Crypto Pyaes Report

Introduction

The Advanced Encryption Standard (AES) is one of the most widely used block ciphers in modern cryptography. It provides strong data protection through a symmetric key algorithm, and its efficiency has made it the backbone of many applications including secure communications, file encryption, and network protocols such as TLS and VPNs. Python's pyaes library offers pure Python implementation of AES, supporting multiple modes of operation such as ECB, CBC, CFB, OFB, and CTR. Unlike AES implementations in C or hardware, which are heavily optimized, the pure-Python version is significantly slower. However, it serves as an excellent benchmark for analyzing interpreter-level performance and for exploring optimization opportunities.

We chose to focus on the crypto_pyaes benchmark from the performance suite because encryption is both computation-intensive and performance-sensitive. Optimizing AES can provide insights into Python's execution model, the cost of arithmetic and lookup-heavy operations, and the trade-offs between readability, correctness, and speed in algorithmic code. Furthermore, since AES is a real-world workload with practical use cases, improvements made here directly translate into more efficient secure applications.

In the standard implementation of pyaes, each encryption and decryption step involve repeated table lookups, word transformations, and arithmetic over finite fields. While functionally correct, these operations can create bottlenecks due to Python's high overhead loops, repeated indexing, and memory management. By examining this behavior, we can identify inefficiencies and propose optimizations that restructure the code for faster execution.

In this project, we introduce an optimized version of pyaes that reduces redundant computations with the usage of precomputation and loop unrolling. The optimized design maintains correctness while improving runtime efficiency, making the benchmark run faster in CTR mode. In addition to software optimizations, we explore a hardware accelerator that performs AES round computations directly on-chip using preloaded tables and keys, offering constant-latency execution and reduced CPU overhead.

Overview

AES Algorithm

The Advanced Encryption Standard (AES) is a symmetric block cipher standardized by NIST and widely adopted for securing digital communications. AES operates on fixed-size blocks of 128 bits (16 bytes), using keys of 128, 192, or 256 bits. The encryption process transforms plaintext into ciphertext through a sequence of substitution and permutation steps organized into *rounds*. The number of rounds depends on the key size: 10 for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys.

Each round consists of the following core transformations:

> SubBytes – A nonlinear substitution step where each byte is replaced using a fixed S-box lookup table.
> ShiftRows – A row-wise permutation that cyclically shifts the rows of the state array.
> MixColumns – A mixing operation that transforms each column using matrix multiplication over a finite field, spreading byte influence across the state.
> AddRoundKey – A bitwise XOR of the state with a round-specific subkey derived from the original key.

The final round omits the MixColumns step, producing the ciphertext. Decryption reverses these operations using inverse transformations.

Modes of Operation of AES

AES by itself encrypts only a single 128-bit block at a time. If applied directly to larger messages, identical 128-bit input blocks would always produce the same ciphertext blocks (ECB). This is insecure, since patterns in the plaintext would be visible in the ciphertext. To overcome this limitation and provide semantic security, modes of operation are used to extend AES to variable-length messages and to introduce randomness or dependency between blocks.

The crypto_pyaes library implements the following modes:

CTR (Counter Mode) – The mode used in our benchmark. CTR turns AES into a stream cipher by encrypting successive values of a counter, which are then XORed with the plaintext to produce the ciphertext. Decryption is identical: the ciphertext is XORed with the same keystream (the encrypted counters), recovering the original plaintext. As a result, CTR mode uses the same AES encryption function for both encryption and decryption, eliminating the need for a separate decryption routine and simplifying implementation while retaining strong security.
ECB (Electronic Codebook) – Directly encrypts each block independently (insecure against pattern leakage).
CBC (Cipher Block Chaining) – Each block is XORed with the previous ciphertext block before encryption.
CFB and OFB (Cipher Feedback, Output Feedback) – Turn AES into a stream cipher by feeding previous outputs into the next encryption.

AES with T-Tables

The pure-Python implementation of pyaes adopts the T-table method, a common software optimization for AES. Instead of performing SubBytes, ShiftRows, and MixColumns separately, precomputed lookup tables (T-tables) merge these transformations into a single step. Each T-table maps one input byte into a 32-bit word that represents the combined effect of substitution, row shifting, and column mixing. Four such tables (T1–T4) are used per round, significantly reducing runtime operations at the cost of higher memory usage.

Benchmark Analysis

The Benchmark

The crypto_pyaes benchmark from the pyperformance suite evaluates the performance of AES in CTR mode. The benchmark script:

1. Initializes an AES object with a 128-bit key.
2. Encrypts a cleartext message of ~23,000 bytes using CTR mode.
3. Reinitializes AES with the same key and counter state.
4. Decrypts the ciphertext back to plaintext.
5. Verifies correctness by comparing with the original message.

This process is repeated multiple times inside a timed loop to measure throughput and latency.

Expected Bottlenecks

The runtime cost of the benchmark is dominated by the encryption and decryption loops, where thousands of table lookups, shifts, and XORs occur for each 16-byte block. Initialization (key expansion, counter setup) is relatively lightweight and performed only once per benchmark iteration. Thus, most of the performance improvement potential lies in optimizing the inner encryption and decryption routines rather than setup or initialization.

Benchmark Baseline Performance

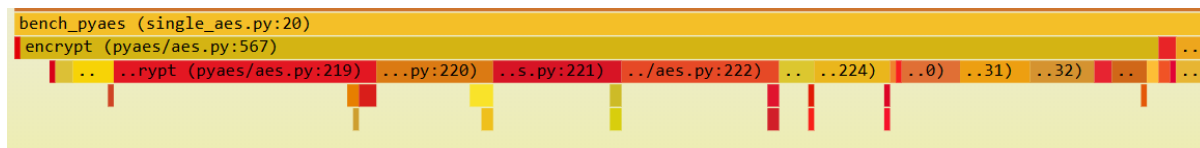crypto_pyaes: Mean +- std dev: 175 ms +- 1 ms

Perf Report + Stats

The perf report showing the distribution of the function calls during the single aes operation can be seen at res/aes/perf_report_aes_baseline.txt

The performance profiling of the baseline AES Python benchmark shows that execution is dominated by interpreter overhead rather than raw AES computation. The single largest contributor is _PyEval_EvalFrameDefault, which alone accounts for roughly 12% of all sampled cycles, reflecting the cost of repeatedly entering and dispatching Python call frames. In addition, significant portions of time are spent in Python's memory debug and allocation routines such as _PyMem_DebugCheckAddress (~7%), read_size_t (~5%), _PyMem_DebugRawFree (~3%), and write_size_t (~3%). Together these functions highlight that memory management and object lifecycle overhead consume more resources than the actual numeric operations. Overall, the perf data demonstrates that the bottleneck lies primarily in Python's interpreter loop and object model rather than the AES algorithm's mathematical complexity.

Metric	Baseline Value
L1-dcache-load-misses	4.327E+08
L1-dcache-loads	3.274E+10
LLC-load-misses	1.287E+06
LLC-loads	4.978E+07
branch-misses	1.638E+08
Branches	2.157E+10
context-switches	1.482E+03
cpu-migrations	2.600E+01
Cycles	4.507E+10
dTLB-load-misses	9.278E+06
dTLB-loads	3.199E+10
iTLB-load-misses	3.598E+06
iTLB-loads	6.681E+06
Instructions	1.101E+11
page-faults	7.838E+04
task-clock	1.752E+04
time-elapsed	1.824E+01

Flamegraph

The Flamegraph of the single baseline aes operation can be seen at
res/aes/flamegraph_aes_baseline.svg

From the flamegraph we see that the hottest lines were 219-224 and 230-232.

In these lines we are mainly seeing the core arithmetic of the AES algorithm. The loops
implement the T-table method, which combines ShiftRows, MixColumns, and AddRoundKey
(XOR) using lookups into precomputed tables. Each iteration processes a 4-byte row of the
AES state, repeatedly performing table lookups and arithmetic operations. Since the same
types of arithmetic are applied many times across rounds, reducing the cost of these
repeated operations - for example by optimizing how the lookups and bitwise operations are
handled - can translate directly into a noticeable performance boost.

Optimization Strategies

From the benchmark analysis, the main bottlenecks in crypto_pyaes lie in the encryption
and decryption routines. In CTR mode, both operations rely on the same underlying
function, encrypt, since encryption and decryption use identical keystream generation. This
means that optimizing encrypt benefits both directions of the operation and has the largest
impact on performance.

We focused our optimization efforts on restructuring this function to reduce Python
interpreter overhead and redundant work. Three main strategies were applied: loop
unrolling, avoiding data copying with double buffering, and removing unnecessary bit masks.

Loop Unrolling

In the original implementation, encryption of each block involves multiple nested loops to
apply transformations round by round. While correct, these loops add significant overhead
in Python because every iteration involves repeated index calculations, loop condition
checks, and function calls.

By unrolling the inner loops, we reduce the number of Python-level iterations and directly
inline repeated operations. This minimizes interpreter overhead, improves instruction

locality, and allows the CPU to execute operations more efficiently. For small, fixed-size structures like AES state arrays (always 16 bytes), loop unrolling can yield measurable performance improvements.

```python
for i in xrange(0, 4):
    a[i] = (self.T5[(t[ i          ] >> 24) & 0xFF] ^
            self.T6[(t[(i + s1) % 4] >> 16) & 0xFF] ^
            self.T7[(t[(i + s2) % 4] >>  8) & 0xFF] ^
            self.T8[ t[(i + s3) % 4]        & 0xFF] ^
            self._Kd[r][i])
```

image 1: Original Library Code Using Loop

```python
# i = 0
a[0] = (self.T1[(t[0] >> 24)] ^
        self.T2[(t[1] >> 16) & 0xFF] ^
        self.T3[(t[2] >> 8)  & 0xFF] ^
        self.T4[t[3] & 0xFF] ^
        self._Ke[r][0])
# i = 1
a[1] = (self.T1[(t[1] >> 24)] ^
        self.T2[(t[2] >> 16) & 0xFF] ^
        self.T3[(t[3] >> 8)  & 0xFF] ^
        self.T4[t[0] & 0xFF] ^
        self._Ke[r][1])
# i = 2
a[2] = (self.T1[(t[2] >> 24)] ^
        self.T2[(t[3] >> 16) & 0xFF] ^
        self.T3[(t[0] >> 8)  & 0xFF] ^
        self.T4[t[1] & 0xFF] ^
        self._Ke[r][2])
# i = 3
a[3] = (self.T1[(t[3] >> 24)] ^
        self.T2[(t[0] >> 16) & 0xFF] ^
        self.T3[(t[1] >> 8)  & 0xFF] ^
        self.T4[t[2] & 0xFF] ^
        self._Ke[r][3])
```

image 2: Optimized Code Using Loop Unrolling

Avoiding Copy with Double Buffering

The original code often copies intermediate AES state arrays between rounds, creating unnecessary memory allocations and object management overhead. To avoid this, we replaced repeated copying with a double-buffering strategy: two alternating buffers are used to hold intermediate states. After each round, the roles of the buffers are swapped instead of creating new lists.

This reduces memory churn and garbage collection pressure, as Python avoids repeatedly allocating and freeing short-lived arrays. It also improves cache locality since the buffers are reused, making subsequent access more efficient.

```python
t = copy.copy(a)
```

image 3: Original Library Code Using Copy

```python
t,a = a,t
```

image 4: Optimized Code Using Double Buffering
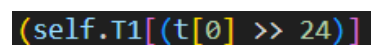
Removing Unnecessary Masks

In the T-table implementation, each round extracts 8-bit components from 32-bit words by shifting and applying bit masks. However, when shifting by 24, the result is already constrained to the range 0–255, making the additional & 0xFF mask redundant.

By eliminating such redundant masks, we reduce the number of operations executed per round. While each mask removal saves only a single integer operation, these operations occur very often in the benchmark, so even small reductions accumulate into noticeable performance gains.

```
(self.T1[(t[ i        ] >> 24) & 0xFF]
```

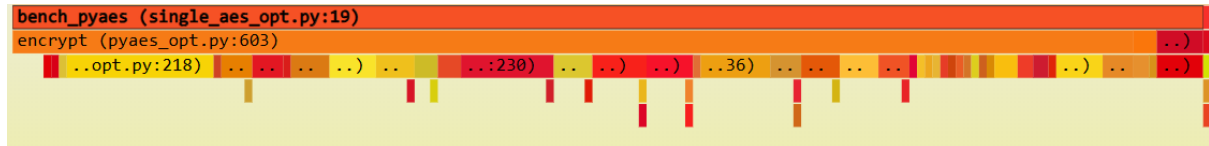*image 5: Original Library Code Using Unnecessary Masks*

```
(self.T1[(t[0] >> 24)]
```

image 6: Optimized Code Removes Unnecessary Masks

Performance Comparison

Flamegraph

The Flamegraph of the single opt aes operation can be seen at
res/aes/flamegraph_aes_opt.svg

With the optimizations made to the loop content, we can now see the hot lines being spread out to the unrolled loop, in an equally distributed method. Additionally, the introduction of double buffering (at line 242 in the optimized code) reduces the cost of buffer copy operations: what previously accounted for about 2.53% of the runtime now takes only 0.24%. This highlights how the optimizations reduced overhead from data movement, allowing more of the runtime to be spent directly on arithmetic operations.

Perf Report

The perf report showing the distribution of the function calls during the single optimized aes operation can be seen at res/aes/perf_report_aes_opt.txt.
The perf report for the optimized AES Python implementation shows essentially the same profile as the baseline case. The dominant costs remain _PyEval_EvalFrameDefault (~12%) along with Python's memory debugging and allocation routines (_PyMem_DebugCheckAddress ~9%, read_size_t ~5%). These results confirm that the overwhelming runtime is still consumed by Python interpreter overhead and object lifecycle management rather than the arithmetic instructions themselves. Since our optimizations - such as loop unrolling and mask removal - targeted redundant arithmetic at the Python level without bypassing the interpreter, the perf output remains visually unchanged.

Performance & Stats Comparison

Benchmark	Baseline Mean (us)	Baseline Std (us)	Optimized Mean (us)	Optimized Std (us)	Delta (us) (Opt - Base)	Improvement (%)
crypto_pyaes	175000	1000	136000	1000	-39000	22.29

With all optimizations applied, we achieved an overall performance improvement of ~22%. This gain aligns with expectations, as the changes directly target redundant work in the AES encryption loop. Loop unrolling avoids repeated index calculations and loop overhead by inlining operations, allowing us to "jump ahead" directly to the T-table lookups, which dominate the cost of each round. Removing redundant masking eliminates unnecessary integer operations that, while individually cheap, accumulate heavily across many rounds. Finally, the introduction of double buffering replaces repeated state-array copies with lightweight buffer swaps, reducing memory churn and improving cache locality. Together, these changes streamline both computation and memory handling, yielding consistent performance gains across the benchmark.

Metric	Baseline Value	Optimized Value	Delta (Opt - Base)	%
L1-dcache-load-misses	4.327E+08	4.051E+08	-2.758E+07	-6%
L1-dcache-loads	3.274E+10	2.599E+10	-6.744E+09	-21%
LLC-load-misses	1.287E+06	1.781E+06	4.935E+05	38%
LLC-loads	4.978E+07	6.048E+07	1.070E+07	21%
branch-misses	1.638E+08	1.491E+08	-1.468E+07	-9%
branches	2.157E+10	1.769E+10	-3.873E+09	-18%
context-switches	1.482E+03	1.386E+03	-9.600E+01	-6%
cpu-migrations	2.600E+01	3.000E+01	4.000E+00	15%
cycles	4.507E+10	3.650E+10	-8.575E+09	-19%
dTLB-load-misses	9.278E+06	7.352E+06	-1.926E+06	-21%
dTLB-loads	3.199E+10	2.579E+10	-6.199E+09	-19%
iTLB-load-misses	3.598E+06	2.956E+06	-6.420E+05	-18%
iTLB-loads	6.681E+06	6.053E+06	-6.285E+05	-9%
instructions	1.101E+11	9.016E+10	-1.998E+10	-18%
page-faults	7.838E+04	7.831E+04	-6.600E+01	0%
task-clock	1.752E+04	1.430E+04	-3.229E+03	-18%
time-elapsed	1.824E+01	1.498E+01	-3.256E+00	-18%

From this comparison of the AES baseline vs. optimized perf stat results, we can clearly see that our optimizations (loop unrolling, double buffering, mask removal) reduced the overall compute and memory overhead. The number of instructions executed dropped by ~18, and the cycle count decreased by a similar amount, which lines up with the ~22% runtime improvement in the benchmark. This shows that we are performing fewer Python-level operations and avoiding redundant work inside the encryption routine.

Most cache and TLB-related metrics also decreased, meaning fewer memory references and better efficiency in the instruction stream. However, LLC-loads actually increased by ~21%, and LLC-load-misses also went up by ~38%. This could be explained by the double-buffer optimization: instead of copying into new arrays, we now reuse two alternating buffers. While this avoids repeated allocations and reduces L1/L2 traffic, it also means that the same two buffers are continuously reused and swapped. This can increase pressure on the LLC because their contents are repeatedly evicted/reloaded as the encryption rounds alternate, compared to the sequential (and more cache-friendly) access of a copy operation.

Hardware Acceleration Proposal - T-Table AES Round Accelerator

The idea is to design a small, fixed-latency hardware block that performs the compute-heavy part of an AES round when using the T-table formulation. The unit stores the four 1 KB T-tables on-chip (total 4 KB) and services four parallel lookups per cycle. It reduces the four 32-bit table values with an XOR tree and, using an internal round-key buffer, XORs in the selected round-key word. To handle the final AES round (which omits MixColumns), the unit also includes an S-Box path so the entire round function can be completed on-chip with constant latency.

At a high level, the CPU provides four byte indices corresponding to a state column after ShiftRows rotation. The accelerator reads T0[idx0], T1[idx1], T2[idx2], and T3[idx3] from its local SRAMs in parallel, XOR-reduces those 32-bit values, and then XORs the result with the appropriate round-key word fetched from an internal key store. The output is the round's 32-bit column result. For the last round, the datapath switches to an S-Box + byte-permute path: four S-Box reads (8-bit each) are performed, bytes are placed per ShiftRows, and the result is XORed with the final round-key word, making the last-round column.

The core consists of four 1 KB single-cycle SRAMs holding T0…T3 (each 256×32), a 32-bit four-input XOR tree, and a compact round-key buffer (≤240 B for AES-256) addressed by {round, word}. A small controller captures input indices and the round selector, advances a two- or three-stage pipeline (SRAM read → XOR reduce → key XOR), and asserts valid_out when the 32-bit column value is ready. For the final-round path, a 256×8 S-Box SRAM (replicated per byte lane or time-multiplexed with a one-cycle bubble) feeds a byte assembly block that applies the ShiftRows permutation before the key XOR. To scale throughput, the block can be synthesized with P parallel lanes (1–4). With P=4, the unit can produce one full 128-bit round result per cycle after pipeline fill. The memory footprint is ~4 KB for T-tables + 256 B–1 KB for the S-Box path (depending on replication) + ≤240 B key store.

Inputs: four 8-bit table indices tidx[3:0] (one per T-table) or, in a variant, a 128-bit state with an internal ShiftRows byte-picker, a round/word selector rk_sel, and a valid_in strobe. Optional programming signals preload the key buffer and (if not ROMed) the tables. Outputs: a 32-bit mixout value equal to T0[idx0] $\oplus$ T1[idx1] $\oplus$ T2[idx2] $\oplus$ T3[idx3] $\oplus$ RK, or, in the final-round mode, SBox/ShiftRows(state_bytes) $\oplus$ RK, plus a valid_out strobe.

Advantages: the unit delivers constant-latency, cache-independent table access, removing the dominant memory stalls of software T-tables and reducing CPU work to feeding indices (or state) and orchestrating rounds. It is area-efficient compared to a fully unrolled AES pipeline, yet can approach similar throughput when instantiated with four lanes. Because tables and keys are on-chip, timing is deterministic and friendly to CTR parallelization. Costs: Extra on-chip memory is required (≈4.25–5 KB including S-Box and key store), and the design is AES-specific. In addition, we introduce a separate unit that handles sensitive data and therefore it exposes new threats and should be treated as a potential risk, especially when passing plaintext data and AES keys to the unit.
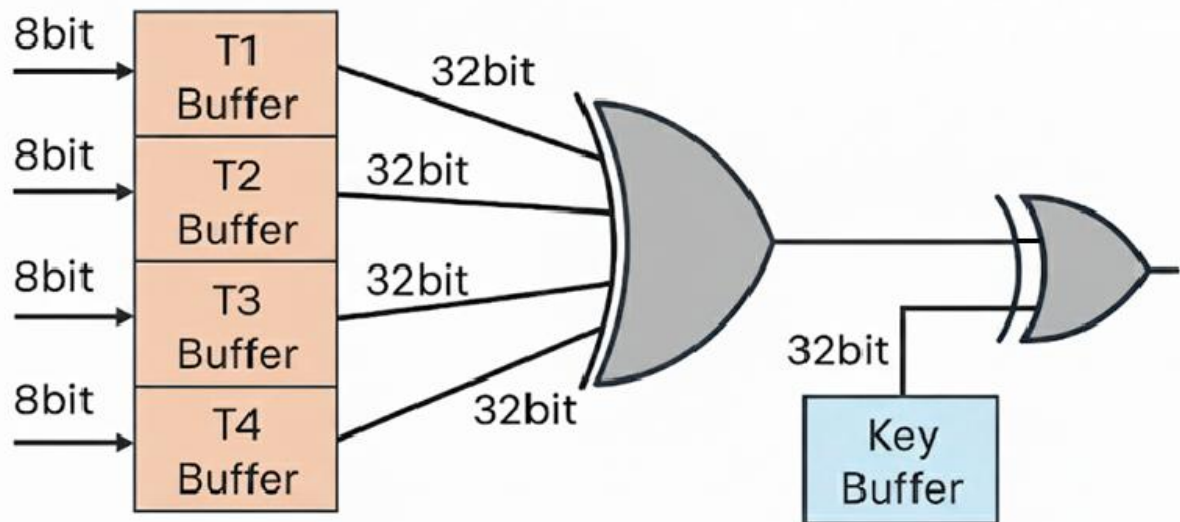
Conclusion

Through this work we demonstrated how both software restructuring and hardware support can meaningfully accelerate AES in CTR mode. On the software side, optimizations such as loop unrolling, double buffering, and mask removal streamlined computation and memory handling, yielding a measurable 22% performance improvement over the baseline Python implementation. On the hardware side, we proposed a dedicated T-table round accelerator with on-chip lookup tables, an integrated S-Box path, and a round-key buffer. This design provides constant-latency AES rounds, reduces CPU overhead, and enables parallelization with predictable timing. Together, these approaches highlight the trade-off between software flexibility and hardware specialization: software techniques deliver immediate gains within existing platforms, while hardware units promise higher throughput and efficiency at the cost of additional area and design complexity. By combining both perspectives, the project emphasizes the impact of thoughtful co-design in achieving secure and efficient cryptographic performance.