# Microservice-based in-network AES solution for FPGA NICs

A THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

**M.Tech**

BY Lasani Hussain MT21042

Computer Science and Engineering

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY DELHI
NEW DELHI– 110020

December 21, 2022

## Abstract

Data centers demand high throughput (100 to 400 Gbps) and sub-millisecond latency. The performance of data center applications heavily depends on the efficiency of the underlying TCP stack. Despite several optimizations, such as kernel bypass and zero copying, TCP processing consumes up to 60% of the entire CPU cycles for short-lived connections. Modern data centers are pushing the TCP processing to programmable data plane hardware (smart NICs) to improve performance and save CPU cycles. However, the user space application processes the transport layer security (TLS) functions, negating the benefits of TCP offload. Some research proposes offloading TLS state and connection and management but ignores the processing of compute-intensive TLS crypto algorithms. We aim to offer in-network crypto primitives that TLS offload solutions can incorporate. Our goal is to design an in-network crypto framework that promises high-speed, low latency, scalability, dynamic reconfiguration, and low-power by leveraging FPGA-based network hardware. This thesis presents an FPGA-based AES offload solution aiming to satisfy the required objectives.

# Chapter 1

# Introduction

Modern data centers can process traffic rates in the range of 100-400 Gbps. Studies have found that a large proportion of this traffic (>90%) are TCP packets.[1] Even with optimizations like kernel bypass, more than 60% of clocks are used for TCP processing. To improve CPU usage, Researchers have offloaded data processing to the NIC while retaining connection and state management on the CPU. Since TCP depends on TLS for security, without offloading TLS protocol, packets need to revisit the kernel stack for encryption and decryption, defeating the offload's purpose. This demands that TLS is also offloaded to NICs but the current design implementation of NIC does not have ample computing power to provide line rates without specialized hardware. TLS supports a variety of cryptographic suites with varying popularity. The addition of a new ASIC accelerator requires enormous development time and product revision which cannot be done for every new algorithm. Hence the new addition of FPGA fabric with current NICs can help us develop architectures where we can generate similar accelerators. We identify the common submodules among the cipher suites listed in TLS and formulate a DAG to denote the dependency of the modules on each other. By having individual IPs(Intellectual Property) for these algorithmic sub-blocks, we can efficiently scale up by provisioning only those IPs which are computation hungry. My work comprised the AES GCM sub-block, wherein I identified the points in the flow of code which can be optimized by the HLS tool, wrote appropriate HLS equivalent code for the C implementation of GCM, and optimized the clock cycles required to execute the algorithm end-to-end. I also checked my implementation for correctness by writing test benches which would validate the HW output against the SW output, using RFC test vectors, to be sure my implementation is robust and fails no corner case. Finally I deployed the code on the ZCU 106 Evaluation kit and the results are promising.

# Chapter 2

# Background

## 2.1 TLS

Cipher suites, often known as Secure Sockets Layer (SSL) or Transport Layer Security (TLS), are collections of instructions that enable secure network communications (SSL). These cipher suites offer the algorithms and protocols necessary to protect communications between clients and servers in the background. The two parties must first communicate which algorithms they support (i.e., the cipher suite) and agree on which ones to utilize before they can start exchanging messages. The exchange of several messages accomplishes this during the so-called "TLS handshake." The server is the one who ultimately chooses the ciphers.
To guarantee the security, functionality, and compatibility of your HTTPS connections, selecting and maintaining the proper cipher suites in both the web server and the client is crucial.
TLS 1.2 supports 37 cipher suites out of which few are listed below :

1. TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
2. TLS_DHE_RSA_WITH_AES_128_CBC_SHA
3. TLS_DHE_RSA_WITH_AES_256_CBC_SHA
4. TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
5. TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
6. TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
7. TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305
8. TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
9. TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305

Typically, a cipher suite appears as a long string of seemingly random data; yet, every segment contains crucial data. This data string typically consists of the following essential elements:

Algorithm for :

- key exchange such as RSA, DH, ECDH, DHE, ECDHE, or PSK
- for digital signatures (authentication)like RSA, ECDSA, or DSA
- for bulk encryption like AES, CHACHA20,
- for the message authentication code (MAC) such as SHA-256, and POLY1305

Figure 2.1: Parts of TLS Cipher Suite

In the diagram 2.1 ECDHE determines that the keys will be exchanged during the handshake using ephemeral Elliptic Curve Diffie Hellman. The authentication algorithm is called ECDSA, or Elliptic Curve Digital Signature Algorithm. The mass encryption algorithm is AES128-GCM, which uses a 128-bit key size and AES in Galois Counter Mode. Finally, the hashing algorithm is SHA-256.

## 2.2 AES GCM

AES with Galois/Counter Mode (AES-GCM) offers authenticated encryption (confidentiality and authentication) as well as the capability to verify the authenticity and integrity of extra authenticated data (AAD) that is transmitted in the open [2].
The AES Encryption algorithm (also known as the Rijndael algorithm) is a symmetric block cipher algorithm with a block/chunk size of 128 bits. It converts these individual blocks using keys of 128, 192, and 256 bits. The AES algorithm uses a substitution-permutation, or SP network, with multiple rounds to produce ciphertext. The number of rounds depends on the key size being used. A 128-bit key size dictates ten rounds, a 192-bit key size dictates 12 rounds, and a 256-bit key size has 14 rounds. Each of these rounds requires a round key, but since only one key is inputted into the algorithm, this key needs to be expanded to get keys for each round, including round 0.

The input to the encryption and decryption algorithms is a single 128-bit block. In FIPS PUB 197, this block is depicted as a 4 * 4 square matrix of bytes. This block is copied into the State array,z which is modified at each stage of encryption or decryption. After the final stage, State is copied to an output matrix.

Block cipher : The underlying block cipher of the mode is composed of two inverse functions for any given key. As in the description of the AES method in [3] choosing a block cipher also entails designating one of the two block cipher functions as the forward cipher function. Using the inverse cipher function is not possible with GCM.
Authenticated Encryption : Given the selection of an approved block cipher and key, there are three input strings to the authenticated encryption function:

- a plaintext, denoted P;
- additional authenticated data (AAD), denoted A; and

- an initialization vector (IV), denoted IV.

The two types of data that GCM secures are plaintext and the AAD. The plaintext and the AAD are both protected by GCM, along with the secrecy of the plaintext. The AAD is not covered by GCM. The AAD may contain fields that specify how the plaintext should be handled, such as addresses, ports, sequence numbers, protocol version numbers, and others inside a network protocol. The bit lengths of the input strings to the authenticated encryption function shall meet the following requirements:

$$len(P) <= 2^{39} - 256; \quad (2.1)$$

$$len(A) <= 2^{64} - 1; \quad (2.2)$$

$$1 <= len(IV) <= 2^{64} - 1; \quad (2.3)$$

## 2.3 HLS

### 2.3.1 HLS Synthesis

High-Level Synthesis is an automated design process that takes an abstract behavioral specification of a digital system and generates a register-transfer level structure that realizes the given behavior. The designer needs to create the overall structure and design of the algorithm in C/C++, considering how it will interact with the outside world. However, they do not need to worry about specific micro-architecture details such as the state machine, datapath, and register pipelines. These can be left to the High-Level Synthesis (HLS) tool to handle, using input constraints such as the clock speed, performance pragmas, and target device to generate optimized Register-Transfer Level (RTL) code.

### 2.3.2

### 2.3.3 Vitis HLS

The Vitis High-Level Synthesis (HLS) tool is used to convert C or C++ code into Register-Transfer Level (RTL) code for implementation on Xilinx programmable logic devices, such as Versal ACAP, Zynq MPSoC, or FPGAs. It can be integrated with the Vivado Design Suite for synthesis, place, and route and the Vitis Core Development Kit for system-level design and application acceleration. Vitis HLS can be used to develop and export Vivado IP for use in hardware designs or Vitis kernels for use in the Vitis application acceleration flow. It automates many of the processes required to implement and optimize C/C++ code in programmable logic, including the inference of pragmas to pipeline loops and functions within the code. The generated RTL can be used as IP within the Vivado tool or Model Composer, or it can be ccustomized to meet specific interface standards. The development process involves:

- designing the algorithm
- verifying the C/C++ code with a testbench
- generating the RTL with HLS

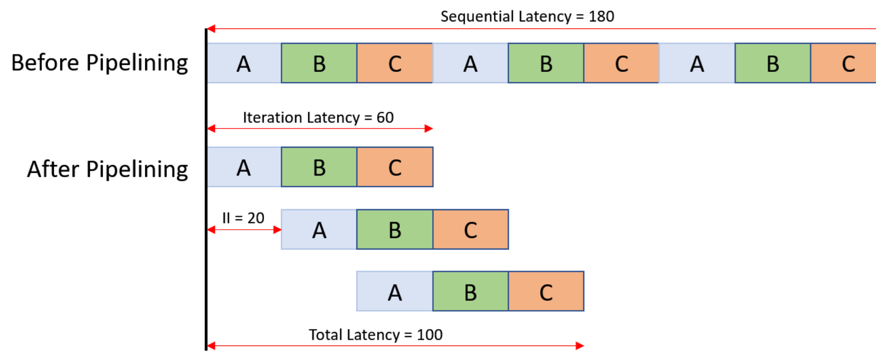


Figure 2.2: Pipeline

- verifying the kernel with C++ outputs, and
- analyzing the synthesis and co-simulation reports to ensure performance goals are met.

### 2.3.4

### 2.3.5 Pipelining Paradigm

Pipelining 2.2 is a technique used to optimize micro-level architecture by allowing multiple tasks to be executed simultaneously using the same resources over time. It is typically used in situations where tasks produce or consume data at a high rate, such as with instruction-level pipelining (ILP).

# Chapter 3

# Related work

## 3.1 TCP Offload

FlexTOE [4] is a flexible and high-performance TCP offload engine for wimpy SmartNICs. It offloads the TCP data path, using the microservices paradigm for parallelization by involving pipelining. For scalability and flexibility, they decompose the TCP data path into fine-grained modules by keeping the stateless design among the states of pipelines and replicating the states since they are stateless. It is also flexible, as it supports the modified TCP. It processes the Memcached application, which scales up to 38% better than the TAS, while saving up to 81% host CPU cycles versus Chelsio. FlexTOE cuts 99.99th-percentile RPC RTT by 3.2 times and 50% versus Chelsio and TAS, respectively.

AccelTCP [5] offloads the TCP layer to the programmable data plane(PDP), i.e smartNICs, by selectively offloading the states of TCP. As it offloads the Connection setup/teardown, segmentation/checksum, and connection splicing to the NPU-SmartNICs, but keeps the TCP data path on the host. AccelTCP enables short-lived connections to perform comparably to persistent connections. It also improves the performance of Redis, a popular in-memory key-value store, and HAProxy, a widely-used layer-7 load balancer, by 2.3x and 11.9x, respectively.

## 3.2 Reconfigurable hardware solution

Autonomous NIC offloads [1]: This work offloads the TLS (i.e L5Ps) without having to offload the entire layer¡= TCP/IP stack into the NIC. The main goal of the L5P-NIC collaboration is to process L5P messages in the NIC in a way that is transparent to the TCP/IP stack that serves as an intermediary. They implement autonomous offloads for two L5Ps: (i) NVMe-over-TCP zero-copy and CRC computation and (ii) https authentication, encryption, and decryption. This implementation increases the throughput by up to 3.3x and reduces the latency by 0.4x and 0.7 x, respectively.

# Chapter 4

# Design

## 4.1 Decomposition of TLS crypto algorihtms

AES block cipher encrypts a 16B plaintext at a time, for plaintexts larger than 16B , the plaintext is divided into blocks of 16B chunks and these we repeatedly encrypt these 16B chunks using AES block cipher, now when we have the 16B ciphertext chunks, how do we combine them to produce the ciphertext, herein comes into picture "modes of operation" for AES. Following are the modes of operation for variable length plaintext encryption :

- ECB mode: Electronic Code Book mode
- CBC mode: Cipher Block Chaining mode
- CFB mode: Cipher FeedBack mode
- OFB mode: Output FeedBack mode
- CTR mode: Counter mode

Refer the TLS crypto suite discussed before. The GCM, CCM, and other modes of operation work in close coordination with AES block cipher as the functioning of these modes differ in how and with what data these modes of operation invoke the underlying AES block cipher with.

For example, in the CBC mode of operation plaintext block, P1 is XORed with IV and then encrypted to get ciphertext block C1 . In the next iteration, P2 is XORed with C1 and then encrypted i.e. Ci acts as IV for the next iteration (i+1) of CBC mode. This is different from how GCM operates as GCM counter blocks themselves are encrypted and later XORed with plaintext. From this, we come to the conclusion that the modes of operation are closely coupled up with the underlying block cipher, but the hash or MAC calculation is a logically different step and comes into the picture once the ciphertext is generated so based on the above observations we can de-couple MAC calculation(SHA) step from modes of operation, depicted in the block diagram4.1 below:

As discussed above, the modes of operations differ in how they act as a wrapper over the AES block, so we can move the common AES block out, to get the following dependency graph, where A-¿B implies A comes later than B in the topological order of algorithm , hence B has to complete its execution before A can begin.

Figure 4.1: Algorihtms in TLS suite

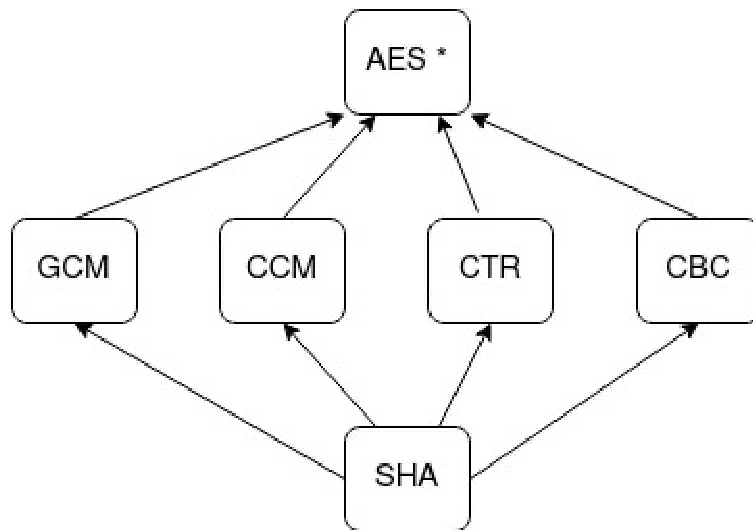

Figure 4.2: Overlapping submodules

In the diagram 4.2 above AES* means for every execution of AES block, Key Expansion is executed internally. So if we have a long plaintext that needs to be encrypted using the GCM mode , then for every 16B chunk of that plaintext that gets encrypted using AES, Key Expansion is executed, which unnecessarily adds an overhead, since the same expanded key can be used for the later 16B chunks encryption belonging to the same plaintext. So it is only logical to move the Key Expansion block out of AES so that we can reduce the number of unnecessary clocks wasted to redo the Key Expansion.

So the DAG becomes the following4.3 : Here the AES block does only the encryption part and Key Expansion is moved out. My work is focussed on optimizing the GCM-AES-KeyExpansion subgraph of the above DAG. In the background section as we discussed the GCM algorithm, GCTR was the function that repeatedly invokes AES for encryption of counter blocks , sending 16B at each such invocation, and the number of such invocations being the number of 16B chunks the plaintext can be divided into. Hence it is logical that we disaggregate the GCM algorithm itself into blocks , which would help us define clear boundaries about where repeated invocations are made to AES IP block. The GCM block further gets divided into Pre GCM, Post GCM and HASH blocks as shown below 4.4 :

The motive behind all the above disaggregations was to identify common overlapping subblocks of algorithms for which we can make separate IPs and this provides us with more software-oriented, microservice-based design, each of the blocks can be used independently as a black box, which additionally can help us with:

- Hardware efficiency: Same FPGA fabric can be used to host multiple different IP's
- Reusability : By identifying IP blocks that consume a large fraction of total computation, we can have multiple instances of such IPs , in our case AES blocks, so that we can perform multiple encryption operations in parallel , making efficient use of hardware resources, increasing throughput and decreasing

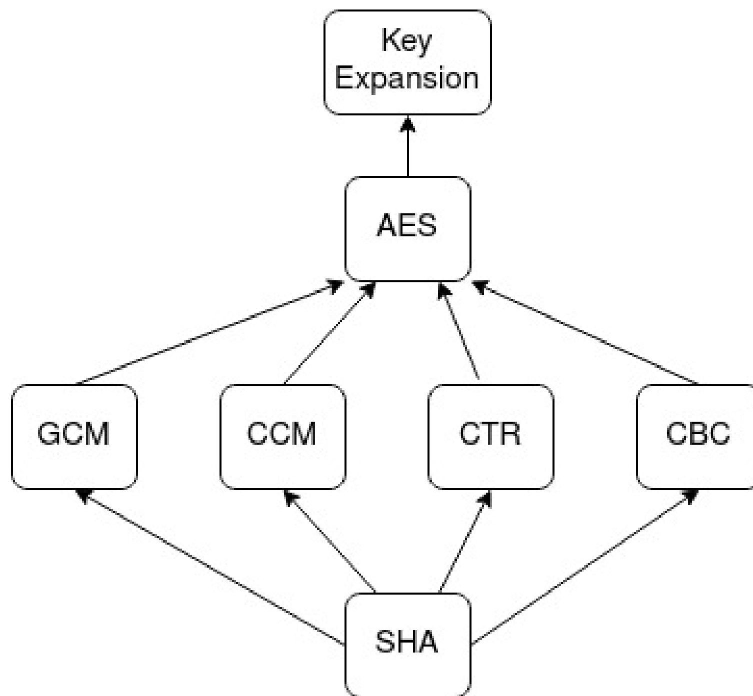


Figure 4.3: Fully disaggregated algorithm blocks

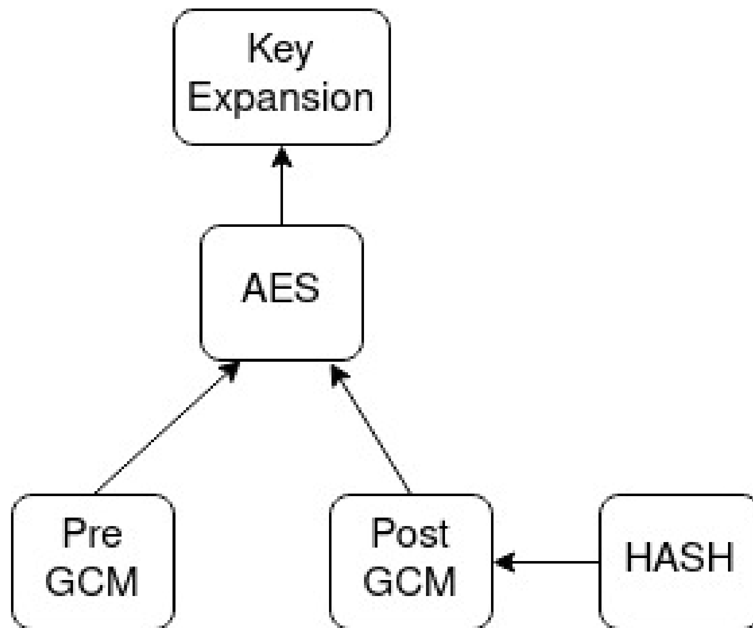


Figure 4.4: Disaggregated GCM

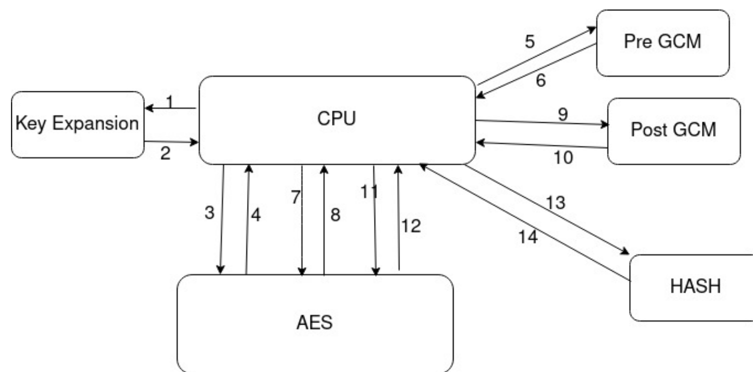


Figure 4.5: GCM Functional Blocks

latency

Following 4.5 is a detailed description of fully disaggregated design of AES GCM algorithm with parameters being exchanged considering everything apart from the CPU is an IP.The operations are numbered in chronological order.NOTE: Steps 1 through 14 happen once except for step 7 and 8 which happen in a loop i.e. CPU sends multiple counter blocks (Ctr) for encryption to AES and gets encrypted counter blocks EncCtr in return.

# Chapter 5

# Guidelines

## 5.1 Guidelines for Algorithm decomposition for FPGA target

- Efficient hardware-friendly code: Code should be written keeping in mind the operations we execute are going to be executed in hardware, so a basic understanding of the mapping of operations to hardware components goes a long way in writing code that executes faster. So we can treat an iterative operation over a fixed-length byte(unsigned 8 bit ) array (say of length k)as equivalent to a single operation on 128*k bits For example take the right shift operation on a byte array of length 16, such that each element is an unsigned 8-bit integer. A software implementation would right shift each of the 16 elements and use a carry variable to track whether the LSB of the current element 'falls off' after incrementing and appropriately set the MSB of the next byte depending on the carry variable, incurring 16 clock cycles. The best way to do this would be to copy all 16 elements into a 128 bit single apuin¡128¿ variable , then do a single right shift over the 128 bits and then copying the 128 bits back into byte array representation incurring a total of 1+1+1=3 clock cycles.

- Pipeline : This can be used in loops that run for a non-fixed number of times i.e. we can't know the number of iterations beforehand. Here we can pipeline the loop so that each iteration gets executed in an overlapped stage of a pipeline. This increases the hardware resource usage but decreases the clock cycles need to execute the complete loop

- Unroll : This is used to execute all operations of a fixed length loop i.e.iterations known during compile time, in a single clock cycle. One should be careful while using this and not treat it as a silver bullet to be applied in any fixed-length loop, which can lead to serious bugs. Some pre-requisite for unrolling a loop, are : There should be no data dependency within loop iterations i.e. inputs for computations done in later iterations should not depend on result of previous iterations. The arrays used for doing I/O in the loop need to be partitioned fully, for the tool to provision parallel memory access.

The above guidelines are not hard and fast, in fact, pipelines can be used in conjunction with loop unrolls, if constraints required by unrolls and pipelines are respected. An example of this combination is the GCTR function (refer to background), where the outer loop runs n times and n is calculated at runtime. We pipeline the outer loop, and then in the inner loop when we process a large array (elements  1500) in batches of size 16, we UNROLL the inner loop which runs 16 times and perform a cyclic partition of the array being operated on by a factor of 16, this way the 16 elements get processed parallelly and this batch processing happens in a pipelined manner where in the next 16 elements get processed in the next stage of the pipeline.

# Chapter 6

# Implementation

## 6.1 Implementation

Refer to 6.1, Parser: The IP has a single input stream and a single output stream, with ports packed to the limit i.e. 128 bits of data read/written in a single cycle, to extract max operations per clock cycle. So any variable length payload that is sent must be preceded by some custom header that encodes the length of the payload the stream carries. Scheduler: We first copy all the data into a single contiguous array of type uint8t, the starting address of which is sent to DMA which treats the array as a contiguous bit stream. Both the input and output cache is flushed and the DMA is configured by the respective device id. We then perform 2 asynchronous DMA transfer operations: one from DMA to IP and the other from IP to DMA. Now we busy wait for the DMA channels to get free, which would only ensure the correctness of data in the output array receiving data from the IP. Load generator(put in testbed ?): We copy and organize our payload data as required by IP along with headers in array allocated via heap memory, in array in[] a. The address of the arrays along with the device ID is passed to function , repeatedly inside a loop running some fixed iterations Generic implementation modules : AXI4 stream: The common interface for connecting components that want to share data is the AXI4-Stream protocol. The interface can be used to link a single data-generating master and a single data-receiving slave. When connecting a higher number of master and slave components, the protocol can also be utilized. The protocol enables the construction of a generic interconnect that can carry out upsizing, downsizing, and routing operations while supporting various data streams utilizing the same set of shared wires. The processing system (PS) : The Processing System IP is the software interface around the Zynq® Ultrascale+™ MPSoC Processing System. The Zynq UltraScale MPSoC family consists of a system-on-chip (SoC) style integrated processing system (PS) and a Programmable Logic (PL) unit, providing an extensible and flexible SoC solution on a single die. The Processing System IP Wrapper acts as a logical connection between the PS and the PL while assisting you to integrate custom and embedded IPs with the processing system using the Vivado® IP integrator. Interconnect: The AXI Interconnect IP connects one or more AXI memory-mapped Master devices to one or

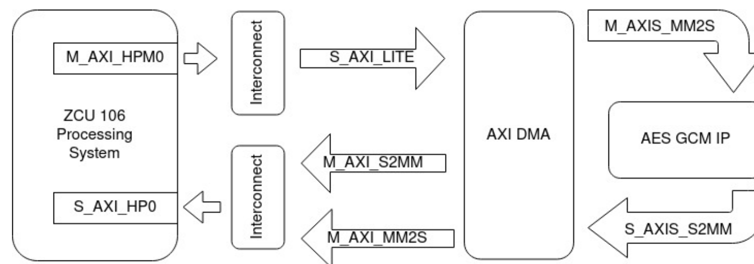


Figure 6.1: PS PL Design

more memory-mapped Slave devices. The AXI interfaces conform to the AMBA® AXI version 4 specifications from ARM®, including the AXI4-Lite control register interface subset. The Interconnect IP is intended for memory-mapped transfers only; AXI4-Stream transfers are not applicable. The AXI Interconnect IP can be used from the Vivado® IP catalog as a pcore from the Embedded Development ToolKit (EDK) or as a standalone core from the CORE Generator™ IP catalog DMA: Direct Memory Access module regulates the disparity in data transfer rates be IP: IP synthesized and exported from vivado hls after being tested against RFC test vectors Complete AES GCM implementation: Here the complete AES GCM algorithm is written in HLS and a single IP block is created, the inputs to which are IV, PT, AAD, and Tlen, and the outputs are CT and Tag. Refer to the figure in background, all the loops in each of the DOT, GHASH, and GCTR functions are optimized (loop unrolls, pipelines inserted at all appropriate locations) except for the pipeline over batches of 16B AES encryption of counter blocks and in the loop where batches of 16B plaintext are XORED with corresponding encrypted counters, which were causing data dependency errors and thereby the IP to fail in the RTL cosimulation stage.

# Chapter 7

# Evaluation

## 7.1 Evaluation

Testbench: We have used Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit for running our experiments. Based on the ZU7EV silicon part and package in the 16 nm FinFET Zynq® UltraScale+TM MPSoC, the ZCU106 is a general-purpose evaluation board for quick prototyping. The ZU7EV gadget combines a dual-core Arm Cortex-R5F real-time CPU with a quad-core Arm Cortex-A53 processing system (PS). Baseline

Container: In this setup, OpenSSL library is used to execute AES GCM on different packet sizes (64B/128B/256B) , with code executing inside a docker container , pinned to a single core of the x86 CPU to have consistency in the metrics we are comparing

PS: Here the C code for AES GCM implementation is executed over the a53 ps. ,so we are performing the computations on the PS part of the FPGA( cpu of FPGA is less powerful than a x86 processor). Even though we save on communication latency (no DMA's involved) , these numbers are worse than GCM Host .

PS+ PL : Here we synthesize the complete code of AES GCM in a single IP with all optimizations added as mentioned in section(implementation.5) ,. The computations take place over the PL part of FPGA, and data is communicated from PS to PL via DMA. PS+ PL scaled : We interpolate our results for the Xilinx SN1000 board using the fraction of LUTs utilized by our IP

# Chapter 8

# Results

## 8.1 Results

Refer below figures for Throughtput 8.2 and Latency8.1 results

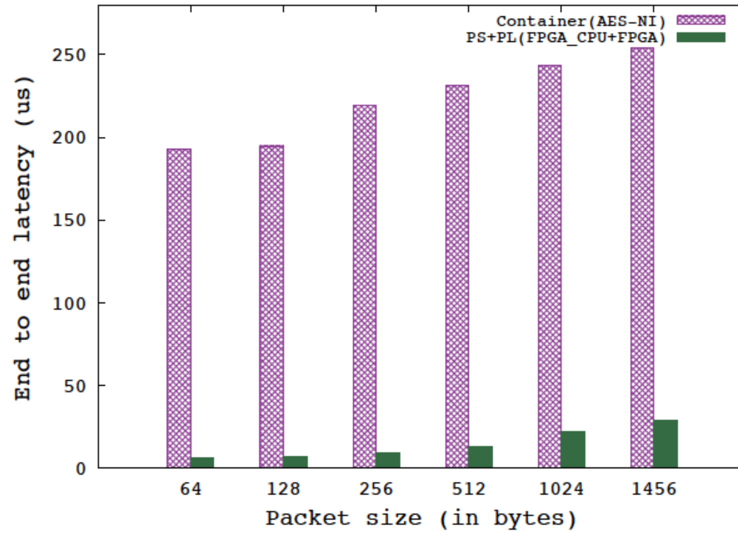


Figure 8.1: Latency

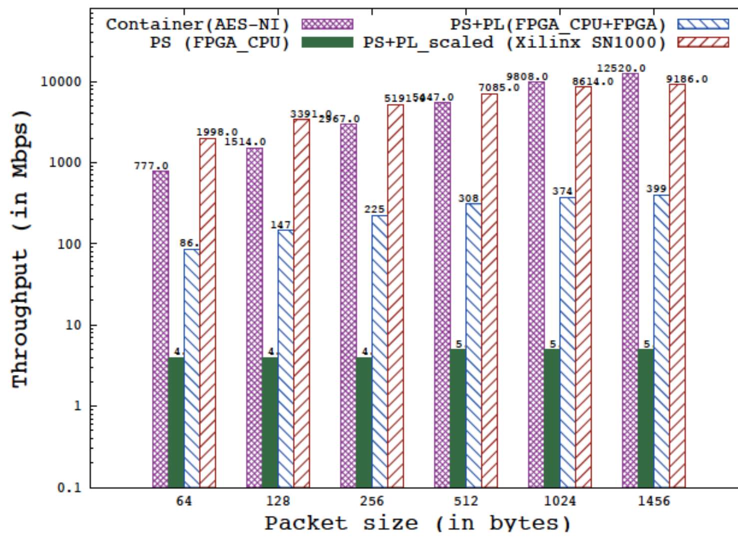


Figure 8.2: Throughput

## Bibliography


[1] B. Pismenny, H. Eran, A. Yehezkel, L. Liss, A. Morrison, and D. Tsafrir, "Autonomous nic offloads," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 18–35, 2021.

[2] D. McGrew, "An interface and algorithms for authenticated encryption," tech. rep., 2008.

[3] A. E. Standard, "Federal information processing standard (fips) publication 197," *National Bureau of Standards, US Department of Commerce, Washington, DC*, 2001.

[4] R. Shashidhara, T. Stamler, A. Kaufmann, and S. Peter, "{FlexTOE}: Flexible {TCP} offload with {Fine-Grained} parallelism," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 87–102, 2022.

[5] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "{AccelTCP}: Accelerating network applications with stateful {TCP} offloading," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 77–92, 2020.