AES Acceleration

By Aly, Linfeng, Kavisha and Idriz

Goal and Objectives

The main focus of this project was to produce and accelerate the AES encryption process on an FPGA board that would allow for a significant speedup when compared to a standard program run on a computer's CPU. The following milestones were used to divide to work into achievable goals:

1. Produce functioning AES software
2. Have PS/PL functioning with the FPGA
3. Optimise the AES program
4. Apply the usage of multicore

In the process of completing these goals, relevant results pertaining to the speed and resource utilisation were taken note of so any acceleration could be identified or to find changes that were in fact detrimental to the speed of the processor.

Algorithm

The AES algorithm used was based on the US government's NIST AES algorithm. For the scope of this task, only the encryption code from this algorithm was required. It follows the following pseudocode:

Algorithm 1 Pseudocode for CIPHER()

```
1:  procedure CIPHER(in, Nr, w)
2:      state ← in                                          ▷ See Sec. 3.4
3:      state ← ADDROUNDKEY(state, w[0..3])                 ▷ See Sec. 5.1.4
4:      for round from 1 to Nr − 1 do
5:          state ← SUBBYTES(state)                         ▷ See Sec. 5.1.1
6:          state ← SHIFTROWS(state)                        ▷ See Sec. 5.1.2
7:          state ← MIXCOLUMNS(state)                       ▷ See Sec. 5.1.3
8:          state ← ADDRO[Go to page 14](state, w[4 ∗ round..4 ∗ round + 3])
9:      end for
10:     state ← SUBBYTES(state)
11:     state ← SHIFTROWS(state)
12:     state ← ADDROUNDKEY(state, w[4 ∗ Nr..4 ∗ Nr + 3])
13:     return state                                        ▷ See Sec. 3.4
14: end procedure
```

5.1.2 SHIFTROWS()

SHIFTROWS() is a transformation of the state in which the bytes in the last three rows of the state are cyclically shifted. The number of positions by which the bytes are shifted depends on the row index r, as follows:

$$s'_{r,c} = s_{r,(c+r) \bmod 4} \qquad \text{for } 0 \le r < 4 \text{ and } 0 \le c < 4. \tag{5.5}$$

5.1.1 SUBBYTES()

SUBBYTES() is an invertible, non-linear transformation of the state in which a substitution table, called an S-box, is applied independently to each byte in the state. The AES S-box is denoted by SBOX().

Let b denote an input byte to SBOX(), and let c denote the constant byte {01100011}. The output byte $b' = \text{SBOX}(b)$ is constructed by composing the following two transformations:

1. Define an intermediate value $\tilde{b}$, as follows, where b^{-1} is the multiplicative inverse of b, as described in Section 4.4:

$$\tilde{b} = \begin{cases} \{00\} & \text{if } b = \{00\} \\ b^{-1} & \text{if } b \neq \{00\}. \end{cases} \tag{5.2}$$

2. Apply the following affine transformation of the bits of $\tilde{b}$ to produce the bits of b':

$$b'_i = \tilde{b}_i \oplus \tilde{b}_{(i+4) \bmod 8} \oplus \tilde{b}_{(i+5) \bmod 8} \oplus \tilde{b}_{(i+6) \bmod 8} \oplus \tilde{b}_{(i+7) \bmod 8} \oplus c_i. \tag{5.3}$$

5.1.3 MIXCOLUMNS()

MIXCOLUMNS() is a transformation of the state that multiplies each of the four columns of the state by a single fixed matrix, as described in Section 4.3, with its entries taken from the following word:

$$[a_0, a_1, a_2, a_3] = [\{02\}, \{01\}, \{01\}, \{03\}]. \tag{5.6}$$

Thus,

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < 4, \tag{5.7}$$

so that the individual output bytes are defined as follows:

$$\begin{aligned} s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}). \end{aligned} \tag{5.8}$$

5.1.4 AddRoundKey()

AddRoundKey() is a transformation of the state in which a round key is combined with the state by applying the bitwise XOR operation. In particular, each round key consists of four words from the key schedule (described in Section 5.2), each of which is combined with a column of the state as follows:

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{(4*round+c)}] \quad \text{for } 0 \leq c < 4 \tag{5.9}$$

Algorithm 2 Pseudocode for KeyExpansion()

```
1:  procedure KeyExpansion(key)
2:      i ← 0
3:      while i ≤ Nk − 1 do
4:          w[i] ← key[4 ∗ i..4 ∗ i + 3]
5:          i ← i + 1
6:      end while                          ▷ When the loop concludes, i = Nk.
7:      while i ≤ 4 ∗ Nr + 3 do
8:          temp ← w[i − 1]
9:          if i mod Nk = 0 then
10:             temp ← SubWord(RotWord(temp)) ⊕ Rcon[i/Nk]
11:         else if Nk > 6 and i mod Nk = 4 then
12:             temp ← SubWord(temp)
13:         end if
14:         w[i] ← w[i − Nk] ⊕ temp
15:         i ← i + 1
16:     end while
17:     return w
18: end procedure
```

Investigation

When finding relevant information on the topic of AES encryption, we found that the information provided by the Federal Information Processing Standards (FIPS) Publication 197 published by the National Institute of Standards and Technology, had more than adequate information on the process of implementing the relevant functions and the order of which these function should be executed to produce the cypher.

Results

We collected results on three different solutions - which can be simplified to PS, PL Single Core and PL Multi Core. The PS is essentially just our software code with no optimisations. The PL Single core adds HLS pragmas (mainly unroll) throughout the whole aes code. It also feeds the input data into AXI-Lite to run this new aes code rather than calling the cipher function directly. The PL Multi Core follows the same logic as its single core counterpart, except it has multiple cores running at once to process the data. The table below shows a comparison between the first and second solution:

Metric	Optimized - PL Single Core	Unoptimized - PS
Platform	PL (RTL via AXI4-Lite) (100Mhz)	PS(A53 Cortex Vitis bare-metal) (553Mhz)
Latency (Cycles)	123	513
Latency (Effective)	110 ns	1401 ns
Iteration Interval(II)	12	1 (Key loops)
BRAM Usage	8	3
LUT Usage	3087	4617
FF Usage	610	885
S-box ROMs	8 instantiated	8 instantiated

Resource	Used	Available	Utilisation
LUTs	1865	117120	1.59%
Registers	2011	234240	0.86%
BRAM (RAMB18)	17	288	5.90%
DSPs	0	1248	0.00%
IOs	1	189	0.53%
Clock BUFG_PS	1	96	1.04%

From the table above we can note that the single core processor outperformed the pure software approach by a factor of 13. This can be credited to the simplification of some functions and further application of pragma directives within Vivado HLS, which enable the FPGA to both pipeline and parallelise elements of the encrypting process.

As can be seen when comparing results from the single core to this multicore setup, the multicore implementation performed with a higher latency than anticipated, providing a speed up. This is due in part to the provisioning overhead innate to having such a configuration as tasks need to be distributed to idle cores. There is a larger bottleneck in the form of the data delivery method, used to provide the FPGA with the bit stream from the computer, specifically with the usage of AXI-Lite rather than AXI-Stream. The potential to improve this aspect of the multicore can be found in the discussion section of the report.

Project Management

The tasks in this project were split among all members evenly. There was some double up in tasks, where some members would participate in pair coding sessions over discord (our main method of communication). Timeline wise, we stuck to it quite well. Preliminary work was finished in W7 and the overall work was finished at the end of Week 9 / beginning of week 10.

Discussion

AXI-Lite vs AXI-Stream

AXI-Lite is a simple data communication method use to allow the FPGA board to communicate with the host computer. It sends 32 bits of data per transaction, meaning it has low throughput, particularly when compared to AXI-Stream which is only limited by the bus size it is provided with, meaning it is able to send a continuous stream of data at a high throughput.
In the final implementation we were able to send data packets to the device via AXI-Lite and not AXI-Stream. This was likely a limiting factor, particularly for the multicore setup, as AXI-Lite's limited data transfer size meant for many operations we had to wait until the entire 128 bit data packet was transferred to our board for processing. If we had extra time, AXI-Stream would have likely been our next goal to reach in order to enable faster data transfer to the board, allowing us to take full advantage of the multicore setup we had devised and implemented, particularly for testing with even larger datasets.

Concluding Remarks

Overall, this was an interesting project to work on. It was an unexpected result of less speedup from the Multicore, but it gives us interesting ideas into further research and development in this area.