SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluation of Data-Intensive Accelerated Functions in Computational Storage Devices

Jiong Liu

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluation of Data-Intensive Accelerated Functions in Computational Storage Devices

Bewertung von datenintensiven beschleunigten Funktionen in Computerspeichern

Author:	Jiong Liu
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Charalampos Mainas, Atsushi Koshiba
Submission Date:	15.02.2023

I confirm that this bachelor’s thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.02.2023 Jiong Liu

Acknowledgments

First of all, I am very grateful to my advisors Charalampos Mainas and Atsushi Koshiba for providing me the opportunity to write my bachelor's thesis and the exciting topic. Additionally, their guidance of debugging, implementation and testing during the time of this project is very helpful to me.

Secondly, I would like to thank my parents and friends for the emotional support during the time of project.

Last but not least, thanks should also go to Xilinx for build the powerful tools and platform.

Abstract

Following the current Big Data trend, modern storage devices can store enormous amounts of data, while still offer users high performance I/O access. These huge quantity of data must be transferred between storage and processing units multiple times due to Von-Nermann Architecture's limitations. In order to address the issue, researchers have been trying to insert computational storage devices between storage devices and computing units. This kind of device is able to processes the data "near to" the storage, thus can provides a platform for programs to conduct user-defined data pre-processing without requiring a lot of data transportation and hence give performance advantages, especially for data-intensive tasks. Despite its promises, building such a program for the user-defined accelerating activities is still a difficult task, since there has not been much study and experimenting with the appropriate API and abstraction. In order to fill up this gap on a specific device, this project focus on the Xilinx Zynq UltraScale+ MPSoC ZCU106 board, a computational storage device from Xilinx Embedded Platform, and aims to provide a software library for some data-intensive functions, so that users are able to use software combined with the hardware acceleration from this board without having any hardware knowledge. Since Xilinx has already a library, called Xilinx Vitis Library, for devices of the other kind of platform, which is the Xilinx Data Center Platform, this project ported this library to the target platform, provided software execution optimizations to some content and evaluate several data-intensive functions between execution on pure CPU and on Xilinx Data Center Platform(Xilinx Alveo U50 Data Center Accelerator Card).

The source code is available on Github. [1]

Contents

1 Introduction

Massive quantities of data can presently be stored on modern storage devices while still providing users with high-performance I/O access. Due to the Von-Nermann Architecture's restrictions, these enormous amounts of data must be transported many times between storage and processing units. To resolve this issue, the computational storage devices are installed between hosts and storage devices. This type of device may process data "near" the storage, giving programs a platform to carry out user-defined data pre-processing without using a lot of data transit. [2] This results in performance benefits, especially for tasks that process a lot of data. However, building such a program for the user-defined accelerating activities is still a difficult task, but there exists already a basic programming environment provided by Xilinx [3] in order to simplify the control of the hardware to a certain degree, which I will cover more details in the background chapter.

But the gap still exists:

- The user may want to use the hardware acceleration provided by the computational storage devices.
- The user may have little hardware knowledge and the learning curve is long.

In order to solve these issues, this project introduce a high-level software library which encapsulates the hardware acceleration in the low-level implementation and provides complete and easy-to-use interfaces. Since Xilinx has already provided a library, called Xilinx Vitis Library, which gives some implementation of the both software and hardware, this project is based on it and has the following main contributions:

1. Introduction of the high-level library based on Xilinx Vitis Library
2. Port of the security library and the data compression library from Xilinx Vitis Library
3. Providing the hardware binary file to configure the platform

This thesis report perform a evaluation for the high-level library compared to other implementations of the data-intensive functions which is running on the CPU. The CPU implementation and the FPGA implementation are compared by their I/O time and the compute time. However, Xilinx does not provides an official support the evaluation board and we tried the configuration from the community but failed to work, the evaluation of the ported library can only be done on the data center platform. The details will be explained in the Evaluation chapter.

In the following chapter, this thesis report will cover the background of this project and the also explain the basic terms from Xilinx in both software and hardware. Afterwards, the thesis report will give a short overview of each libraries and the overview of this project structure. Then, the design of the programming model including the hardware kernel execution and software part distribution will be explained. At the end, this thesis report will evaluate the libraries implemented in this project and take a view of future works.

2 Background

In this chapter, we firstly would like to explain the design flow of Xilinx Vitis Unified Software Platform which is used in this project as a building tool. Additionally, this chapter also contains basic terms as background of this project.

2.1 Vitis Unified Software Platform Overview

Xilinx Vitis Unified Software Platform [4] aims to provide a highly similar design flow for developing different devices from different platforms. It contains the structure shown in Figure 2.1 [5].

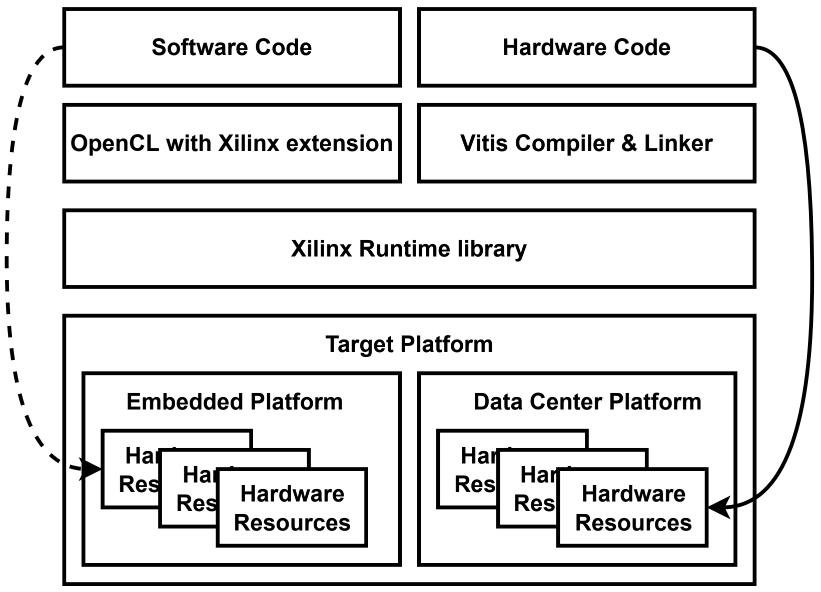


Figure 2.1: Structure of Xilinx Unified Software Platform

In this structure, the layer at bottom is the target platforms. These platforms contains hardware resources that can be represented as hardware code in Vitis HLS or RTL and configured at run time (solid arrow). These hardware resources are also the execution target of the software code (dashed arrow).

Based on the platform layer, Xilinx Runtime library (XRT) [6] is built in order to facilitates communication between the software code (running on an embedded Arm or x86 host) and the target platform. Thus, it includes user-space libraries and APIs, kernel drivers, board utilities, and firmware for the run time configuration.

Through the XRT, Xilinx provides the special OpenCL [7] version with some extensions [8] in order to simplify the design flow and abstract it to a relative high level framework. In addition, Xilinx also provides Vitis Compiler & Linker, which relies on XRT, to transfer the hardware code into configurable bit stream.

This project uses the version 2020.2 [9]. This is because the board design is using this version. If this project uses a higher version, it may cause incompatibility.

2.2 Vitis Programming

After the explanation of the Vitis Unified Software Platform, in this section, the report is going to introduce the Vitis Programming and Execution Model. A program is called Vitis Program if this program use the Xilinx Vitis Unified Platform in order to realize the software and hardware co-acceleration. In most cases, this program consists of two parts: A software program running on a standard processor such as an X86 processor, or ARM embedded processor. This part is also called the Processing System part. ; a Xilinx device binary (xclbin) containing hardware accelerated functions, or so called hardware kernels, which can be configured to the device during the run time. This part is also called the Programmable Logic part [10].

The PS part is most likely a host executable written in C++ & OpenCL and can be executed on the corresponding host CPU. The executable uses OpenCL with some Xilinx extensions to finish the interaction with the hardware kernels in the PL part. In the contrast, the hardware kernels are written in Vitis HLS or RTL (either Verilog or VHDL) and run within the PL part Xilinx devices. [11]

The interaction between PS part and PL part is illustrated in Figure 2.2 [12]. Red arrows represents the data transfer, while black arrows means the building phase. Hardware code is built by the Vitis Compiler from XRT into a hardware kernel which will be configured during the run time. Software code uses the XRT APIs to execute the XRT driver in order to transfer data to the hardware platform. After this, the hardware platform transfer the data through hardware interfaces to the kernel. The data transfer from kernel to software is only the reversed version of steps above.

2.3 Platform Explained

In Xilinx Data Center Platforms, the software runs on an x86 server and the kernels are executed in an FPGA on a PCIe-attached accelerator card, whereas in Xilinx Embedded Platforms, the software runs on an ARM processor of a Xilinx MPSoC device and the kernels are executed within the same device.

For this project, the platforms are Alveo U50 Data Center Accelerator Card [13] and Zynq UltraScale+ MPSoC ZCU106 [14].

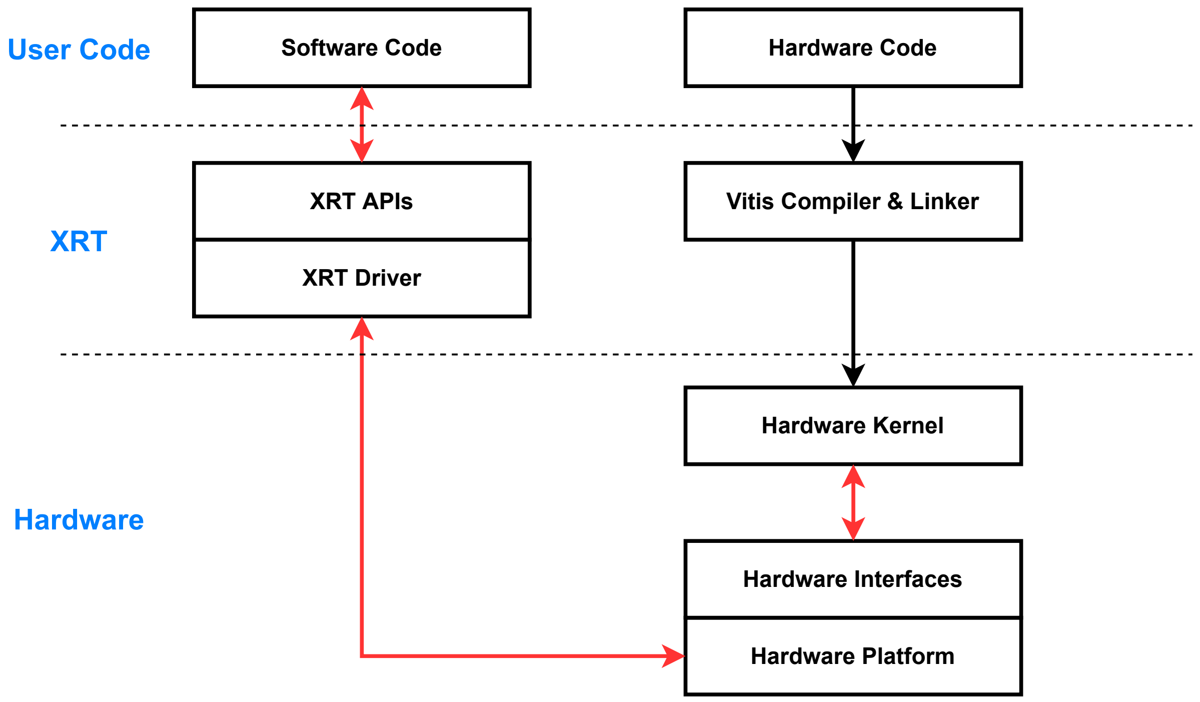


Figure 2.2: Illustration of data transfer between software code and hardware kernel

2.4 Vitis Build Process

As we all know, the compilation of C/C++ code needs two steps: compile and link. In the first step, the compiler compiles the source code into Object files (.o files). And then in the second step, the linker links the Object files into an executable file. Similar to this process, the hardware code needs also compile and link steps. But the different point is that the compiler and linker is provided by Xilinx Vitis. In the first step, the source code of hardware kernel is compiled into Xilinx Object files (.xo files) and in the second step, these Xilinx Object files are linked in order to build a FPGA binary file (.xclbin file). These two steps and the comparison to C/C++ building process is illustrated in Figure 2.3.

2.5 Vitis Build Targets

Comparing the build process for C/C++ and hardware kernels, there is one step more for hardware, namely the build target selection. There are three different build target in Vitis: software emulation, hardware emulation and hardware.

In software emulation, the hardware code is complied only to host executable and run on the host processor. There is no hardware compilation required in this build target. Thus, the speed of compilation is high. But without the hardware compilation, this build target cannot represent the real executing environment. Due to this fact, it is always used to identify syntax error and verify the correctness of the hardware code.

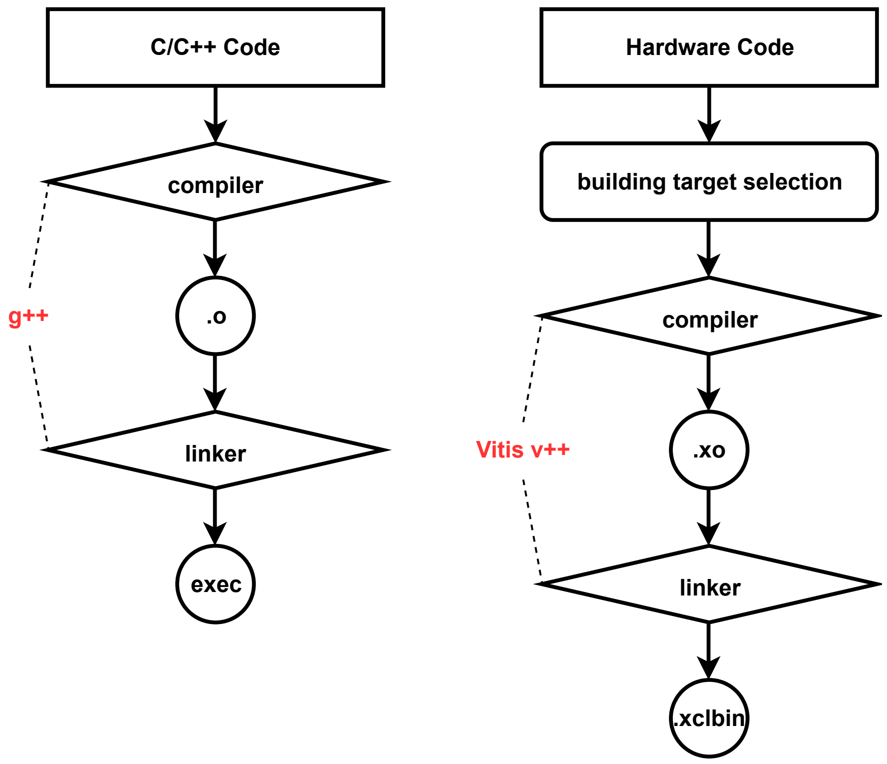


Figure 2.3: Illustration of the Vitis Build Process

In contrast to software emulation, the hardware compilation is enabled in hardware emulation. This emulation still runs in the host processor and it tries to emulate the real hardware environment. Thus the speed of compilation is a little bit slow and so as the execution speed. Due to this fact, it is always used to test the hardware correctness for small data sets.

The hardware build target is for the real hardware, so it has the slowest compilation speed. Thus, it is always the last step of the build process [15].

2.6 Vitis Library

Vitis Library [16] is provided by Xilinx and offers an open-sourced implementation of libraries of acceleration functions based on XRT. However, it targets the data center platform and does not provide a complete API for all sizes of data set. Thus, this project mainly depends on the Vitis Library and make the extension to the target platform of this project and also to all sizes of data set.

2.7 Terms for Programmable Logic

In this section, I would like to explain the terms that are used for PL part.

2.7.1 Vitis HLS

Vitis HLS [17] is essentially a tool to transfer a restricted C/C++ function into RTL for implementations in PL part. It aims to help developers to implement high quality RTL code within a relative short time. Developers can also use different Marcos to make optimizations for loops, arrays, memory IO and so on. The use cases and its purposes of optimizations in this project will be explained in the Implementation section.

Because Vitis HLS is targeted for automatic RTL code generation, it does not support some C/C++ constructs. Some of them are:

- System call
- Dynamic memory such as new and delete
- Recursive functions
- Virtual functions and pointers

2.7.2 Component & Kernel

A component is a function in Vitis HLS code for implementing one or more functionalities. A kernel is a special component. It is the top-level function and combines multiple components. In the code, a kernel must be surrounded with the keyword "extern" as shown in Figure 2.4, so that Vitis HLS can recognize it.

```
extern "C" {
    void kernel(/* parameter list */){
        // implementation
    }
}
```

Figure 2.4: An example for a kernel in Vitis HLS

In the parameter list, it is allowed to use scalars, pointers and streams. Streams is going to be introduced in section 2.6.3.

2.7.3 arbitrary precise data type

The Vitis HLS provides a data type to represent a number in arbitrary precise: the ap_int<> and ap_uint<>. An example shows how to use it:

```
ap_uint<64> number=16; // number is 64 bits long
ap_uint<1> b=true; //b is 1 bit long, either 0 or 1.
                   //Different from the bool from C/C++, which is one byte
```

Figure 2.5: An example for arbitrary precise data type

2.7.4 Stream

hls::stream

Vitis HLS provides hls::stream as a temporary data storage between components as shown in Figure 2.5. It is also a precondition to realize the dataflow optimization which realize that different components can work at the same time without waiting the dependent component to finish.

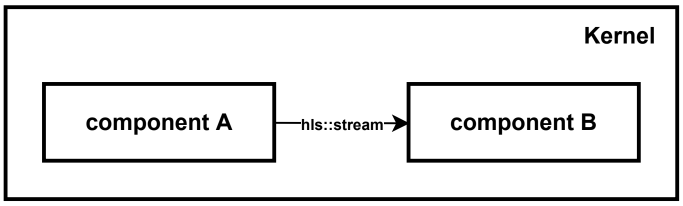


Figure 2.6: Illustration of components connection using hls::stream

AXI Stream

AXI Stream is used for inter-kernel communication as illustrated in Figure 2.6. It can transfer data from one kernel to another. The specific use case is to build the multi-kernel model, which a kernel is responsible for getting the data from the buffer and pushing it on the AXI bus, another kernel processes the data available on the AXI bus and pushes the result also on the AXI bus and the rest one gets the result from the AXI bus and writes it into the target buffer.

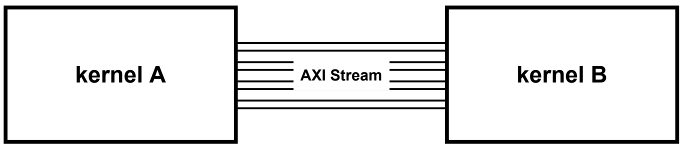


Figure 2.7: Illustration of kernel connection using AXI stream

2.8 Terms for Processing System

In this section, I would like to explain the terms that are used for PS part.

2.8.1 OpenCL

Kernel

OpenCL provides the cl::Kernel object as the virtual mapping of the hardware kernel. This object can control the behaviour of the hardware kernel during the run time, such as specify the parameter list, schedule the execution together with the command queue.

Buffer

OpenCL provides the cl::Buffer object as the virtual mapping of the global memory in the PL part. It supports either creating and then mapping a memory buffer with the same size into the program's virtual address space or using directly the page aligned memory buffer in the program's virtual address space and then mapping it to the PL part. After the creation, data transfer can be scheduled together with the command queue.

Command Queue & Event

OpenCL provides the cl::CommandQueue object as the global schedule and execution. Normally, the execution order of the scheduled tasks is the same as the order of the time where they are scheduled. However, with the command queue configured with "OUT_OF_ORDER" property and the OpenCL events, an asynchronous execution could be build in order to overlap the kernel execution. This use case of this optimization approach will be explained in the following section. A example for out of order command queue is shown in Figure 2.7.

Squares identify tasks to be executed, while circles are OpenCL events. The blue arrow means that after the task finished, the event that the arrow points will be set to status "complete". However, the red arrow means the task this arrow points to can only be executed when the event is completed. Thus, the task B and C can be executed in parallel.

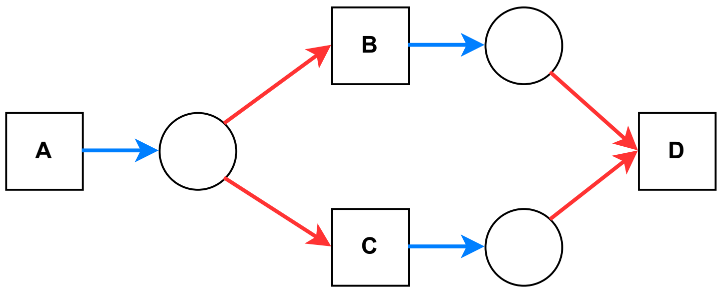


Figure 2.8: Illustration of components connection using hls::stream

2.9 Execution Model

A simple execution model can be summarized into following steps:

- The host application writes the data needed by a kernel into the global memory of the FPGA device.
- The host program sets up the input parameters of the kernel.
- The host program triggers the execution of the kernel.
- The host program transfers data from global memory back into the host memory.

2.10 Security Library

2.10.1 DES & AES algorithm overview

DES and AES are two types of Symmetric-key algorithm. That means that the encryption and decryption use the same key. The length of the key for DES is 64 bits and the size of blocks is also 64 bits, while for AES, the length of the key can be one of 128, 192 and 256 bits and the size of blocks is 128 bits. The encryption and decryption algorithm and its implementation is not part of this project. The hardware Vitis HLS code is provided by Xilinx.

There are five post-processing modes of DES and AES, which are Electronic Codebook Book (ECB) [18], Cipher Block Chaining (CBC) [19], Counter (CTR) [20], Cipher Feedback (CFB) [21] and Output Feedback (OFB) [22]. The workflow of them is shown in Figure 2.9 2.13 [23].

2.11 Data Compression Library

2.11.1 File structures of different compression algorithms

In order to explain the process of the data compression, I display in the following subsections the file structure of each data compression algorithm.

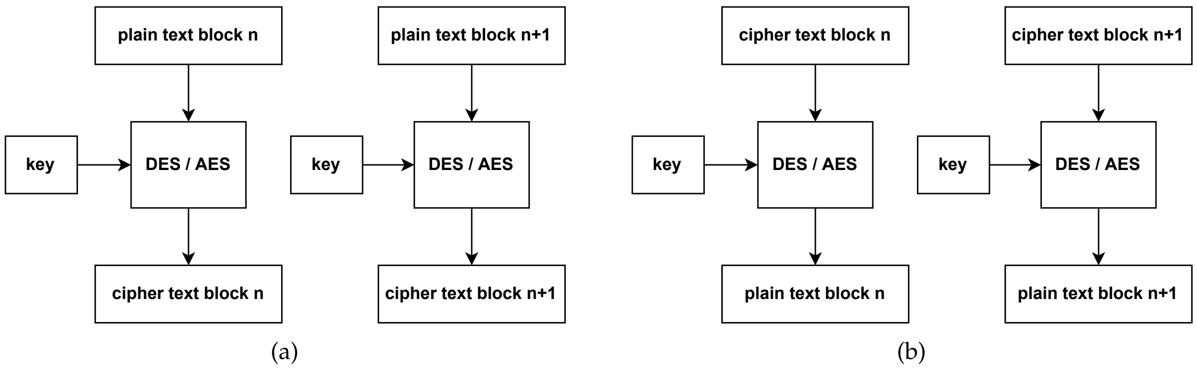


Figure 2.9: Electronic Codebook Book (ECB) encryption (a) and decryption (b)

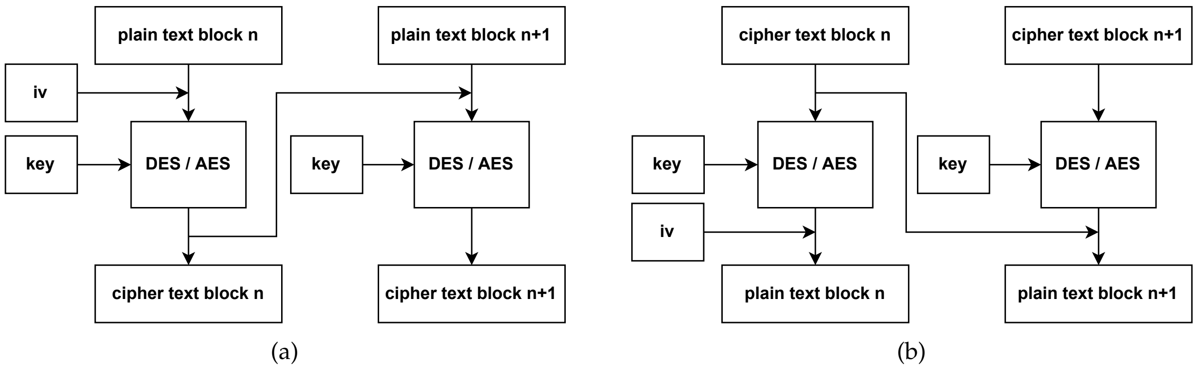
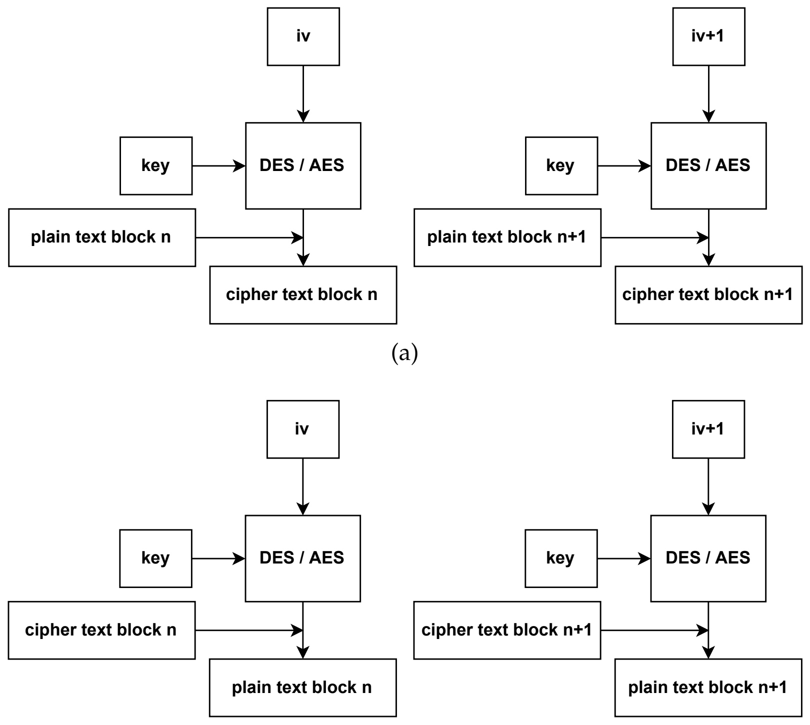


Figure 2.10: Cipher Block Chaining (CBC) encryption (a) and decryption (b)



(b)

Figure 2.11: Counter (CTR) encryption (a) and decryption (b)

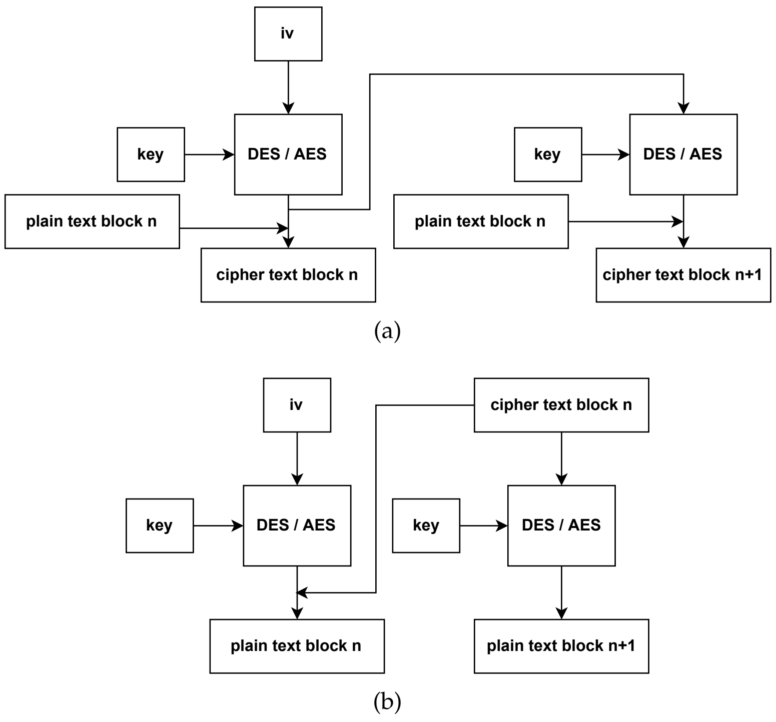


Figure 2.12: Cipher Feedback (CFB) encryption (a) and decryption (b)

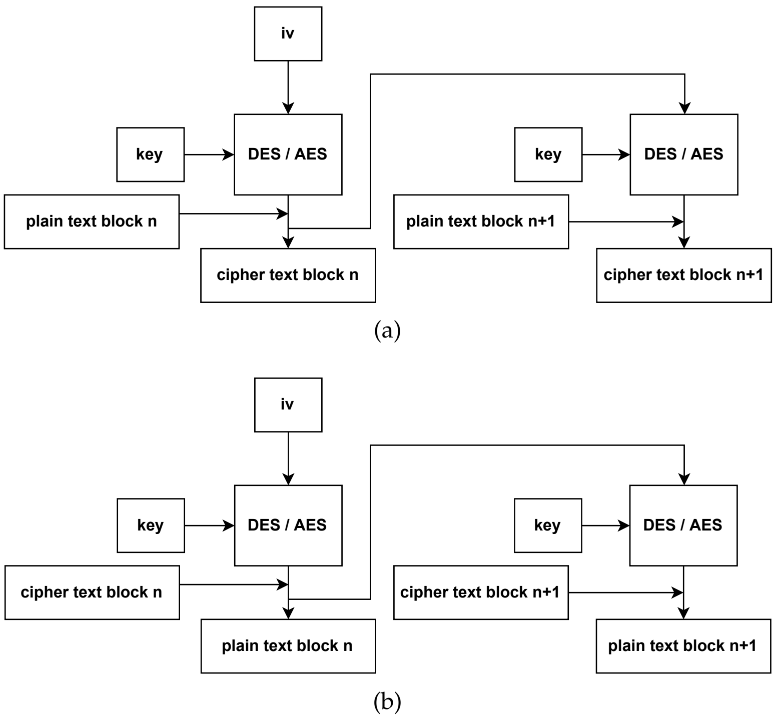


Figure 2.13: Output Feedback (OFB) encryption (a) and decryption (b)

Gzip

The Gzip file structure is shown in Figure 2.14.

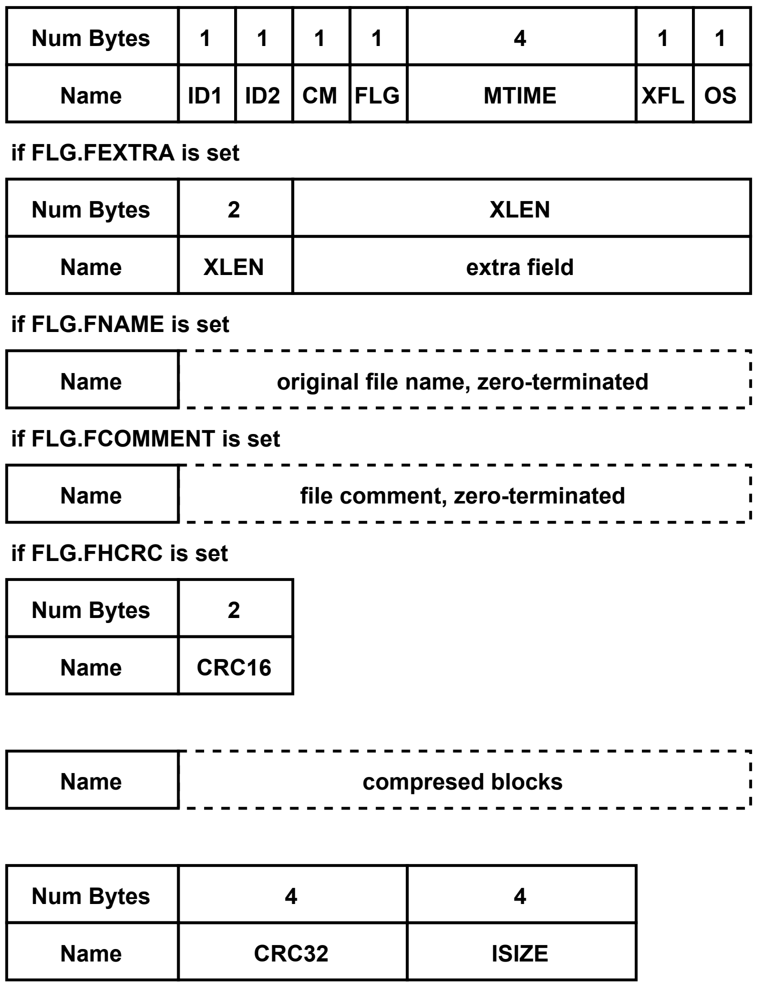

Num Bytes | 1 | 1 | 1 | 1 | 4 | 1 | 1
Name | ID1 | ID2 | CM | FLG | MTIME | XFL | OS
if FLG.FEXTRA is set
Num Bytes | 2 | XLEN
Name | XLEN | extra field
if FLG.FNAME is set
Name | original file name, zero-terminated
if FLG.FCOMMENT is set
Name | file comment, zero-terminated
if FLG.FHCRC is set
Num Bytes | 2
Name | CRC16
Name | compresed blocks
Num Bytes | 4 | 4
Name | CRC32 | ISIZE

Figure 2.14: Gzip file structure

The FLG byte structure is shown in Figure 2.15.

The first part of the Gzip file is the header containing two bytes identifiers, namely ID1 and ID2, and the value is fixed: ID1=0x1f and ID2=0x8b. These identifier aims to identify that this file is a Gzip file. After the identifiers follows the compressed method CM whose value is fixed to 8 which means the algorithm uses the deflate method. Then comes the FLG for the configuration of the information which follows this part. Finally, at the end of this part, there are modification time MTIME, extra flag XFL and the operating system OS. In this project, the FLG is set to 8, MTIME remains 0 for simplicity, XFL is set to 0 and OS is set to 3, which means UNIX operating system.

After the first part, three optional parts which depend on the FLG follows. In this project, the file name is always set to 'x' for simplicity. Then, the compressed blocks are added after

0	FTEXT
1	FHCRC
2	FEXTRA
3	FNAME
4	FCOMMENT
5	reserved
6	reserved
7	reserved

Figure 2.15: Gzip FLG bitmap

the optional parts. At last, the last part which contains the chunk sum for the compressed data and the original size is added to the tail of the file.

Although the compression will generate only a specific version of Gzip file, the decompression is targeted to all versions of Gzip file.

Lz4

The Lz4 file contains several Lz4 frames. In Figure 2.16, a Lz4 frame structure is displayed.

Num Bytes	4	3~15
Name	Magic Number	Frame Descriptor

Name	compresed blocks

Num Bytes	4	0~4
Name	End Mark	Checksum

Figure 2.16: Lz4 frame structure

The 4 bytes Magic Number is fixed to 0x184D2204 in order to identify that this file is a Lz4 file. There is a Frame Descriptor with variable length from 3 bytes to 15 bytes. This will be explained in the next paragraph. The compressed blocks will be added after the Frame Descriptor. And the End Mark and the Checksum follow after them. The End Mark is fixed to 4 bytes of 0x00 and the length of the Checksum depends on the FLG in the Frame Descriptor.

Num Bytes	1	1	0~8	0~4	1
Name	FLG	BD	Content Size	Dictionary ID	HC

Figure 2.17: Lz4 frame descriptor structure

0	DictID
1	reserved
2	C.Checksum
3	C.Size
4	B.checksum
5	B.indep
6	version
7	

(a)

0	reserved
1	
2	
3	
4	Block MaxSize
5	
6	
7	reserved

(b)

Figure 2.18: Lz4 FLG bitmap (a) and BD bitmap (b)

In Figure 3.10, the structure of the Frame Descriptor is shown. It contains a 1 byte FLG, BD and HC. The Content Size and the Dictionary ID is optional and whether they are included depends on the corresponding bits in the FLG byte. In Figure 2.17, I display the bitmap of the FLG byte and the BD byte. The DictID is not considered in this project. For the compression, the C.Checksum (content checksum), C.Size (content size) and B.Checksum (block checksum) are not set and the B.Indep (block independence) is set. Thus, the Content Size and the Dictionary ID do not exist in this project for simplicity. From the BD byte, we can get the BlockMaxSize. In this project, the maximum size of blocks is fixed to 64 KB for simplicity.

The data blocks contains a 4-bytes block size and then block size bytes data follows. Whether there is a block checksum, depends on the B.Checksum bits in the Frame Descriptor.

Num Bytes	4	block size	0~4
Name	block size	data	block checksum

Figure 2.19: Lz4 Data Block structure

Although the Lz4 compression in this project generates a specific version of Lz4 file, the decompression of Lz4 is targeted to all types of Lz4 files.

Snappy

A snappy file consist of chunks. There are 3 mainly used chunks: stream identifier, uncompressed data chunk and compressed data chunk. The structure of a chunk is generally the same and displayed in Figure 2.20.

Num Bytes	**1**	**3**	**chunk size**
Name	**ID**	**chunk size**	**data**

Figure 2.20: Snappy Chunk structure

There is a chunk identifier at the front of a chunk. This identifier is fixed for each type of chunks: stream identifier has the chunk identifier 0xff, the compressed chunk has the chunk identifier 0x00 and the uncompressed chunk has the chunk identifier 0x01. After the chunk identifier follows the chunk size which is three bytes long. Finally, the data is added after the chunk size. The data for the stream identifier chunk is always the same and is a string of "sNaPpY" in ASCII. The data of compressed chunk and uncompressed chunk is the corresponding the compressed data and the uncompressed data.

Since there is no additional configuration of the snappy file, the compression and the decompression in this project are targeted to all data sets and all snappy files.

Zstd

A Zstd file contains multiple Zstd Frames. A Zstd Frame starts with a Magic Number 0xfd2fb528. Then comes the Frame Header whose length varies from 2 bytes to 14 bytes. After the Frame Header follow the data blocks. At the end of the Zstd Frame may exist a Content Checksum. The existence of the checksum depends the flag in the Frame Header.

Num Bytes	**4**	**2~14**
Name	**Magic Number**	**Frame Header**

Name	**compressed blocks**

Num Bytes	**0~4**
Name	**Content Checksum**

Figure 2.21: Zstd Frame structure

The Zstd Frame Header starts with a Frame Header Descriptor whose bitmap is shown in Figure 2.23. The following three parts are optional and whether they should be provided

Num Bytes	1	0~1	0~4	0~8
Name	Frame Header Descriptor	Window Descriptor	Dictionary ID	Frame Content Size

Figure 2.22: Zstd Frame Header structure

0	Dictionary ID flag
1	
2	Content Checksum Flag
3	Reserved bit
4	Unused bit
5	Single Segment Flag
6	Frame Content Size Flag
7	

Figure 2.23: Zstd Frame Header Descriptor bitmap

depends on the Frame Header Descriptor. When the Frame Content Size Flag is 1, 2 or 3, then the length of the Frame Content Size is 2, 4, or 8. Under the condition that the Frame Content Size Flag is 0, it is a little bit complex: if the Single Segment Flag is set, then the length of the Frame Content Size is 1. Otherwise the length of the Frame Content Size is 0, which means that it it not provided. The length of the Dictionary ID depends on the Dictionary ID flag and the value of it is $2^{\text{value of Dictionary ID flag}}$. Whether the Window Descriptor exists is the reversed condition of the Single Segment Flag. When the Single Segment Flag is set, the Window Descriptor is not provided and vice versa.

3 Overview

3.1 Basic Information

This project evaluates three libraries of data-intensive functions which are security library, data compression library and the database library.

- security library: one of the most famous encryption and decryption algorithm, namely Data Encryption Standard (DES) [24] and Advanced Encryption Standard (AES) [25], is accelerated with different modes and key length.
- data compression library: four different compression and decompression algorithms are accelerated. These algorithms are Gzip [26], Lz4 [27], snappy [28] and Zstd [29].
- database library: the basic database operations like join and aggregation is accelerated.

3.2 General Structure

This project contains a four layer structure that is shown in Figure 3.1.

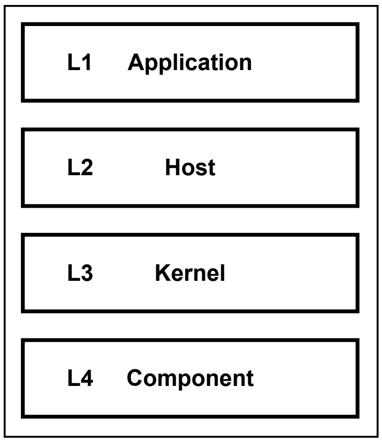


Figure 3.1: Illustration of components connection using hls::stream

The explanation from bottom to top:

- Layer 4, the hardware component layer. This layer contains hardware components that are available to be built into a kernel.

- Layer 3, the hardware kernel layer. This layer combines the components from Layer 4 and build the hardware kernels that can be executed from PS part.
- Layer 2, the host layer. This layer executes the hardware kernels from Layer 2 and provides the functionality to process small data set in blocks.
- Layer 1, the application layer. This layer executes the functionality from L3 blocks by blocks in order to process all size of the data sets. Additionally, this layer also provides a complete and user friendly API for developers to use.

3.3 Detailed overview for Security Library

3.3.1 Functionality of each layer

- Layer 4: The hardware implementation of DES and AES algorithms.
- Layer 3: Utilization of the DES and AES components to build hardware kernel for different processing modes.
- Layer 2: Execution the hardware kernel to process a chunk of data.
- Layer 1: Dividing the data set that needs to be processed into multiple chunks and using the Layer 3 to process it.

3.3.2 Abstract workflow

Users firstly give the data to Layer 1. Then Layer 1 executes the API from Layer 2 specific for blocks and record the initial vector so that the users can input continuously without calculating it themselves. Layer 3 initializes the hardware, creates the virtual hardware object for execution and finally triggers the hardware to execution. In the hardware, the Layer 2 kernel gets the data from the buffer and gives it to the hardware component provided by Xilinx. The result is also written by the kernel so that the Layer 3 can have access to it.

3.4 Detailed overview for Data Compression Library

3.4.1 Functionality of each layer

- Layer 4: The hardware implementation of the compression and decompression algorithms provided by Xilinx.
- Layer 3: the hardware kernels provided by Xilinx.
- Layer 2: Execution the hardware kernel to process the whole set of data.
- Layer 1: Global Control of the host components designed in Layer 2.

3.4.2 Abstract workflow

Since the result of the compression and decompression is unknown, this thesis report provides a solution to solve it, namely to let the processing of the data run in a separate thread, which is implemented by Layer 3. The Layer 4 is responsible for the creation and the stop of the thread as well as the deal with the user input and output. The Layer 3 takes the user input as a stream and processes it in a similar way as the security library: it divide the input into blocks and for each blocks, the kernel will be executed once. The hardware can be treated as a block box which accepts the input from the user and returns the processed data as result. This result will be output by Layer 3 also as stream and the Layer 4 will give it out to user.

3.5 Design Goal

3.5.1 High Level

The goal of this project is to hide the hardware implementation from the user in order to decrease the learning curve of using hardware acceleration. Thus, the software library has to be on a high level and provides an abstract API. It should abstract the implementation details for the hardware implementation to a interface that is available for users who are assumed to have no hardware basic knowledge.

3.5.2 Usability

Although the software library should hide the hardware implementation of the acceleration, it should also not provide a complex user interface. Otherwise, the user still needs to spend huge amounts of time to keep in mind how to use it and the learning curve is also increased, which is the opposite way of the goal of this project. Thus, the interface of the software library should not be too difficult to use. The learning curve should be small and the interface should be designed to be similar with some of the CPU implementations.

3.5.3 Performance

The CPU implementations which are the benchmarks of this project have also a plenty of optimizations and is relative fast. The performance goal of this project is that the performance should be at least within an order of magnitude of the performance of the CPU implementation. Thus, the software library should bring an acceptable performance, since it uses the hardware acceleration for those data-intensive functions.

4 Design

4.1 Security Library

4.1.1 Hardware Kernel Design

In this section, the report will explain the hardware kernel design for encryption and decryption in two different versions.

Kernel API

The kernel needs 2 memory buffer which are the buffer for plain text and the buffer for cipher text. Because both the encryption and the decryption need these buffers, the kernel API can be the same. The only difference is that for the encryption kernel, the input buffer is the plain text and the otuput buffer is the cipher text, while for the decryption kernel, the input is the cipher text and the output is the plain text. Additionally, the kernel needs also the key and the initial vector. Because the key can be 128 bits, 192 bits and 256 bits long and the initial vector is 128 bits long, it is not able to store them in a single scalar like a long int. Thus, they also need buffers to be stored. The summary of the kernel API is shown in Figure 4.1 and 4.2.

```
extern "C"{
void kernel(
        ap_uint<128> *inputTextBuffer,
        ap_uint<128> *cipherKey,
        ap_uint<128> *initVec,
        ap_uint<128> *outputTextBuffer,
        int size);
}
```

Figure 4.1: Kernel API with initial vector

Dataflow

The design of the kernel using dataflow is shown in Figure 4.3. There are three components in this design: MM2S (memory to stream reader), the corresponding DES or AES component for a specific mode and the S2MM (stream to memory writer). The green filled arrows represent hls::stream. The MM2S reads the data from the global memory and pushes them to the hls::stream connected to it. S2MM works similarly but reversed: it pops data from hls::stream

```
extern "C"{
void kernel(
        ap_uint<128> *inputTextBuffer,
        ap_uint<128> *cipherKey,
        ap_uint<128> *outputTextBuffer,
        int size);
}
```

Figure 4.2: Kernel API without initial vector

and writes them back to the global memory. The core component, DES or AES component, pops data from hls::stream and processes them. Finally, the result will be pushed into the hls::stream connected to the component S2MM.

The advantage of this design is that each component can work independently with other components. Thus, the read operation from the global memory and the write operation to the global memory in S2MM is deposit from the the main processing component, thus providing a theoretically higher throughput. But, this design uses external memory to store the temporary data in the stream, which means that this design uses more resources. Additionally, the maximum depth of the stream is fixed. If a component want to put more data to a stream than it can store, this operation is undefined and will result in the crash of the whole PL part. Due to these facts, it is more suitable to use this design on the process that depends on the previous result.

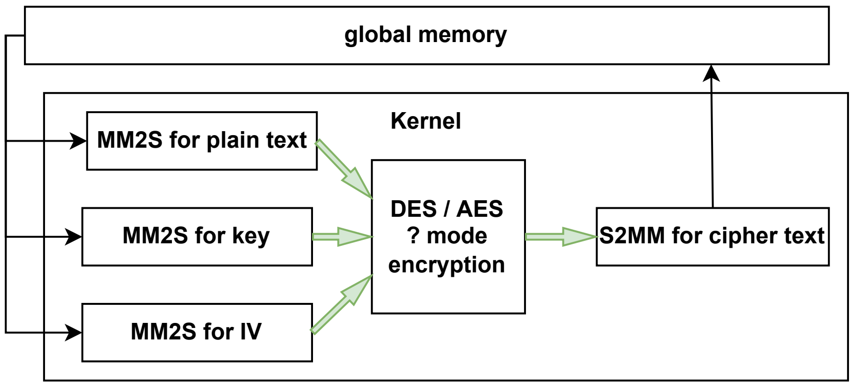


Figure 4.3: Illustration of encryption kernel using dataflow

Memory

The design of the encryption kernel using memory is shown in Figure 4.4. In this design, there is only one component, which is the DES or AES component corresponding to the specific mode. This component is modified from the one which accepts the hls::stream as input. This design accesses the global memory when needed, for example, when the plain text, key and initial vector is going to be used, thus is more suitable for the process that does not depend on the previous results. But the disadvantage of this design is that the read and write operation from and to the global memory needs the time of one clock, thus the further operations need to wait on that. However, this design does not use any external resources.

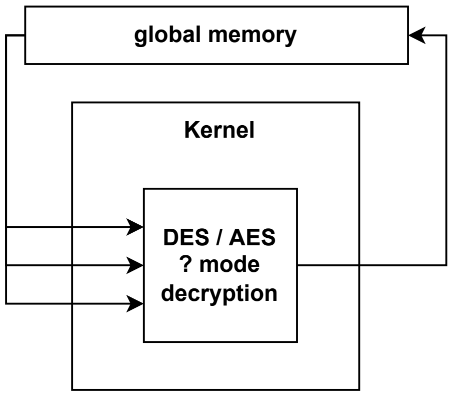


Figure 4.4: Illustration of kernel using memory

4.1.2 Host Layer Design

This layer is responsible for encrypt or decrypt a chunk from the PS part. I design the workflow for kernel with and without initial vector:

1. create a buffer at PL part and its mapped buffer at PS part
2. initialize the buffer to the corresponding values
3. migrate the values in PS part buffer to PL part
4. set the kernel parameter properly (with or without initial vector)
5. trigger the kernel to execute
6. migrate the result from PL part to PS part
7. copy the result in the target buffer

Since the kernels have the only two APIs with and without initial vector, the API in Figure 4.5 is designed for process the data in a single chunk for AES.

For DES, the API is similar. Here, I use two Enum types: the Type is for selecting encryption or decryption and the KeyLength is for selecting the length of the key.

```
template<Type T, KeyLength K>
void aesWithIV(uint64_t *in, uint64_t *key, uint64_t *iv, uint64_t *out,
    uint32_t size);

template<Type T, KeyLength K>
void aesWithoutIV(uint64_t *in, uint64_t *key, uint64_t *out, uint32_t size);
```

Figure 4.5: host API for AES

```
enum class Type{ENC, DEC};
enum class KeyLength{U128, U192, U256};
```

Figure 4.6: Enum class detail

4.1.3 Application Layer Design

This layer combines the processing of a single chunk together, increase the targeted data set to all possible sizes and provides a complete API for user. Using the CBC mode encryption as an example, I design the API in Figure 4.7.

User only needs to provide the initial vector for the first chunk using the first methods. For the other chunks, the initial vector can be calculated automatically. The initial vector is stored in a global buffer. The calculation of the initial vector is different for different mode and this will be explained in the chapter Implementation.

```
void aesEncrypt(void *src, void* dest, uint32_t size, void *key,
    uint32_t keyLength, void* iv);
void aesEncrypt(void *src, void* dest, uint32_t size, void *key,
    uint32_t keyLength);
```

Figure 4.7: Application API for AES CBC mode as an example

4.2 Data Compression Library

4.2.1 Hardware Kernels

Although the hardware kernels are provided by Xilinx, it is still necessary to explain their design here, in order to introduce the corresponding implementation of kernel's usage.

Gzip

The Gzip compression kernel accepts a segment of memory as input and processes this segment as a data block. The result of the Gzip compression kernel is the compressed data block and the checksum data of this data block.

```
void xilGzipZlibCompress(const ap_uint<GMEM_DWIDTH>* in,
                 ap_uint<GMEM_DWIDTH>* out,
                 uint32_t* compressd_size,
                 uint32_t* checksumData,
                 uint32_t input_size,
                 bool checksumType);
```

Figure 4.8: API of Gzip compression kernel

However, the Gzip decompression kernel read the whole data set from the AXI stream and puts the output also on the AXI stream.

```
void xilGzipZlibDecompress(hls::stream<ap_axiu<16, 0, 0, 0> >& inaxistreamd,
              hls::stream<ap_axiu<MULTIPLE_BYTES * 8, 0, 0, 0> >& outaxistreamd);
```

Figure 4.9: API of Gzip compression kernel

In order to put the data set on the AXI stream and read the result provided by the Gzip decompression kernel, two additional kernels, MM2S and S2MM, are needed to finish the mentioned task. the character "S" in the names is not the hls::stream but the AXI stream. These three kernels build a kernel to kernel communication via the AXI bus.

```
void xilMM2S(uintMemWidth_t* in,
             uint32_t inputSize,
             uint32_t last,
             hls::stream<ap_axiu<c_inStreamDwidth, 0, 0, 0> >& outStream);
void xilS2MM(uintMemWidth_t* out,
             uint32_t* encoded_size,
             uint32_t* status_flag,
             uint32_t read_block_size,
             hls::stream<ap_axiu<OUTPUT_BYTES * 8, 0, 0, 0> >& inStream);
```

Figure 4.10: API of MM2S and S2MM kernel

The kernel to kernel communication is similar to component to component communication but essentially different. The communication between components is through the hls::stream and it is an API in order to simplify the FIFO structure in hardware. The data transfer via hls::stream is still within the kernel. However, the communication between kernels uses hardware resources outside the kernels: the AXI bus. In order to make the communication between kernels possible, special configuration for the AXI stream connections need to be provided during the hardware link phase as the code in Figure 4.12.

```
sc=xilMM2S_1.outStream:xilGzipZlibDecompressStream_1.inaxistreamd
sc=xilGzipZlibDecompressStream_1.outaxistreamd:xilS2MM_1.inStream
```

Figure 4.11: Gzip kernels connection configuration

Lz4

The API of Lz4 compression and decompression kernel are similar. They first need the size of the uncompressed block. As Input of the compression kernel, the uncompressed data is divided in to blocks and the size of input buffer is given. Thus, the number of blocks can be calculated. The result of the compression kernel is the compressed data blocks with their size stored in an integer array. The compressed data block is aligned to size of the uncompressed block.

```
void xilLz4Compress(const xf::compression::uintMemWidth_t* in,
              xf::compression::uintMemWidth_t* out,
              uint32_t* compressd_size,
              uint32_t* in_block_size,
              uint32_t block_size_in_kb,
              uint32_t input_size);
```

Figure 4.12: API of Lz4 compression kernel

Similarly, the decompression kernel needs the compressed data blocks aligned to the size of uncompressed block as the input. But in the decompression kernel, the number of blocks is directly asked to provide. The result of the decompression kernel is the uncompressed data with their sizes stored in an integer array.

```
void xilLz4Decompress(const ap_uint<MULTIPLE_BYTES * 8>* in,
                ap_uint<MULTIPLE_BYTES * 8>* out,
                uint32_t* in_block_size,
                uint32_t* in_compress_size,
                uint32_t block_size_in_kb,
                uint32_t no_blocks);
```

Figure 4.13: API of Lz4 decompression kernel

Snappy

The API of the Snappy compression and decompression kernel are almost the same as the Lz4 compression and decompression kernels.

```
void xilSnappyCompress(const xf::compression::uintMemWidth_t* in,
                   xf::compression::uintMemWidth_t* out,
                   uint32_t* compressd_size,
                   uint32_t* in_block_size,
                   uint32_t block_size_in_kb,
                   uint32_t input_size);

void xilSnappyDecompress(const ap_uint<MULTIPLE_BYTES * 8>* in,
                   ap_uint<MULTIPLE_BYTES * 8>* out,
                   uint32_t* in_block_size,
                   uint32_t* in_compress_size,
                   uint32_t block_size_in_kb,
                   uint32_t no_blocks);
```

Figure 4.14: API of Snappy compression and decompression kernel

Zstd

The Zstd compression and decompression kernels are all based on kernel to kernel communication and use the same kernels as Gzip to read and write data from AXI bus. The Zstd compression kernel reads a segment of data and compress it to multiple Zstd frames. However, the Zstd decompression kernel can only accept one kernel at one time. The API of these two kernels is listed in Figure.

```
void xilZstdCompress(
       hls::stream<ap_axiu<STREAM_IN_DWIDTH, 0, 0, 0> >& axiInStream,
       hls::stream<ap_axiu<STREAM_OUT_DWIDTH, 0, 0, 0> >& axiOutStream);

void xilZstdDecompress(
       hls::stream<ap_axiu<c_instreamDWidth, 0, 0, 0> >& inaxistreamd,
       hls::stream<ap_axiu<c_outstreamDWidth, 0, 0, 0> >& outaxistreamd);
```

Figure 4.15: API of Snappy compression and decompression kernel

4.2.2 Host Layer Design

The data compression and decompression process will not operate on the same data block for a second time. Thus, I design the compression and decompression process based on stream.

There is a workshop that is responsible for the process of compression or decompression. The workshop consists of a data analyzer and a kernel executor. The data analyzer reads data from the input stream and processes it according to the specific algorithm and the kernel API in roder to prepare for the kernel execution. The result of processing will be stored in a

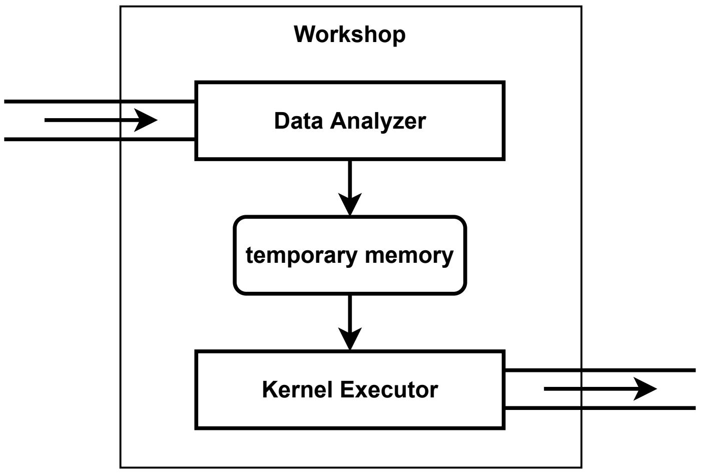


Figure 4.16: Illustration of the Data Compression Host Layer Design

temporary memory. Then the Kernel executor will trigger the kernel to execute. The result of the kernel executor will be pushed into the output stream.

4.2.3 Application Layer Design

This promise of this layer is to provide a complete API for user to use. Thus, this layer provides methods for adding the header and footer of some compress algorithms. Additionally, this layer also provides an easy-to-use API for user to write data that is not processed and read the result. For the Gzip compress algorithm, the API looks like the code shown in Figure .

```
uint32_t writeGzipHeader(uint8_t* out);
uint32_t writeGzipFooter(uint8_t *out, uint32_t fileSize);

void pushGzipCompression(void* src, uint32_t size, bool first, bool last);
uint32_t popGzipCompression(void *dest, uint32_t size, bool &last);
void pushGzipDecompression(void* src, uint32_t size, bool first, bool last);
uint32_t popGzipDecompression(void *dest, uint32_t size, bool &last);
```

Figure 4.17: API of Application layer: Gzip

5 Implementation

In this chapter, the report will explain the implementation details of the design from the previous chapters.

5.1 Application

This class stores the basic information for the XRT. The information includes: the device information encapsulated in the OpenCL object cl::Device; the context information of the runtime encapsulated in the OpenCL object cl::Context and the configuration of the PL part encapsulated in the OpenCL object cl::Program. It is worth noting that the cl::Device and the cl::Context is constant during the run time, while the cl::Program object can be change in order to change the configuration of the PL part, thus change the hardware kernel the user can use. Due to these facts, I implemented the Application class as a Lazy Singleton, which means that there is only one instance of this class during the runtime and the initialization is on demand. Thus, the copy constructor and the copy assignment should be set to delete. In order to avoid race condition for multi-threading circumstances, the initialization of the instance should be done by only one thread and the synchronization is necessary. However, even if the synchronization is needed, we can use a trick to avoid using locks: the initialization of the local static variable. During the initialization of the local static variable, the compiler will automatically avoid multiple threads performing the initializing operation, resulting in a more elegant and easy to understand Singleton implementation shown in Figure 5.1.

```
class Singleton{
private:
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) =delete;
public:
    static Singleton* getInstance(){
        static Singleton instance; // only after C++11
        return &instance;
    }
};
```

Figure 5.1: Singleton implementation

5.2 Pointer

Because the OpenCL 2 follows the C++ programming technique RAII (Resource Acquisition Is Initialization). This means that the resource allocation should be performed during the instance construction and the resource release should be performed during the instance deconstruction. This technique ensures that the possibility that the resource may leak is relative low. However, the locally created variable will be freed when the program goes out of the scope where the variable is defined because of this technique. But, sometimes people want it lifetime to be longer. This class resolve this issue by encapsulating a smart pointer inside. It has three sub classes which are KernelPointer, BufferPointer and CommandQueuePointer. These subclass extend the OpenCL provided class and add helper functions inside it to make it easier to use.

5.3 Pool

This class aims to collect the previous class: Pointer in order to provide a easy to use tool for implementation of huge number of instances.

5.4 Security Library API

5.4.1 hardware kernels

MM2S

In Vitis HLS, the MM2S is simply a for-loop that reads the data one by one and push it to the hls::stream. Since two loop is indepentdent with each other, I make the loop work in pipeline mode and write the marco "#pragma HLS pipeline II = 1", in order to make sure that the initial interval of two loops is one clock of time. The endStream is a hls::stream of bool variables. It indicates whether the data at the same position in the dataStream is valid or not.

```
void mm2s(ap_uint<128>* in, int inputSize,
    hls::stream<ap_uint<128>> &dataStream,
    hls::stream<bool> &endStream)
{
        for(int i=0;i<inputSize;i++){
                dataStream<<in[i];
                endStream<<0;
        }
        endStream<<1;
}
```

Figure 5.2: MM2S implementation

S2MM

Similar to MM2S, S2MM works in a similar way. It is a for-loop that pops the data from the hls::stream and writes it into the global memory.

```
void s2mm(ap_uint<128>* out,
    hls::stream<ap_uint<128>> &dataStream,
    hls::stream<bool> &endStream)
{
        int cnt=0;
        bool e;
        endStream>>e;
        while(!e){
                dataStream>>out[cnt++];
                endStream>>e;
        }
}
```

Figure 5.3: S2MM implementation

Modification of Xilinx provided component

Xilinx provides components with similar APIs such as the API shown in Figure 5.4.

```
template <unsigned int _keyWidth = 256>
void aesCbcEncrypt(
    // stream in
    hls::stream<ap_uint<128> >& plaintext,
    hls::stream<bool>& plaintext_e,
    // input cipherkey and initialization vector
    hls::stream<ap_uint<_keyWidth> >& cipherkey,
    hls::stream<ap_uint<128> >& initialization_vector,
    // stream out
    hls::stream<ap_uint<128> >& ciphertext,
    hls::stream<bool>& ciphertext_e);
```

Figure 5.4: Example Xilinx's API for AES

However, the key stream and the initial vector stream is read only once. Thus, they can be changed into memory access and the read operations of them should be changed from reading from hls::stream to reading from memory. For the dataflow design, the rest is still the same.

For the memory design, it does not depends on the hls::stream, so the data stream should be transformed into memory and the end identifier stream is not necessary and removed. Instead of the end identifier stream, the memory design needs the number of blocks so that it can proceed data using a for-loop depends on the number. Thus, all the read operation of data should be transformed from reading from hls::stream to reading from memory.

5.4.2 Host Layer

The implementation of the host layer is rather straightforward. It just follows the workflow described in the Design chapter. First of all, the OpenCL Program is initialized, and the PL part is configured. Then, the BufferPointer instances for input, output, key and initial vector and the KernelPointer instance is created. The example for the CBC mode is shown in Figure 5.6.

```
CommandQueuePointer cqPointer;
KernelPointer kPointer;
Pool<BufferPointer, 4> bufferPool;
const size_t numULL=keyLengthToNumBits(K)/64;

//create the buffers, the command queue and the kernel
cqPointer.create(Application::getContext(), Application::getDevice(),
    CL_QUEUE_PROFILING_ENABLE);
kPointer.create(Application::getProgram<Lib::AES128_CBC>(),
    "aes128CbcEnc");
BufferPointer &inPointer=bufferPool.get<0>().create(Application::getContext(),
    CL_MEM_READ_WRITE, size*sizeof(uint64_t)*numULL, NULL);
//...

//map the buffer to PS part
uint64_t *inMapped = (uint64_t *)cqPointer->enqueueMapBuffer(*inPointer,
    CL_TRUE, CL_MAP_WRITE | CL_MAP_READ, 0, size*sizeof(uint64_t)*numULL);
//...
```

Figure 5.5: Example code for the kernel execution: first step

After that, the input, key and the initial vector is migrated to the PL part and the parameter list for the kernel is set like in Figure 5.7.

In the next step, the kernel is triggered to execute in Figure 5.8.

Finally, the result is migrated back and copied into the target output as in Figure 5.9.

5.4.3 Application Layer

In the application layer, the data set that is going to be processed is divided into multiple chunks by the user and the host API is executed multiple times to process chunk by chunk.

```
//kernel parameter setting
kPointer->setArg(0, *inPointer);
//...

//copy the data into the mapped buffer in PS part
for(size_t i=0;i<size*numULL;i++) inMapped[i]=in[i];
for(size_t i=0;i<numULL;i++) keyMapped[i]=key[i];
for(size_t i=0;i<numULL;i++) ivMapped[i]=iv[i];

//migrate the data to PL part
cqPointer->enqueueMigrateMemObjects({*inPointer, *keyPointer, *ivPointer},
    0);
```

Figure 5.6: Example code for the kernel execution: second step

```
//trigger the kernel to execution
cqPointer->enqueueTask(*kPointer);
```

Figure 5.7: Example code for the kernel execution: third step

```
//migrate the data back to PS part
cqPointer->enqueueMigrateMemObjects({*outPointer, *ivPointer},
    CL_MIGRATE_MEM_OBJECT_HOST);
cqPointer->finish();

//copy the result to the target place
for(size_t i=0;i<size*numULL;i++) out[i]=outMapped[i];
```

Figure 5.8: Example code for the kernel execution: final step

However, except the first chunk, the initial vector needs to be calculated for the other chunks. The workflow of each mode shows how to calculate it.

ECB mode

There is no initial vector in this mode.

CBC & CFB mode

The initial vector for the $n - th$ chunk is the last block of the cipher text in the $(n - 1) - th$ chunk, no matter it is encryption or decryption.

OFB mode

The initial vector for the $n - th$ chunk is the XOR of the plain text and the cipher text of the last block in the $(n - 1) - th$ chunk, mo matter it is encryption or decryption.

CTR mode

The initial vector for the $n - th$ chunk is the initial vector of the $(n - 1) - th$ chunk plus the number of block in the $(n - 1) - th$ chunk.

5.5 Data Compression Library API

5.5.1 Stream

Stream is a template class which aims to provide a queue data structure with thread safe property. It encapsulates the queue data structure from STL and adds synchronization techniques to simulate the producer-consumer model. The instance type stored in the inner STL queue depends on the template parameter. The core methods of this class are the push and pop functions and will be explained.

```
template<typename T>
void Stream<T>::push(T val) {
    std::unique_lock<std::mutex> lk(mut);
    cv.wait(lk, [this] { return queue.size()<maxSize; });
    queue.push(val);
    cv.notify_all();
}
```

Figure 5.9: Stream's push function

The push function has interaction with shared resource, namely the STL queue data structure. Thus, a lock needs to be added before the push function operates on it. The maximum size of Stream is fixed and the push function should never allows the number of stored instances exceed the maximum size. Because of this, the thread needs to wait until the size of the queue is smaller than the fixed maximum size. This is realized by a conditional variable provided by STL. After the push operation, the size of the queue must have one instance more than before the push operation. Thus, the pop function is allowed to be executed and I have to notify the threads that wait on this conditional variable.

```
template<typename T>
T Stream<T>::pop() {
    std::unique_lock<std::mutex> lk(mut);
    cv.wait(lk, [this] { return queue.size()>0; });
    T val=queue.front();
    queue.pop();
    cv.notify_all();
    return val;
}
```

Figure 5.10: Stream's pop function

The pop operation is similar to the push operation. The only difference is that before the pop operation, the thread has to wait until the queue is not empty, namely the size of the queue is strictly more then zero. After the pop operation, there must be a stored instance less than before the pop operation and a push function can be executed. In this case, it is still necessary to notify the threads that wait on this conditional variable.

5.5.2 ByteStream

The ByteStream inherits the template class Stream and the type of its stored instance is a C++ struct called ByteItem which stores the payload, a pointer to a byte array, namely a segment of memory, its size, the used size which records how much is already popped and a Boolean variable last which identifies whether the current ByteItem is the last one.

The push and pop with a specific size from a buffer into the ByteStream are common. However, the number of bytes to pop cannot always fit the byte segments stored in the ByteItems. In order to make the implementation of these two functionalities simple, three methods: available(), empty() and use() are added to the ByteItem. The available() method calculates the available number of bytes in the current ByteItem by substracting the used size from the total size. The empty() method reports whether the current ByteItem is empty by comparing the used size with the total size. The use() method try to increase the used size with a specific number.

With the methods implemented by the ByteItem, the two mentioned process can be implemented in an easier way. The push method firstly copy the whole buffer to push into a new buffer controlled by ByteItem. The reason of this step is that I cannot ensure the user provided buffer is still valid when I use it in the future. Then the ByteItem is pushed into the

```
struct ByteItem{
        uint32_t size;
        uint32_t used;
        uint8_t *payload;
        bool last;

        ByteItem(uint32_t size, uint8_t *payload, bool last);
        uint32_t available();
        void use(uint8_t *dest, uint32_t size);
        bool empty();
};
```

Figure 5.11: struct ByteItem

ByteStream.

```
void ByteStream::push(void *SRC, uint32_t size, bool last){
        uint8_t *src=(uint8_t*)SRC;
        uint8_t *payload=new uint8_t[size];
        memcpy(payload, src, size);
        std::unique_lock<std::mutex> lk(mut);
        cv.wait(lk, [this] { return queue.size()<maxSize; });
        queue.emplace(size, payload, last);
        cv.notify_all();
}
```

Figure 5.12: ByteStream's push method

The implementation of the pop method is a little bit complex. Firstly, I would like to regular to process in such a way that only the ByteItem which is not empty can still stay in the ByteStream. For the given size to pop, I will keep popping the ByteItem from the queue. If the available size of the current ByteItem is less than the number of bytes the user wants to pop, then write the data from the whole payload to the target buffer. In the other case where the available size of the current ByteItem is more than the number of bytes the user wants to pop, I have to only write the front of the available payload to the target buffer and update the used size in the ByteItem. The update of the number of bytes is necessary after processing a ByteItem by subtracting the number of byte that is written to the target buffer.

```
uint32_t ByteStream::pop(void *DEST, uint32_t size, bool &last){
        uint8_t *dest=(uint8_t*)DEST;
        std::unique_lock<std::mutex> lk(mut);
        uint32_t popped=0;
        while(size){
                cv.wait(lk, [this] { return queue.size()>0; });
                ByteItem &item=queue.front();
                uint32_t canUse=std::min(size, item.available());
                item.use(dest, canUse);
                dest+=canUse;
                popped+=canUse;
                size-=canUse;
                if(item.empty()){
                        delete[] item.payload;
                        if(item.last){
                                last=true;
                                queue.pop();
                                return popped;
                        }
                        queue.pop();
                        cv.notify_all();
                }
        }

        last=false;
        return popped;
}
```

Figure 5.13: ByteStream's push method

5.5.3 Data Analyzer

Compression of Gzip & Zstd

Since the compression hardware kernels of these two compression algorithms accept a segment of data and compress it to a block for Gzip and multiple frames for Zstd. Thus, the Data Analyzer of these two compression algorithms only need to read a fixed size of data from the input stream.

Compression of Lz4 & Snappy

The compression hardware kernels of these two compression algorithms share almost the same API. Thus, the process of these two Data Analyzers are similar. They read a fixed size of data from the input stream and then divide it into several blocks and summarize the number of blocks and the block size of each block.

Decompression of Gzip

The Gzip decompression kernel accepts the whole compressed data set as input, thus the Data Analyzer of Gzip decompression needs only to repeatedly read a fixed size of bytes from stream and wait the Kernel Executor to process it.

Decompression of Lz4 & Snappy

The Lz4 and Snappy decompression kernel accepts the compressed data blocks as input. Thus, the Data Analyzer of Lz4 and Snappy have to firstly read a fixed number of blocks from the input stream, then process it according to the file structure of Lz4 and Snappy explained in the Overview chapter, in order to get rid of the unnecessary data such as the size and the checksum. Then, the Data Analyzers should also summarize the size of the compressed blocks and write it to the target buffer. Hence, the decompression uses the copy for an additional time comparing to the compression of Lz4 and Snappy.

Decompression of Zstd

The Zstd decompression kernel can decompress a Zstd frame at one time. This is the result of testing which initially pushes multiple frames and then see whether the decompression works. However, it only works when only one frame is inputed. Thus, the Data Analyzer of the Zstd decompression algorithm should read from the input stream multiple times in order to get the whole Zstd frame. Then, the Data Analyzer writes it to the target buffer and waits the Kernel Executor to process it.

5.5.4 Kernel Executor

The Kernel Executor triggers the hardware kernels to execute. The data which the hardware kernels need should already be prepared by the Data Analyzer. There are two types of the kernel design, namely one that directly access the global memory on the PL part and one that using the AXI stream as input and output.

The hardware kernels that directly access the global memory are executed in the similar way as the hardware kernel execution for the security library: the buffer in the PL part is created firstly and then mapped to the PS part. The PS part copies the data into the buffer and then migrates it to the PL part. The hardware kernels in the PL part use the buffer to execution. After the execution of the hardware kernel, the PS part migrates the result back and pushes it into the output stream.

However, the hardware kernels that use the AXI stream is not going to be executed, because it only reads the data from the AXI stream and the result is also written on it. Thus, I have to execute the MM2S kernel and the S2MM kernel at the same time, since the processing kernel reads and writes the data at the same time. The execution model is shown in Figure 5.14.

```
std::thread chunkWriter([this]{
    // the execution of the MM2S kernel
    // normally, it will be executed multiple times
    // in order to input the whole data which the hardware kernel needs
});

std::thread chunkReader([this]{
    // the execution of the S2MM kernel
    // normally, it will be executed multiple times
    // in order to output the whole data which the hardware kernel generates
});

chunkWriter.join();
chunkReader.join();
```

Figure 5.14: Execution model for hardware kernels using AXI stream

The execution of the MM2S and the S2MM kernel is running in two separate threads in order to prevent unexpected thread blocking.

5.5.5 Application layer

In order to prevent the blocking of thread for user input and output, the workshop is implemented to run in a separate thread. If the user set the "first" parameter in the push method as true, then the thread is created and starts to run. If the user set the "last" parameter in the pop method as true, then the thread starts to wait the workshop for finish processing, since the last chunk is going to be popped.

6 Evaluation

6.1 General Information

In this section, I will display the environment of the performance test. The test program is running on a machine called Hinoki with the FPGA board Xilinx Alevo U50 attached on it. Hinoki has a x86_64 Architecture and the CPU is Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz. The program is compiled using -O3 optimization option and with g++. The execution time is measured using the steady_clock of the "chrono" library from STL. The data sets for testing are in different sizes from 1 GiB to 64 GiB.

6.1.1 Performance Analysis of Security Library

For the performance measurement of the security, there are many modes for DES and AES. Additionally, the length of the key varies from 128 bits to 256 bits. In order to reduce the combinations to test, we focus on the AES algorithm with the width of the key is 256 bits, since it is commonly used. For the test data sets, we choose the 4 GiB and 32 GiB data sets. For each test, we repeat it for five times and use the average values in order to reduce the randomness from the execution.

The benchmark is the Openssl library [30]. This library provides a CPU implementation of the same functions we want to test.

For each test, we test the compute time of the implementation, the time for host I/O, which is the time for reading and writing the data set. For FPGA implementation, there is additionally another time that needs to be measure, which is the FPGA I/O time. Because the computation is not executed in the host part, some time is needed to transfer the data to and back from the FPGA device.

AES Cbc mode

From the test result, we can see that the compute time of encryption running in FPGA in fast two times of the compute time running in CPU. The reason of this may be that encryption of Cbc mode is dependent between blocks. In other words, one block can be processed only after the last board is processed. This property wastes the advantage of parallel execution of the FPGA. Also the frequency of the FPGA board is much lower than CPU. These facts results in the longer compute time. However, the decryption of FPGA implementation takes a much shorter time than the CPU implementation because the dependence between blocks disappears.

The FPGA executions have additionally a FPGA I/O time. This time includes the time for the data to transfer from or to the FPGA board. This also costs huge amounts of time.

The host I/O time is almost the same for both FPGA implementation and the CPU implementation.

The total time is the sum of the three types of time. Due to the additional FPGA I/O time and a longer compute time, the encryption of the FPGA implementation is slower than the CPU implementation. Considering the data set with bigger file, this performance gap has no trend to decrease.

However, for the decryption, although the FPGA implementation for the 4 GiB data set is a little bit slower than the CPU implementation, the execution time of the FPGA implementation for the 32 GiB data set is shorter. The reason of this change may be that with the scaling of the data set, the performance gap of the decryption is also scaled. When the size of data set is big enough, the compute time difference is also big enough to eliminate the disadvantage of the FPGA I/O time.

AES Cfb mode

With the same property with the Cbc mode, there also exists the dependence between for encryption, but the decryption is independent between blocks. Thus, the encryption of the FPGA implementation is slower than the CPU implementation. However, the difference of the compute time in decryption is very huge even for smaller data sets. And the difference becomes bigger for bigger data sets, which results in a fast half of the execution time.

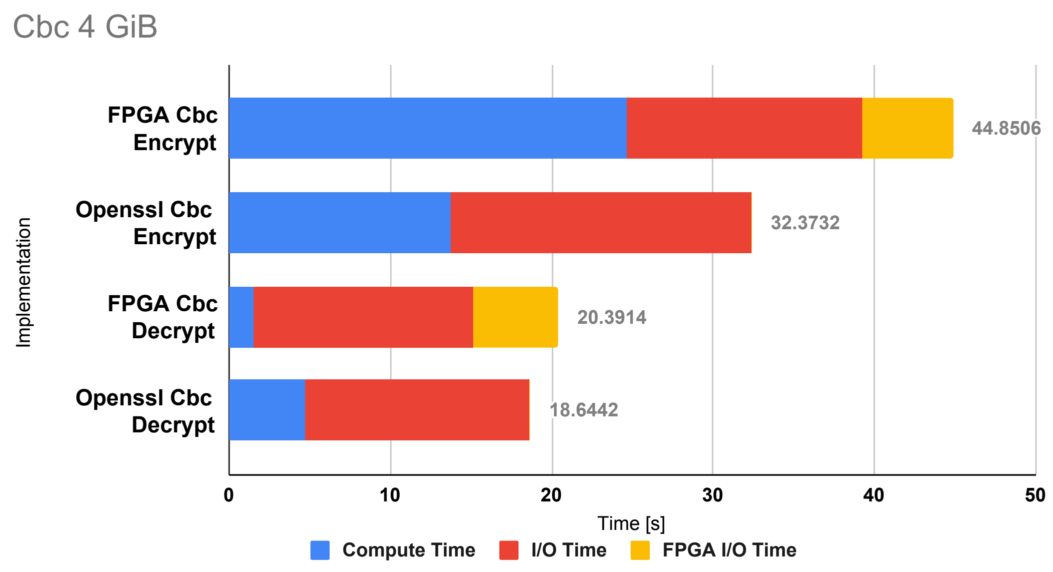


Figure 6.1: Performance of AES Cbc mode for 4 GiB files

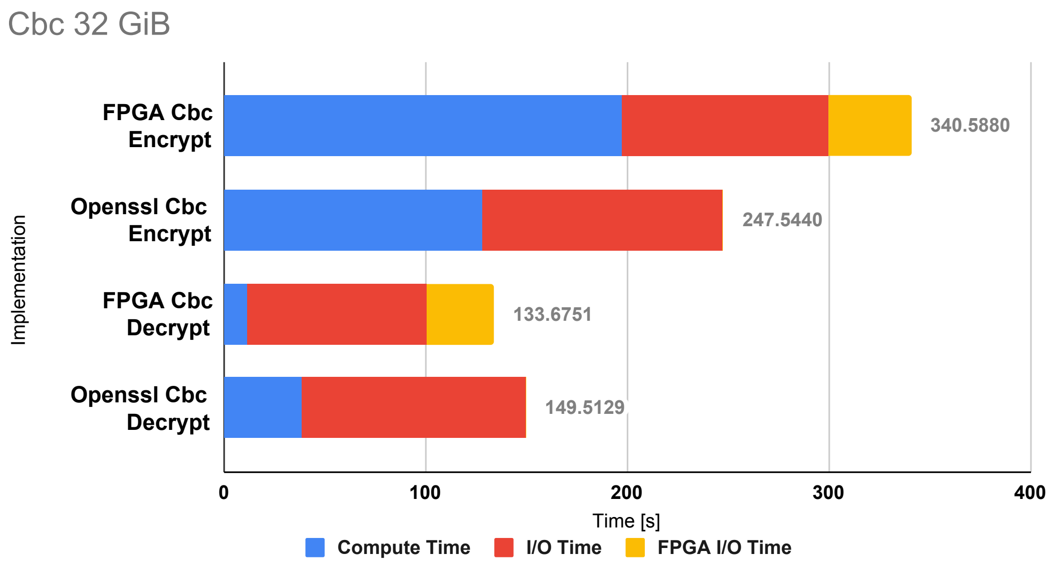


Figure 6.2: Performance of AES Cbc mode for 32 GiB files

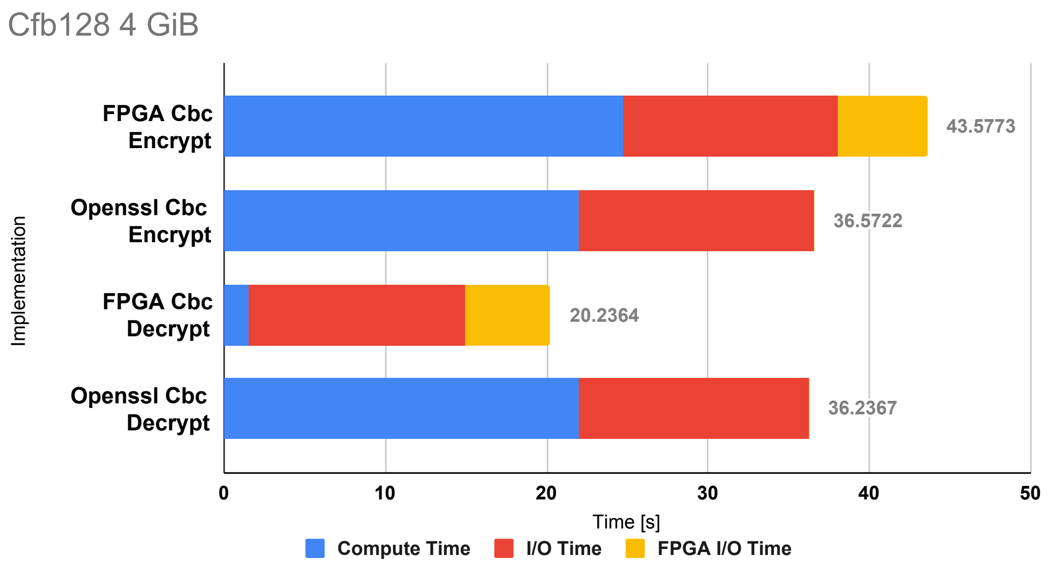


Figure 6.3: Performance of AES Cfb mode for 4 GiB files

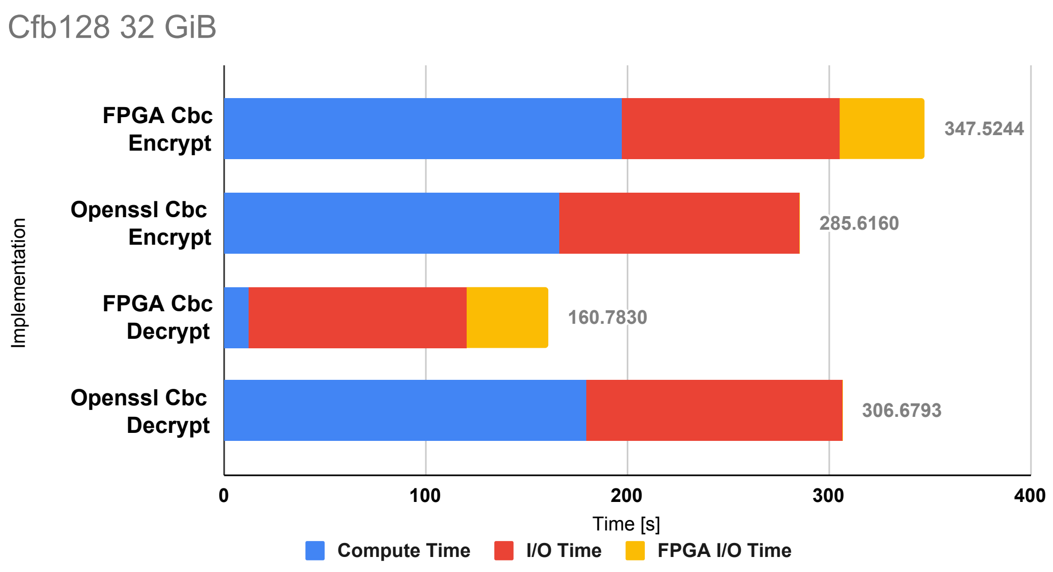


Figure 6.4: Performance of AES Cfb mode for 32 GiB files

AES Ecb mode and CTR mode

These two modes have the same property, which is that both the encryption and the decryption doesn't have the the dependence between blocks. From the test result, we can see that the compute time of the FPGA implementation is much shorter than the CPU implementation. However, the FPGA implementation is always a little bit slower, because the disadvantage of the additional FPGA I/O.

6.1.2 Performance Analysis of Data Compression library

In this subsection, the performance of the four compression algorithms are measured.

Gzip

For Gzip compression algorithm, the benchmark is the CPU Gzip utility from Linux [31]. From Figure 6.9, we can find that the FPGA implementation is more than 20x faster than the CPU implementation for compression and 8x faster for decompression and the bigger the data set is, the more performance gain we can get. This may results from that the Gzip compression algorithm is very suitable for the hardware acceleration due to its parallel property.

Lz4

The benchmark of the Lz4 compression algorithm is the Lz4 utility of Linux [32]. We can conclude from Figure 6.10 that for small data sets, the FPGA implementation is almost the

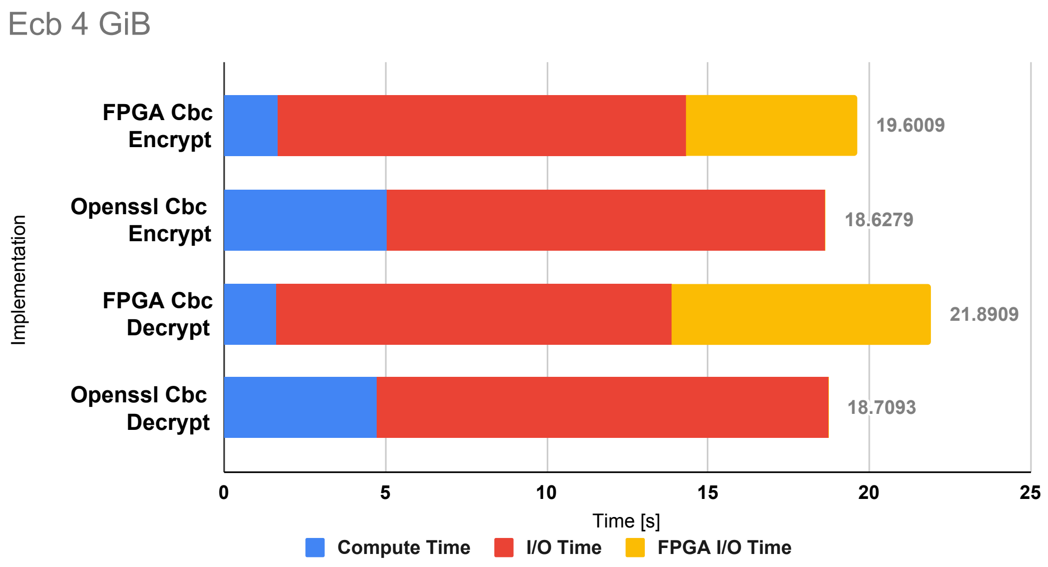


Figure 6.5: Performance of AES Ecb mode for 4 GiB files

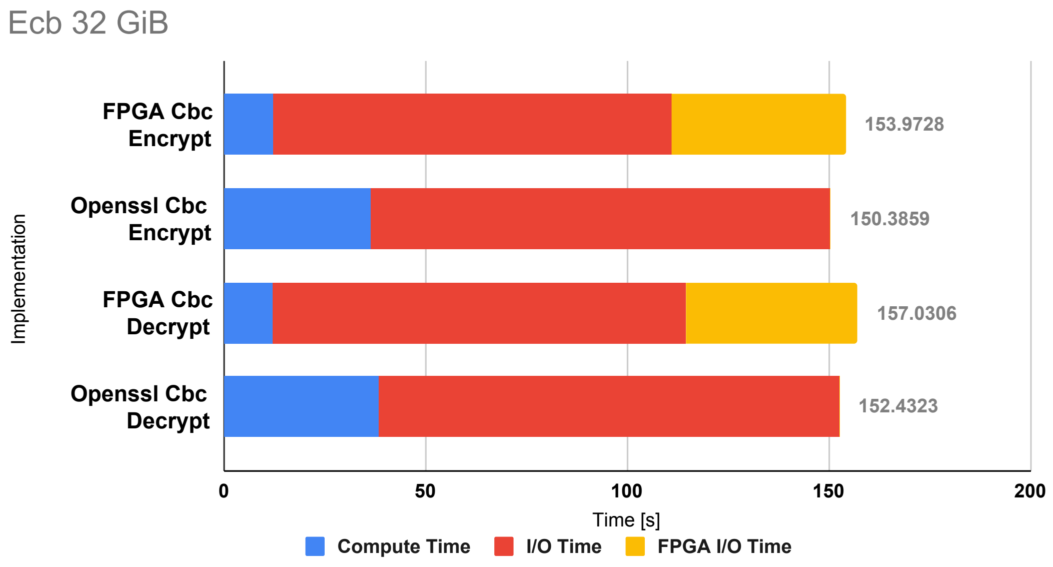


Figure 6.6: Performance of AES Ecb mode for 32 GiB files

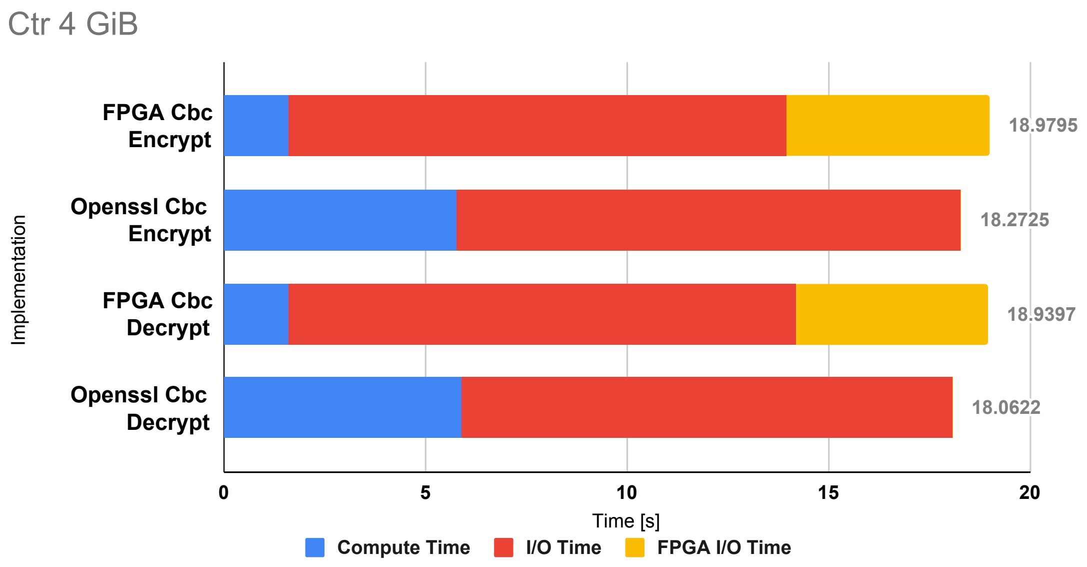


Figure 6.7: Performance of AES Ctr mode for 4 GiB files

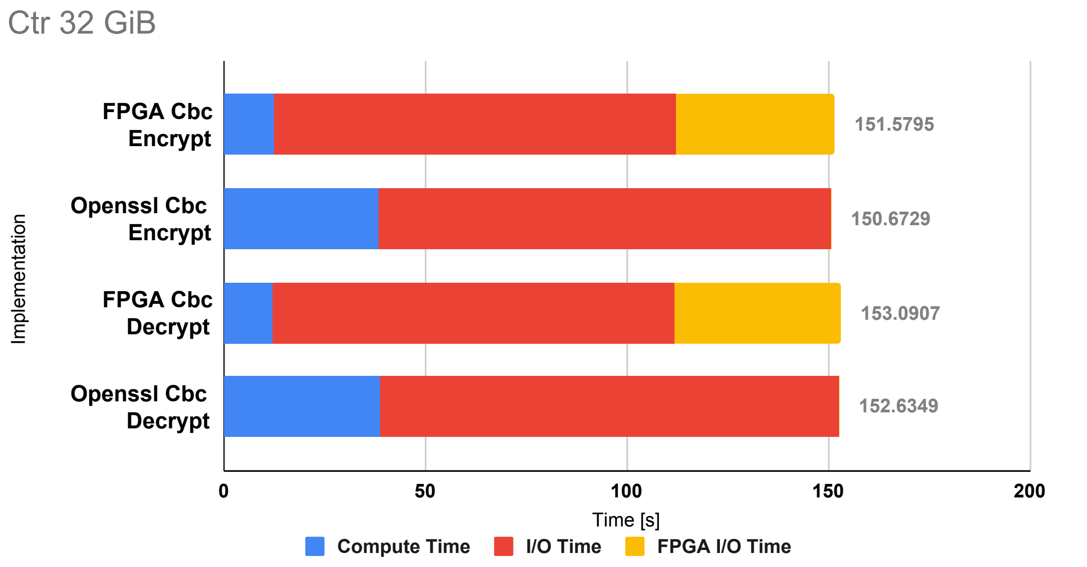


Figure 6.8: Performance of AES Ctr mode for 32 GiB files

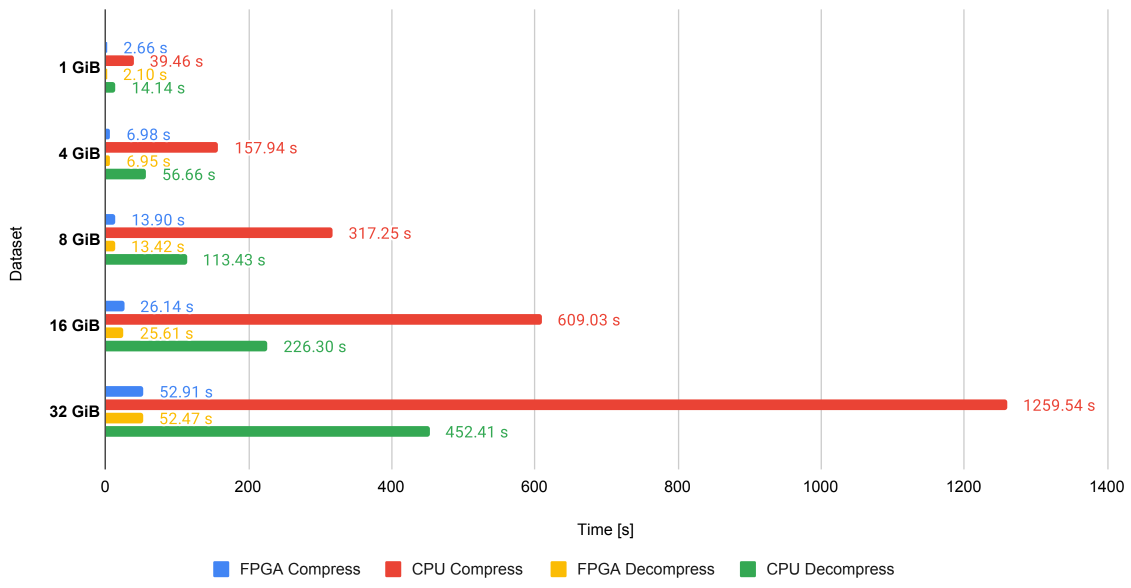


Figure 6.9: Performance of Gzip compression algorithm

same fast than the CPU implementation. However, with the increasing of the size of data sets, the FPGA implementation becomes faster and the difference is also bigger and bigger. The reason of this maybe the FPGA initiation and additional I/O for the FPGA implementation. But for the decompression, the FPGA implementation is only a little bit faster than the compression, unlike that the compression of the CPU compression is much faster than its decompression.

In order to find the reason for that, this report tries to measure the time separately for the host and FPGA execution. Since we have multiple data sets with different size, here the average time per GiB is measured. From Figure 6.11, we can find that for the host time for compression is longer than for the decompression although for the implementation of the decompression, we use an additional copy. This is because the compressed file is with a much smaller size. However, the FPGA time for compression is much shorter than the decompression. This is the opposite situation of the CPU implementation: the decompression is faster than the compression. This issue should be solved in the hardware level by Xilinx and is out of the scope of this thesis report.

Snappy

The structure of Snappy compression algorithm is similar to Lz4. So the performance is also in the similar way.

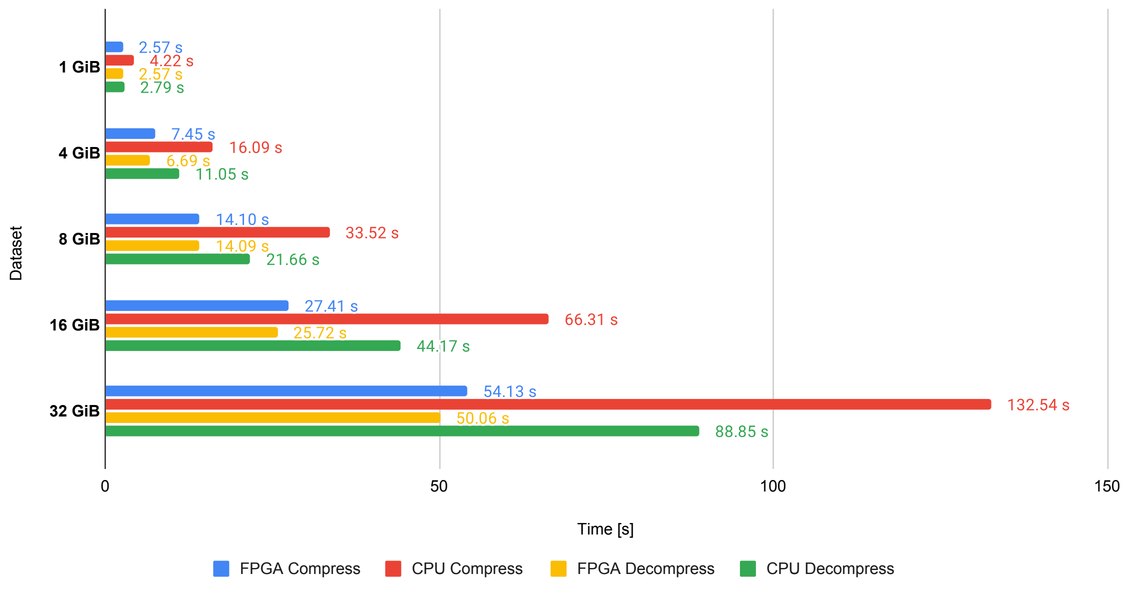


Figure 6.10: Performance of Lz4 compression algorithm

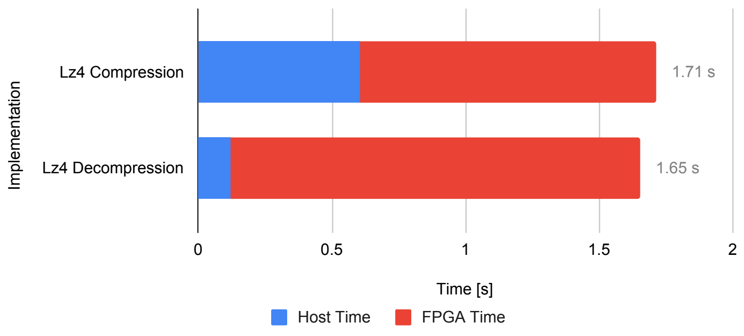


Figure 6.11: Average Time of Lz4 per GiB

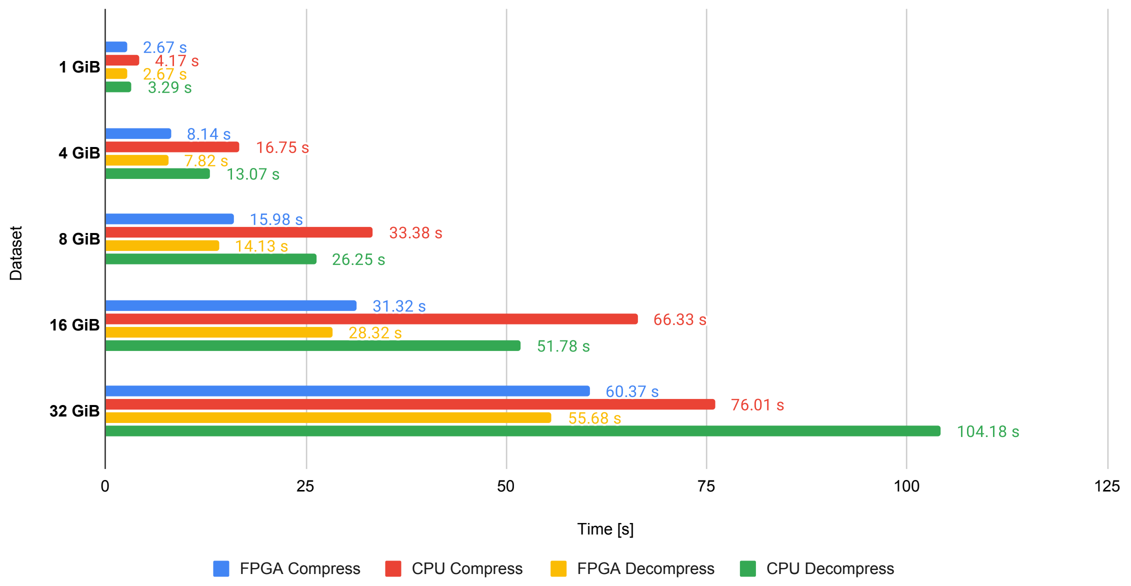


Figure 6.12: Performance of Snappy compression algorithm

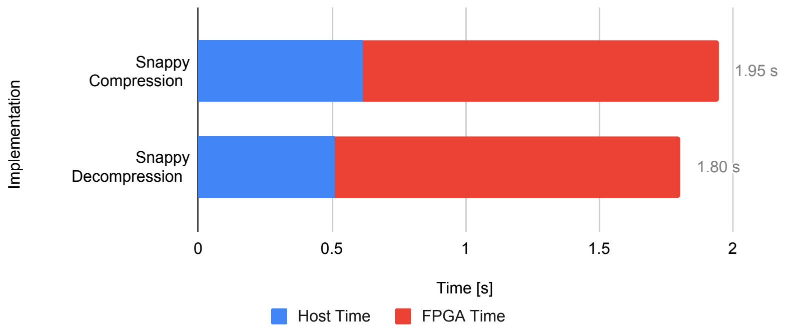


Figure 6.13: Average Time of Snappy per GiB

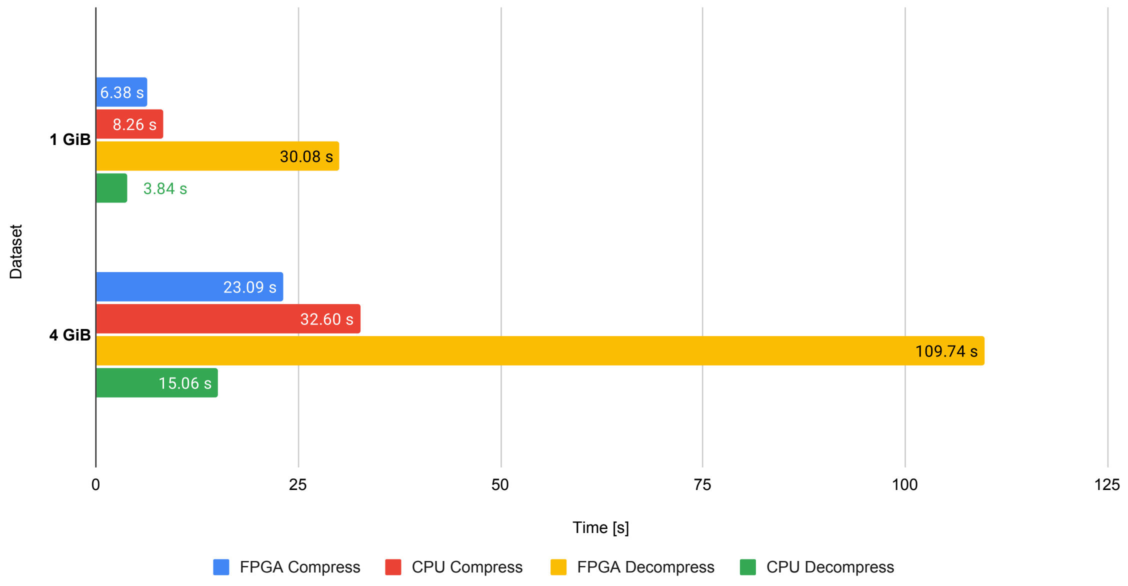


Figure 6.14: Performance of Zstd compression algorithm

Zstd

From Figure 6.12, we can find that the Zstd compression of the FPGA implementation is about 1.3x faster than the CPU implementation.

However, the Zstd decompression is relative slow. That's because the compression of Xilinx hardware kernel will divide the input data into blocks and packages them so that each block is in its own Zstd frame. But the size of the blocks is relative small, which means that the size of a Zstd frame is also small and this will leads to the fact that the compressed data set is too fragmented. The Xilinx decompression kernel accepts only a Zstd frame at a time and the decompression time for a single frame is small, but with too many fragments, the decompression time is thus much longer. From the Figure 6.15, we may find that the most time is spent on the FPGA computation. The improvement needs to be done in hardware.

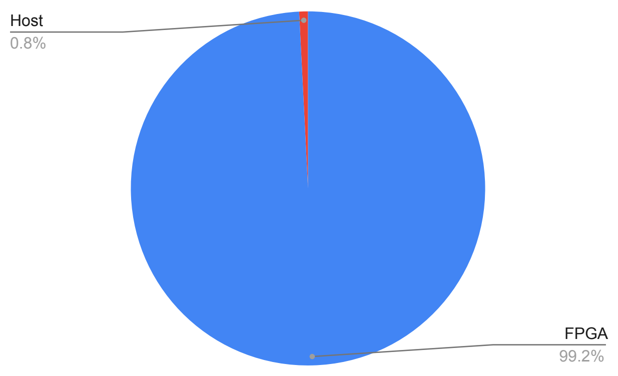


Figure 6.15: Zstd Decompression Time

7 Related Work

In this chapter, the report introduce some projects that related to computational storage devices.

7.0.1 Biscuit: A Framework for Near-Data Processing of Big Data Workloads

Biscuit [33] is a near-data processing framework designed for modern SSDs. It enables programmers to create data-intensive applications that can operate distributed but smoothly on the host machine and storage system. Targeting to provide a high-level programming model, Biscuit expands on the idea of dataflow, where typed and data ordered ports are used for communication between data processing activities. However, Tasks that are executed on the host system and the storage system are not distinguished by Biscuit. As a result, Biscuit exhibits desirable qualities like generality and expressiveness, promotes code reuse, and naturally exposes concurrency. And it brings a over 15x performance gain for the top 5 TPC-H queries.

7.0.2 YourSQL: A High Performance Database System Leveraging In-Storage Computing

YourSQL [34] is a database system that uses the the additional computing capabilities from computational storage devices for accelerating the data-intensive queries. By loading the data scanning of a query to user-programmable solid-state disks, YourSQL achieves extremely early data filtering. YourSQL is researched based on a recent branch of MariaDB, which is a variant of MySQL. When compared to a vanilla system, YourSQL speeds up the execution of all TPC-H queries by 3.6 seconds. Additionally, the six TPC-H queries with the highest performance increases had an average speed-up of over 15, which is impressive. Significant energy savings can be detected as a result of this significant reduction in execution time.

7.0.3 Willow: A User-Programmable SSD

Willow [35] enables developers to expand and enhance the semantics of an SSD with capabilities tailored to particular applications without compromising file system security. The SSD Apps running on Willow reduces the stress that IO processing exerts on the host CPU while providing apps with low-latency, high-bandwidth access to the SSD's contents. A lot of freedom is offered by the development architecture for SSD Apps, which also allows for the parallel operation of several SSD Apps in Willow and the execution of trusted code. Six SSD Apps are implemented and measured their performance. As a result, defining

SSD semantics in software is simple and advantageous, and Willow enables a variety of IO-intensive applications to take use of a tailored SSD interface.

8 Summary and Conclusion

Overall, in this project, we build complete and easy-to-use interfaces for the security and data compression library based on Xilinx Vitis Library. These interfaces contains the hardware accelerations for data-intensive functions, but the user can learn how to use it without having any hardware knowledge. From the evaluation, we can have the conclusion that the data-intensive functions with hardware accelerations have in most cases better performance than the functions that are implemented using the CPU and its optimization techniques. Thus, our original goal to unload the data-intensive functions to the computational storage devices is generally a feasible idea. However, due to the time limits, only two libraries are ported and there are some other data-intensive functions that still need to be migrated. Additionally, some hardware accelerations such as overlapped execution is not implemented in this project. Last but not least, the target board is currently not available to evaluate the ported library since Xilinx does not provide an official support and we tried the configuration from the community but failed, it is also necessary to evaluate the library on the target board, when Xilinx provides the support in the future.

9 Future Work

9.1 Complete the other libraries

Due to the time limits, only two libraries of the Xilinx Vitis Library is successfully ported and tested. However, there are still some libraries that are related to the data-intensive functions, but we are not able to port it. For example:

- Vitis Database Library
- Vitis Data Analytic library
- Vitis Graph library
- Vitis HPC library
- ...

9.2 Hardware Acceleration Methods

In this project, we only use the simple hardware kernel execution model. However, this model can be better by using the overlapped execution shown in Figure 9.1 and 9.2 [36].

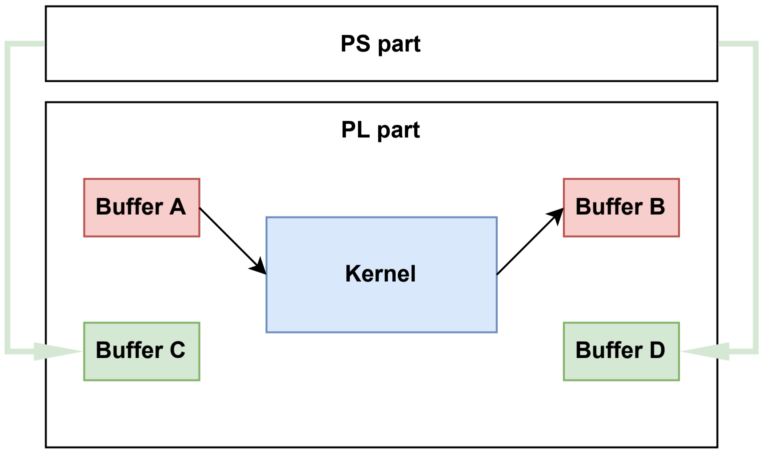


Figure 9.1: Illustration of kernel's behaviour at time n

The currently used RAM usually has two ports and the read and write operation of these two ports are independent. Thus, the idea of the overlapped execution can be realized. At

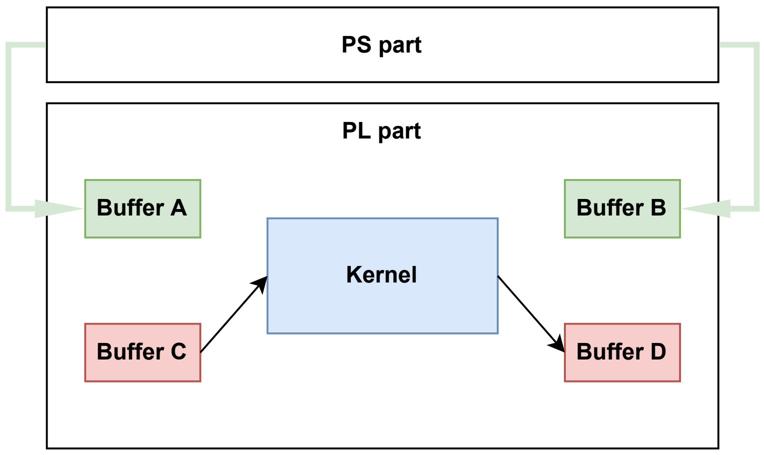


Figure 9.2: Illustration of kernel's behaviour at time n+1

some time n, the hardware kernel is using the Buffer A for input and Buffer B for output. Thus, the buffer C and Buffer D are not used and are able to be read and written from the PS part. At time n+1, the hardware kernel starts to read from Buffer C and Buffer D and the Buffer A and Buffer B is free again. The result of the last kernel execution is currently in the Buffer B, which should be migrated during the current execution of the kernel and before the kernel is executed for the next time. Thus, the hardware kernel execution model is overlapped and get a higher throughput.

9.3 Evaluation board

Due to the fact that Xilinx do not provide an official configuration support for the Xilinx Zynq UltraScale+ MPSoC ZCU106 board and we tried the configuration from the community but failed, the actual execution on the board is yet not evaluated. This needs to be finished when Xilinx provides the official configuration.

List of Figures

List of Tables

Bibliography

[1] Jiong Liu. *bsc-project*. https://github.com/Ljiong201108/bsc-project (Last accessed on 2023-01-28).

[2] Jaeyoung Do, Antonio Barbalace. "Computational Storage: Where Are We Today?" In: ().

[3] Xilinx. *Xilinx - Adaptable. Intelligent | together we advance*. https://www.xilinx.com/ (Last accessed on 2023-01-28).

[4] Xilinx. *Vitis Software Platform*. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html (Last accessed on 2023-01-28).

[5] Xilinx. *Vitis Getting Started Tutorial*. https://github.com/Xilinx/Vitis-Tutorials/tree/2022.2/Getting_Started/Vitis (Last accessed on 2023-01-28).

[6] Xilinx. *Xilinx Runtime Library (XRT)*. https://www.xilinx.com/products/design-tools/vitis/xrt.html (Last accessed on 2023-01-28).

[7] OpenCL. *OpenCL Overview - The Khronos Group Inc*. https://www.khronos.org/opencl/ (Last accessed on 2023-01-28).

[8] Xilinx. *Xilinx OpenCL extension - XRT Master documentation*. https://xilinx.github.io/XRT/master/html/opencl_extension.html (Last accessed on 2023-01-28).

[9] Xilinx. *vitis-documentation-2020.2*. https://docs.xilinx.com/v/u/2020.2-English/ug1416-vitis-documentation (Last accessed on 2023-01-28).

[10] Xilinx. *Vitis Getting Started Tutorial*. https://github.com/Xilinx/Vitis-Tutorials/tree/2022.2/Getting_Started/Vitis (Last accessed on 2023-01-28).

[11] Xilinx. *Vitis Getting Started Tutorial*. https://github.com/Xilinx/Vitis-Tutorials/tree/2022.2/Getting_Started/Vitis (Last accessed on 2023-01-28).

[12] Xilinx. *Vitis Getting Started Tutorial*. https://github.com/Xilinx/Vitis-Tutorials/tree/2022.2/Getting_Started/Vitis (Last accessed on 2023-01-28).

[13] Xilinx. *Alveo U50 Data Center Accelerator Card*. https://www.xilinx.com/products/boards-and-kits/alveo/u50.html (Last accessed on 2023-01-28).

[14] Xilinx. *Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit*. https://www.xilinx.com/products/boards-and-kits/zcu106.html (Last accessed on 2023-01-28).

[15] Xilinx. *Vitis Getting Started Tutorial*. https://github.com/Xilinx/Vitis-Tutorials/tree/2022.2/Getting_Started/Vitis (Last accessed on 2023-01-28).

[16] Xilinx. *Vitis Library*. https://www.xilinx.com/products/design-tools/vitis/vitis-libraries.html (Last accessed on 2023-01-28).

[17] Xilinx. *Introduction to Vitis HLS*. `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS` (Last accessed on 2023-01-28).

[18] Wikipedia. *Electronic Code Book Mode – Wikipedia*. `https://de.wikipedia.org/wiki/Electronic_Code_Book_Mode` (Last accessed on 2023-01-28).

[19] Wikipedia. *Cipher Block Chaining Mode – Wikipedia*. `https://de.wikipedia.org/wiki/Cipher_Block_Chaining_Mode` (Last accessed on 2023-01-28).

[20] Wikipedia. *Counter Mode – Wikipedia*. `https://de.wikipedia.org/wiki/Counter_Mode` (Last accessed on 2023-01-28).

[21] Wikipedia. *Cipher Feedback Mode – Wikipedia*. `https://de.wikipedia.org/wiki/Cipher_Feedback_Mode` (Last accessed on 2023-01-28).

[22] Wikipedia. *Output Feedback Mode – Wikipedia*. `https://de.wikipedia.org/wiki/Output_Feedback_Mode` (Last accessed on 2023-01-28).

[23] Xilinx. *Vitis Security Library*. `https://xilinx.github.io/Vitis_Libraries/security/2022.1/index.html` (Last accessed on 2023-01-28).

[24] Wikipedia. *Data Encryption Standard – Wikipedia*. `https://de.wikipedia.org/wiki/Data_Encryption_Standard` (Last accessed on 2023-01-28).

[25] Wikipedia. *Advanced Encryption Standard – Wikipedia*. `https://de.wikipedia.org/wiki/Advanced_Encryption_Standard` (Last accessed on 2023-01-28).

[26] G. Project. *Gzip - GNU Project - Free Software Foundation*. `https://www.gnu.org/software/gzip/` (Last accessed on 2023-01-28).

[27] *LZ4 - Extremely fast compression*. `http://lz4.github.io/lz4/` (Last accessed on 2023-01-28).

[28] Google. *google/snappy: A fast compressor/decompressor*. `https://github.com/google/snappy` (Last accessed on 2023-01-28).

[29] Facebook. *Zstandard - Real-time data compression algorithm*. `http://facebook.github.io/zstd/` (Last accessed on 2023-01-28).

[30] Openssl. *openssl*. `https://www.openssl.org/` (Last accessed on 2023-01-28).

[31] L. man page. *gzip*. `https://linux.die.net/man/1/gzip` (Last accessed on 2023-01-28).

[32] *LZ4 - Extremely fast compression*. `http://lz4.github.io/lz4/` (Last accessed on 2023-01-28).

[33] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, Duck-hyun Chang. "Biscuit: A Framework for Near-Data Processing of Big DataWorkloads". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture* 1063-6897.16 (2016), pp. 153–165. DOI: `10.1109/ISCA.2016.23`.

[34] Insoon Jo, DuckHo Bae, Andre S. Yoon, JeongUk Kang, Sangyeun Cho, Daniel DG Lee, Jaeheon Jeong. "YourSQL: A HighPerformance Database System Leveraging InStorage Computing". In: *Proceedings of the VLDB Endowment* 9.12 (2016), pp. 924–935.

[35] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, Steven Swanson. "Willow: A User-Programmable SSD". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)* (2014), pp. 67–80. DOI: `978-1-931971-16-4`.

[36] Xilinx. *Overlapping Data Transfers with Kernel Computation*. `https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Overlapping-Data-Transfers-with-Kernel-Computation` (Last accessed on 2023-01-28).