Pynqrypt: a FPGA-accelerated AES Implementation for PYNQ

FPGA101
Roberto Alessandro Bertolini

December 18, 2022

Abstract

Nowadays cryptography is a fundamental part of our daily life: it is used to protect and ensure the integrity of our data, to authenticate users and to provide secure communications. However secure encryption and decryption protocols come at a high cost in terms of computational power required, a requirement which might become a bottleneck in many applications and devices. In this report we will analyze how the use of FPGA accelerators can help overcome this problem in some cases, shifting the burden of the cryptographic operations from the CPU to the FPGA.

1 Introduction

The need of protecting and hiding data has been around for thousands of years, which makes cryptography and cryptoanalysis an especially wide field of study: there are countless algorithms and methods to encrypt and decrypt data, ranging from the classical techniques, like the Caesar cipher, to the modern AES encryption standard, up to the new quantum-secure ciphers. But what makes a good encryption algorithm? The Ceasar cipher, for example, is very easy to implement, but it is just as easy to break, by simply brute-forcing all the possible combinations. On the other hand, the one-time-pad is an unbreakable encryption technique, but it is almost never used in practice because it is very hard to implement correctly, since the key must be as long as the message itself, never reused, and kept secret by the communicating parties.
One of the most widely used encryption algorithms is the Advanced Encryption Standard (AES), which a symmetric-key block cipher, meaning that the same key is used for both encryption and decryption, and that it operates on blocks of a set size, which is 128 bits. AES is considered cryptographically secure, as there are known attacks that are able to break it, but they are all computationally infeasible and only marginally better than just brute-forcing all the possible key combinations.

1.1 AES-CTR

AES-CTR is a mode of operation based on the AES algorithm. It is a stream cipher, meaning that it can encrypt a stream of data of any length (not necessarily a multiple of the block size), and it is based on a counter, which gets incremented at each block encryption. A very important property of AES-CTR is that it is higly parallelizable, because each block can be

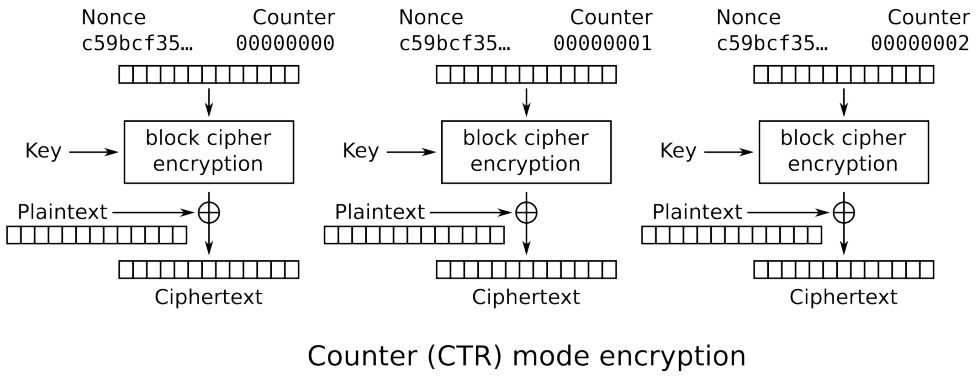


Figure 1: AES-CTR requires both a key and a nonce, which is a random value used to initialize the counter. For each block, the counter value is encrypted with the key, and the result is XORed with the plaintext to get the ciphertext. Then the counter is incremented by one. As XOR is its own inverse, the decryption flow is exactly the same. [1]

encrypted and decrypted indipendently from the others, just by knowing the counter value of that specific block. Another interesting behavior of AES-CTR is that the encryption and decryption operations are exactly the same, so the same code can be shared between the two operations. This makes AES-CTR a very good candidate for hardware acceleration, because the same implementation can be reused for both encryption and decryption, and because its parallelizability allows the FPGA to perform better.

1.2 The Hardware

The tests were conducted on a PYNQ Z2, an open-source development board for Xilinx Zynq SoCs, which features both a dual-core ARM CPU and a programmable FPGA fabric. The CPU runs at 650 MHz and, unlike most modern designs, lacks the AES instruction set extensions, which makes it particularly slow at performing AES operations. The FPGA runs at a variable frequency up to 250 MHz and is interconnected with the CPU via a high-speed AXI bus. The board also features 512 Mbytes of DDR3 RAM, with a maximum bandwidth of 1050 Mbps, an Ethernet port, a microSD card slot, and other peripherals.

2 Methodology

Since the main goal of this project was to evaluate whether the use of an FPGA accelerator can be used to improve the performance of AES-CTR, the CPU and the FPGA were compared by measuring the time they took to perform the same operation on the same data and block key. As in a real world scenario the data to be encrypted is usually of a variable length, the tests were performed on a set of 4 different data sizes, which are:

1. 16 bytes (1 block)
2. 1024 bytes (64 blocks)
3. 256 Kbytes (16 Kblocks)
4. 16 Mbytes (1 Mblocks)

The time measurements were repeated 10 times for each data size, and the average was then reported, excluding the highest and the lowest times.
As the FPGA implementation would be used through the Python API, we chose, for the sake of simplicity and consistency, to use a CPU implementation which featured Python bindings, which would enable use to run the benchmarks in the same environment one after the other.
The CPU implementation we chose was the one provided by the PyCryptodome library, which, while exposing a Python API, implements the AES algorithm in C and builds it from source code during the install process, so it should be able to leverage all the available architecture optimizations.
The FPGA implementation was written using Vitis HLS and the C++ language, and it was then exported to Vivado as an IP, from which a bitstream was generated and loaded onto the FPGA. As a secondary goal of the project was to evaluate how easily a software implementation can be ported to hardware, we decided not to use any of the already existing AES-CTR implementations for the FPGA, but to write our own from scratch, starting from one that would work on the CPU. With each iteration of implementation, we tried to improve the hardware performance by optimizing the code, documenting which changes gave a positive or negative impact on the performance. To ensure the correctness of the implementation, at each iteration we validated the results against both the previously mentioned PyCryptodome library, and the OpenSSL library, which is widely trusted and used in many applications.

3 Design Iterations

3.1 Iteration 1

The first iteration of the design was a very simple and generic implementation. A few modifications had to be made to the original code, in order to make it compile in Vitis HLS:

1. All the C++ standard library calls had to be replaced with generic C calls, as the HLS compiler does not support advanced C++ features (like std::copy, which had to be replaced with memcopy).

2. The original code defined a class, called Pynqrypt, which contained all the methods needed to perform the encryption and decryption operations. As Vitis HLS can expose only one top-level function as a Vivado IP, we had to write a new function which just instantiates the Pynqrypt object and calls the appropriate method.

3. A few interface pragmas had to be added to the previously mentioned function, so that the HLS compiler would know how to properly expose the function arguments in the generated IP.

This first implementation achieved the following results:

	CPU	Iteration 1
1 block	0.11 ms	0.16 ms
64 blocks	0.21 ms	0.42 ms
16 Kblocks	24.52 ms	67.86 ms
1 Mblocks	1641.08 ms	4333.76 ms
Max throughput	78 Mbps	29.54 Mbps

Table 1: Comparison between the CPU and Iteration 1 on the FPGA

As can be seen from the table, the FPGA implementation is significantly slower "out of the box", most likely due to the fact that the HLS compiler is not able to optimize the code much, due to the generic nature of the implementation.

3.2 Iteration 4

Iteration 4 was the last fully cross-platform iteration. In terms of optimization, we applied the following changes from Iteration 1:

1. The functions `Pynqrypt::ctr_compute_nonce`, `Pynqrypt::ctr_xor_block`, `Pynqrypt::aes_mix_columns`, `Pynqrypt::aes_shift_rows`, `Pynqrypt::aes_sub_bytes` were inlined, to enable some pipeline optimizations.

2. The loops in the `Pynqrypt::aes_mix_columns` and `Pynqrypt::aes_sub_bytes` functions were unrolled.

These were all minor and cost-free changes, so they didn't have a significant impact on the performance, as can be seen from the results:

	CPU	Iteration 1	Iteration 4
1 block	0.11 ms	0.16 ms	0.16 ms
64 blocks	0.21 ms	0.42 ms	0.34 ms
16 Kblocks	24.52 ms	67.86 ms	45.91 ms
1 Mblocks	1641.08 ms	4333.76 ms	2928.72 ms
Max throughput	78 Mbps	29.54 Mbps	43.71 Mbps

Table 2: Comparison between the CPU, Iteration 1 and Iteration 4 on the FPGA

3.3 Iteration 5

Iteration 5 saw a significant improvement in performance, mainly due to the switch from 16-element array of bytes to a 128 bit integer, implemented using the ap_uint<128>type. This change removed the need for a lot of loops in the code, which were previously used to perform bitwise operations on the array elements. The downside of this change is that the code is now platform-dependent, as the ap_uint type is specific to Xilinx FPGAs, and, as far as we know, closed source. Other than that, no further optimizations were made.

	CPU	Iteration 4	Iteration 5
1 block	0.11 ms	0.16 ms	0.17 ms
64 blocks	0.21 ms	0.34 ms	0.28 ms
16 Kblocks	24.52 ms	45.91 ms	33.08 ms
1 Mblocks	1641.08 ms	2928.72 ms	2107.78 ms
Max throughput	78 Mbps	43.71 Mbps	60.73 Mbps

Table 3: Comparison between the CPU, Iteration 4 and Iteration 5 on the FPGA

3.4 Iteration 6

Iteration 6 was the first to outperform the CPU implementation. The gains were achieved by optimizing the main loop function, removing a double copy

of the data from the input buffer to the current block state. This is due to the fact that ap_uint internally stores the data in little endian format, but the AES algorithm expects the data to be big endian, so the data had to be reversed before being encrypted, and then reversed again after the encryption. This reversal was previously performed by copying the whole 16-byte block from the input buffer to the local block variable, then looping over it to reverse the endianness. Now the reversal is performed by looping over the 16 bytes of the block, and copying them to the local variable in reverse order.

	CPU	Iteration 5	Iteration 6
1 block	0.11 ms	0.17 ms	0.15 ms
64 blocks	0.21 ms	0.28 ms	0.21 ms
16 Kblocks	24.52 ms	33.08 ms	24.72 ms
1 Mblocks	1641.08 ms	2107.78 ms	1573.01 ms
Max throughput	78 Mbps	60.73 Mbps	81.37

Table 4: Comparison between the CPU, Iteration 5 and Iteration 6 on the FPGA

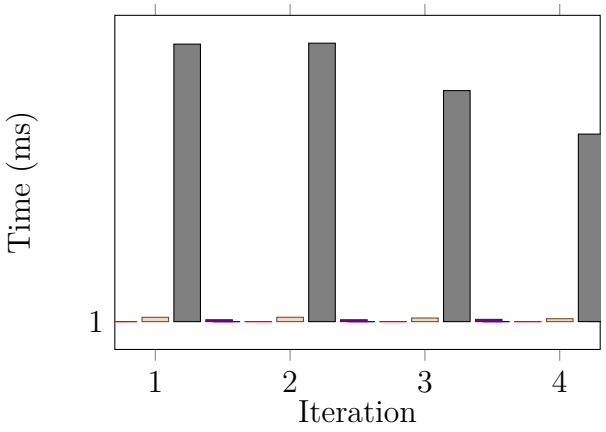
3.5 Iteration 7

Iteration 7 saw a big jump in performance. During the AES encryption flow, the block has to be substituted multiple times by performing a byte-by-byte lookup in a 256-element array, which is called the S-box. This operation is very expensive, as it requires a lot of memory accesses, and it is also very repetitive, as the same S-box is used multiple times. We tried multiple approaches to tackle this problem, but the one that gave the best results was applying an "ARRAY_RESHAPE" pragma to the array. This is an expensive operation, as it increases the memory footprint of the array, but it allows the compiler to optimize the accesses to the array.

3.6 Iteration 8

Applying the same "ARRAY_RESHAPE" pragma to two other arrays, which are used as lookup tables for fast multiplication, gave a huge performance boost.

3.7 Iteration 9



4 Results

We can see from the previous graph that the FPGA implementation is able to vastly outperform the CPU implementation, especially for larger data sizes, with just a bit of tweaking and optimization, and shouldn't increase overhead too much for smaller data sizes. The results also show that the performance of the FPGA implementation is somehow bottlonecked by the system, as the throughput doesn't scale linearly when the PL clock was increased. We suspect that the culprit is the DDR3 memory, which, as we mentioned before, has a maximum bandwidth of 1050 Mbps. In order to achieve a throughput of 441 Mbps, we actually need to transfer 882 Mbps of data from and to the memory, which is suspiciously close to its maximum bandwidth.

5 Comparison with Other Hardware

We have successfully proven that the FPGA accelerator is a viable solution for accelerating AES-CTR encryption on the Pynq-Z2 board, but our hardware choice was particularly unfavourable to

References

[1] WhiteTimberwolf. *CTR encryption*. File: `CTR_encryption_2.svg`. ***Public domain***, `via Wikimedia Commons`. 2013. URL: `https://commons.wikimedia.org/wiki/File:CTR_encryption_2.svg`.