Pynqrypt: a FPGA-accelerated AES Implementation for PYNQ

FPGA101
Roberto Alessandro Bertolini

December 15, 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 very hard to get right, 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 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 encrypted and decrypted indipendently from the others, just by knowing

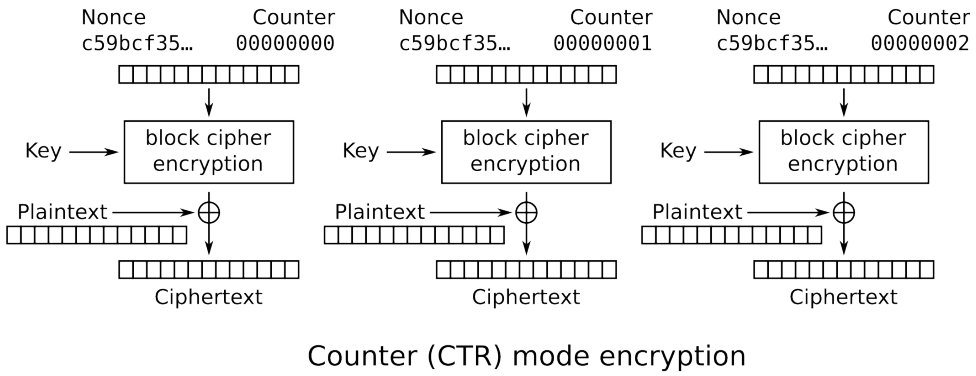

Figure 1: AES-CTR requires both a key and a nonce, which is a random value used to initialize the counter. For each block, the counter value is encrypted with the key, and the result is XORed with the plaintext to get the ciphertext. Then the counter is incremented by one.

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 up to 250 MHz and is interconnected with the CPU via a high-speed AXI bus. // TODO doublecheck this The board also features 512 Mbytes of DDR3 RAM, with a maximum bandwidth of 1050 Mbps.

2 Methodology

As the main goal of this project is 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 required 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.
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.
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.
The FPGA implementation was written using Vitis HLS and the C++ language, and it was then exported as a Vivado IP, which was then loaded as a bitstream through the Pynq framework. As one of the main goals of this project was to analyze how targeted optimizations can greatly improve the real world performance of an FPGA implementation, we decided to start from scratch and we wrote a purposedly unoptimized and generic encryption library. With each iteration of the design we then added optimizations and targeted features, noting the performance improvements and the changes made. 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 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:

Table 1: Iteration 1 results

Implementation	1 block	64 blocks	16 Kblocks	1 Mblock
CPU	0.0001 s	0.0001 s	0.0001 s	0.0001 s
Iteration 1	0.0001 s	0.0001 s	0.0001 s	0.0001 s

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. // TODO add code example here

   This first implementation achieved the following results: