Crypto_pyaes report

Introduction

This benchmark is a python implementation if the AES block-cipher in CTR mode.

AES – **A**dvanced **E**ncryption **S**tandard is a symmetric block-cipher that is widely used to secure digital data. It was established in 2001, mainly to replace DES (due to increase computing size, making DES breakable using a brute-force attack within a matter of hours / minutes. AES operates on fixed data blocks of 128 bits and supports key lengths of 128/192/256 bits. AES basically repeats 4 major functions for encryption: sub bytes, shift rows, mix columns, and XOR with some subset of the key. Since the algorithm is symmetric, decryption is mainly reverse order of the encryption steps.

I picked AES due to its enormous usage and popularity worldwide, for example, using ChatGPT I asked: "roughly, how much data is being encrypted and decrypted per day using AES?"

And the answer was astonishing (33 EB/day is the recent traffic online overall):

33 EB/day × (~0.85 HTTPS) × (~0.8 AES share) ≈ **22–25 EB/day** of AES in transit

Moreover, AES employs some heavy calculation for each round, where each round involves repeated lookups, heavy arithmetic, and of course, a lot of loops.

AES is being used vastly around the world, and improving this workload might have a real impact on world applications.

On the technical side of things, AES also offers few different types of calculations: bitwise operations (e.g. xor, compute bound) table lookups (memory access bound), and even some linear algebra (e.g. mix columns operations). All those calculations can be optimized using a different skill set, as they touch different aspects of computer architecture.

In addition, offering a hardware accelerator for the AES operations/ part of its operations sounds to me like an interesting part, and a very feasible task as well.

In my project, we will use flamegraph as the "front hint" on where I should focus to improve the benchmark performance, analyze the "hot lines" assembly (well, its byte code), and I will show a few improvements to those lines, while keeping the algorithm correct, but faster 😊

In addition, I will offer a hardware accelerator, that will enable users to speed up the AES calculation even more, and will analyze its PPA (Power, performance Area).

AES overview

As discussed briefly on the introduction section, AES is a symmetric encryption algorithm widely used throughout the world.

AES supports three length of keys: 128 bits, 192 bits, and 256 bits. The AES operates on rounds. Meaning that each round takes the previous round output and pass the input through the different stages of AES. The number of rounds changes, depending on the length of the key:

# key length	# rounds
128	10
192	12
256	14

Each round consists of four main transformations: substitution bytes, shift rows, mix columns, and key addition.

Substitution bytes for example is critical, since it is the only part in the transformation that is not linear. (if it was linear, the entire encryption could have been easily reversed, hence, not really a strong encryption). Mix columns for example is important, since it is the reason that even a single bit change will influence the entire 128 bits encryption. (That means that changing one bit of the input / key will change the entire encrypted data).

AES modes

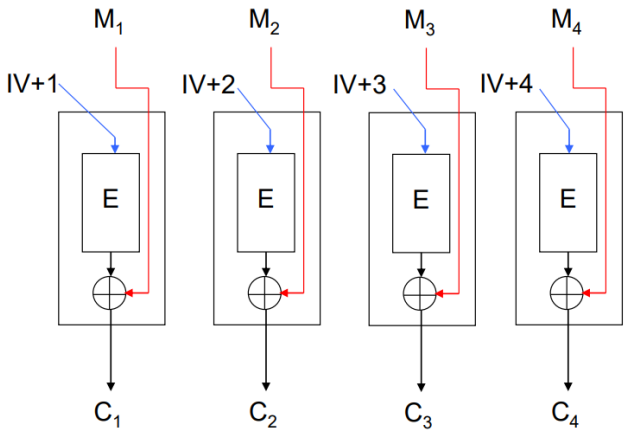
AES offers different types of modes of operations:

mode	Description
ECB	Each 128-bit block is encrypted independently with the same key
CBC	Each block of plaintext is XORed with the previous ciphertext block before encryption. The first block uses an Initialization Vector (IV).
CFB/ OFB	Turns AES into a self-synchronizing stream cipher. Encrypts the previous ciphertext (or IV) and XORs with the plaintext / output.

CTR	Converts AES into a parallelizable stream cipher by encrypting a counter value combined with a nonce.

Each operation has different pros and cons, and in the benchmark itself they used CTR. The main advantage of the CTR is that it is fully parallelizable (since you compute each round starting keys in advance).

CTR can be explained using the following scheme:



Benchmark initial analysis

The benchmark is using CTR mode. To speed up the calculation process, what pyaes does, like similar other compute heavy workloads is using the classic trade off : computation vs memory. To save calculations, many AES implementations, like the one here – use a precomputed results of the entire operations (4 steps we already mentioned previously). For each input byte, we already know the output. Thus, each AES operation is now a "simple" array lookup (and also some XOR's operations). This is a classis trade off between calculation and memory. In Python, this approach gains significant speed up, however, array lookup in Python and XOR is slower than C approach (due to Python objects encapsulation and memory management).

First setup steps

Pyperformance benchmark evaluates the AES as follows:

Initialize a dummy text, with roughly 23k bytes.

Uses a 128-bit key.

Then, the benchmark runs on a loop with the following steps:

1. Creates an AES object – CTR mode.
2. Encrypt the dummy text

3. Decrypt the dummy text back to clear text.
4. Destroys the AES object for the next loop.

In addition, the benchmark makes sure that the clear text is equal to the decrypted text.

First, what I did was to look at the source code of the AES implementation, so that I would understand how they implemented the algorithm.

I saw that the implementation uses the mentioned T tables lookup implementation, and I suspected that the most amount of time spent on the algorithm will be getting the values from the tables, since python indexing is relatively slow, as well as the fact that this is most of the job "left" to do.

Then, in order to understand verify where the hot lines in the code are, I used py-spy library which shows which lines the benchmarks spends the most time on, on a python level.

Unoptimized flame graph version

I ran the benchmark, (without the overhead of pyperformance, since I wanted to look into the AES code only)

And got the following flamegraph:

As we can see, the flamegraph is divided into two almost identical parts: the encryption part, and the decryption part (which makes sense, since in CTR mode the encryption and the decryption is the same) main hot lines which are repeated into both of those operations are lines 219 – 224, and 229 -233.

If we look at the corresponding lines, we reach to this for loop:

```
216         # Apply round transforms
217         for r in xrange(1, rounds):
218             for i in xrange(0, 4):
219                 a[i] = (self.T1[(t[ i          ] >> 24) & 0xFF] ^
220                         self.T2[(t[(i + s1) % 4] >> 16) & 0xFF] ^
221                         self.T3[(t[(i + s2) % 4] >>  8) & 0xFF] ^
222                         self.T4[ t[(i + s3) % 4]        & 0xFF] ^
223                         self._Ke[r][i])
224             t = copy.copy(a)
```

```
227         result = [ ]
228         for i in xrange(0, 4):
229             tt = self._Ke[rounds][i]
230             result.append((self.S[(t[ i           ] >> 24) & 0xFF] ^ (tt >> 24)) & 0xFF)
231             result.append((self.S[(t[(i + s1) % 4] >> 16) & 0xFF] ^ (tt >> 16)) & 0xFF)
232             result.append((self.S[(t[(i + s2) % 4] >>  8) & 0xFF] ^ (tt >>  8)) & 0xFF)
233             result.append((self.S[ t[(i + s3) % 4]        & 0xFF] ^  tt       ) & 0xFF)
234
```

As suspected, these are indeed the heavy lines. Which consists of the main work that we need to do in AES.

Lines 219 – 224 consist of all of the rounds, with exception of the last round, which is a bit different (and corresponding to lines 229 – 233).

optimizations pool

First, I would like to point out a few optimizations options I opted not to do:

Usage of threads – most of the time, threads is the first naïve answer on how to speed things up. However, specifically in python, each thread is limited to python's GIL (Global interpreter lock) which means that only one thread can execute bytecode at the same time. Since the loops we have here is pure python, without any I/O bound – threads will have to serialize. Thus, I suspect that the main gain coming from thread will not be that significant. However, I can use process, which "removes" the GIL limitation. However, processes are much heavier than threads. To conclude, due to the native bottleneck we have in this benchmark, I decided not to use threads.

Usage of AES instruction set - just like hardware accelerator, and due to the fact, the AES is extremely popular, X86 CPUs (both Intel and AMD) offers, and AES instruction set. ISA specific instructions for sure will speed up our work, **but** it is really dependent on the CPU you run, and the presented code will not be faster on ARM chips, such as Apple M series. Thus, I decided not to go with this path.

Instead, I will present two approaches I took, one is pure python optimizations, with no extra libraries needed, just straight forward optimizations that it was very easy to understand.

The second will take the first approach, but on a deeper level, which will mainly revolve around using more efficient Python libraries (hint: they are written in pure C/ C++) and usage of JIT compilers for the "hot lines".

Examine the byte code of the hot lines

I wanted to understand why the lines presented are so slow, since, at least in my pure intuition, they are just list access, and xor. Both should be relatively quick

instructions in a decent CPU (XOR is easy, and list access should be very good with proper Cache).

What I did was to inspect the raw python byte code for those specific lines.

I used the dis module in python, which provides byte code for functions. Down below, we can see the huge overhead the loop creates:

For example, the following byte code is for line:

self.T2[(t[(i + s1) % 4] >> 16) & 0xFF]

```
LOAD_DEREF      self
LOAD_ATTR       T2
LOAD_FAST       t
LOAD_FAST       i
LOAD_FAST       s1
BINARY_OP       +               # i + s1
LOAD_CONST      4
BINARY_OP       %               # (i + s1) % 4
BINARY_SUBSCR                   # t[(i+s1)%4]
LOAD_CONST      16
BINARY_OP       >>              # >> 16
LOAD_CONST      255
BINARY_OP       &               # & 0xFF
BINARY_SUBSCR                   # T2[...]
```

These lines are repeated, and some are redundant. For example, we see that each line is doing repeated attribute lookup, and this could easily be avoided by doing the attribute lookup once outside of the loop. In addition, there is the calculation of the rotation in the algorithm, which is predictable and static. This can also be hardcoded and removed.

To conclude, my focus is; after understanding why we spend so much time on this loop, is to make it simple.

In addition, I will also employ a loop unrolling for the inner loop – this a loop for 4 iterations only, creating a loop in Python is heavy, and unrolling it 4 times will also reduce overhead (on normal CPU loops translate into branches which in turn split the instructions, which increase FE bound).

All in all, this should yield a decent performance improvement.

Now, for the code optimizations:

```
# first, get the needed attributes only ones, and not for each loop
T1,T2,T3,T4,Ke, S = self.T1, self.T2, self.T3, self.T4, self._Ke, self.S
t0, t1, t2, t3 = t[0], t[1], t[2], t[3]
```

Before the major loop body, I'm taking the needed variables from the class. Taking values from a class takes up a decent overhead, so it is best to do it only

once. (you can see in the byte code snippet that you have LOAD ATTR for example, that will be repeated in all the loop lines).

Next up, I unroll the inner loop, with removing the unneeded code that was written purely to make the code generic.

```
    # lets unroll the loop
    a0 = T1[(t0 >> 24) & 0xFF] ^ T2[(t1 >> 16) & 0xFF] ^ T3[(t2 >>  8) & 0xFF] ^ T4[ t3 & 0xFF] ^ Ke[r][0]
    a1 = T1[(t1 >> 24) & 0xFF] ^ T2[(t2 >> 16) & 0xFF] ^ T3[(t3 >>  8) & 0xFF] ^ T4[ t0 & 0xFF] ^ Ke[r][1]
    a2 = T1[(t2 >> 24) & 0xFF] ^ T2[(t3 >> 16) & 0xFF] ^ T3[(t0 >>  8) & 0xFF] ^ T4[ t1 & 0xFF] ^ Ke[r][2]
    a3 = T1[(t3 >> 24) & 0xFF] ^ T2[(t0 >> 16) & 0xFF] ^ T3[(t1 >>  8) & 0xFF] ^ T4[ t2 & 0xFF] ^ Ke[r][3]
    t0, t1, t2, t3 = a0, a1, a2, a3
```

For example, (s1, s2, s3) = [1, 2, 3] so instead of the original code that did:

(i + s1) – I can manually calculate the result and remove a lot of redundancy.

Lastly, I also replaced the original t = copy.copy(a), with a manual assignment. Copy is costly – it is a function call, it allocates memory, it is not needed. Instead, I used 4 local variables, which use STORE_FAST byte code, which is much faster.

Lastly, I wrote the last round without any loops, as it's only loop of 4:

```
    res = [0] * 16 # we can save appends here if we declare the size from the start
    k0, k1, k2, k3 = Ke[rounds]

    # i = 0
    res[0] = (S[(t0 >> 24) & 0xFF] ^ (k0 >> 24)) & 0xFF
    res[1] = (S[(t1 >> 16) & 0xFF] ^ (k0 >> 16)) & 0xFF
    res[2] = (S[(t2 >>  8) & 0xFF] ^ (k0 >>  8)) & 0xFF
    res[3] = (S[  t3       & 0xFF] ^  k0       ) & 0xFF

    # i = 1
    res[4] = (S[(t1 >> 24) & 0xFF] ^ (k1 >> 24)) & 0xFF
    res[5] = (S[(t2 >> 16) & 0xFF] ^ (k1 >> 16)) & 0xFF
    res[6] = (S[(t3 >>  8) & 0xFF] ^ (k1 >>  8)) & 0xFF
    res[7] = (S[  t0       & 0xFF] ^  k1       ) & 0xFF

    # i = 2
    res[8] = (S[(t2 >> 24) & 0xFF] ^ (k2 >> 24)) & 0xFF
    res[9] = (S[(t3 >> 16) & 0xFF] ^ (k2 >> 16)) & 0xFF
    res[10] = (S[(t0 >>  8) & 0xFF] ^ (k2 >>  8)) & 0xFF
    res[11] = (S[  t1       & 0xFF] ^  k2       ) & 0xFF

    # i = 3
    res[12] = (S[(t3 >> 24) & 0xFF] ^ (k3 >> 24)) & 0xFF
    res[13] = (S[(t0 >> 16) & 0xFF] ^ (k3 >> 16)) & 0xFF
    res[14] = (S[(t1 >>  8) & 0xFF] ^ (k3 >>  8)) & 0xFF
    res[15] = (S[  t2       & 0xFF] ^  k3       ) & 0xFF
```

And predefined the size of the list res – since I know the size will be 16, and I don't need to call append like the original algorithm did.

Second key optimization I decided to do is the usage of NumPy + numba compile JIT. First, there is no need to really elaborate on NumPy, but I'll give a short outline of it – it is a python library with a huge emphasis on speed and efficiency, since it uses C / Fortran library under the hood, and avoids the huge python interpreter overhead. NumPy lacks the flexibility of lists (we need to define the array size from the start) but for AES the array size is known and fixed, so it is easy to make the needed changes when I want to use NumPy in our project. It also allows for vector operations instead of pure python code.

Numba is a JIT – Just in time compiler. It uses LLVM to turn Python code (and in our case, the encrypt function which does the heavy lifting) into a very optimized machine code. This should give us a decent boost in performance, this boost "bypasses" the need to go line by line with the interpreter, injecting machine code (pure assembly) directly into the CPU.

Main changes, will be to take the previous changes I made (unroll, avoid copy) to the next level – which will be implementing the encrypt function to use NumPy and Numba on top of it:

```
@njit(cache=True)
def _encrypt_block_numba(plaintext_u8, Ke_u32, T1, T2, T3, T4, S):
```

I created a wrapper function, and told Numba that we will compile this function on the fly, and also added cache=True, since I want this to be saved and reused (we are running the benchmarks on a loop)

```
    # ---- Main rounds ----
    for r in range(1, rounds):
        # a0
        a0 = (T1[(t0 >> 24) & 0xFF] ^
              T2[(t1 >> 16) & 0xFF] ^
              T3[(t2 >>  8) & 0xFF] ^
              T4[(t3      ) & 0xFF] ^
              Ke_u32[r,0])

        # a1
        a1 = (T1[(t1 >> 24) & 0xFF] ^
              T2[(t2 >> 16) & 0xFF] ^
              T3[(t3 >>  8) & 0xFF] ^
              T4[(t0      ) & 0xFF] ^
              Ke_u32[r,1])
        # a2
        a2 = (T1[(t2 >> 24) & 0xFF] ^
              T2[(t3 >> 16) & 0xFF] ^
              T3[(t0 >>  8) & 0xFF] ^
              T4[(t1      ) & 0xFF] ^
              Ke_u32[r,2])
        # a3
        a3 = (T1[(t3 >> 24) & 0xFF] ^
              T2[(t0 >> 16) & 0xFF] ^
              T3[(t1 >>  8) & 0xFF] ^
              T4[(t2      ) & 0xFF] ^
              Ke_u32[r,3])

        t0, t1, t2, t3 = a0, a1, a2, a3
```

next section the main rounds, and they are mainly the same the previous optimization work I made, with the only exception is that now we are running on NumPy arrays, which in turn will provide a huge performance boost, with the addition of the option to use vectorize operation in the assembly code – with the help of Numba.

```
# ---- Final round  ----
out = np.empty(16, dtype=np.uint8)

tt = Ke_u32[rounds,0]
out[ 0] = np.uint8((S[(t0 >> 24) & 0xFF] ^ (tt >> 24)) & 0xFF)
out[ 1] = np.uint8((S[(t1 >> 16) & 0xFF] ^ (tt >> 16)) & 0xFF)
out[ 2] = np.uint8((S[(t2 >>  8) & 0xFF] ^ (tt >>  8)) & 0xFF)
out[ 3] = np.uint8((S[(t3      ) & 0xFF] ^  tt       ) & 0xFF)

tt = Ke_u32[rounds,1]
out[ 4] = np.uint8((S[(t1 >> 24) & 0xFF] ^ (tt >> 24)) & 0xFF)
out[ 5] = np.uint8((S[(t2 >> 16) & 0xFF] ^ (tt >> 16)) & 0xFF)
out[ 6] = np.uint8((S[(t3 >>  8) & 0xFF] ^ (tt >>  8)) & 0xFF)
out[ 7] = np.uint8((S[(t0      ) & 0xFF] ^  tt       ) & 0xFF)

tt = Ke_u32[rounds,2]
out[ 8] = np.uint8((S[(t2 >> 24) & 0xFF] ^ (tt >> 24)) & 0xFF)
out[ 9] = np.uint8((S[(t3 >> 16) & 0xFF] ^ (tt >> 16)) & 0xFF)
out[10] = np.uint8((S[(t0 >>  8) & 0xFF] ^ (tt >>  8)) & 0xFF)
out[11] = np.uint8((S[(t1      ) & 0xFF] ^  tt       ) & 0xFF)

tt = Ke_u32[rounds,3]
out[12] = np.uint8((S[(t3 >> 24) & 0xFF] ^ (tt >> 24)) & 0xFF)
out[13] = np.uint8((S[(t0 >> 16) & 0xFF] ^ (tt >> 16)) & 0xFF)
out[14] = np.uint8((S[(t1 >>  8) & 0xFF] ^ (tt >>  8)) & 0xFF)
out[15] = np.uint8((S[(t2      ) & 0xFF] ^  tt       ) & 0xFF)

return out
```

and here the final round, again, like the previous work, but with Numpy on top of it.

Next, I used a script to fetch the assembly, and for example for the line:

a0 = T1[t0>>24] ^ T2[t1>>16] ^ T3[t2>>8] ^ T4[t3] ^ Ke[r,0]

I got (and asked ChatGPT to add some comments):

```
shrq    $24, %rdx               ; rdx = (t3>>24)        (used later)
shrl    $14, %eax               ; eax ~ ((t0>>24)<<2)   (byte*4 address)
andl    $1020, %eax
movq    296(%rsp), %r10         ; r10 = T1
movl    (%r10,%rax), %ecx       ; ecx = T1[(t0>>24)]
movq    240(%rsp), %rdi         ; rdi = T2
xorl    (%rdi,%rdx,4), %ecx     ;       ^ T2[(t3>>24)]  [used later again]

shrq    $24, %rsi               ; rsi = (t0>>24)    (prep for a1 later)

movq    %r8, %rbx               ; rbx = t1
movl    %ebx, %eax
shrl    $14, %eax               ; eax ~ ((t1>>24)<<2)
andl    $1020, %eax
movl    (%r10,%rax), %r15d      ; r15d = T1[(t1>>24)]
xorl    (%rdi,%rsi,4), %r15d    ;        ^ T2[(t0>>24)]
movzbl %bl, %eax
movq    %rax, 24(%rsp)          ; spill (t1&0xFF)
movq    %r11, %rdx              ; r11 = t2 (from previous iter)
movzbl %dh, %eax                ; eax = (t2>>8)&0xFF
movq    352(%rsp), %rsi         ; rsi = T3
xorl    (%rsi,%rax,4), %r15d    ;        ^ T3[(t2>>8)]
movzbl %bh, %eax                ; eax = (t1>>8)&0xFF
movq    408(%rsp), %r12         ; r12 = T4
xorl    (%r12,%ebp,4), %r15d    ;        ^ T4[(t3)&0xFF]
...
xorl    -12(%r9), %r15d         ;        ^ Ke_u32[r,0]
```

To understand, this code – which is 1 inner loop (which was unrolled)

Contains **just 27 assembly instructions.** In the previous change that I did (which already removed some bytecode), I asked ChatGPT to estimate the number of assembly instructions, and this is the answer I got:

roughly **1,500–3,100 CPU assembly instructions** for that exact sequence on **CPython**.

Results section

To gather the results properly, what I did was to create a script with three stages:

1. Run the native python AES implementation
    a. Run that creates the flame graph
    b. Run perf stat for 5 times (between each run I flush the cache)
    c. Run the benchmark using pyperformnace wrappers.
2. Run the first optimized python AES implementation
    a. Run that creates the flame graph
    b. Run perf stat for 5 times (between each run I flush the cache)
    c. Run the benchmark using pyperformnace wrappers.
3. Run the second optimized python AES implementation
    a. Run that creates the flame graph
    b. Run perf stat for 5 times (between each run I flush the cache)
    c. Run the benchmark using pyperformnace wrappers.

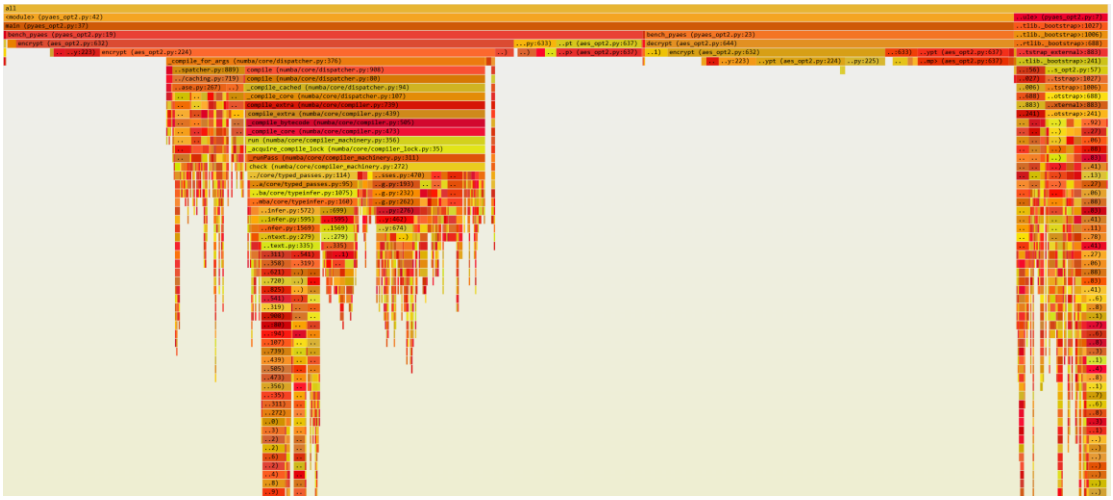
Optimized flame graph version 1

This is the first version flame graph:

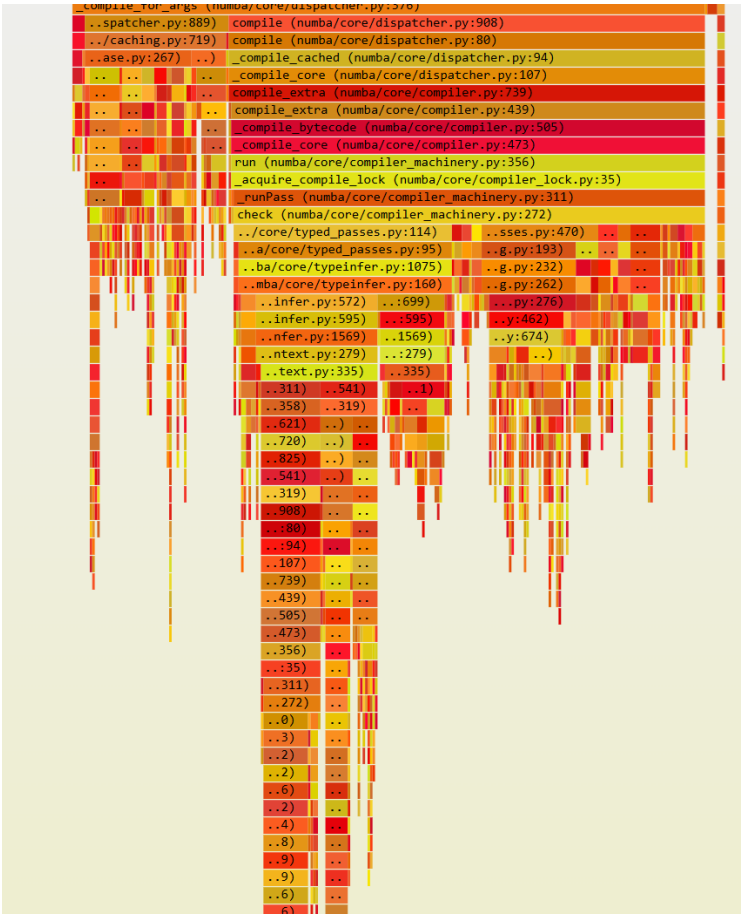
What we clearly see is that now the runtime is evenly distributed between the unrolled lines, which makes total sense, since now the computation is spread across different lines. In addition, the copy operation that was replaced with swap now costs less. (2.24% > 1.1%).

Optimized flame graph version 2

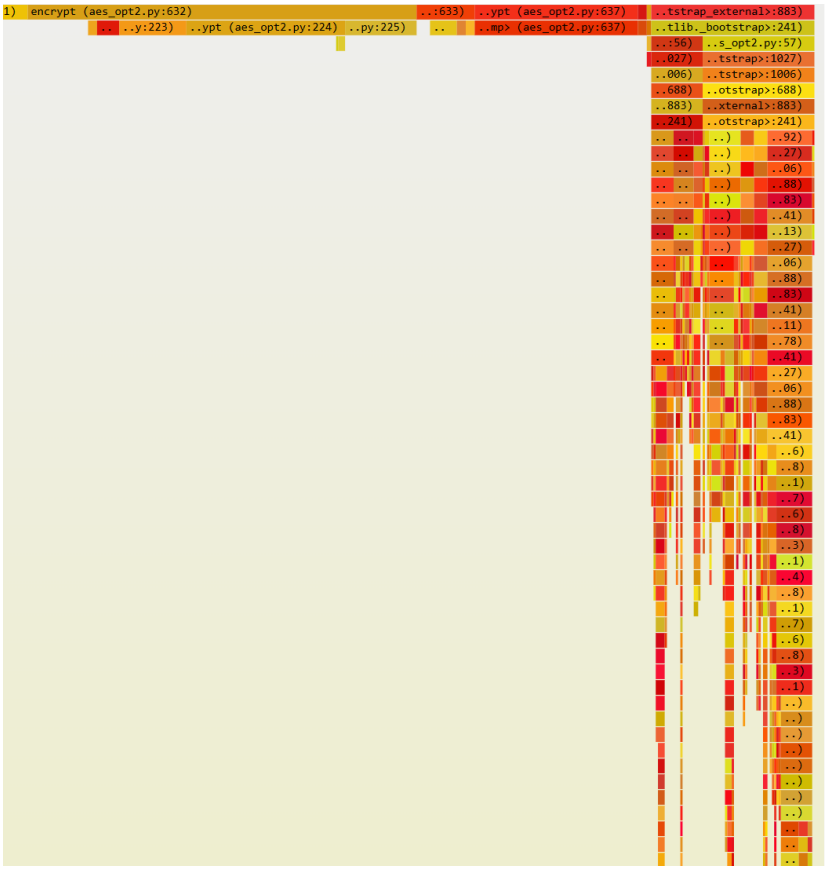
This is the second version flame graph:



The second flame graph contains a **lot** of information, lets try to zoom in a bit:



this part is a zoom in on the left part of the flame graph, and what we can see is that most of the work is on numba uitls and compile functions. This is some decent overhead; **however**, we will later see that it is worth it, and also that this is a one time overhead for each AES object creation. Since we later see that for the decrypt function, which for CTR mode, as explained, is symmetric with encrypt – the time it takes to run the same function is shorter. This make sense, since the first run of AES runs as byte code, but on the next stages (and runs) we will use a compiled version. If we zoom in on the second part of the flame graph we indeed see that it is now faster:



I do want to clarify that the speedup Numba provides might have been even bigger, since in the benchmark we use, AES object is being destructed and constructed for each iteration. On a normal version, I suspect that the speed up would have been bigger.

Numerical results

Full results will be attached to this report on an HTML and excel, here is the full results attached as a table:

	pyaes_clean	pyaes_opt	pyaes_opt2
time	60.18	45.35	6.31
instructions	372,868,099,444	293,980,849,497	38,831,029,761
cycles	156,219,787,035	117,723,432,614	30,675,604,409
IPC	2.39	2.50	1.27
context-switches	184.00	153.60	725.00
page-faults	12991.80	5660.80	35100.80
branches	94,858,787,161	75,801,414,107	8,699,136,704
branch-misses	311,479,352	219,708,360	36518528.20
L1-dcache-loads	87,130,684,252	64,014,087,531	8,836,779,338
L1-dcache-load-misses	444,117,324	323,044,504	254,816,828
LLC-loads	26636483.40	19675232.40	28637104.00
LLC-load-misses	167637.00	128563.20	1112054.60
dTLB-loads	87,135,513,438	64,052,016,711	8,966,818,319
dTLB-load-misses	4693632.40	4265469.40	3415343.80
iTLB-loads	5186091.80	2090725.20	36975271.80
iTLB-load-misses	1053577.20	604830.80	4181043.00
Speedup	1.00	1.33	9.54

Pyaes clean is running the benchmark without any modifications to the AES library, pyaes_opt is the first version – only code changes, and pyaes_opt2 is the second version which provides Numba + NumPy optimizations.

Looking at the results, it is clear that the second version achieves a tremendous speedup of 9.5x! in addition, the first version also provides a decent speedup of 1.33x. what is a bit odd, is that although the second version achieves tremendous speedup, the IPC of that version is lower by half~. We can clearly see that it is due to an increase in LLC loads, context switches, page-faults, and ITLB misses. If I think about it and correlate these results with the unique flame graph we got, it makes sense. This is mainly because Numba offloads a lot of the work to JIT functions, LLVM functions, this increases the code fragmentation significantly. Instead of running the entire code from the python bytecode, the first run (which is encrypt) runs pure python bytecode, goes to Numba functions, so it will JIT this function, and the next call, goes to the JIT version. We can clrealy see that the DTLB loads shrink massively – since we are working on contiguous arrays, no more python objects.

I suspect that the main increase in these metrics is mainly due to the "warmup" stage which is significant here. Since, although we run the benchmark on a loop, each iteration creates a new AES object. If we were to change it so that the AES object will be the same in all of the loops, I'm certain that the huge warmup overhead caused by NumPy array allocations, Numba JIT, will pay its dividends. Right now we run the efficient JIT function only once per warmup stage (again – as stated – in each encrypt there is a decent overhead caused by LLVM + Numba util that creates the JIT compiled function, then in decrypt we use that function) if we were to keep the AES object alive in all of the loops – the overhead would have been negligible.

To conclude, while some metrics increase, due to the increase the codebase (jumping from Python code, Numpy code, Numba code, etc') there are a lot of metrics that gain from this optimzition, and the clear one which is time – improved a lot!

The first version, which is straightforward Python code optimization, shows clear results in all the metrics, and speedup is also nice. This makes sense, since we touched on the core heart of the benchmark, and optimized it through careful analysis of the bottleneck. The first version doesn't require any new library and is very easy to deploy. The second version achieves better performance boost but requires some more "getting hands dirty" in terms of code changes.

Hardware accelerator proposal

As I already mentioned in the introduction section, on X86 there is already a unique instruction set for AES, due to its increasing popularity. Therefore, I believe that a successful hardware accelerator should opt to replace those instructions, this means that, if the hardware will be good, it can be embedded into all of x86 processors in the world, will offer backward compatibility, and will also allow offer high speedup compared to traditional calculation in a CPU, without any special hardware added for it. It will take up some decent area in the CPU, so my suggestion would be to only add this hardware on a special CPU (for those who require cryptography calculations) if we look at the AES instruction set we see the following instructions (taken from Wikipedia)

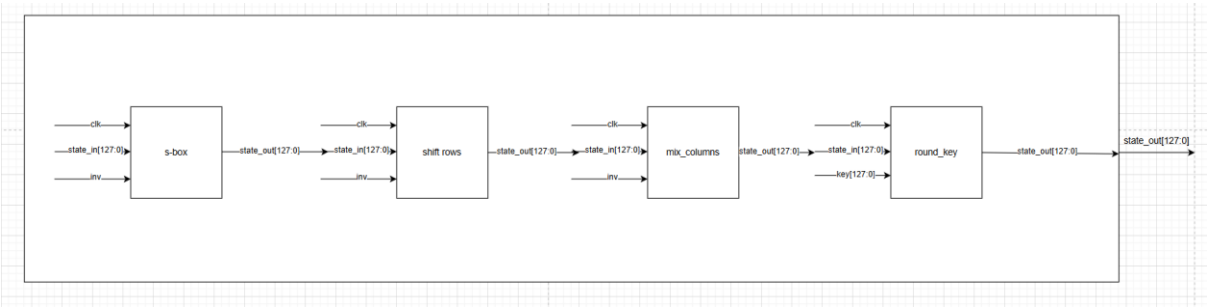
AESENC	Perform one round of an AES encryption flow
AESENCLAST	Perform the last round of an AES encryption flow
AESDEC	Perform one round of an AES decryption flow
AESDECLAST	Perform the last round of an AES decryption flow
AESKEYGENASSIST	Assist in AES round key generation[note 1]
AESIMC	Assist in AES decryption round key generation. Applies Inverse Mix Columns to round keys.

Since I want this hardware to be able to sit on a SOC of a CPU, I will not take a T-table approach. This is mainly because will take up a lot of chip memory (which is very expensive in a CPU).

Instead, we will create a native pipeline of pure AES.

I will only focus on AESENC, mainly due to the fact that AESENCLAST is the same – just skips the mix columns part, so we can just add bypass to it.

The state diagram should look like this:



S-box implementation

The S-box implementation will be based on ROM. Meaning that we will have 16 independent S-boxes, that is done to support 16 bytes per round. Each S-box operates on 8 bits input. On a fully pipelined path, it should take 1 cycle (each cycle the 16 s-boxes produce the output of 128-bit input)

It takes as input clk, state_in which is the input from plaintext / previous round and inv bit – which is used for decryption (the inverse operation)

Shift rows

Shift rows is an extremely easy task in hardware, since they are just moving wires around. 0 combinatorial logic, so only small RC delay (the time the signal travel from place to place).

Mix columns

Mix columns operation is implemented using a matrix multiplication. Where the 128 bits are organized as a 4x4 matrix. It will be multiplied by a 4x4 mix column matrix:

$$\begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix}$$

Inputs, is again the 128 bits to be going through the AES round, clk, and inv (again, for decryption the matrix is different)

Round key

Round key is one of the easy tasks to do in hardware, a simple bitwise xor between the key in that round, and the mixed bits that went through the AES stage.

PPA estimations

Regarding area, most of the area taken is the S-box. Which is 256 inputs times 8 bits – 256 bytes. Since we have 16, we get in total 4KB.

The rest of the area is at gate levels, and I'll neglect it.

I'll focus on throughput, and each cycle spit out 128 bit, let's assume 2GHZ frequency (modern CPU can reach to 5 GHZ easily), so, ignoring pipeline warm, we should get $f \times 128\ bits = 256GB\ per\ second$.

If we assume that we have a fully pipelined diagram (which is partly correct, since we have 10 rounds) for 25KB data size (which is roughly the amount of input we got in pyperformance) we get:

$$\frac{25 \times 1024 \times 8}{256 \times 10^9} \approx 0.8\,\mu s$$

Which is far faster than the software approach.

Summary

In this report I provided a thorough analysis of AES, went through the initial analysis of this benchmark, showcased why it is critical to make it better, used the techniques learned in class to understand where I should focus in optimizing this benchmark, proposed two solid software optimizations, that both have some advantages and disadvantages. In addition, I provided an end-to-end suggestion for an AES hardware accelerator, that is tailored to fully integrate into the popular X86 chips we have in the world, which can be sold separately as a special CPU for cryptography operations.