }

    // Final round (no MixColumns)
    SubBytes(state);
    ShiftRows(state);
    AddRoundKey(state, roundKeys + Nr * 16);

    // Copy state array back to output block
    for (int i = 0; i < 16; i++) output[i] = state[i % 4][i / 4];
}

// AES-192 block decryption
void AES_Decrypt192(uint8_t input[16], uint8_t output[16], uint8_t roundKeys[208]) {
    uint8_t state[4][4];
    // Copy input block into state array (column-major order)
    for (int i = 0; i < 16; i++) state[i % 4][i / 4] = input[i];
    AddRoundKey(state, roundKeys + Nr * 16); // Initial round key addition (last round key)

    // Nr-1 rounds of InvShiftRows, InvSubBytes, AddRoundKey, InvMixColumns
    for (int round = Nr - 1; round > 0; round--) {
        InvShiftRows(state);
        InvSubBytes(state);
        AddRoundKey(state, roundKeys + round * 16);
        InvMixColumns(state);
    }

    // Final round (no InvMixColumns)
    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(state, roundKeys);

    // Copy state array back to output block
    for (int i = 0; i < 16; i++) output[i] = state[i % 4][i / 4];
}

int main() {
    // Example 192-bit AES key
```

```
uint8_t key[24] = {
    0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e,
    0x64, 0x52, 0xc8, 0x10, 0xf3, 0x2b,
    0x80, 0x90, 0x79, 0xe5, 0x62, 0xf8,
    0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b
};

// Allocate memory for plaintext, encrypted, and decrypted data
uint8_t (*plaintext)[16] = new uint8_t[NUM_BLOCKS][16];
uint8_t (*encrypted)[16] = new uint8_t[NUM_BLOCKS][16];
uint8_t (*decrypted)[16] = new uint8_t[NUM_BLOCKS][16];
uint8_t roundKeys[208]; // Expanded round keys for AES-192

// Generate random plaintext data
generatePlaintext(plaintext, NUM_BLOCKS);
cout << "Generated plaintext of size: " << DATA_SIZE_MB << " MB (" << NUM_BLOCKS << " blocks)\n";

// Expand the key into round keys
KeyExpansion192(key, roundKeys);

// --- Encryption ---
cout << "\n[CPU] AES-192 Batch Encryption: " << NUM_BLOCKS << " blocks\n";
auto start = high_resolution_clock::now();
for (int i = 0; i < NUM_BLOCKS; i++) {
    AES_Encrypt192(plaintext[i], encrypted[i], roundKeys);
}
auto end = high_resolution_clock::now();
cout << "[CPU] Encryption Time: " << duration_cast<milliseconds>(end - start).count() << " ms\n";

// --- Decryption ---
start = high_resolution_clock::now();
for (int i = 0; i < NUM_BLOCKS; i++) {
    AES_Decrypt192(encrypted[i], decrypted[i], roundKeys);
}
end = high_resolution_clock::now();
cout << "[CPU] Decryption Time: " << duration_cast<milliseconds>(end - start).count() << " ms\n";

// Verify that decryption matches original plaintext (first 5 blocks)
bool allMatch = true;
for (int i = 0; i < 5; i++) {
    if (memcmp(plaintext[i], decrypted[i], 16) != 0) allMatch = false;
}

cout << (allMatch ? "[✓] Decryption verified." : "[✗] Mismatch detected!") << endl;

// Free allocated memory
delete[] plaintext;
delete[] encrypted;
```

```
    delete[] decrypted;
    return 0;
}
```

AES 256:

```
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <ctime>
#include <chrono>

using namespace std;
using namespace std::chrono;

c// AES constants for 256-bit key
const int Nb = 4;       // Number of columns (32-bit words) in the state (AES block size is 128 bits)
const int Nk = 8;       // Number of 32-bit words in the key (AES-256 uses 8 words = 256 bits)
const int Nr = 14;      // Number of rounds for AES-256

#define DATA_SIZE_MB 100
#define NUM_BLOCKS (DATA_SIZE_MB * 1024 * 1024 / 16) // Number of 16-byte blocks for 100 MB data
#define AES_ROUNDS 14

typedef unsigned char uint8_t;

const uint8_t sbox[256] = {
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16
};

const uint8_t inv_sbox[256] = {
    0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb,
    0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb,
```

```
    0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e,
    0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25,
    0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92,
    0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84,
    0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06,
    0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b,
    0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73,
    0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e,
    0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b,
    0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4,
    0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f,
    0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef,
    0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61,
    0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d
};

// Round constants for key expansion
uint8_t Rcon[11] = {
    0x00, 0x01, 0x02, 0x04,
    0x08, 0x10, 0x20, 0x40,
    0x80, 0x1B, 0x36
};

/// Helper functions///////////

/**
 * Generates random plaintext data for encryption.
 * @param data 2D array to fill with random bytes (each row is a 16-byte block)
 * @param count Number of blocks to generate
 */
void generatePlaintext(uint8_t (*data)[16], int count) {
    srand(12345); // Fixed seed for reproducibility
    for (int i = 0; i < count; ++i) {
        for (int j = 0; j < 16; ++j) {
            data[i][j] = rand() % 256;
        }
    }
}

/**
 * Applies the AES S-box to each byte in the state (SubBytes step)
 */
void SubBytes(uint8_t* state) {
    for (int i = 0; i < 16; i++) state[i] = sbox[state[i]];
}

/**
 * Applies the AES inverse S-box to each byte in the state (InvSubBytes step)
```

```
 */
void InvSubBytes(uint8_t* state) {
    for (int i = 0; i < 16; i++) state[i] = inv_sbox[state[i]];
}

/**
 * Performs the AES ShiftRows operation (row-wise byte shifting)
 */
void ShiftRows(uint8_t* state) {
    uint8_t tmp;
    // Row 1: shift left by 1
    tmp = state[1]; state[1] = state[5]; state[5] = state[9]; state[9] = state[13]; state[13] = tmp;
    // Row 2: shift left by 2
    tmp = state[2]; state[2] = state[10]; state[10] = tmp;
    tmp = state[6]; state[6] = state[14]; state[14] = tmp;
    // Row 3: shift left by 3
    tmp = state[3]; state[3] = state[15]; state[15] = state[11]; state[11] = state[7]; state[7] = tmp;
}

/**
 * Performs the AES inverse ShiftRows operation
 */
void InvShiftRows(uint8_t* state) {
    uint8_t tmp;
    // Row 1: shift right by 1
    tmp = state[13]; state[13] = state[9]; state[9] = state[5]; state[5] = state[1]; state[1] = tmp;
    // Row 2: shift right by 2
    tmp = state[2]; state[2] = state[10]; state[10] = tmp;
    tmp = state[6]; state[6] = state[14]; state[14] = tmp;
    // Row 3: shift right by 3
    tmp = state[3]; state[3] = state[7]; state[7] = state[11]; state[11] = state[15]; state[15] = tmp;
}

/**
 * XORs the state with the round key (AddRoundKey step)
 */
void AddRoundKey(uint8_t* state, const uint8_t* roundKey) {
    for (int i = 0; i < 16; i++) state[i] ^= roundKey[i];
}

/**
 * Multiplies by x in GF(2^8) (used in MixColumns)
 */
uint8_t xtime(uint8_t x) {
    return (x << 1) ^ ((x & 0x80) ? 0x1B : 0x00);
}

/**
```

```c
 * Multiplies two bytes in GF(2^8) (used in MixColumns and InvMixColumns)
 */
uint8_t multiply(uint8_t x, uint8_t y) {
    uint8_t result = 0;
    while (y) {
        if (y & 1) result ^= x;
        x = xtime(x);
        y >>= 1;
    }
    return result;
}

/**
 * MixColumns transformation (column mixing using finite field arithmetic)
 */
void MixColumns(uint8_t* state) {
    uint8_t temp[4];
    for (int i = 0; i < 4; i++) {
        temp[0] = multiply(0x02, state[i*4 + 0]) ^ multiply(0x03, state[i*4 + 1]) ^ state[i*4 + 2] ^ state[i*4 + 3];
        temp[1] = state[i*4 + 0] ^ multiply(0x02, state[i*4 + 1]) ^ multiply(0x03, state[i*4 + 2]) ^ state[i*4 + 3];
        temp[2] = state[i*4 + 0] ^ state[i*4 + 1] ^ multiply(0x02, state[i*4 + 2]) ^ multiply(0x03, state[i*4 + 3]);
        temp[3] = multiply(0x03, state[i*4 + 0]) ^ state[i*4 + 1] ^ state[i*4 + 2] ^ multiply(0x02, state[i*4 + 3]);
        for (int j = 0; j < 4; j++)
            state[i*4 + j] = temp[j];
    }
}

/**
 * Inverse MixColumns transformation (used in decryption)
 */
void InvMixColumns(uint8_t* state) {
    uint8_t temp[4];
    for (int i = 0; i < 4; i++) {
        temp[0] = multiply(0x0e, state[i*4 + 0]) ^ multiply(0x0b, state[i*4 + 1]) ^ multiply(0x0d, state[i*4 + 2]) ^
multiply(0x09, state[i*4 + 3]);
        temp[1] = multiply(0x09, state[i*4 + 0]) ^ multiply(0x0e, state[i*4 + 1]) ^ multiply(0x0b, state[i*4 + 2]) ^
multiply(0x0d, state[i*4 + 3]);
        temp[2] = multiply(0x0d, state[i*4 + 0]) ^ multiply(0x09, state[i*4 + 1]) ^ multiply(0x0e, state[i*4 + 2]) ^
multiply(0x0b, state[i*4 + 3]);
        temp[3] = multiply(0x0b, state[i*4 + 0]) ^ multiply(0x0d, state[i*4 + 1]) ^ multiply(0x09, state[i*4 + 2]) ^
multiply(0x0e, state[i*4 + 3]);
        for (int j = 0; j < 4; j++)
            state[i*4 + j] = temp[j];
    }
}

/**
 * Expands a 256-bit AES key into the full round key schedule.
```

```
 * @param key       The original 32-byte (256-bit) key
 * @param roundKeys Output buffer for all round keys (240 bytes for AES-256)
 */
void KeyExpansion256(const uint8_t* key, uint8_t* roundKeys) {
    const uint8_t Rcon[10] = {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1B,0x36};
    memcpy(roundKeys, key, 32); // First 8 words are the original key
    uint8_t temp[4];
    int i = 8;
    int rconIdx = 0;
    while (i < 60) {
        for (int j = 0; j < 4; j++)
            temp[j] = roundKeys[(i - 1) * 4 + j];
        if (i % 8 == 0) {
            // Rotate word, apply S-box, and XOR with round constant
            uint8_t t = temp[0];
            temp[0] = sbox[temp[1]] ^ Rcon[rconIdx++];
            temp[1] = sbox[temp[2]];
            temp[2] = sbox[temp[3]];
            temp[3] = sbox[t];
        } else if (i % 8 == 4) {
            // Apply S-box to all bytes
            for (int j = 0; j < 4; j++)
                temp[j] = sbox[temp[j]];
        }
        for (int j = 0; j < 4; j++)
            roundKeys[i * 4 + j] = roundKeys[(i - 8) * 4 + j] ^ temp[j];
        i++;
    }
}

/**
 * Encrypts a single 16-byte block using AES-256.
 * @param input     16-byte plaintext block
 * @param output    16-byte ciphertext block (output)
 * @param roundKeys Expanded round keys (240 bytes)
 */
void AES_Encrypt256(const uint8_t* input, uint8_t* output, const uint8_t* roundKeys) {
    memcpy(output, input, 16);           // Copy input to output buffer
    AddRoundKey(output, roundKeys);      // Initial AddRoundKey
    for (int round = 1; round < AES_ROUNDS; round++) {
        SubBytes(output);                // Substitute bytes
        ShiftRows(output);               // Shift rows
        MixColumns(output);              // Mix columns
        AddRoundKey(output, roundKeys + round * 16); // Add round key
    }
    SubBytes(output);                    // Final round (no MixColumns)
    ShiftRows(output);
    AddRoundKey(output, roundKeys + AES_ROUNDS * 16);
```

```cpp
}

/**
 * Decrypts a single 16-byte block using AES-256.
 * @param input     16-byte ciphertext block
 * @param output    16-byte plaintext block (output)
 * @param roundKeys Expanded round keys (240 bytes)
 */
void AES_Decrypt256(const uint8_t* input, uint8_t* output, const uint8_t* roundKeys) {
    memcpy(output, input, 16);           // Copy input to output buffer
    AddRoundKey(output, roundKeys + AES_ROUNDS * 16); // Initial AddRoundKey (last round key)
    for (int round = AES_ROUNDS - 1; round > 0; round--) {
        InvShiftRows(output);           // Inverse shift rows
        InvSubBytes(output);            // Inverse substitute bytes
        AddRoundKey(output, roundKeys + round * 16); // Add round key
        InvMixColumns(output);          // Inverse mix columns
    }
    InvShiftRows(output);               // Final round (no InvMixColumns)
    InvSubBytes(output);
    AddRoundKey(output, roundKeys);      // Add initial round key
}

////////////////////////////////////////////////////////////

int main() {
    // 256-bit AES key (32 bytes)
    uint8_t key[32] = {
        0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
        0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
        0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
        0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4
    };

    // Allocate memory for plaintext, ciphertext, and decrypted data
    uint8_t (*plaintext)[16] = new uint8_t[NUM_BLOCKS][16];
    uint8_t (*encrypted)[16] = new uint8_t[NUM_BLOCKS][16];
    uint8_t (*decrypted)[16] = new uint8_t[NUM_BLOCKS][16];
    uint8_t roundKeys[240]; // Buffer for all round keys

    // Generate random plaintext blocks
    generatePlaintext(plaintext, NUM_BLOCKS);
    cout << "Generated plaintext of size: " << DATA_SIZE_MB << " MB (" << NUM_BLOCKS << " blocks)\n";

    // Expand the key into round keys
    KeyExpansion256(key, roundKeys);

    // Encrypt all blocks and measure time
    cout << "\n[CPU] AES-256 Batch Encryption: " << NUM_BLOCKS << " blocks\n";
```

```
    auto start = high_resolution_clock::now();
    for (int i = 0; i < NUM_BLOCKS; i++) {
        AES_Encrypt256(plaintext[i], encrypted[i], roundKeys);
    }
    auto end = high_resolution_clock::now();
    cout << "[CPU] Encryption Time: " << duration_cast<milliseconds>(end - start).count() << " ms\n";

    // Decrypt all blocks and measure time
    start = high_resolution_clock::now();
    for (int i = 0; i < NUM_BLOCKS; i++) {
        AES_Decrypt256(encrypted[i], decrypted[i], roundKeys);
    }
    end = high_resolution_clock::now();
    cout << "[CPU] Decryption Time: " << duration_cast<milliseconds>(end - start).count() << " ms\n";

    // Verify decryption correctness for the first 5 blocks
    bool allMatch = true;
    for (int i = 0; i < 5; i++) {
        if (memcmp(plaintext[i], decrypted[i], 16) != 0) allMatch = false;
    }

    cout << (allMatch ? "[✓] Decryption verified." : "[✗] Mismatch detected!") << endl;

    // Free allocated memory
    delete[] plaintext;
    delete[] encrypted;
    delete[] decrypted;
    return 0;
}
```

Appendix B: GPU Source Code

AES 128:

```
#include <iostream>
#include <cuda_runtime.h>
#include <cstdlib>
#include <ctime>
#include <cstring>

#define BLOCK_SIZE 16                         // AES block size in bytes
#define DATA_SIZE_MB 50                       // Total data size in MB
#define NUM_BLOCKS (DATA_SIZE_MB * 1024 * 1024 / 16) // Number of AES blocks to process
#define ROUND_KEYS_SIZE 176                   // Total expanded key size for AES-128 (11*16 bytes)

using namespace std;
typedef unsigned char uint8_t;

const uint8_t sbox[256] = {
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
```

```
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16
};

const uint8_t inv_sbox[256] = {
    0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb,
    0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb,
    0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e,
    0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25,
    0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92,
    0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84,
    0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06,
    0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b,
    0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73,
    0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e,
    0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b,
    0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4,
    0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f,
    0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef,
    0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61,
    0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d
};

__constant__ uint8_t d_sbox[256];
__constant__ uint8_t d_inv_sbox[256];


// =====================
//  Device Helper Functions
// =====================

// SubBytes transformation using S-box
__device__ void SubBytes(uint8_t* state) {
    for (int i = 0; i < 16; i++) {
        state[i] = d_sbox[state[i]];
```

```
    }
}

// InvSubBytes transformation using inverse S-box
__device__ void InvSubBytes(uint8_t* state) {
    for (int i = 0; i < 16; i++) {
        state[i] = d_inv_sbox[state[i]];
    }
}

// ShiftRows transformation (left shift rows 1-3)
__device__ void ShiftRows(uint8_t* state) {
    uint8_t tmp;
    // Row 1
    tmp = state[1]; state[1] = state[5]; state[5] = state[9]; state[9] = state[13]; state[13] = tmp;
    // Row 2
    tmp = state[2]; state[2] = state[10]; state[10] = tmp;
    tmp = state[6]; state[6] = state[14]; state[14] = tmp;
    // Row 3
    tmp = state[3]; state[3] = state[15]; state[15] = state[11]; state[11] = state[7]; state[7] = tmp;
}

// Inverse ShiftRows transformation (right shift rows 1-3)
__device__ void InvShiftRows(uint8_t* state) {
    uint8_t tmp;
    // Row 1
    tmp = state[13]; state[13] = state[9]; state[9] = state[5]; state[5] = state[1]; state[1] = tmp;
    // Row 2
    tmp = state[2]; state[2] = state[10]; state[10] = tmp;
    tmp = state[6]; state[6] = state[14]; state[14] = tmp;
    // Row 3
    tmp = state[3]; state[3] = state[7]; state[7] = state[11]; state[11] = state[15]; state[15] = tmp;
}

// AddRoundKey step: XOR state with round key
__device__ void AddRoundKey(uint8_t* state, uint8_t* roundKey) {
    for (int i = 0; i < 16; i++) {
        state[i] ^= roundKey[i];
    }
}

// xtime: multiply by 2 in GF(2^8)
__device__ uint8_t xtime(uint8_t x) {
    return (x << 1) ^ ((x & 0x80) ? 0x1B : 0x00);
}

// General multiplication in GF(2^8)
__device__ uint8_t multiply(uint8_t x, uint8_t y) {
```

```
    uint8_t result = 0;
    while (y) {
        if (y & 1) result ^= x;
        x = xtime(x);
        y >>= 1;
    }
    return result;
}

// MixColumns transformation for diffusion
__device__ void MixColumns(uint8_t* state) {
    uint8_t temp[4];
    for (int i = 0; i < 4; i++) { // For each column
        temp[0] = multiply(0x02, state[i*4 + 0]) ^ multiply(0x03, state[i*4 + 1]) ^ state[i*4 + 2] ^ state[i*4 + 3];
        temp[1] = state[i*4 + 0] ^ multiply(0x02, state[i*4 + 1]) ^ multiply(0x03, state[i*4 + 2]) ^ state[i*4 + 3];
        temp[2] = state[i*4 + 0] ^ state[i*4 + 1] ^ multiply(0x02, state[i*4 + 2]) ^ multiply(0x03, state[i*4 + 3]);
        temp[3] = multiply(0x03, state[i*4 + 0]) ^ state[i*4 + 1] ^ state[i*4 + 2] ^ multiply(0x02, state[i*4 + 3]);
        for (int j = 0; j < 4; j++)
            state[i*4 + j] = temp[j];
    }
}

// Inverse MixColumns for decryption
__device__ void InvMixColumns(uint8_t* state) {
    uint8_t temp[4];
    for (int i = 0; i < 4; i++) { // For each column
        temp[0] = multiply(0x0e, state[i*4 + 0]) ^ multiply(0x0b, state[i*4 + 1]) ^ multiply(0x0d, state[i*4 + 2]) ^
multiply(0x09, state[i*4 + 3]);
        temp[1] = multiply(0x09, state[i*4 + 0]) ^ multiply(0x0e, state[i*4 + 1]) ^ multiply(0x0b, state[i*4 + 2]) ^
multiply(0x0d, state[i*4 + 3]);
        temp[2] = multiply(0x0d, state[i*4 + 0]) ^ multiply(0x09, state[i*4 + 1]) ^ multiply(0x0e, state[i*4 + 2]) ^
multiply(0x0b, state[i*4 + 3]);
        temp[3] = multiply(0x0b, state[i*4 + 0]) ^ multiply(0x0d, state[i*4 + 1]) ^ multiply(0x09, state[i*4 + 2]) ^
multiply(0x0e, state[i*4 + 3]);
        for (int j = 0; j < 4; j++)
            state[i*4 + j] = temp[j];
    }
}

// =====================
//  Encryption & Decryption Kernels
// =====================

// Each thread encrypts one 16-byte block
__global__ void encrypt_kernel(uint8_t* input, uint8_t* output, uint8_t* roundKeys) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= NUM_BLOCKS) return;
```

```
    uint8_t state[16];
    // Copy input block to local state
    for (int i = 0; i < 16; i++) state[i] = input[idx * 16 + i];

    AddRoundKey(state, roundKeys); // Initial round key

    // 9 main rounds
    for (int round = 1; round < 10; round++) {
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, roundKeys + round * 16);
    }
    // Final round (no MixColumns)
    SubBytes(state);
    ShiftRows(state);
    AddRoundKey(state, roundKeys + 160);

    // Write encrypted block to output
    for (int i = 0; i < 16; i++) output[idx * 16 + i] = state[i];
}

// Each thread decrypts one 16-byte block
__global__ void decrypt_kernel(uint8_t* input, uint8_t* output, uint8_t* roundKeys) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= NUM_BLOCKS) return;

    uint8_t state[16];
    // Copy input block to local state
    for (int i = 0; i < 16; i++) state[i] = input[idx * 16 + i];

    AddRoundKey(state, roundKeys + 160); // Initial round key (last round key)

    // 9 main rounds (in reverse)
    for (int round = 9; round > 0; round--) {
        InvShiftRows(state);
        InvSubBytes(state);
        AddRoundKey(state, roundKeys + round * 16);
        InvMixColumns(state);
    }
    // Final round (no InvMixColumns)
    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(state, roundKeys);

    // Write decrypted block to output
    for (int i = 0; i < 16; i++) output[idx * 16 + i] = state[i];
}
```

```
// ====================
//  Key Expansion (Host)
// ====================

// Expand 128-bit key into 11 round keys (AES-128)
void KeyExpansion(const uint8_t* key, uint8_t* roundKeys) {
    const uint8_t Rcon[10] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36};

    // Copy the original 16-byte key as the first round key
    memcpy(roundKeys, key, 16);

    // Generate the rest of the round keys
    for (int i = 4; i < 44; i++) {
        uint8_t temp[4];
        // Get previous word
        for (int j = 0; j < 4; j++) {
            temp[j] = roundKeys[(i-1)*4 + j];
        }
        // Every 4th word: apply key schedule core
        if (i % 4 == 0) {
            // RotWord: rotate left by 1 byte
            uint8_t t = temp[0];
            temp[0] = temp[1];
            temp[1] = temp[2];
            temp[2] = temp[3];
            temp[3] = t;
            // SubWord: apply S-box
            for (int j = 0; j < 4; j++) {
                temp[j] = sbox[temp[j]];
            }
            // XOR with round constant
            temp[0] ^= Rcon[i/4 - 1];
        }
        // XOR with word 4 positions back
        for (int j = 0; j < 4; j++) {
            roundKeys[i*4 + j] = roundKeys[(i-4)*4 + j] ^ temp[j];
        }
    }
}

// ====================
//  Main Function
// ====================

int main() {
    // Example 128-bit AES key
    uint8_t key[16] = {
```

```
    0x2b, 0x7e, 0x15, 0x16,
    0x28, 0xae, 0xd2, 0xa6,
    0xab, 0xf7, 0x15, 0x88,
    0x09, 0xcf, 0x4f, 0x3c
};

// Allocate host memory for input, encrypted, and decrypted data
uint8_t* h_input = new uint8_t[NUM_BLOCKS * BLOCK_SIZE];
uint8_t* h_encrypted = new uint8_t[NUM_BLOCKS * BLOCK_SIZE];
uint8_t* h_decrypted = new uint8_t[NUM_BLOCKS * BLOCK_SIZE];
uint8_t roundKeys[ROUND_KEYS_SIZE];

// Fill input with random data
srand(12345);
for (int i = 0; i < NUM_BLOCKS * BLOCK_SIZE; i++)
    h_input[i] = rand() % 256;

// Expand the key for all AES rounds
KeyExpansion(key, roundKeys);

// Copy S-boxes to device constant memory
cudaMemcpyToSymbol(d_sbox, sbox, 256);
cudaMemcpyToSymbol(d_inv_sbox, inv_sbox, 256);

// Allocate device memory
uint8_t *d_input, *d_output, *d_decrypted, *d_roundKeys;
cudaMalloc(&d_input, NUM_BLOCKS * BLOCK_SIZE);
cudaMalloc(&d_output, NUM_BLOCKS * BLOCK_SIZE);
cudaMalloc(&d_decrypted, NUM_BLOCKS * BLOCK_SIZE);
cudaMalloc(&d_roundKeys, ROUND_KEYS_SIZE);

// Set up CUDA kernel launch dimensions
dim3 blockDim(128);
dim3 gridDim((NUM_BLOCKS + blockDim.x - 1) / blockDim.x);

// CUDA event objects for timing
cudaEvent_t start, stop;
float time_total = 0, time_H2D = 0, time_encrypt = 0, time_decrypt = 0, time_D2H = 0;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cout << "\n[GPU] AES-128 Batch Encryption: " << NUM_BLOCKS << " blocks\n";

// Host to Device memory copy timing
cudaEventRecord(start);
cudaMemcpy(d_input, h_input, NUM_BLOCKS * BLOCK_SIZE, cudaMemcpyHostToDevice);
cudaMemcpy(d_roundKeys, roundKeys, ROUND_KEYS_SIZE, cudaMemcpyHostToDevice);
cudaEventRecord(stop);
```

```
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_H2D, start, stop);

// Encryption kernel timing
cudaEventRecord(start);
encrypt_kernel<<<gridDim, blockDim>>>(d_input, d_output, d_roundKeys);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_encrypt, start, stop);

// Decryption kernel timing
cudaEventRecord(start);
decrypt_kernel<<<gridDim, blockDim>>>(d_output, d_decrypted, d_roundKeys);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_decrypt, start, stop);

// Device to Host memory copy timing
cudaEventRecord(start);
cudaMemcpy(h_encrypted, d_output, NUM_BLOCKS * BLOCK_SIZE, cudaMemcpyDeviceToHost);
cudaMemcpy(h_decrypted, d_decrypted, NUM_BLOCKS * BLOCK_SIZE, cudaMemcpyDeviceToHost);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_D2H, start, stop);

// Total GPU time
time_total = time_H2D + time_encrypt + time_decrypt + time_D2H;

// Print timing breakdown
printf("[GPU Timing Breakdown]\n");
printf("  Host → Device Copy : %.3f ms\n", time_H2D);
printf("  Encryption Kernel  : %.3f ms\n", time_encrypt);
printf("  Decryption Kernel  : %.3f ms\n", time_decrypt);
printf("  Device → Host Copy : %.3f ms\n", time_D2H);
printf("  ------------------------------\n");
printf("  Total GPU Time      : %.3f ms\n", time_total);

// Verify correctness: decrypted data should match original input
bool match = true;
for (int i = 0; i < NUM_BLOCKS * BLOCK_SIZE; i++) {
    if (h_input[i] != h_decrypted[i]) {
        printf("[✗] Mismatch at byte %d: input=%02x, decrypted=%02x\n", i, h_input[i], h_decrypted[i]);
        match = false;
        break;
    }
}
printf("%s\n", match ? "[✓] Decryption matches original plaintext." : "[✗] Decryption mismatch!");
```

```
    // Cleanup device and host memory
    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_decrypted);
    cudaFree(d_roundKeys);
    delete[] h_input;
    delete[] h_encrypted;
    delete[] h_decrypted;

    return 0;
}
```

AES 192:

```
#include <iostream>
#include <cuda_runtime.h>
#include <cstdlib>
#include <ctime>
#include <cstring>

#define BLOCK_SIZE 16
#define DATA_SIZE_MB 100
#define NUM_BLOCKS (DATA_SIZE_MB * 1024 * 1024 / 16)
#define ROUND_KEYS_SIZE 208
#define AES_ROUNDS 12

typedef unsigned char uint8_t;
using namespace std;

const uint8_t sbox[256] = {
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16
};
```

```
const uint8_t inv_sbox[256] = {
    0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb,
    0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb,
    0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e,
    0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25,
    0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92,
    0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84,
    0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06,
    0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b,
    0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73,
    0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e,
    0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b,
    0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4,
    0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f,
    0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef,
    0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61,
    0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d
};

__constant__ uint8_t d_sbox[256];
__constant__ uint8_t d_inv_sbox[256];

////////////////////////////////////////////////////
// Helper Functions
////////////////////////////////////////////////////

// Applies the SubBytes transformation using the S-box
__device__ void SubBytes(uint8_t* state) {
    for (int i = 0; i < 16; i++) {
        state[i] = d_sbox[state[i]];
    }
}

// Applies the inverse SubBytes transformation using the inverse S-box
__device__ void InvSubBytes(uint8_t* state) {
    for (int i = 0; i < 16; i++) {
        state[i] = d_inv_sbox[state[i]];
    }
}

// Performs the ShiftRows transformation on the state array
__device__ void ShiftRows(uint8_t* state) {
    uint8_t tmp;
    // Row 1 left shift by 1
    tmp = state[1]; state[1] = state[5]; state[5] = state[9]; state[9] = state[13]; state[13] = tmp;
    // Row 2 left shift by 2
    tmp = state[2]; state[2] = state[10]; state[10] = tmp;
    tmp = state[6]; state[6] = state[14]; state[14] = tmp;
```

```
    // Row 3 left shift by 3
    tmp = state[3]; state[3] = state[15]; state[15] = state[11]; state[11] = state[7]; state[7] = tmp;
}

// Performs the inverse ShiftRows transformation on the state array
__device__ void InvShiftRows(uint8_t* state) {
    uint8_t tmp;
    // Row 1 right shift by 1
    tmp = state[13]; state[13] = state[9]; state[9] = state[5]; state[5] = state[1]; state[1] = tmp;
    // Row 2 right shift by 2
    tmp = state[2]; state[2] = state[10]; state[10] = tmp;
    tmp = state[6]; state[6] = state[14]; state[14] = tmp;
    // Row 3 right shift by 3
    tmp = state[3]; state[3] = state[7]; state[7] = state[11]; state[11] = state[15]; state[15] = tmp;
}

// XORs the state with the round key
__device__ void AddRoundKey(uint8_t* state, const uint8_t* roundKey) {
    for (int i = 0; i < 16; i++) {
        state[i] ^= roundKey[i];
    }
}

// Multiplies a byte by x (i.e., {02}) in GF(2^8)
__device__ uint8_t xtime(uint8_t x) {
    return (x << 1) ^ ((x & 0x80) ? 0x1B : 0x00);
}

// Multiplies two bytes in GF(2^8)
__device__ uint8_t multiply(uint8_t x, uint8_t y) {
    uint8_t result = 0;
    while (y) {
        if (y & 1) result ^= x;
        x = xtime(x);
        y >>= 1;
    }
    return result;
}

// MixColumns transformation for AES encryption
__device__ void MixColumns(uint8_t* state) {
    uint8_t temp[4];
    for (int i = 0; i < 4; i++) {
        temp[0] = multiply(0x02, state[i*4 + 0]) ^ multiply(0x03, state[i*4 + 1]) ^ state[i*4 + 2] ^ state[i*4 + 3];
        temp[1] = state[i*4 + 0] ^ multiply(0x02, state[i*4 + 1]) ^ multiply(0x03, state[i*4 + 2]) ^ state[i*4 + 3];
        temp[2] = state[i*4 + 0] ^ state[i*4 + 1] ^ multiply(0x02, state[i*4 + 2]) ^ multiply(0x03, state[i*4 + 3]);
        temp[3] = multiply(0x03, state[i*4 + 0]) ^ state[i*4 + 1] ^ state[i*4 + 2] ^ multiply(0x02, state[i*4 + 3]);
        for (int j = 0; j < 4; j++)
```

```
            state[i*4 + j] = temp[j];
    }
}

// Inverse MixColumns transformation for AES decryption
__device__ void InvMixColumns(uint8_t* state) {
    uint8_t temp[4];
    for (int i = 0; i < 4; i++) {
        temp[0] = multiply(0x0e, state[i*4 + 0]) ^ multiply(0x0b, state[i*4 + 1]) ^ multiply(0x0d, state[i*4 + 2]) ^
multiply(0x09, state[i*4 + 3]);
        temp[1] = multiply(0x09, state[i*4 + 0]) ^ multiply(0x0e, state[i*4 + 1]) ^ multiply(0x0b, state[i*4 + 2]) ^
multiply(0x0d, state[i*4 + 3]);
        temp[2] = multiply(0x0d, state[i*4 + 0]) ^ multiply(0x09, state[i*4 + 1]) ^ multiply(0x0e, state[i*4 + 2]) ^
multiply(0x0b, state[i*4 + 3]);
        temp[3] = multiply(0x0b, state[i*4 + 0]) ^ multiply(0x0d, state[i*4 + 1]) ^ multiply(0x09, state[i*4 + 2]) ^
multiply(0x0e, state[i*4 + 3]);
        for (int j = 0; j < 4; j++)
            state[i*4 + j] = temp[j];
    }
}

// Expands a 192-bit key into the full set of AES round keys
void KeyExpansion192(const uint8_t* key, uint8_t* roundKeys) {
    const uint8_t Rcon[10] = {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1B,0x36};
    memcpy(roundKeys, key, 24); // Copy initial key
    uint8_t temp[4];
    int i = 6;
    int bytesGenerated = 24;
    int rconIdx = 0;

    while (bytesGenerated < 208) {
        for (int j = 0; j < 4; j++) temp[j] = roundKeys[(i - 1) * 4 + j];
        if (i % 6 == 0) {
            // Rotate, apply S-box, and add round constant
            uint8_t t = temp[0];
            temp[0] = sbox[temp[1]] ^ Rcon[rconIdx++];
            temp[1] = sbox[temp[2]];
            temp[2] = sbox[temp[3]];
            temp[3] = sbox[t];
        }
        // XOR with word 6 positions earlier
        for (int j = 0; j < 4; j++) {
            roundKeys[i * 4 + j] = roundKeys[(i - 6) * 4 + j] ^ temp[j];
        }
        i++;
        bytesGenerated += 4;
    }
}
```

```
/////////////////////////////////////////////////////
// CUDA Kernels
/////////////////////////////////////////////////////

// Kernel for AES-192 encryption of many blocks in parallel
__global__ void aes192_encrypt_kernel(uint8_t* input, uint8_t* output, const uint8_t* roundKeys) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= NUM_BLOCKS) return; // Out-of-bounds check

    uint8_t state[16];
    // Load one block from global memory to local state
    for (int i = 0; i < 16; i++) state[i] = input[idx * 16 + i];

    // Initial AddRoundKey
    AddRoundKey(state, roundKeys);
    // AES main rounds
    for (int round = 1; round < AES_ROUNDS; round++) {
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, roundKeys + round * 16);
    }
    // Final round (no MixColumns)
    SubBytes(state);
    ShiftRows(state);
    AddRoundKey(state, roundKeys + AES_ROUNDS * 16);

    // Write encrypted block back to global memory
    for (int i = 0; i < 16; i++) output[idx * 16 + i] = state[i];
}

// Kernel for AES-192 decryption of many blocks in parallel
__global__ void aes192_decrypt_kernel(uint8_t* input, uint8_t* output, const uint8_t* roundKeys) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= NUM_BLOCKS) return; // Out-of-bounds check

    uint8_t state[16];
    // Load one block from global memory to local state
    for (int i = 0; i < 16; i++) state[i] = input[idx * 16 + i];

    // Initial AddRoundKey (last round key)
    AddRoundKey(state, roundKeys + AES_ROUNDS * 16);
    // AES main rounds
    for (int round = AES_ROUNDS - 1; round > 0; round--) {
        InvShiftRows(state);
        InvSubBytes(state);
        AddRoundKey(state, roundKeys + round * 16);
```

```
        InvMixColumns(state);
    }
    // Final round (no InvMixColumns)
    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(state, roundKeys);

    // Write decrypted block back to global memory
    for (int i = 0; i < 16; i++) output[idx * 16 + i] = state[i];
}

////////////////////////////////////////////////////////
// Main Function
////////////////////////////////////////////////////////

int main() {
    // Example 192-bit AES key
    uint8_t key[24] = {
        0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e,
        0x64, 0x52, 0xc8, 0x10, 0xf3, 0x2b,
        0x80, 0x90, 0x79, 0xe5, 0x62, 0xf8,
        0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b
    };

    // Allocate host memory for input, encrypted, and decrypted data
    uint8_t* h_input = new uint8_t[NUM_BLOCKS * BLOCK_SIZE];
    uint8_t* h_encrypted = new uint8_t[NUM_BLOCKS * BLOCK_SIZE];
    uint8_t* h_decrypted = new uint8_t[NUM_BLOCKS * BLOCK_SIZE];
    uint8_t roundKeys[ROUND_KEYS_SIZE]; // Buffer for expanded round keys

    // Fill input with random data
    srand(12345);
    for (int i = 0; i < NUM_BLOCKS * BLOCK_SIZE; i++)
        h_input[i] = rand() % 256;

    // Expand the AES key
    KeyExpansion192(key, roundKeys);

    // Copy S-boxes to device constant memory
    cudaMemcpyToSymbol(d_sbox, sbox, 256);
    cudaMemcpyToSymbol(d_inv_sbox, inv_sbox, 256);

    // Allocate device memory
    uint8_t *d_input, *d_output, *d_decrypted, *d_roundKeys;
    cudaMalloc(&d_input, NUM_BLOCKS * BLOCK_SIZE);
    cudaMalloc(&d_output, NUM_BLOCKS * BLOCK_SIZE);
    cudaMalloc(&d_decrypted, NUM_BLOCKS * BLOCK_SIZE);
    cudaMalloc(&d_roundKeys, ROUND_KEYS_SIZE);
```

```
// Configure CUDA kernel launch parameters
dim3 blockDim(128);
dim3 gridDim((NUM_BLOCKS + blockDim.x - 1) / blockDim.x);

// CUDA events for timing
cudaEvent_t start, stop;
float time_total = 0, time_H2D = 0, time_encrypt = 0, time_decrypt = 0, time_D2H = 0;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cout << "\n[GPU] AES-192 Batch Encryption: " << NUM_BLOCKS << " blocks\n";

// Host to Device memory copy timing
cudaEventRecord(start);
cudaMemcpy(d_input, h_input, NUM_BLOCKS * BLOCK_SIZE, cudaMemcpyHostToDevice);
cudaMemcpy(d_roundKeys, roundKeys, ROUND_KEYS_SIZE, cudaMemcpyHostToDevice);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_H2D, start, stop);

// AES encryption kernel timing
cudaEventRecord(start);
aes192_encrypt_kernel<<<gridDim, blockDim>>>(d_input, d_output, d_roundKeys);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_encrypt, start, stop);

// AES decryption kernel timing
cudaEventRecord(start);
aes192_decrypt_kernel<<<gridDim, blockDim>>>(d_output, d_decrypted, d_roundKeys);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_decrypt, start, stop);

// Device to Host memory copy timing
cudaEventRecord(start);
cudaMemcpy(h_encrypted, d_output, NUM_BLOCKS * BLOCK_SIZE, cudaMemcpyDeviceToHost);
cudaMemcpy(h_decrypted, d_decrypted, NUM_BLOCKS * BLOCK_SIZE, cudaMemcpyDeviceToHost);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_D2H, start, stop);

// Sum total GPU time
time_total = time_H2D + time_encrypt + time_decrypt + time_D2H;

// Print timing breakdown
```

```
    printf("[GPU Timing Breakdown]\n");
    printf("  Host → Device Copy : %.3f ms\n", time_H2D);
    printf("  Encryption Kernel  : %.3f ms\n", time_encrypt);
    printf("  Decryption Kernel  : %.3f ms\n", time_decrypt);
    printf("  Device → Host Copy : %.3f ms\n", time_D2H);
    printf("  -----------------------------\n");
    printf("  Total GPU Time      : %.3f ms\n", time_total);

    // Verify that decrypted output matches original input
    bool match = true;
    for (int i = 0; i < NUM_BLOCKS * BLOCK_SIZE; i++) {
        if (h_input[i] != h_decrypted[i]) {
            printf("[✗] Mismatch at byte %d: input=%02x, decrypted=%02x\n", i, h_input[i], h_decrypted[i]);
            match = false;
            break;
        }
    }

    printf("%s\n", match ? "[✓] Decryption matches original plaintext." : "[✗] Decryption mismatch!");

    // Free device and host memory
    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_decrypted);
    cudaFree(d_roundKeys);
    delete[] h_input;
    delete[] h_encrypted;
    delete[] h_decrypted;

    return 0;
}
```

AES 256:

```
#include <iostream>
#include <cuda_runtime.h>
#include <cstdlib>
#include <ctime>
#include <cstring>

#define BLOCK_SIZE 16                          // AES block size in bytes
#define DATA_SIZE_MB 100                       // Amount of data to process (in MB)
#define NUM_BLOCKS (DATA_SIZE_MB * 1024 * 1024 / 16)  // Number of 16-byte blocks to process
#define ROUND_KEYS_SIZE 240                    // Total bytes for AES-256 expanded key schedule (15*16)
#define AES_ROUNDS 14                          // Number of rounds for AES-256

typedef unsigned char uint8_t;
using namespace std;
```

```
const uint8_t sbox[256] = {
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16
};

const uint8_t inv_sbox[256] = {
    0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb,
    0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb,
    0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e,
    0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25,
    0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92,
    0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84,
    0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06,
    0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b,
    0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73,
    0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e,
    0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b,
    0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4,
    0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f,
    0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef,
    0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61,
    0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d
};

__constant__ uint8_t d_sbox[256];
__constant__ uint8_t d_inv_sbox[256];

/////////////////////////////////////////////////////
// Helper Functions for AES Transformations
/////////////////////////////////////////////////////

// SubBytes transformation: substitute each byte in the state using the S-box
__device__ void SubBytes(uint8_t* state) {
```

```
    for (int i = 0; i < 16; i++) {
        state[i] = d_sbox[state[i]];
    }
}

// InvSubBytes transformation: substitute each byte in the state using the inverse S-box
__device__ void InvSubBytes(uint8_t* state) {
    for (int i = 0; i < 16; i++) {
        state[i] = d_inv_sbox[state[i]];
    }
}

// ShiftRows transformation: cyclically shift rows of the state to the left
__device__ void ShiftRows(uint8_t* state) {
    uint8_t tmp;
    // Row 1: shift left by 1
    tmp = state[1]; state[1] = state[5]; state[5] = state[9]; state[9] = state[13]; state[13] = tmp;
    // Row 2: shift left by 2
    tmp = state[2]; state[2] = state[10]; state[10] = tmp;
    tmp = state[6]; state[6] = state[14]; state[14] = tmp;
    // Row 3: shift left by 3
    tmp = state[3]; state[3] = state[15]; state[15] = state[11]; state[11] = state[7]; state[7] = tmp;
}

// InvShiftRows transformation: cyclically shift rows of the state to the right
__device__ void InvShiftRows(uint8_t* state) {
    uint8_t tmp;
    // Row 1: shift right by 1
    tmp = state[13]; state[13] = state[9]; state[9] = state[5]; state[5] = state[1]; state[1] = tmp;
    // Row 2: shift right by 2
    tmp = state[2]; state[2] = state[10]; state[10] = tmp;
    tmp = state[6]; state[6] = state[14]; state[14] = tmp;
    // Row 3: shift right by 3
    tmp = state[3]; state[3] = state[7]; state[7] = state[11]; state[11] = state[15]; state[15] = tmp;
}

// AddRoundKey transformation: XOR the state with the round key
__device__ void AddRoundKey(uint8_t* state, const uint8_t* roundKey) {
    for (int i = 0; i < 16; i++) {
        state[i] ^= roundKey[i];
    }
}

// xtime: multiply by 2 in GF(2^8)
__device__ uint8_t xtime(uint8_t x) {
    return (x << 1) ^ ((x & 0x80) ? 0x1B : 0x00);
}
```

```
// multiply: multiply two bytes in GF(2^8)
__device__ uint8_t multiply(uint8_t x, uint8_t y) {
    uint8_t result = 0;
    while (y) {
        if (y & 1) result ^= x;
        x = xtime(x);
        y >>= 1;
    }
    return result;
}

// MixColumns transformation: mix each column of the state matrix (encryption)
__device__ void MixColumns(uint8_t* state) {
    uint8_t temp[4];
    for (int i = 0; i < 4; i++) {
        temp[0] = multiply(0x02, state[i*4 + 0]) ^ multiply(0x03, state[i*4 + 1]) ^ state[i*4 + 2] ^ state[i*4 + 3];
        temp[1] = state[i*4 + 0] ^ multiply(0x02, state[i*4 + 1]) ^ multiply(0x03, state[i*4 + 2]) ^ state[i*4 + 3];
        temp[2] = state[i*4 + 0] ^ state[i*4 + 1] ^ multiply(0x02, state[i*4 + 2]) ^ multiply(0x03, state[i*4 + 3]);
        temp[3] = multiply(0x03, state[i*4 + 0]) ^ state[i*4 + 1] ^ state[i*4 + 2] ^ multiply(0x02, state[i*4 + 3]);
        for (int j = 0; j < 4; j++)
            state[i*4 + j] = temp[j];
    }
}

// InvMixColumns transformation: mix each column of the state matrix (decryption)
__device__ void InvMixColumns(uint8_t* state) {
    uint8_t temp[4];
    for (int i = 0; i < 4; i++) {
        temp[0] = multiply(0x0e, state[i*4 + 0]) ^ multiply(0x0b, state[i*4 + 1]) ^ multiply(0x0d, state[i*4 + 2]) ^
multiply(0x09, state[i*4 + 3]);
        temp[1] = multiply(0x09, state[i*4 + 0]) ^ multiply(0x0e, state[i*4 + 1]) ^ multiply(0x0b, state[i*4 + 2]) ^
multiply(0x0d, state[i*4 + 3]);
        temp[2] = multiply(0x0d, state[i*4 + 0]) ^ multiply(0x09, state[i*4 + 1]) ^ multiply(0x0e, state[i*4 + 2]) ^
multiply(0x0b, state[i*4 + 3]);
        temp[3] = multiply(0x0b, state[i*4 + 0]) ^ multiply(0x0d, state[i*4 + 1]) ^ multiply(0x09, state[i*4 + 2]) ^
multiply(0x0e, state[i*4 + 3]);
        for (int j = 0; j < 4; j++)
            state[i*4 + j] = temp[j];
    }
}

////////////////////////////////////////////////////////
// AES-256 Key Expansion (host-side)
////////////////////////////////////////////////////////

// KeyExpansion256: expands a 256-bit key into round keys for all AES rounds
void KeyExpansion256(const uint8_t* key, uint8_t* roundKeys) {
    const uint8_t Rcon[10] = {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1B,0x36};
```

```
    memcpy(roundKeys, key, 32); // Copy original key as first 8 words (32 bytes)
    uint8_t temp[4];
    int i = 8;
    int rconIdx = 0;

    while (i < 60) { // AES-256 needs 60 words (4 bytes each)
        for (int j = 0; j < 4; j++)
            temp[j] = roundKeys[(i - 1) * 4 + j];

        if (i % 8 == 0) {
            // Rotate, substitute, and XOR with Rcon for every 8th word
            uint8_t t = temp[0];
            temp[0] = sbox[temp[1]] ^ Rcon[rconIdx++];
            temp[1] = sbox[temp[2]];
            temp[2] = sbox[temp[3]];
            temp[3] = sbox[t];
        } else if (i % 8 == 4) {
            // Substitute for every 4th word (except first)
            for (int j = 0; j < 4; j++)
                temp[j] = sbox[temp[j]];
        }

        // XOR with word 8 positions earlier
        for (int j = 0; j < 4; j++) {
            roundKeys[i * 4 + j] = roundKeys[(i - 8) * 4 + j] ^ temp[j];
        }
        i++;
    }
}

////////////////////////////////////////////////////////
// CUDA Kernels for AES-256 Encryption/Decryption
////////////////////////////////////////////////////////

// aes256_encrypt_kernel: encrypts each 16-byte block independently in parallel
__global__ void aes256_encrypt_kernel(uint8_t* input, uint8_t* output, const uint8_t* roundKeys) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= NUM_BLOCKS) return; // Out-of-bounds check

    uint8_t state[16];
    // Load input block into local state
    for (int i = 0; i < 16; i++) state[i] = input[idx * 16 + i];

    // Initial round key addition
    AddRoundKey(state, roundKeys);

    // Main AES rounds
    for (int round = 1; round < AES_ROUNDS; round++) {
```

```
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, roundKeys + round * 16);
    }
    // Final round (no MixColumns)
    SubBytes(state);
    ShiftRows(state);
    AddRoundKey(state, roundKeys + AES_ROUNDS * 16);

    // Store encrypted block to output
    for (int i = 0; i < 16; i++) output[idx * 16 + i] = state[i];
}

// aes256_decrypt_kernel: decrypts each 16-byte block independently in parallel
__global__ void aes256_decrypt_kernel(uint8_t* input, uint8_t* output, const uint8_t* roundKeys) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= NUM_BLOCKS) return; // Out-of-bounds check

    uint8_t state[16];
    // Load encrypted block into local state
    for (int i = 0; i < 16; i++) state[i] = input[idx * 16 + i];

    // Initial round key addition (last round key)
    AddRoundKey(state, roundKeys + AES_ROUNDS * 16);

    // Main AES rounds (in reverse)
    for (int round = AES_ROUNDS - 1; round > 0; round--) {
        InvShiftRows(state);
        InvSubBytes(state);
        AddRoundKey(state, roundKeys + round * 16);
        InvMixColumns(state);
    }
    // Final round (no InvMixColumns)
    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(state, roundKeys);

    // Store decrypted block to output
    for (int i = 0; i < 16; i++) output[idx * 16 + i] = state[i];
}

////////////////////////////////////////////////////
// Main Program: AES-256 Batch Encryption/Decryption Test
////////////////////////////////////////////////////
int main() {
    // Example 256-bit AES key (32 bytes)
    uint8_t key[32] = {
```

```
    0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
    0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
    0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
    0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4
};

// Allocate host memory for input, encrypted, and decrypted data
uint8_t* h_input = new uint8_t[NUM_BLOCKS * BLOCK_SIZE];
uint8_t* h_encrypted = new uint8_t[NUM_BLOCKS * BLOCK_SIZE];
uint8_t* h_decrypted = new uint8_t[NUM_BLOCKS * BLOCK_SIZE];
uint8_t roundKeys[ROUND_KEYS_SIZE];

// Fill input with random data
srand(12345);
for (int i = 0; i < NUM_BLOCKS * BLOCK_SIZE; i++)
    h_input[i] = rand() % 256;

// Expand the key for all AES rounds
KeyExpansion256(key, roundKeys);

// Copy S-boxes to GPU constant memory
cudaMemcpyToSymbol(d_sbox, sbox, 256);
cudaMemcpyToSymbol(d_inv_sbox, inv_sbox, 256);

// Allocate device memory
uint8_t *d_input, *d_output, *d_decrypted, *d_roundKeys;
cudaMalloc(&d_input, NUM_BLOCKS * BLOCK_SIZE);
cudaMalloc(&d_output, NUM_BLOCKS * BLOCK_SIZE);
cudaMalloc(&d_decrypted, NUM_BLOCKS * BLOCK_SIZE);
cudaMalloc(&d_roundKeys, ROUND_KEYS_SIZE);

// Set up CUDA kernel launch configuration
dim3 blockDim(128); // 128 threads per block
dim3 gridDim((NUM_BLOCKS + blockDim.x - 1) / blockDim.x); // Enough blocks to cover all data

// CUDA events for timing
cudaEvent_t start, stop;
float time_total = 0, time_H2D = 0, time_encrypt = 0, time_decrypt = 0, time_D2H = 0;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cout << "\n[GPU] AES-256 Batch Encryption: " << NUM_BLOCKS << " blocks\n";

// Copy input data and round keys from host to device (timed)
cudaEventRecord(start);
cudaMemcpy(d_input, h_input, NUM_BLOCKS * BLOCK_SIZE, cudaMemcpyHostToDevice);
cudaMemcpy(d_roundKeys, roundKeys, ROUND_KEYS_SIZE, cudaMemcpyHostToDevice);
```

```
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_H2D, start, stop);

// Launch encryption kernel (timed)
cudaEventRecord(start);
aes256_encrypt_kernel<<<gridDim, blockDim>>>(d_input, d_output, d_roundKeys);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_encrypt, start, stop);

// Launch decryption kernel (timed)
cudaEventRecord(start);
aes256_decrypt_kernel<<<gridDim, blockDim>>>(d_output, d_decrypted, d_roundKeys);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_decrypt, start, stop);

// Copy results back from device to host (timed)
cudaEventRecord(start);
cudaMemcpy(h_encrypted, d_output, NUM_BLOCKS * BLOCK_SIZE, cudaMemcpyDeviceToHost);
cudaMemcpy(h_decrypted, d_decrypted, NUM_BLOCKS * BLOCK_SIZE, cudaMemcpyDeviceToHost);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_D2H, start, stop);

// Sum up total GPU time
time_total = time_H2D + time_encrypt + time_decrypt + time_D2H;

// Print timing breakdown
printf("[GPU Timing Breakdown]\n");
printf("  Host → Device Copy : %.3f ms\n", time_H2D);
printf("  Encryption Kernel  : %.3f ms\n", time_encrypt);
printf("  Decryption Kernel  : %.3f ms\n", time_decrypt);
printf("  Device → Host Copy : %.3f ms\n", time_D2H);
printf("  ----------------------------\n");
printf("  Total GPU Time      : %.3f ms\n", time_total);

// Verify that decrypted data matches original input
bool match = true;
for (int i = 0; i < NUM_BLOCKS * BLOCK_SIZE; i++) {
    if (h_input[i] != h_decrypted[i]) {
        printf("[X] Mismatch at byte %d: input=%02x, decrypted=%02x\n", i, h_input[i], h_decrypted[i]);
        match = false;
        break;
    }
}
```

```
    printf("%s\n", match ? "[✓] Decryption matches original plaintext." : "[✗] Decryption mismatch!");

    // Free device and host memory
    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_decrypted);
    cudaFree(d_roundKeys);
    delete[] h_input;
    delete[] h_encrypted;
    delete[] h_decrypted;

    return 0;
}
```