Analyzing the Generation and Optimization of an FPGA Accelerator using High Level Synthesis

Sebastian Kaltenstadler
Ulm University
Ulm, Germany
sebastian.kaltenstadler@missinglinkelectronics.com

Stefan Wiehler
Missing Link Electronics
Neu-Ulm, Germany
stefan.wiehler@missinglinkelectronics.com

Ulrich Langenbach
Beuth University of Applied Sciences
Berlin, Germany
ulrich.langenbach@beuth-hochschule.de

Abstract—**Multi-Processor System-on-Chip FPGAs can utilize programmable logic for compute intensive functions, using so-called Accelerators, implementing a heterogeneous computing architecture. Thereby, Embedded Systems can benefit from the computing power of programmable logic while still maintaining the software flexibility of a CPU. As a design option to the well-established RTL design process, Accelerators can be designed using High-Level Synthesis. The abstraction level for the functionality description can be raised to algorithm level by a tool generating HDL code from a high-level language like C/C++. The Xilinx tool Vivado HLS allows the user to guide the generated RTL implementation by inserting compiler pragmas into the C/C++ source code. This paper analyzes the possibilities to improve the performance of an FPGA accelerator generated with Vivado HLS and integrated into a Vivado block design. It investigates, how much the pragmas affect the performance and resource cost and shows problems the tool has with coding style.**

I. Introduction

For modern computing systems it is getting more popular to use heterogeneous computer architectures to further increase computing power. There are multiple ways to compensate the stagnation of single core performance of CPUs, ranging from instruction set extensions, multi core processors and GPUs to coprocessors on FPGAs. Cryptographic and hashing functions for example can be accelerated on an FPGA. The advantages of an implementation on an FPGA are almost ASIC-like computing performance, quick adaption to new protocols and standards as well as low energy consumption [7]. To develop such a coprocessor, the logic of the algorithm has to be described with VHDL/Verilog on Register Transfer Level. This is complicated because of the high level of detail and the high susceptibility to errors due to the low level of abstraction on Register Transfer Level. To lower development time, one can raise the abstraction level to Algorithm level through High-Level Synthesis (HLS). Instead of a complicated description of the accelerator in VHDL/Verilog, High-Level Synthesis uses standard C/C++ code to describe the logic. The HLS-Tool Vivado-HLS offers compiler pragmas to further define the hardware architecture of the C/C++ code. With those pragmas, the developer can create different implementations of the same algorithm without touching the functionality by just inserting or deleting one line in the source code. With those capabilities, Vivado-HLS can be used for design space exploration. In this paper, an FPGA coprocessor is used to accelerate AES encryption and decryption calls from the Linux Kernel Crypto API.

II. Definitions and Abbreviations

A. Definitions

This paragraph specifies how common FPGA build flow terms are used in this paper.

Synthesis is the whole process of High-Level Synthesis. It basically summarizes all design steps from Vivado HLS.

Implementation summarizes all steps from Vivado.

Vivado Synthesis is the Synthesis step inside of the Implementation.

Bitstream is the output of the implementation. It is used to program the FPGA.

B. Abbreviations

This paragraph gives a short summary of all abbreviations used in this paper.

AES stands for Advanced Encryption Standard. See section III-A for an explanation.

BRAM stands for Block RAM. BRAM is one of the resources on an FPGA. BRAMs are arranged in slices of 36 KBit.

FF stands for Flip Flop. FFs are one of the resources on an FPGA.

HLS stands for High-Level Synthesis. See section III-C for a brief explanation.

II stands for Initiation Interval. See section III-B for an explanation.

LUT stands for Lookup table. LUTs are one of the resources on an FPGA. They build the logic gates inside of the FPGA.

RTL stands for Register-Transfer Level.

III. Basics

This chapter gives a short summary of the most important basics of the paper. It explains how AES works, what the design steps are within Vivado HLS and how optimization with HLS works.

A. AES

The Advanced Encryption Standard (AES) [4] is an encryption algorithm developed in 2000 by Joan Daemen and Vincent Rijmen and is one of the most important encryption algorithms today. It is a symmetric block cipher, which means, it uses the same key for en- and decryption. Block cipher means, it encrypts and decrypts blocks of data of a constant size, in this case the block size is 128 bits or 16 Bytes respectively. These 16 Bytes are arranged in a 2-dimensional 4x4-array called state. The algorithm consists of four base operations repeated in multiple rounds. These operations are AddRoundKey, SubBytes, MixColumns and ShiftRows. AddRoundKey is a simple XOR-connection between the state and the round key. SubBytes replaces all Bytes of the state according to a substitution box, called S-Box. In this paper, the SubBytes operation was implemented using arrays with precomputed values as Lookup tables (this does not mean the LUT hardware resource on the FPGA, but the basic concept of Lookup tables in software). ShiftRows does a cyclic shift on the rows of the state according to its row number. MixColumns mixes the 4 Bytes so every input Byte affects every output Byte. MixColumns is like SubBytes implemented using precomputed arrays as Lookup Tables. To encrypt more than 16 Bytes, a operation mode is required. In this work Cipher Block Chaining (CBC) is used. In this mode, the ciphertext of a block depends on the plaintext and the ciphertext of the previous block. This data dependency cannot be resolved; thus the encryption cannot be pipelined. In decryption however this dependency does not exist, which enables pipelining.

B. Performance of digital circuits

To evaluate the results of High-Level Synthesis, we need to measure the performance of an FPGA accelerator which is determined by its throughput. The throughput is influenced by many different characteristics of the accelerator, which are listed below:

Clock period is the time period of one clock cycle. All registers in the design are connected to the same clock to synchronize read and write operations.

Blocksize is the amount of data that can be read and computed at once. The unit is bit or byte. In this paper, the blocksize is 128 bit or 16 Byte.

Latency is the number of clock cycles after reading data until the result is available at the output registers.

Initiation Interval is the number of clock cycles after reading a block, until the circuit can read new data.

With these quantities the throughput can be computed as shown in eq. (1).

$$BW = \frac{BS_{total} f}{L_{init} + L_{single} + II(n-1)} \tag{1}$$

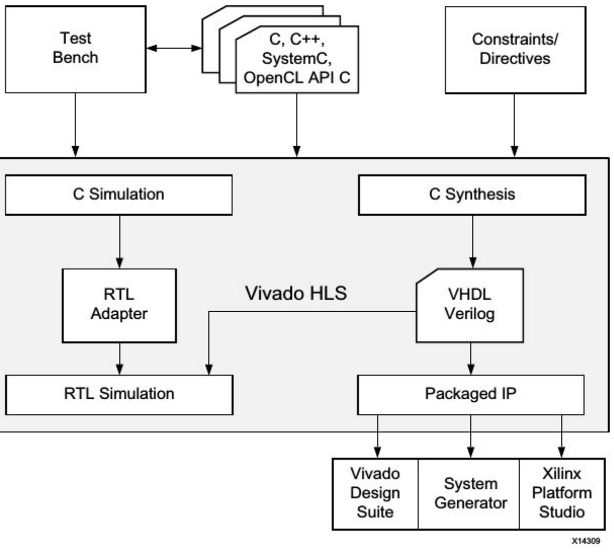

Fig. 1. Design flow of Vivado HLS [2].

with the throughput BW, the total amount of data BS_{total}, the clock frequency f, the latency for initialization process L_{init}, the latency for a single block of data L_{single}, the initial interval II and the total number of data blocks n, which is BS_{total}/16 Byte. Without pipelining, the initiation interval and the latency for a single block are the same, so the terms are interchangeable. Since the initial latency is constant with around 1000 clock cycles, it does not influence the throughput for a big amount of data. This simplifies the formula to eq. (2).

$$BW = \frac{BS_{total} f}{IIn} \tag{2}$$

This formula assumes, that there is no input data stream stall when processing a stream at the input of the accelerator and the data is read from the output immediately, so it does not get slowed down by back pressure. The result of this formula is used as a metric for the performance of the generated accelerators.

C. High-Level Synthesis with Vivado HLS

High-Level Synthesis is the Synthesis of a hardware description on Register-Transfer-Level (RTL) from a description on algorithm level. In this paper, we generate an AES accelerator from C/C++ source code with Vivado HLS. For a more detailed introduction to HLS see [6]. Figure 1 shows the design flow of Vivado-HLS. The source code can be any C/C++ implementation, as long as it only makes no use of dynamic memory allocation. It is recommended to use a generic implementation instead of an optimized one for a special compiler or processor. Since an FPGA works differently than a normal CPU, optimizations for a CPU are not suited for an FPGA and might even worsen the results. The correctness on the algorithm level of the code can be checked with the C-Simulation using a testbench. Now, the interface of the accelerator has to be declared with the Interface pragma.

It describes how the interface has to be generated from the parameters of the top level function and which type of bus has to be used. In our case, we used an AXI-Stream interface for the data input and output with an additional AXI-Lite interface for control signals like starting the encryption or changing the encryption key. Once the code is synthesizeable, it can be optimized using additional pragmas. A list of all used optimization pragmas and a short description is given below:

Loop Tripcount lets the user specify a minimum and maximum number of iterations for a loop. This does not influence the synthesis, but helps to get a precise latency estimation.

Array Partition partitions an array in multiple smaller arrays. The default behavior is to only generate one input and output port for every array. By partitioning it into smaller arrays with an input and an output port for each sub-array, the manipulation of single cells of the array can be parallelized.

Loop Unroll generates multiple instances of the code body of a loop. If there are no data dependencies between loop iterations, they can be parallelized by creating multiple instances of the body.

Pipeline creates a pipelined architecture for a specified function. This increases the throughput by reducing the initiation interval.

Inline eliminates the hierarchy level of sub-functions and dissolves their logic into the logic of the caller function. The default behavior of Vivado HLS generates one module for every function in the source code with a sub-module for every sub-function. The logic optimization only works inside a module on one hierarchy level. By eliminating those borders with inlining, it is easier for the optimization to simplify and shorten the RTL description.

Before starting the C-Synthesis, we need to specify a target frequency, that specifies the frequency, at which the accelerator should operate.
The C-Synthesis generates the VHDL/Verilog code from the C/C++ source code. The RTL code can be verified using the C/RTL-Co-Simulation. Now the code gets packaged and exported with the IP-Packager and can later be included into a Vivado block design. More details to Vivado HLS can be found in [2].

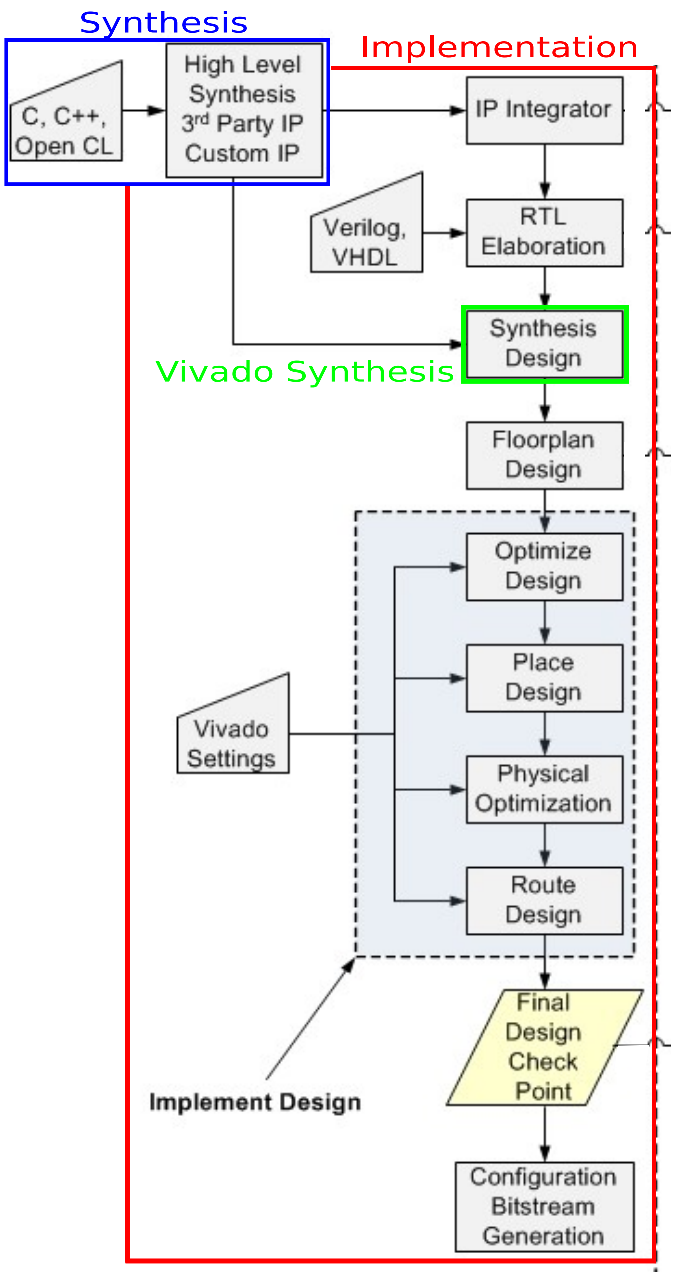

Fig. 2. Design flow using Vivado HLS and Vivado [3].

IV. Test Setup

Figure 2 shows the complete design flow with Vivado HLS and Vivado. A detailed view of the Synthesis is depicted in fig. 1. Both tools, Vivado and Vivado HLS, were used in version 2016.2. Newer versions could not be used because of incompatibilities with the hardware driver seen in fig. 4. Starting point is a C/C++ implementation of the AES-Algorithm. The one used for this paper can be found at [8]. After running through all the steps explained in section III-C Vivado HLS returns estimations for the resource demand and the performance of the accelerator, including the initiation interval and latency. The user can go through the whole hierarchy of the design and see these estimations for every single sub-module.

The generated IP Core is part of the block design displayed at fig. 3 at the position highlighted in red. The clock is set to the estimated clock given by Vivado-HLS. After implementation, Vivado returns the actual resource costs and a timing analysis, including signal paths that fail the timing constraints.

Performance tests are conducted on a Xilinx ZC706 board, featuring a Zynq 7045 SoC. Through a custom device driver, explained in [1], the Linux OS on the processing system (PS) of the Zynq SoC on the board is able to accelerate AES calls of the Linux Kernel Crypto API with the FPGA

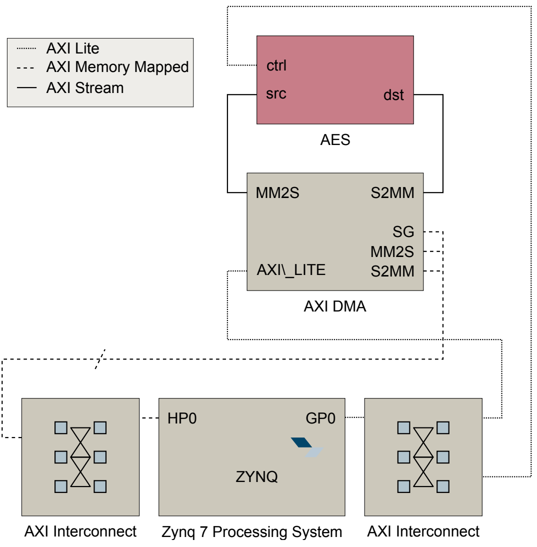

Fig. 3. Vivado block design [1].

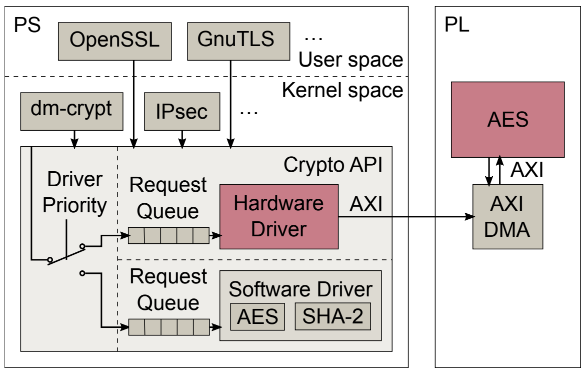

Fig. 4. Test setup with connection to the Crypto API [1].

accelerator. Figure 4 shows the whole test setup. This does not always work, there are two different types of failure that we observe. The first one is the creation of incorrect logic. When the driver is loaded, the Crypto-API verifies the correctness with a testbench. If the generated logic is incorrect, this returns an error message stating that the ciphertext is wrong. With the other type of failure, the measurement halts at the initialization of the measurement. Since both, driver and functionality, always stay the same for all tests, this points to a failure in the Synthesis. In the results paragraph we do not differ between the two types of failure and only state if the design passed the test or not.

The accelerator is generated with different sets of optimization pragmas in five tests. Each test contains 9 designs with the same set of pragmas, but a different target frequency. It ranges from 100 MHz to 260 MHz in steps of 20 MHz. For higher target frequencies than 260 MHz, the design always fails to meet the timing constraints, so these implementations/frequencies are not considered in the evaluation. The different optimizations are as follows:

Test 1 contains no pragmas for optimizing the architecture. The Interface pragma is inserted, because it is necessary for the tool to synthesize the IP core. Also the Loop Tripcount pragma is inserted to generate more precise estimation results for performance. With this pragma, the user is able to define a maximum and minimum amount of loop iterations for the latency estimation.

Test 2 contains the Array Partition pragma. By default, for every array in the source code, the Synthesis generates one BRAM with only one read and write port. This decreases the performance, because the single array elements can only be read or modified in a sequential manner. By partitioning the array into multiple smaller memory blocks with a read and write port for each block, access to different array elements can be parallelized.

Test 3 extends Test 2 with added Loop Unrolling pragma. It creates multiple instances of the loop body to calculate the results in parallel as long as there are no data dependencies in between the loop iterations.

Test 4 extends Test 3 with added Pipeline pragma. The Pipeline pragma allows the user to define an Initiation Interval for the pipeline. We used 10, 3 and 1 clock cycle as Initiation Interval to test the influence of different intervals on the performance and the resource cost.

Test 5 extends Test 4 with an initiation interval of 1 clock cycle with added Inline pragma. The Inline Pragma shifts the inlined function on a higher hierarchy level and eliminates hierarchical borders. This does not optimize the architecture directly, but the following optimization step in the Synthesis has a lot more freedom to combine operations and getting rid of unnecessary registers in between operations, which reduces latency and resource cost.

Since the goal is to test the capabilities of the tool and compare the influence of different optimizations, the optimizations focus on the decryption of AES. The encryption and decryption consist of the same operations in different order, but with the operation mode CBC, pipelining is only possible for the decryption. The results in section V only contains the results of the decryption, since the rest (encryption and key expansion) are not included in the optimization process.

V. Results

The highest RTL hierarchy level of the decryption always looks the same, no matter which optimizations were applied. The RTL description is displayed in fig. 5. While the two blocks CBC-XOR and AES-Decryption are directly influenced by the C/C++ source code, the FSM (finite state machine) is automatically generated.

The tool always returns a specific estimation for resource costs and performance. In the following paragraphs, the ranges given for some values are the the minimum and maximum values for the 9 different target frequencies.

A. Test 1: no pragmas

The code without any optimization pragmas generates an implementation with low resource costs. The estimated de-

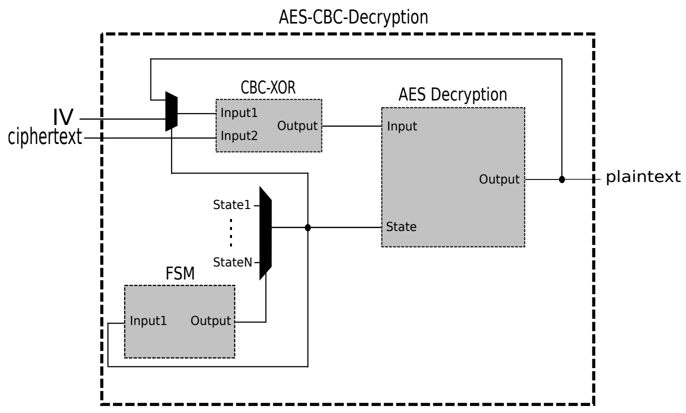

Fig. 5. Synthesized Logic AES-CBC-Decryption

mand of LUTs ranges from 3654 to 3660, the amount of FFs ranges from 592 to 1155 and 9 BRAM slices are required. While the estimation of the BRAM is accurate, the actually required amount of LUTs and FFs is lower than the estimation. The required amount of LUTs ranges from 593 to 638, the required amount of FFs from 573 to 689. The minimal latency ranges from 1011 to 1683, the maximal latency from 1371 to 2235 clock cycles, the initiation interval is identical. This leads to an estimated throughput of up to 3.06 MB/s according to eq. (2) for a target frequency of 260 MHz. The maximum for the target frequency seems to be around 180 MHz, since all higher frequencies fail to meet the timing constraints after the implementation. The actual measurement of the throughput failed for all designs in Test 1.

B. Test 2: array partitioning

Due to the added array partitioning, the estimated and the actual performance and resource requirements rise. The BRAM estimation and the actually required amount rise to 40 BRAM slices. The estimated number of LUTs range from 4918 to 5044, the FFs from 905 to 1946. The minimal latency ranges from 416 to 895 clock cycles, the maximum from 562 to 1203. The highest estimated throughput is 6.55 MB/s for a target frequency of 240 MHz. The actual resource requirements are again lower than the estimation. The LUT demand ranges from 2363 to 2679, the FF demand from 880 to 1212. All designs passed the functionality test, the highest throughput was 5.16 MB/s for a target frequency of 240 MHz.

C. Test 3: loop unrolling

The loop unrolling further increases the performance and the resource costs. The estimated number of LUTs drops to a range from 4602 to 4883. This is because the loop calculations now happen in parallel. Before, there was only one instance, which was reused for every loop iteration. That required a control logic which disappears for parallel computation. The amount of registers rises, since every register has to be duplicated for every parallel path. This leads to an increase of the estimated FFs to a range of 1590 to 2529. The required BRAM stays the same, since the methods that require BRAM do not contain loops. The minimal latency ranges from 50 to 131, the maximum from 70 to 182. The actual amount of LUTs ranges from 1774 to 2338, the FFs range from 1592 to 2012. The BRAM demand and estimation are identical. Apart from a target frequency of 180 MHz, all designs passed the measurement test with the highest throughput of 26.15 MB/s for a target frequency of 120 MHz.

D. Test 4: pipelining

1) Initiation Interval = 10 clock cycles: Pipelining only has a small impact on the minimal latency, but the maximal latency is now equal to the minimal latency. This is necessary, because pipelining needs a constant latency. For an initiation interval of 10 clock cycles it ranges from 44 to 131 clock cycles, the initiation interval is the specified 10 clock cycles. The LUTs estimation ranges from 7519 to 9693, the estimated FFs range from 1982 to 4247. The BRAM estimation rises to 80 slices. Since pipelining requires additional instances of all base operations, the resource consumption increases. 189 MHz seems to be the maximum for the estimated frequency, since no design achieves a higher frequency. The highest estimated throughput is 301 MB/s. After implementation, the required amount of LUTs is between 2617 and 4028. The required FFs range from 1736 to 3820. Apart from a target frequency of 180 MHz, all designs pass the measurement test. The highest measured throughput is 27.24 MB/s for a target frequency of 100 MHz. This is significantly lower than the estimation of 301 MB/s. The reason for this is explained in section VI, since it influences all pipelined designs.

2) Initiation Interval = 3 clock cycles: The resource costs increase again, since the synthesis now creates 5 instances of all base operations. The latency ranges from 44 to 88 clock cycles. The LUTs estimation ranges from 17499 to 22541, the FFs from 2381 to 7097. The BRAM estimation rises to 200 slices. The maximum for the estimated frequency seems to be 189 MHz again, the maximal estimated throughput is 1006 MB/s. The implementation only needs between 4800 and 6635 LUTs and between 2271 and 3915 FFs. Apart from a target frequency of 180 MHz all designs pass the measurement test. The highest throughput was achieved by the design with a target frequency of 140 MHz with 31.28 MB/s, which is again significantly lower than the estimations.

3) Initiation Interval = 1 clock cycle: There are now 14 instances of every base operation. This leads to a LUT estimation between 44790 and 45047 and a FF estimation between 3077 and 15603. The BRAM estimation and utilization rises to 528 slices. The latency drops to a range from 42 to 84 clock cycles, the highest estimated throughput is 3019 MB/s for target clocks between 180 and 260 MHz. The actually required amount of LUTs ranges from 4653 to 5366, the FFs from 1085 to 6857. Target frequencies from 120 to 180 MHz fail, the rest passes the measurement test. The highest throughput is 30.36 MB/s.

E. Test 5: Inlining

Inlining the base operations decreases the latency to a range from 28 to 68 clock cycles. The estimated amount of LUTs ranges from 13660 to 15327 and the estimated FFs range from 3431 to 13239. The estimated BRAM usage stays at 528 slices. The maximal estimated throughput is 93 MB/s for a target frequency of 140 MHz when taking into account that pipelining does not work. The actual LUT usage ranges from 5713 to 8163, the FFs from 3288 to 9918. No design passed the measurement test.

VI. Analysis

All pipelined designs have an increased resource utilization in comparison to the designs without pipelining due to additional instances. The analysis with the Integrated Logic Analyzer shows, that there is always valid data at the input and the output always waits for the accelerator to read data. So the problem does not originate from the environment, but from the accelerator itself.

So for example we investigate the design of Test 4 with an initiation interval of 10 clock cycles and a target frequency of 100 MHz. This design should achieve up to 181.81 MB/s, but the measurement only shows a throughput of 27.24 MB/s. The synthesis shows, that it created an additional instance of every base operation and they still exist after the implementation. Pipelining with an initiation interval of 10 clock cycles means, that one instance can only be occupied for 10 clock cycles by a single data block.

AES consists of many rounds, as explained in section III-A, which consist of the four base operations AddRoundKey, SubBytes, MixColumns and ShiftRows. Depending on the key size there are between 10 and 14 complete rounds per data block. The scheduling diagram in Vivado HLS shows, that SubBytes and MixColumns are occupied for 2 clock cycles in each round. This sums up to 20 and 28 cycles for SubBytes and MixColumns each for every data block. Divided by two because of the additional instances of the base operations, this results in 10 to 14 clock cycles with an initiation interval of 10 clock cycles. So there need to be at least 3 instances of every base operation to actually enable pipelining with an initiation interval of 10 clock cycles.

For the same reason, the design with an initiation interval of 3 clock cycles would need at least 14 instances of every base operation. For an initiation interval of 1 clock cycle and a target frequency of 100 MHz the High-Level Synthesis generates 14 instances of all operations plus an additional AddRoundKey instance for the first round. Because of the operations SubBytes and MixColumns always being occupied for 2 clock cycles at a time, it is still not possible to achieve the requested initiation interval.

To check if this problem is specific for the tool version, we repeated the synthesis step of test 4 with tool version 2017.2, which was the newest available version at the time. The synthesis results though stayed the same. The resource estimations were identical as well as the generated architecture. Even with this version, there were always to few instances of the base operations to enable pipelining.

The goal of this paper was to evaluate the High-Level Synthesis with Vivado HLS. The generation of an FPGA accelerator with High-Level Synthesis is faster and easier compared to writing HDL code. It is possible to generate a completely different architecture for the accelerator within 60 to 90 minutes including synthesis and implementation. Optimization pragmas like array partitioning and loop unrolling work just like they are supposed to. This enables the user to generate accelerators that are faster than most software solutions for fitting problems like encryption or hash algorithms.

Pipelining however does not provide the throughput it should. Inlining can even change the logic and thus breaking the design if applied at the wrong place. The tool seems to heavily depend on the correct coding style. During the whole optimization process, only once an error message occurred stating that the placed optimization pragma does not work at this position. This was when trying to pipeline the encryption despite the operation mode making it impossible. In every other case, the tool stated a correct synthesis, the problems were only observed when actually loading the design to an FPGA and measuring the actual throughput. This behavior is not reliable and not usable for real-life applications. The problems with pipelining and inlining keep the user from creating high performance, high throughput designs.

Another less important problem is, that the resource estimations, especially for the LUTs, is way higher than the actual usage after implementation. A possible reason for this could be an incorrect state machine which creates unnecessary states and logic that later gets optimized away through logic optimization in the implementation.

VII. Outlook

There are multiple ways of progressing with this work. In this paper, we changed the target frequency for the High-Level Synthesis. By keeping the same design and increasing the frequency at the implementation step by step could show, how accurate the frequency estimation is and give another possibility to increase the throughput. One could also compare the results of different implementations to find the best coding style for High-Level Synthesis. One could also try out single pragmas for optimization to see the individual effect of the pragmas, but the results of this step highly depend on the algorithm and its implementation, so there would not be a general knowledge gain out of these tests.

VIII. Conclusion

In this work we showed how to generate an AES FPGA accelerator with High-Level Synthesis. It turned out to be faster and easier compared to standard RTL development with VHDL/Verilog. Especially for developers, who are not experienced in RTL development, it is a way to still profit from the compute power of an FPGA. It is also useful for design space exploration, since it it possible to generate a completely different architecture within minutes by just inserting or removing a compiler pragma.

However the tool is not ready to be used for real-life applications yet. A main flaw is the difficulties with the coding style. It is probably necessary to write optimized code for the High-Level Synthesis, comparable to code optimized for special processors. Yet this would take away the biggest advantage of being easy to use. In the current state it is not possible for the user to generate reliable high performance, high throughput designs.
Another problem is the inaccuracy of the resource estimations, apart from the BRAM estimations. The estimations were always too high. The real LUT usage is only a fraction of the estimation. This would make it possible to implement a design even though the tool estimates a usage of more than 100%. In the current state, the tool has to improve on its reliability before it can be integrated into a professional real-life workflow.

REFERENCES

[1] S.Wiehler, *CPU-Offloading von Transformationsfunktionen aus dem Linux-Kernel*. 2016.

[2] Xilinx, *Vivado Design Suite User Guide: High-Level Synthesis(UG902)*, v.16.2. 2016.

[3] Xilinx, *https://www.xilinx.com/content/dam/xilinx/imgs/applications/isolation-design-flow/idf-flowchart.jpg.thumb.319.319.png*. 16.1.18.

[4] National Institute of Standards and Technology, *FIPS PUB 197: Advanced Encryption Standard (AES)*. 26.1.2001.

[5] Xilinx, *Vivado Design Suite: AXI Reference Guide(UG1037)*. 15.6.2017.

[6] Coussy, P. and Gajski, D. D. and Meredith, M. and Takach, A., *An Introduction to High-Level Synthesis*. 2009.

[7] Andre Dehon, *Fundamental Underpinnings of Reconfigurable Computing Architectures*. 3.3.15.

[8] kokke, *https://github.com/kokke/tiny-AES128-C*. 2017.