Pynqrypt: a FPGA-accelerated encryption library for PYNQ

FPGA101
Roberto Alessandro Bertolini

December 27, 2022

Abstract

Data encryption and decryption is a computationally expensive task to perform, as most encryption algorithms aren't specifically optimized for running on modern computer architectures. In this report we will analyze how a very common algorithm, AES-CTR, can be reimplemented on a FPGA to provide a noticeable improvement in throughput on low-power or embedded devices, shifting most of the operations to the FPGA and thus freeing up the main processor. We will also note the various design considerations and the optimizations applied and how much they contributed to the overall improvement in throughput.

1 Introduction

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 use cases, especially in low-power and embedded devices where energy efficiency or thermal constraints severely hinder hardware performance.

There are two different categories of cryptographic algorithms: symmetric and asymmetric. Symmetric algorithms use the same key for both encryption and decryption, while asymmetric algorithms use a pair of two different keys, one for encryption and one for decryption. Asymmetric algorithms are generally used for authentication and key exchange, while symmetric algorithms are used for the actual encryption and decryption of data. The most common symmetric algorithm is the Advanced Encryption Standard (AES), which supersed the Data Encryption Standard (DES) in 2001 and became the de facto standard for symmetric encryption [3].

1.1 AES

AES [5] is a symmetric block cipher, which means that it encrypts data in blocks of a fixed size, which is 16 bytes (128 bits), with a key of 16, 24 or 32 bytes (128, 192 or 256 bits). It is based on a substitution-permutation network: during the encryption process, the block is XORed, shifted, substituted or mixed byte-by-byte multiple times, in a reversible process that is easy to compute but hard to bruteforce without knowledge of the key used. Since all of this is done using simple bitwise or mathematical operations, it

can be easily implemented on a FPGA, which makes it a perfect candidate for attempting to accelerate this task.
AES only defines the method for encrypting a single block, but it doesn't specify how to chain multiple blocks together, which is the reason why it is usually used in conjunction with a mode of operation, which defines how to encrypt a stream of data.

1.2 AES-CTR

AES-CTR [4] 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 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 is easily exploited by the FPGA.

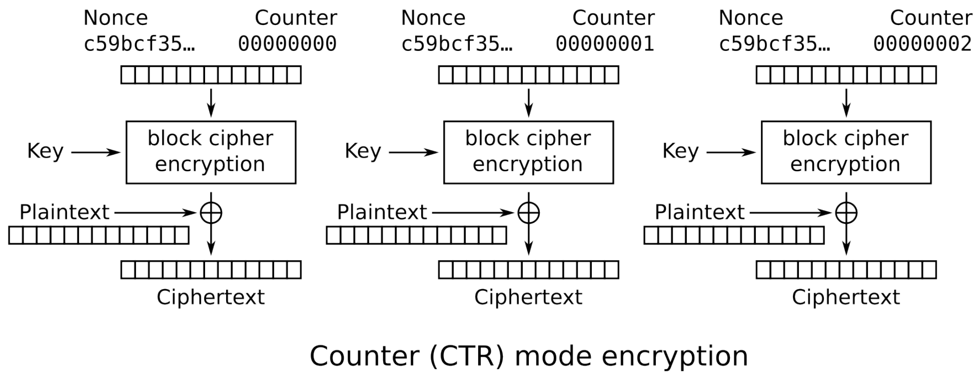

Figure 1: AES-CTR requires both a key and a nonce, which is a random value used to initialize the counter. For each block, the counter value is encrypted with the key, and the result is XORed with the plaintext to get the ciphertext. Then the counter is incremented by one. As XOR is its own inverse, the decryption flow is exactly the same. [6]

1.3 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 [2], 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. For brevity, we decided not to include iterations which didn't improve the performance significantly, nor to show code changes, but everything is available in the GitHub repository [1] tagged by iteration version. The benchmark results are available in the GitHub repository as well, in a Jupyter notebook [2].

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:

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

[1] `https://github.com/MrIndeciso/pynqrypt`

[2] `https://github.com/MrIndeciso/pynqrypt/blob/main/docs/results.ipynb`

	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

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

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

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 Mbps

3.5 Iteration 7

Iteration 7 saw a big jump in performance. During the AES encryption flow, the 16 byte block has to be "substituted" multiple times by performing a byte-by-byte lookup in a 256-element constant array, which is called the S-box. Previously Vitis HLS stored this array in a single BRAM, which meant that the lookup couldn't be performed in parallel, because the BRAM only has a single read port. By applying the pragma "ARRAY_RESHAPE" to the S-box array, we were able to split it into 256 single byte arrays, which occupied a BRAM each. With this change, the lookup can be performed in parallel, and the throughput increases significantly. This change comes at no extra cost, because the Pynq board has 630K of fast BRAM available to be used, but in other devices, if no BRAM is available, it will have to be built using LUTs, increasing the IP footprint significantly.

	CPU	Iteration 6	Iteration 7
1 block	0.11 ms	0.15 ms	0.16 ms
64 blocks	0.21 ms	0.21 ms	0.22 ms
16 Kblocks	24.52 ms	24.72 ms	19.16 ms
1 Mblocks	1641.08 ms	1573.01 ms	1216.49 ms
Max throughput	78 Mbps	81.37 Mbps	105.22 Mbps

3.6 Iteration 8

More lookup tables are used in the `Pynqrypt::aes_mix_columns` function, which is used to perform the MixColumns step of the AES algorithm. While lookup tables aren't strictly required in this step, they are used to avoid performing a lot of multiplications over Galois fields, which are computationally expensive. The same limitations of the S-box apply here, so we applied the same pragma optimization.

	CPU	Iteration 7	Iteration 8
1 block	0.11 ms	0.16 ms	0.16 ms
64 blocks	0.21 ms	0.22 ms	0.15 ms
16 Kblocks	24.52 ms	19.16 ms	7.60 ms
1 Mblocks	1641.08 ms	1216.49 ms	475.91 ms
Max throughput	78 Mbps	105.22 Mbps	268.96 Mbps

The FPGA is now more than three times faster than the CPU on the largest data size, which is a huge improvement and would net a perceivable speedup in real-world applications.

3.7 Iteration 9

For the last iteration we manually unrolled `Pynqrypt::aes_sub_bytes` just for the last encryption round, enabling some additional pipelining. Having reached a result we were satisfied with, we decided to stop optimizing the code and we moved on to raising the clock frequency of the PL. Even though Vitis HLS reported a maximum design frequency of around 136 MHz, we found the IP to be stable up to 200 MHz, a 100% increase in clock frequency from the default settings. We decided not test the IP at the maximum supported frequency of 250 MHz because our aim was not to push the limits of the hardware, but to observe how the performance scaled with the clock frequency. In a production environment we wouldn't recommend overriding the default clock frequency without adequate characterization of the IP, as it could lead to unexpected behavior.

	CPU	Iteration 8	Iteration 9
1 block	0.11 ms	0.16 ms	0.15 ms
64 blocks	0.21 ms	0.15 ms	0.14 ms
16 Kblocks	24.52 ms	7.60 ms	6.80 ms
1 Mblocks	1641.08 ms	475.91 ms	427.36 ms
Max throughput	78 Mbps	268.96 Mbps	299.51 Mbps

	Iter. 9	Iter. 9 (143 MHz)	Iter. 9 (200 MHz)
1 block	0.15 ms	0.16 ms	0.15 ms
64 blocks	0.14 ms	0.15 ms	0.15 ms
16 Kblocks	6.80 ms	5.44 ms	4.66 ms
1 Mblocks	427.36 ms	338.40 ms	290.04 ms
Max throughput	299.51 Mbps	378.25 Mbps	441.31 Mbps

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, while inflicting only a small penalty for the smallest data size. The results also show that the performance of the FPGA implementation is somehow bottlonecked by the system, as the throughput doesn't scale linearly with the increase in PL clock frequency. 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 data at a speed of 882 Mbps from and to the memory, which is close to the 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 favourable for this task. Modern x86_64 and ARM CPUs have specific hardware instructions for performing AES encryption (`AES-NI` [1] o x86_64 and `Cryptography Extensions` [2] on ARM), which are orders of magnitude faster than the software implementation and much faster than our FPGA implementation.

	Iter. 9 (200 MHz)	Ryzen 3 5300U
1 block	0.15 ms	0.01 ms
64 blocks	0.15 ms	0.01 ms
16 Kblocks	4.66 ms	0.33 ms
1 Mblocks	290.04 ms	22.45 ms
Max throughput	441.31 Mbps	5.94 Gbps

As we can see our FPGA is of no use in this scenario, but it is still a viable solution for other applications, such as embedded systems where the CPU is not powerful enough to perform the encryption in software, or where the CPU is already busy with other tasks.

References

[1] Intel Corporation. *Intel® Advanced Encryption Standard (AES) New Instructions*. URL: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.

[2] Arm Ltd. *ARM® Cortex®-A53 MPCore Processor Cryptography Extension*. URL: https://documentation-service.arm.com/static/5e9077ee8259fe2368e2acf7.

[3] Miles E. Smid. *Development of the Advanced Encryption Standard*. National Institute of Standards and Technology, 2021. URL: https://nvlpubs.nist.gov/nistpubs/jres/126/jres.126.024.pdf.

[4] National Institute of Standards and Technology. *Recommendation for Block Cipher Modes of Operation*. National Institute of Standards and Technology, 2001. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf.

[5] National Institute of Standards and Technology. *Specification for the Advanced Encryption Standard*. National Institute of Standards and Technology, 2001. URL: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf.

[6] 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.