Master's Thesis

Evaluating Performance and Productivity of OpenCL on HiFP2.0 Algorithm

1810445 Minh Tien Nguyen

Supervisor: Professor Yasushi Inoguchi

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

March 2021

Abstract

OpenCL (Open Computing Language) is a high-level synthesis & cross-platform standard for building high-performance computing programs and be able to distribute to various processors and hardware accelerators, which contains central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs). OpenCL was born to help software developers accelerating their algorithms on hardware with eases. Since the first release in 2008, many large firms have supported and developed OpenCL (such as Intel, AMD, Nvidia, Xilinx). Now, it is being maintained by Khronos Group.

Audio fingerprinting is an efficient technique to represent the characteristics of an audio file. It is lightweight in representation, then requires a small memory capacity and computational cost for management.

HiFP2.0 is an audio fingerprinting algorithm. By just requiring a small computational cost, it can acquire high throughput and then take advantage of a high-speed network, which is 40Gbps Ethernet or 100Gbps Ethernet.

In the previous works, Araki succeeded in developing HiFP2.0 executed on CPU and had good results in throughput and accuracy [1]. However, it is not sure about the ability when applying the algorithm on other platforms.

In this study, we implement HiFP2.0 in OpenCL executing on Intel FPGA Arria 10, which contains two types of programming models: task-based and data-based parallelism. We tend to provide an insight into how well the OpenCL works on HiFP2.0 algorithm. We investigate the performance and productivity aspects.

To summarize our results, some main points can be listed by the following:

1. The number of code lines can be reduced from 5 - 5.54 times when using OpenCL (a high-level synthesis language) instead of VHDL (a register-transfer level programming language) on an FPGA.

2. The optimized algorithm of OpenCL can achieve more 17.4% performance than the origin, which is intensionally designed for CPUs.

3. The data-based method has the best throughput when applying 512 work-items per work-group and 50 work-groups per execution.

Keywords: OpenCL, HiFP2.0, High-level synthesis, FPGA, Arria 10.

Contents

This dissertation was prepared according to the curriculum for the collaborative education program organized by Japan Advanced Institute of Science

and Technology and University of Engineering and Technology, Vietnam National University.

List of Figures

List of Tables

List of Abbreviations

FPID	Fingerprint Identification
HLS	High-Level Synthesis
RTL	Register-Transfer Level

Chapter 1

Introduction

1.1 Motivation and Problem Statement

1.1.1 The tough of FPGA programming

In the Information Technology (IT) industry, software and hardware are essential parts that must be developed to deliver better services to clients. Through the flow of the IT development, components of IT are constructed level by level, and the new ones are depended and based on the previous blocks. Since 1946 - when the first computer ENIAC was invented at the University of Pennsylvania, Human has developed and improved computer more and more useful.

Nowadays, a computer becomes more and more complex. One person cannot understand all the parts of a computer system. Software engineers can develop and apply algorithms into programs. The hardware designer can develop high-performance architectures for circuits. Because of the differences between the two platforms they work at, the one cannot do the job of the other one.

FPGAs were born in the 1980s, but they are not popularly used until now. In the past, FPGAs stayed in research, development, and prototyping chips. FPGAs have explored their potential ability in High-performance computing. FPGAs can beat GP-GPU in some jobs, and it can be seen in [14].

However, like a knife, FPGAs also have a bad side. In the past, by just supporting the Register-transfer level (RTL) on FPGAs, the productivity of development programs on FPGAs is very low, it is pointed out in [10].

High-level synthesis was born to reduce the tough of FPGA programming. By taking advantage of High-level languages like C/C++, software engineers can quickly implement their algorithms to run on dedicated devices, which had been done previously by hardware designers.

OpenCL (Open Computing Language) is a high-level synthesis & cross-platform standard for building high-performance computing programs and be able to distribute to various processors and hardware accelerators, which contains central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs). Since the first release in 2008, OpenCL has been maintained and operated by Khronos Group with support from many large firms, such as Apple, Intel, AMD, Nvidia, Samsung, Xilinx, Qualcomm. An overview of the OpenCL community is shown in figure 1.1 [1]. By 2020, the latest version of OpenCL is 3.0.

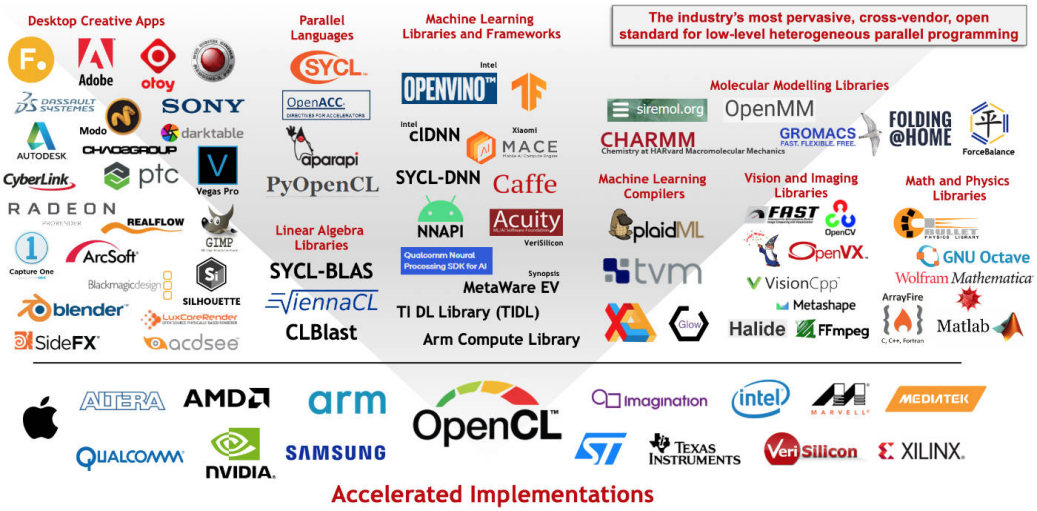

Figure 1.1: Supports from organizations using OpenCL, reprinted from Khronos Group

1.1.2 The raise of audio content in social networks and streaming platforms

In modern life, people listen to music every day. At the end of 2019, Spotify has more than double users, compared to 2016, by their report shown in Figure 1.2.

Music has been one of the most popular types of information globally, and there are hundreds of music streaming platforms and related services available on the Internet now. The music data is accumulated day by day and becomes big data in several companies. Some of the music collections

[1] https://www.khronos.org/opencl

available on the server have reached a scale of ten million tracks. By 2020, there exist about 60 million songs in Apple Music's database [2] and 50 million songs in Spotify's database [3].

The vast audio database has opened many issues in retrieving, searching, and organizing the music content.

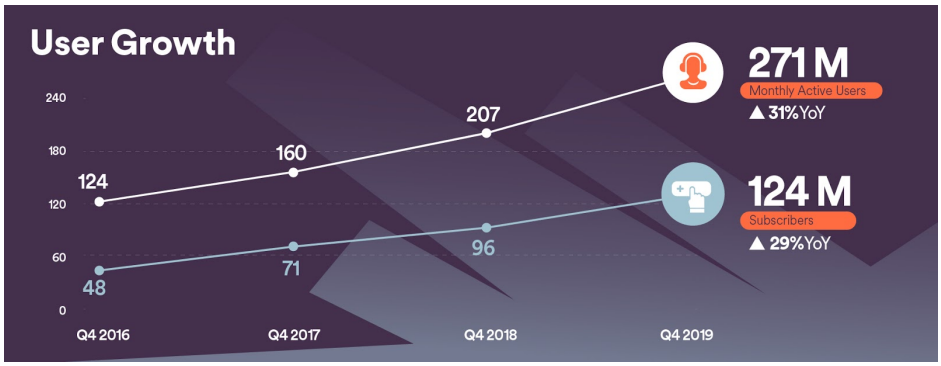

Figure 1.2: Spotify reports fourth quarter and full-year 2019 earnings, reprinted from Spotify

To cope with the searching audio problem, Jaap Haistma proposed a Content-based Audio Identification framework with two fundamental processes: Fingerprint extraction & Database Searching. An overview of Haistma's proposed framework is shown in Figure 1.3.

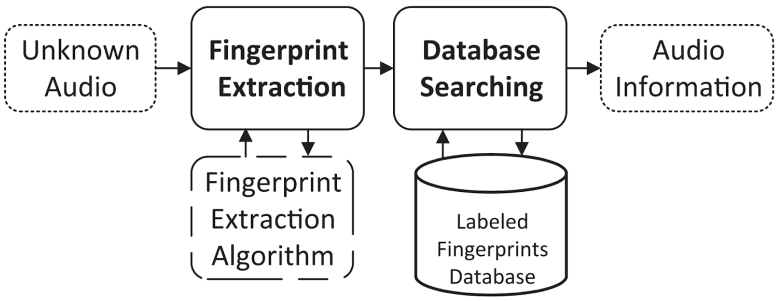

Figure 1.3: Content-based Audio Identification Framework, proposed by Haitsma [2]

[2] https://www.apple.com/apple-music

[3] https://newsroom.spotify.com/2020-02-05/spotify-reports-fourth-quarter-and-full-year-2019-earnings

A good audio fingerprint system should acquire the following requirements:

- **Robust**: The origin audio file can be transformed into many formats to serve many purposes. For example, the raw format is for high-end devices, presenting the best quality of sound. However, the archived versions (have distortion) like mp3 are suitable for lower-quality devices, which requires more transfer speed and less in quality. The two audio files' waveforms are different but have the same content that should be identified as 1.
- **Reliable**: The system should discriminate among many different audio contents with high accuracy.
- **Portable**: The size of an audio fingerprint should be small. The number of audio files in a real database is enormous (maybe more than 60 million), and the searching procedure is often conducted on RAM (random-access memory). The size of an audio fingerprint has to be small enough to fit the database on RAM.
- **Fast**: The most crucial feature of a searching system is speed. Users do not want to wait too long for a searching session on a server. In a high workload system, the long searching time may lead to declined service, affecting badly other parts of the system.
- **Scalable**: The number of music songs is increasing day by day, then the audio fingerprint system has to comfort adding more items after the first initiation.

All five requirements are essential and depend on each other. For example, the robustness can affect reliability because when the system can discriminate among different songs well, it may be more sensitive to distortion. Due to the accuracy aspect increasing, the speed may decrease because the algorithm may use more data for processing.

In this study, we focus on the first process - Fingerprint Extraction, which is mainly to generate a fingerprint of an audio file. We hope that, by doing this study, more information to improve an audio fingerprint system is discovered.

1.2 Methodology

Two questions we try to answer are:

1. How well is the performance OpenCL can give?
2. How much effort is different in using high-level synthesis versus register-transfer level?

To answer the first question, we implement an algorithm initially developed for CPUs (a sequential algorithm) by OpenCL. The algorithm we choose is HiFP2.0, which is designed for generating audio fingerprinting. OpenCL offers two programming models, which are: Single-task kernel and ND-range kernel. By modifying many kernels and applying optimizations, we try to find the best implementation of the HiFP2.0 algorithm. Finally, we examine the speedup of OpenCL on our implementations. We increase the utilization of resources by OpenCL, which are related to the number of work-items and work-groups, then we measure how well the performance changes.

To answer the second question, we make a comparison between an implementation using OpenCL and VHDL. The method used in OpenCL is Single-task kernel. The measurement we use is the number of code lines. A program contains many types of code for many purposes, such as configuration, data loading, data transfering, data processing, communication, so on. A lot of them are similar to other programs, which is almost template. The programming effort spend on template code is not too much. Our study focuses on the core part of the program, which is different from others, to implement it using OpenCL and VHDL, then compare the number of code lines.

1.3 Contribution

The research field about FPID generation is wide, and impossible to cover it all in one work. In this study, we choose to implement the HiFP2.0 algorithm on an FPGA by using OpenCL (a high-level synthesis) and investigate its performability and productivity. We expect a little performance lost to gain better productivity—the easier in approach, the more chances for FPGAs coming to users and life.

Our contributions are summarized as below:

- Develop a program to realize the HiFP2.0 algorithm on an FPGA by High-level synthesis. Although this thesis is all about FPGAs' technologies, the main idea can be used for applying HiFP2.0 on GPGPU. Because the main structure of OpenCL (the framework used in our implementation) is almost the same as CUDA (the framework used to implement GPGPU application).

- Improve the perfomance of HiFP2.0 algorithm. HiFP2.0 algorithm is originally designed for C language, which is run on a CPU. In our study, we try to modify the original algorithm and find a best way to execute HiFP2.0 on an FPGA by OpenCL. Our result show that, the optimized HiFP2.0 implementation can be achieved by using ND-range kernel.
- Evaluate the productivity of OpenCL. OpenCL is a High-level synthesis technology, it was born to reduce the tough of FPGA application development. In our study, we implement the core part of HiFP2.0 in both OpenCL and VHDL and then make a comparison of number of code lines on them.

1.4 Thesis Structure

This thesis is constructed as follows:

- **Chapter 1 - Introduction** shows purposes and contributions for our study.
- **Chapter 2 - Audio Fingerprinting** provides a big picture in this research field, which is related to FPID generation techniques.
- **Chapter 3 - OpenCL, High-level synthesis & cross-platform standard** introduces the basic concepts of OpenCL - a standard for HLS development on FPGAs.
- **Chapter 4 - Implementation of HiFP2.0 using OpenCL** shows our approaches to implement HiFP2.0 in OpenCL. Some optimization strategies also are shown.
- **Chapter 5 - Evaluation** presents experimental results.
- **Chapter 6 - Conclusion, Limitation & Future work** summarizes our works, the OpenCL's and our implementation's limitation, and the possible works in the future.

Chapter 2

Audio Fingerprinting

2.1 Introduction

Since 100 years ago, the human fingerprint research field has discovered that every person has their unique fingerprint, which benefited many other fields, such as forensic science. All nations around the world use fingerprints to identify people. The "fingerprint" means to unique thing used for discriminating one thing from other things. Hence, audio fingerprinting is a method for extracting a feature from an audio file used to discriminate with other audio files. An audio fingerprint is extracted from an audio file and gotten all characteristics of that audio file. Figure 2.1 is an example of a human fingerprint, and figure 2.2 is an audio fingerprint extracted from a random song using the HiFP2.0 algorithm.

Figure 2.1: A human fingerprint

In figure 2.2, the black dots are represented for bit 1 and the whites for bit 0. The top part contains all whites; it means the first part of the target song is quiet.

Figure 2.2: An audio fingerprint extracted by HiFP2.0 algorithm (4096 bits)

2.2 Definition

Recall that an audio fingerprint is a feature that represents the characteristics of an audio file. Hence, the size of an audio fingerprint is less than the song/audio file's size. We use a fingerprint function F to map the audio file X to its audio fingerprint F(X). The threshold T for deciding whether the two fingerprints are made from the same audio/song file or not.

Then, we have:

- $||F(X) - F(Y)|| < T$, X and Y are similar
- $||F(X) - F(Y)|| >= T$, X and Y are dissimilar

There is another way to define an audio fingerprint by a cryptographic hash function H(). However, it has known that not suitable because of the strict (sensitive) of the cryptographic hash function. The two songs' similarity is a perceptual similarity, realizable by the human auditory system, but may be completely different from a mathematical comparison. It happens due to the difference in the file format or the distortion of an audio/song. For example, people cannot hear/realize the differences between a song in raw format and a compressed format (mp3), but their waveforms may differ. The difference of waveform leads to feature made by cryptographic hash function become different, although the input has a 1-bit difference. There is also a study that explained this problem [6].

2.3 Applications

The social network and online entertainment have been developing. Going with it is the use of audio content increased, then they need an efficient way to manage the audio content database. Some researchers point out the applications in [3, 4, 9, 16] .Audio fingerprint plays an important role in their systems. The uses of them may be transparent to end-users. Some examples of real-life application can be told are:

- **Filtering illegal audio content**
  Nowadays, people use social network platforms for uploading personal songs or videos to share with their friends, family. Some parts of their audio/video may contain illegal content produced by a singer or organization for commercial purposes and be licensed by social network platforms. Each licensed music uploaded in the provider's system generates an audio fingerprint for searching/comparing with the one user uploaded. If the user's audio/video were known as containing licensed audio, the system would refuse to upload.

- **Organizing music library**
  The record of a song may exist in many types of format and distribute in any place on the Internet. For example, a human can download a raw format of "Let it be - The Beatles" from website A and have an mp3 format from website B. When building the playlist, the song may be played two times due to redundancy. Although humans can hear the content of 2 audio files as the same content, the two audio files' waveforms are different, leading to the player not discriminating. In this problem, an audio fingerprint can be used to identify songs and remove redundant songs in a playlist.

- **Detecting unknown songs/rythm**
  Do you remember when you randomly hear a song or rhyme but cannot precisely remember its name? Then this is a problem the audio fingerprint can help. An audio fingerprint application could be installed on a mobile phone. When users need to find the name of any songs hearing, open an application when they can keep the radio. The application may send the record to an audio fingerprint server to identify the song and then give back the information needed.

2.4 Related works

2.4.1 Robust Hash Extraction Algorithm

Haitsma's research pointed out the valuable information of audio content is hidden in the frequency domain [7]. Then on each chosen frame of audio content, one Fourier Transform calculating is conducted. After that, an FPID is generated by comparing the energy differences.

Because this algorithm uses Fourier Transform, it requires a floating-point operation, which consumes enormous computational cost.

Figure 2.3 is an overview of the Robust Hash Extraction Algorithm.

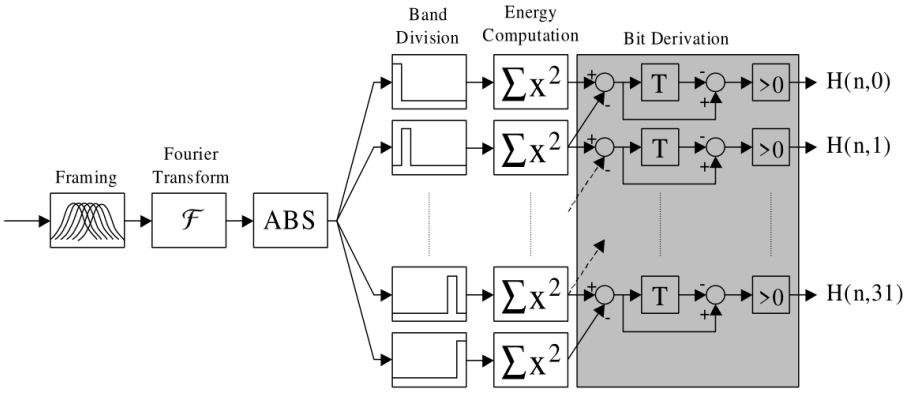

Figure 2.3: Overview of Robush Hash Extraction Algorithm, cited from [7]

2.4.2 HiFP2.0 Algorithm

HiFP2.0 algorithm is proposed by Araki [1], which stands for High-speed Audio FingerPrinting. As the name the algorithm has, HiFP2.0 can reduce the time for generating the fingerprint by using only integer operation. The overview of HiFP2.0 can be seen in Figure 2.4.

HiFP2.0 algorithm has 2 main processes:

- Multi-level subband decomposition using Haar wavelet transform
- Feature extraction/ FPID generation

The central idea of HiFP2.0 is to extract the orientation of the low band of an input audio signal. Figure 2.5 shows the imagination of how HiFP2.0 works. For each 4-samples, if the orientation from the first sample to the last

sample increases, we have bit 1 in the FPID. If the orientation decreases, we have a 0-bit in the FPID. Because the HiFP2.0 algorithm uses integer calculation instead of floating-point calculation, it is faster than other methods used Fourier Transform [1]. The algorithm of HiFP2.0 can be seen in Figure 2.6 and 2.7.

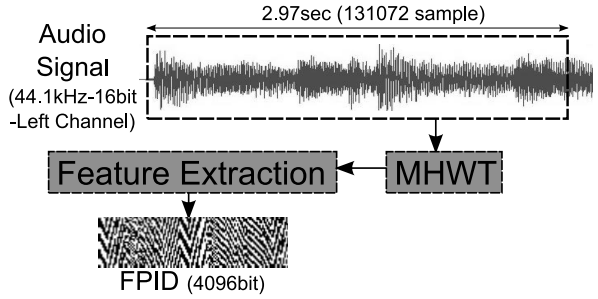

Figure 2.4: HiFP2.0 framework, cited from [1]

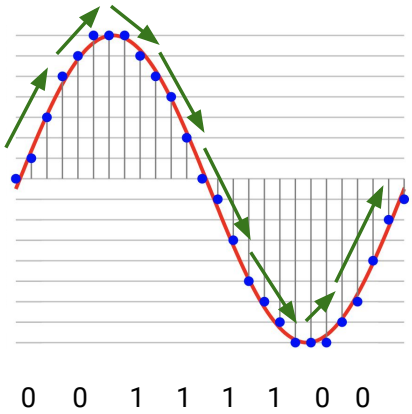

Figure 2.5: Imagination of how HiFP2.0 works

2.5 Chapter Summary

This chapter presents the necessary information on an audio fingerprint. We know that there are two ways to increase a program's speed: Do fewer software instructions or equip more hardware resources. HiFP2.0 algorithm chooses the first approach. By using integer calculation, the computational cost of HiFP2.0 is small compared to other methods using floating-point operations but still does the discrimination well.

```
1. MHWT(wav[] ← Input signal,
2.              n ← Number of samples of input signal,
3.              m ← Number of output samples){
4.     for ( ; n > m ; n/ = 2 ) then
5.       for ( i = 0 ; i < n/2 ; i + + ) then
6.          Hi[i] = (wav[2 × i] − wav[2 × i + 1])/2;
7.          Lo[i] = (wav[2 × i] + wav[2 × i + 1])/2;
8.       end for
9.       wav[] ← Lo[];
10.    end for
11.    return (Hi, Lo);}
```

Figure 2.6: HiFP2.0 framework, originally developed by Araki, cited from [1]

```
1. HiFP2.0(wav[] ← PCM data){
2.     n ← 131,072; /*Number of samples of PCM data*/
3.     m ← 16,384; /*Number of output samples*/
4.     Hi[], Lo[] ← MHWT(wav[], n, m);
5.     j ← 0;
6.     for (i = 0 ; i < m − 4 ; i+ = 4 ) then
7.       tmp = Lo[i] − Lo[i + 4];
8.       if tmp > 0 then
9.          FPID[j] = 1;
10.      else
11.         FPID[j] = 0;
12.      end if
13.      j + +;
14.    end for
15.    FPID[m − 1] = 0;
16.    return FPID; }
```

Figure 2.7: HiFP2.0 framework, originally developed by Araki, cited from [1]

In this study, we use the HiFP2.0 algorithm for implementation on FPGA using OpenCL.

Chapter 3

OpenCL - High-level synthesis & cross-platform standard

3.1 Introduction

In this chapter, we discuss the main points of the OpenCL platform, which are used to understand our implementation in the next chapter better.

3.2 OpenCL Architecture

The OpenCL architecture is defined by 3 concepts:

- Platform model
- Execution model
- Memory model

3.2.1 Platform Model

OpenCL's platform model contains two parts, which are host and OpenCL devices. One host is connected to one or many devices. A host is a computer with a CPU running an operating system (Windows, Linux, macOS, ...). A device may be CPU, GPU, DSP, or FPGA. This model supports both task-based and data-based parallelism. One OpenCL device contains many compute units (CUs), and one compute unit contains many processing elements (PEs). The overview of OpenCL's platform can be seen in figure 3.1.

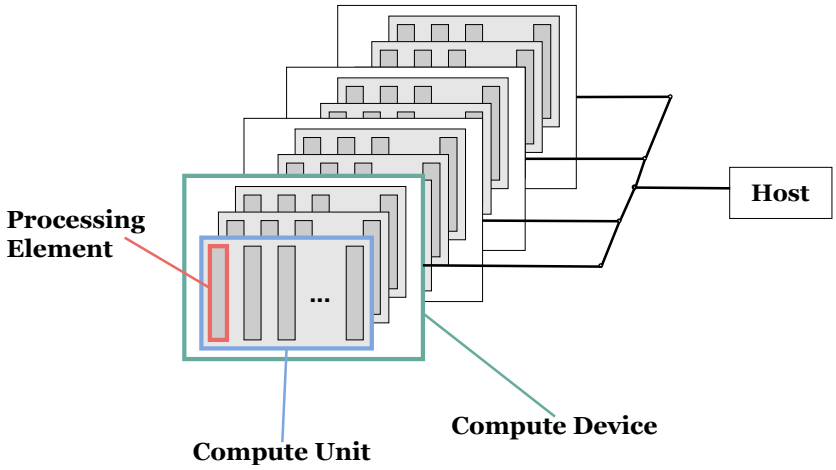

Figure 3.1: OpenCL's platform model

The processing elements execute instructions as SIMD (Single Instruction, Multiple Data) or SPMD (Single Program, Multiple Data). One of OpenCL's advantages is running on many different device types, likes Java (write once, run everywhere). While a device's hardware is different from others among many providers, the standard concepts are not changed.

The number of CUs or PEs is different among different devices, and then engineers have to construct their programs to process data with many sizes of CUs or PEs. The information of a device can only be determined at runtime, then the decision of dividing data should be made at the runtime.

The ATI Radeon HD 5870 GPU has 20 SIMD units mapped to 20 compute units in OpenCL.

3.2.2 Execution Model

The OpenCL's execution model has two components: A host program and kernels. Kernels contain executable instructions that can be run on many devices. In the SIMD mode, the kernel's instructions are executed by each processing element of devices.

Host program

The host program is executed by the host side and programmed in high-level languages like C/C++, Java, and Python using SDK (Software Development Kit) from hardware providers. For example, Intel provides OPAE (Open Programmable Acceleration Engine) - a software framework for managing

and accessing programable accelerators (FPGAs). The OPAE contains SDK, drivers, libraries.

The host can create a context to manage the execution of kernels by the following resources:

- **Devices**: The OpenCL's devices for executing kernels.
- **Program Objects**: The program compiled by the hardware provider's compiler is transferred to the device side when a kernel call is made from a host. One program can contain many kernels.
- **Kernels**: OpenCL functions contains instructions for executing in device side.
- **Memory Objects**: A collection of memory is used by devices.

A command queue is used for two sides' communication. A host can send a message to a device through a queue. If there are many devices, each device uses one queue for communication. The content of the communication of a queue is isolated from the others. Three types of commands that can be called within a queue are:

- **Kernel execution command**: To indicate which kernel in a program object to be used.
- **Memmory command**: For transferring data from host to device or from device to host.
- **Synchronization command**: In the default, all commands in a queue are executed concurrently. The synchronization command uses events to force them from executing sequentially.

Main actions have to be made in Host side are:

- Query platforms
- Query devices
- Create contexts
- Create/built programs
- Create command-queues
- Create kernels

- Create buffers
- Write data to buffers
- Enqueue tasks into command-queues
- Release resources

3.2.3 Memory Model

OpenCL has a hierarchical memory model, which consists of 4 regions:

- **Host memory**: The memory is placed on the host side of OpenCL's platform model but visible to both sides. Typically the memory is a DDR type. The problem data has to be loaded and prepared on the host side, then transferred to Global memory on the device side.
- **Global memory and Constant memory**: The memory is placed on the device side of OpenCL's platform model. Like the Host memory, Global and Constant memory are visible to both sides. Constant memory use for storing kernel parameters. Because they are designed at the top of the memory model's hierarchical structure, they are visible to all lower components of the device: compute unit and processing elements.
- **Local memory**: The memory is placed on the device side of OpenCL's platform model. This type of memory can be visible to all processing elements in each compute unit. The access scope is located only in a Compute unit.
- **Private memory**: This memory type has a smaller scope than Local memory and only be visible to one processing element that is pair with it.

The overview of OpenCL's memory model is shown in figure 3.2.

A host and a device always use Host memory and Global memory to transfer data in and out. The first time transfer is for loading data to the device to prepare for after execution. The last time transfer is for returning the result from the device back to the host.

On the device side, the Global memory is the most extensive capacity memory, follows local memory, and the last is Private memory. By considering the speed, Private memory is the fastest, the Local memory is the 2nd, and the lowest memory is Global memory.

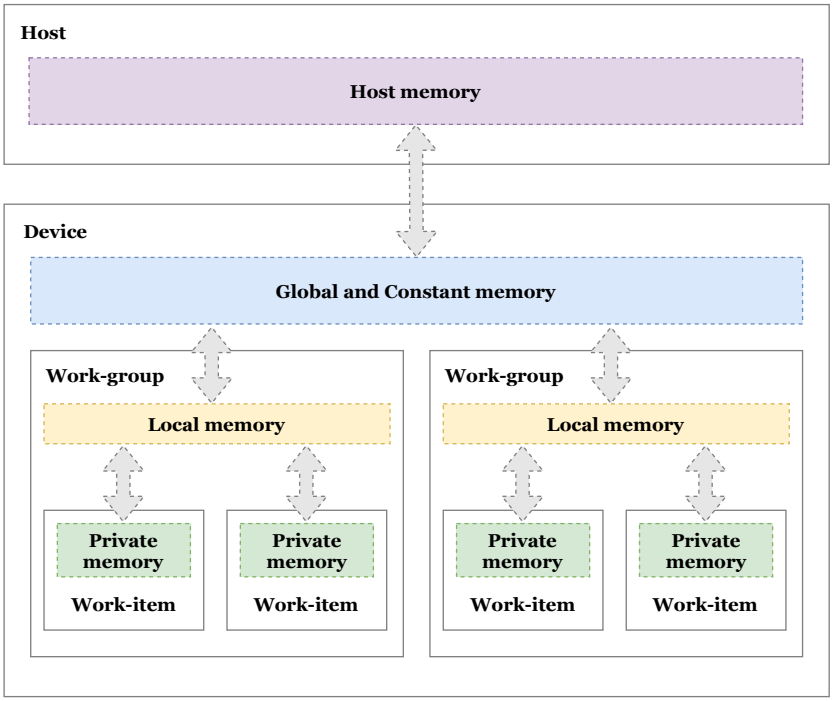

Figure 3.2: OpenCL's memory model

3.3 Optimization Strategy

3.3.1 Loop-unrolling

Loop-unrolling is an OpenCL optimization technique. The applied program uses more memory resources in order to reduce the total of clock circles used.

Figure 3.3 shows an example of using this technique. The number of clock cycles reduces from 4 to 1.

Loop-unrolling is triggered when the OpenCL's compiler meets the **#pragma unroll** before the for-loop.

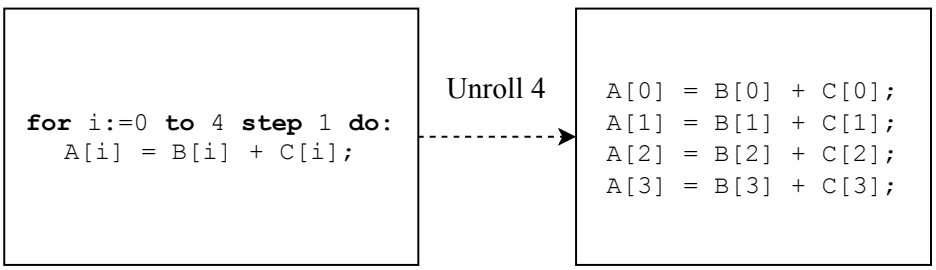

Figure 3.3: An example of using Loop-Unrolling

3.3.2 Data-preserving

By using Data-preserving, we can reduce the loading time between work-item and memory.

To implement this technique, we need to load data from Global Memory into the Private Memory, then conduct processing on the new data, instead of using Global data like normal.

3.4 Productivity improvement

In the past, the Register-transfer level had been the only technology for developing FPGA applications. Because the resource is specific for each device, such as the number of ALUTS, FFs, so on, and RTL design of an algorithm highly depends on the hardware architecture. It is not easy to reuse the same RTL design for many FPGA devices effectively. The high-level synthesis was born to reduce the tough of developing FPGA applications and generalize using the same program for many FPGA devices.

3.4.1 Definition

Productivity can be measured in time spend and the number of code lines. Some studies used the whole source code of a program to make a comparison on the number of code lines. We know that a program is constructed by many types of source code, which can be the same among software. The most important part of a program that made a difference is the core algorithm. In our study, we use the number of code lines of the core part of an algorithm to make a productivity comparison between HLS and RTL language. We expect the less number of code lines the better productivity.

3.4.2 Related works

In this section, we list several studies that investigate in performability and productivity aspect of OpenCL:

- Fäerber et al. [5], by using OpenCL instead of Verilog, their implementation of the Cherenkov Angle Reconstruction algorithm can be reduced nearly 13.6 times.
- Hill et al. [8], by observing the whole design flow of both VHDL and OpenCL, the author states that 6 times increment in productivity is possible.

- Memeti et al. [12] compare the productivity among OpenCL, OpenMP, OpenACC, and CUDA. They use CodeStat for measurement and quantifying the parallelization effort. By investigating many algorithms, they find that the parallelization effort (of the number of code lines of OpenCL in a total of code lines of a program) is 2.8% - 9.63%.

3.5 Chapter Summary

This chapter presented the main concepts and information about OpenCL. Although the standard was born in 2008, OpenCL applications are not so popular compared to NVIDIA's CUDA. One of the reasons is that OpenCL is an open standard, which is developed by many companies. Then many features of OpenCL are not implemented and stable yet.

There are not so many studies that investigated in performability and productivity aspect of the OpenCL. It is a considerable downside affecting the standard's development.

Chapter 4

Implementation of HiFP2.0 using OpenCL

4.1 Introduction

OpenCL standard defines two types of kernels, which are Single-task kernel and ND-range kernel. Due to the uncertainty about which one is better than the other, especially when the device we used in our study is an FPGA, implementing two types of the kernel is needed to see the differences between the two. The implementation using a Single-task kernel may be fully extracted parallel capability into hardware. However, on the other side, the implementation using an ND-range kernel may take advantage of an engineer's heuristic experience.

This chapter presents two approaches to implement HiFP2.0 using High-level synthesis: ND-range kernel & Single-task kernel.

The complete source code is published at *https://github.com/m-inh/hifp*.

4.2 Processing Flow

The HiFP2.0 program gives an audio file; after processed in 2 modules, an FPID is generated.

The two modules of processing flow are listed below:

- **Pre-processing**: This module is to extract the wave data in the given audio file. After extracting the audio file's header (in RIFF format), the appropriate wave data is read. After all, 131,072 wave samples are loaded into Host memory.

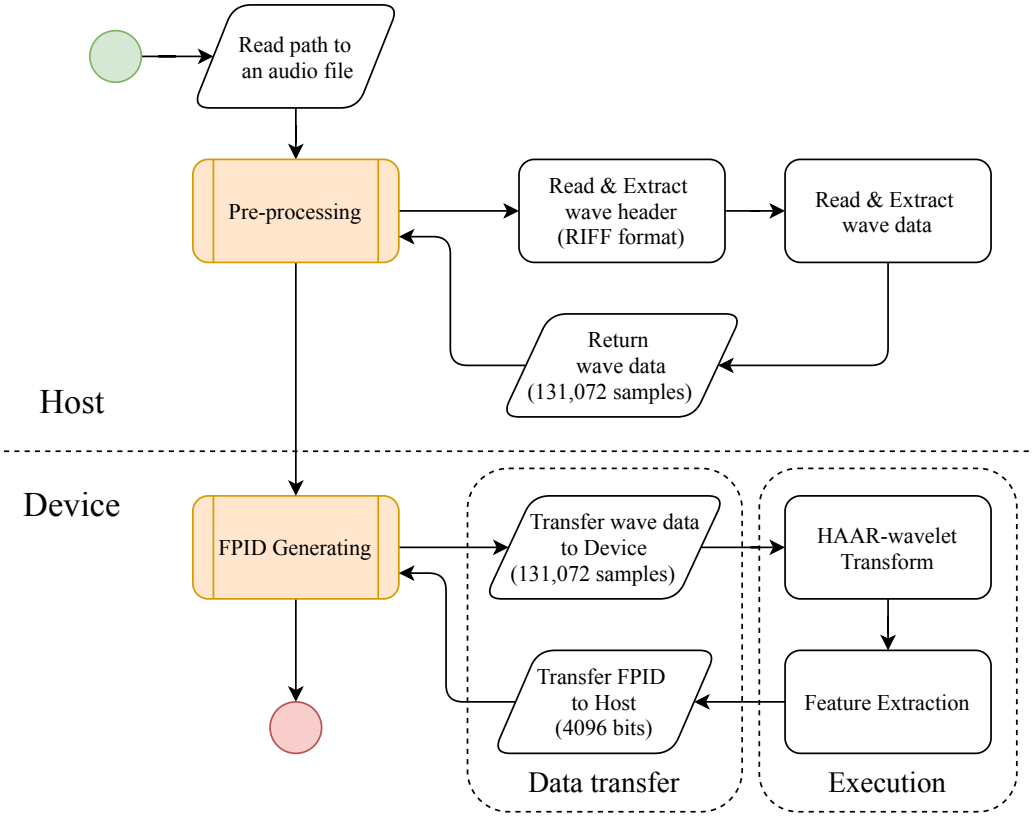

Figure 4.1: Processing flow of HiFP2.0 program using OpenCL

- **FPID Generating**: In this module, the wave samples given from the previous step are transferred to the device side, then be processed at the Execution phase. After the kernel execution, an FPID containing 4096-frames is transferred back to the host. FPID is the outcome we need.

The overview of the entire process is shown in Figure 4.1.

4.3 3-Stages of Development Process

Figure 4.2 shows the complete stages of the development process.

Starting development with C or C++ version is a usual way to develop an OpenCL program. C was born in 1972. Until now, there are many stable libraries, frameworks built-in, and based on C/C++ ecosystems. Starting with C/C++, in the 1st stage, the first version of the algorithm can build and take advantage of libraries and frameworks' diversity. We can apply any appropriate testing frameworks to guarantee the result as correct. In the first step, we need a runnable and correct program, at least.

After the C/C++ version is done, the 2nd stage is to implement the Single-task Kernel (OpenCL) algorithm. We have to use FPGA vendors' resources (like libraries or frameworks, simulation tools) and modify the source code to make it executable on both the Host and Device sides. The vendor's compiler conducts optimization on our program automatically, then the source code from C/C++ is not changed so much. OpenCL's language is mainly based on C, so the transformation should be straightforward.

Finally, the 3rd stage is to implement the algorithm in the ND-range kernel style. In this implementation, more insight and a deep understanding of the algorithm have to be used. Unlike the Single-task kernel version, we have to divide the input data and distribute them into many work-items and work-groups in this work. The performance and resource depend on how we design the data partition and the parameter choices. In this part, the Host side's work is almost untouched, and we focus on the Device side (or kernels). The procedure of modifying Single-task kernel to ND-range kernel seems popular to other studies [17].

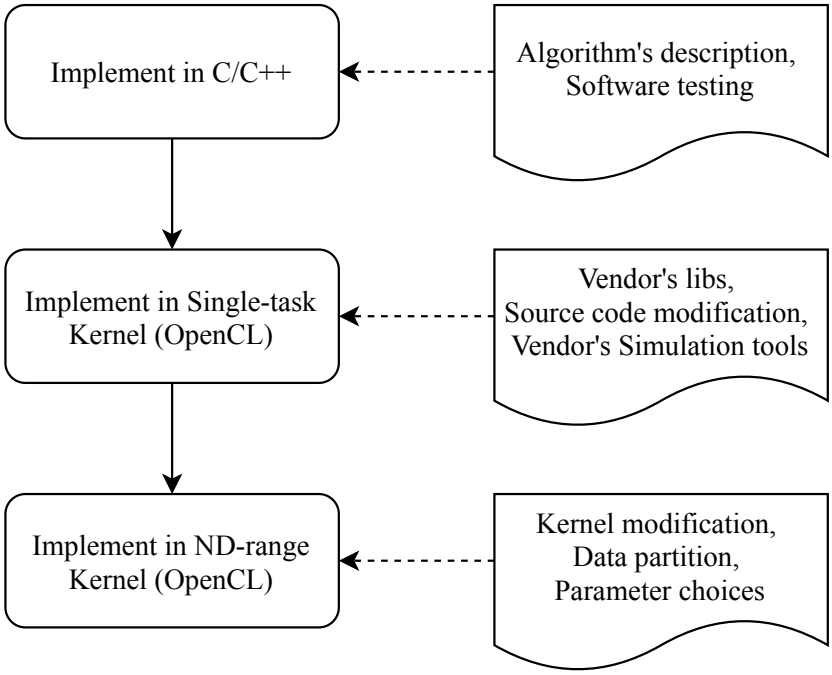

Figure 4.2: 3-Stages of Development process

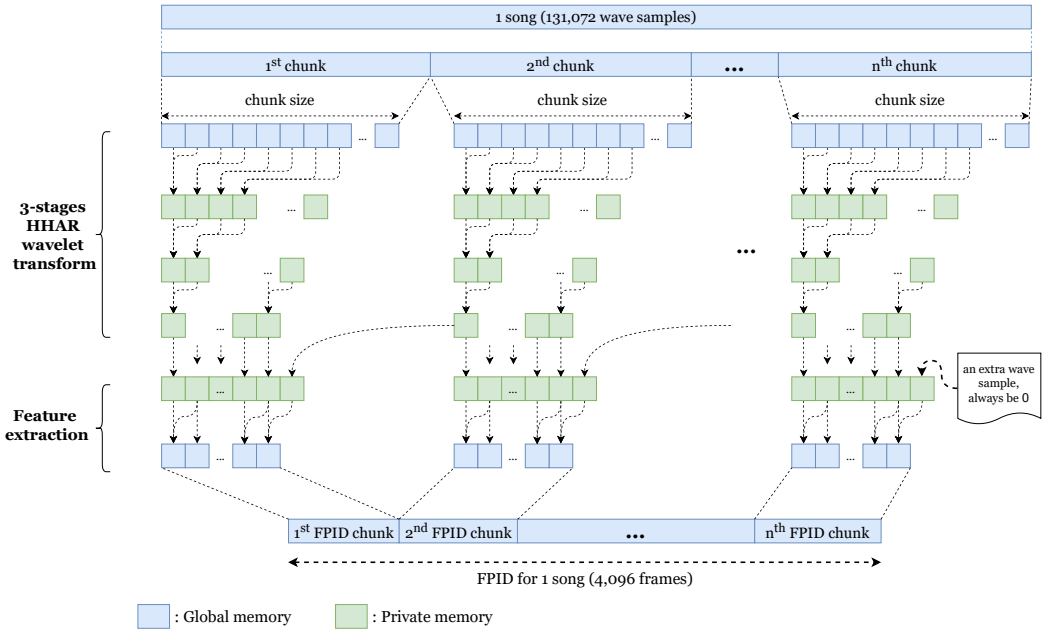

Figure 4.3: Processing one song using Single-task kernel

4.4 Kernel Implementation

4.4.1 Single-task Kernel

With the Single-task kernel style, the code we wrote is the same as the original code, which is initially designed for CPU programs. OpenCL's compiler optimizes the kernel code automatically, and the optimization is mainly about pipelining the data in the program. The overview of the implementation using a Single-task kernel can be seen in figure 4.3.

Firstly, we need to implement the algorithm in C or C++, then break the part of the code, which can be accelerated, to let it be executed in a device. This part of the code is organized in one OpenCL kernel.

According to Figure 4.1, the Single-task kernel contains two processes in the Execution phase: HAAR-wavelet Transform & Feature Extraction.

Figure 4.4 show the idea for 3-stages wavelet transform of HiFP2.0. When conducting the HAAR-wavelet transform in the CPUs, the execution is sequential. The 3-stages wavelet transform needs 7 instructions to complete, and then the CPU version needs 7 clock cycles. However, for FPGAs, the dedicated circuit for the program is built, then 3 cycles are enough.

Figure 4.5 shows the idea for the Feature Extraction stage. In this stage, we compare the value of each 2-adjacent samples. If a wave sample's value is higher than the previous one, a 0-value bit is returned; otherwise, a 1-value bit is produced. In order to simplify the source code, an input array has 4097 wave samples given. The last wave sample has 0 in value. As a result, the last frame of output FPID always has a 1-value bit.

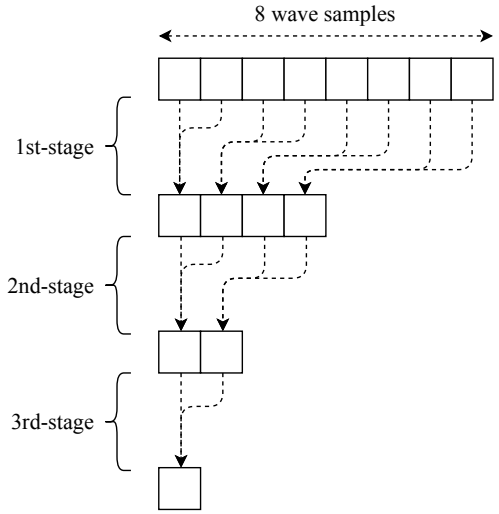

Figure 4.4: 3-stages wavelet transform

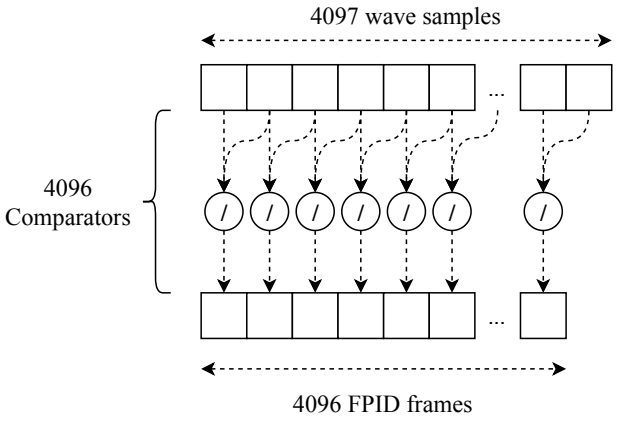

Figure 4.5: Feature extraction

Algorithm 1 shows the complete implementation of HiFP2.0 using a Single-task kernel. The input is an array stored 131,072 wave samples. The output is an array stored Fingerprint ID, which has 4,096 frames. To store all wave samples after the HAAR-wavelet transform process, we use an array declared in line 1 as *dwt_wave*. After the HAAR-wavelet transform process, 4096 wave samples are returned. To simplify the source code in the later process, we add one more 0-value sample at the last position of *dwt_wave* array. In the ideal situation, 4096 work-items are chosen to execute this kernel, and then they are nice fit into the data we have. 131,072 wave samples in the 1st process are assigned to 4096 work-items, and 4097 DWT wave samples in the 2nd process are assigned to 4096 work-items. It opens a chance for us to implement an efficient FPGA algorithm because the task is divided into many independent sub-tasks. From line 3 to line 19, the 1st stage of HiFP2.0 is conducted. We use an array to store the 8-first samples on each 32 wave samples of the input wave samples, and it is called *wave_tmp[8]*. The for loop in part between line 11 to line 16 is to conduct 3-stages HAAR-wavelet transform. The imagination of the 3-stages HAAR-wavelet transform is shown in figure 4.4. After the for-loop of the HAAR-wavelet transform, the wave sample we need is at the 1st position of *wave_tmp* array. In the line 22, we assign the last value of *dwt_wave* to 0. From line 25 to line 31, the Feature Extraction process is conducted. The last frame of output *FPID* is 1, due to the last wave sample in *dwt_wave* always be 0.

4.4.2 ND-range Kernel

In the Single-task kernel implementation, optimization is conducted automatically by the vendor's compiler. If the compiler works well in every situation,

Algorithm 1: Single-task kernel implementation

Input : Wave data array: WAVE_SAMPLES (131,072 samples)
Output: Fingerprint ID: FPID (4096 frames)

```
1  dwt_wave[4097];
2
3  /* 1st stage of HiFP2.0: HAAR-wavelet transform */
4  for i:=0 to 4096 step 1 do
5  |  wave_tmp[8] ← Load 8 wave samples from WAVE_SAMPLES;
6  |
7  |  /* Conduct HAAR-wavelet transform on 8-samples */
8  |  for k:=8 to 1 step k/=2 do
9  |  |  for l:=0 to k/2 step 1 do
10 |  |  |  wave_tmp[l] := (wave_tmp[l*2] + wave_tmp[l*2 + 1]) / 2;
11 |  |  end
12 |  end
13 |
14 |  dwt_wave[i] := wave_tmp[0];
15 end
16
17 /* The last wave sample is 0 */
18 dwt_wave[4097] := 0;
19
20 /* 2nd stage of HiFP2.0: Feature extraction */
21 for i:=0 to 4096 step 1 do
22 |  if dwt_wave[i] > dwt_wave[i+1] then
23 |  |  FPID[i] := 1;
24 |  else
25 |  |  FPID[i] := 0;
26 |  end
27 end
```

the ND-range kernel is not needed. However, in practice, we know that the idea environment may not work. Many conditions affect the compiler decisions, which should do and do not. One of the possible reason is memory architecture. The OpenCL has 3 memory types, which are: Global memory, Local memory, Private memory. It is hard for the compiler to decide on what memory type should be used in many situations. The memory model is tightly related to what work-items and work-groups to be used, so in some situations, the algorithm has to be changed to adapt. In the ND-range kernel style, heuristic experience may improve the algorithms, which are originally designed to be run by a CPU.

Processing one song per execution using one work-group

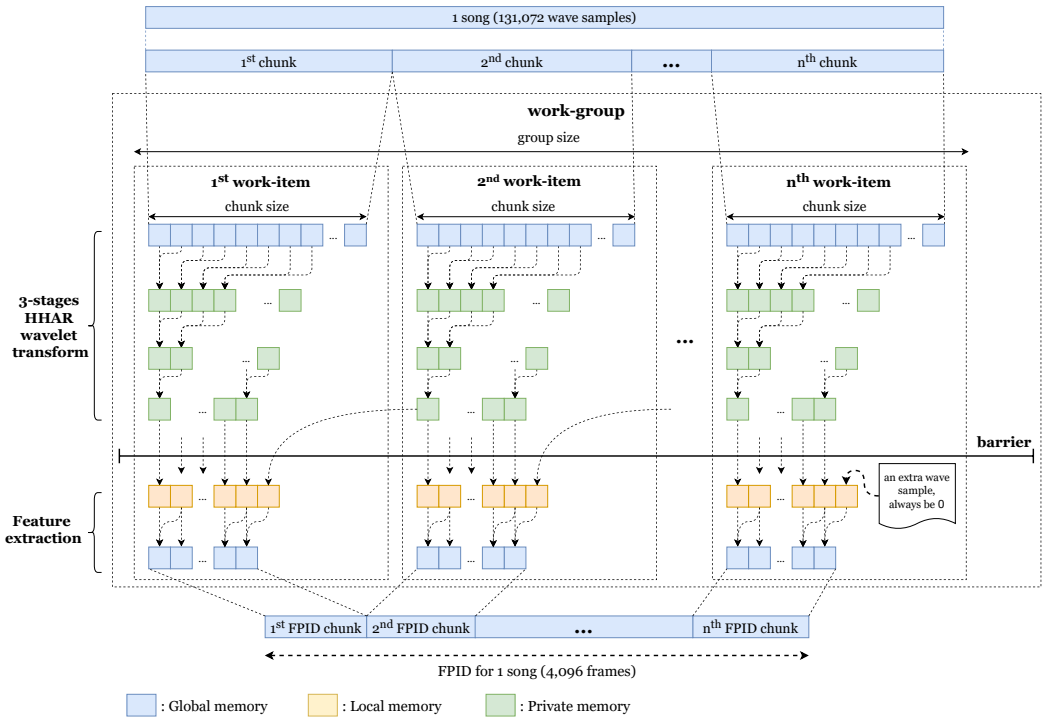

Figure 4.6: Processing one song per work-group using ND-range kernel

We need to complete the Single-task kernel version to implement an algorithm in the ND-range kernel style. The ND-range kernel is based on previous work, and the main changes are mostly conducted in the kernel code, which is run by the device side. A way for distributing the input data into many work-items and work-groups has to be investigated. The overview of the implementation is shown in figure 4.6.

Algorithm 2 shows the complete pseudo kernel code for HiFP2.0 using the ND-range kernel style. Due to the 1st-stage of HiFP2.0 (HAAR-wavelet transform) has to be done before going next, we have 2 options to implement the kernel. The first option is to use 2 kernels, one for HAAR-wavelet transform, one for Feature Extraction. The second option is to use one kernel with a Local memory type in OpenCL. The more kernels used, the more Global memory is hit. Global memory is the lowest memory type of OpenCL standard, so the first option is not optimal. To use the second option, we construct the kernel to process one song per work-group using the Local memory. To synchronize 2-stages of HiFP2.0, we use *barrier()* function between them.

The first input of this kernel is an array called WAVE_SAMPLES, which stores 131,072 wave samples. The second input is the number of work-items per work-groups called GROUP_SIZE; by using this parameter, the workload per work-item can be specified within the kernel. Table 4.1 shows the relation between work-group size and the number of wave samples assigned to one work-item (called chunk_size). We have $chunk\_size = \frac{4096}{GROUP\_SIZE}$. The more work-items are used, the fewer number of wave samples be assigned to each work-item.

Work-group size	Number of wave samples per work-item
4096	1
2048	2
1024	4
512	8
256	16
128	32
64	64
32	128

Table 4.1: Relation between work-group size and the number of wave samples be assigned to one work-item

Processing multiple-songs per execution using multiple work-groups

It is straightforward for changing from processing one song per execution to processing multiple songs per execution. Instead of loading one song per execution, the host is changed to load multiple songs and save them into the host's memory until they reach the designated number of songs and then transfer them to the device at once. The size of global work is changed from

Algorithm 2: ND-range kernel implementation (one-song)

Input : Wave data array: WAVE_SAMPLES (131,072 samples)
Work-group size: GROUP_SIZE
Output: Fingerprint ID: FPID (4096 frames)

```
1  lid := get_local_id();
2  chunk_size := 4096 / GROUP_SIZE;
3  fpid_offset := lid * chunk_size;
4
5  local dwt_wave[4097];
6
7  /* 1st stage of HiFP2.0: HAAR-wavelet transform, run in parallel */
8  for i:=0 to chunk_size step 1 do
9  |  wave_offset := (fpid_offset + i) * 32;
10 |  wave_tmp[8] ← Load 8 wave samples from WAVE_SAMPLES;
11 |
12 |  /* Conduct HAAR-wavelet transform on 8-samples */
13 |  for k:=8 to 1 step k/=2 do
14 |  |  for l:=0 to k/2 step 1 do
15 |  |  |  wave_tmp[l] := (wave_tmp[l*2] + wave_tmp[l*2 + 1]) / 2;
16 |  |  end
17 |  end
18 |  dwt_wave[fpid_offset+i] := wave_tmp[0];
19 end
20
21 /* The last wave sample always be 0 */
22 dwt_wave[4097] := 0;
23
24 barrier();
25
26 /* 2nd stage of HiFP2.0: Feature extraction, run in parallel */
27 for i:=0 to chunk_size step 1 do
28 |  if dwt_wave[fpid_offset+i] > dwt_wave[fpid_offset+i+1] then
29 |  |  FPID[fpid_offset+i] := 1;
30 |  else
31 |  |  FPID[fpid_offset+i] := 0;
32 |  end
33 end
34
35 return FPID;
```

131,072 to $131,072 * N$, with N is the number of songs per one execution.

Figure 4.7 shows how multiple songs be assigned to multiple work-groups.

Algorithm 3 shows the whole idea for the kernel code written for the device side. Due to the global memory being changed for storing multiple songs, then the work-group has to indicate the data it needs. The work-group's input wave samples is located by wave_global_offset, which is calculated by: $wave\_global\_offset = group\_id * 131,072$. The work-group's ID presented by group_id is retrieved from OpenCL API. The output FPID is also located by the same way, calculated by $fpid\_global\_offset = group\_id * 4096$.

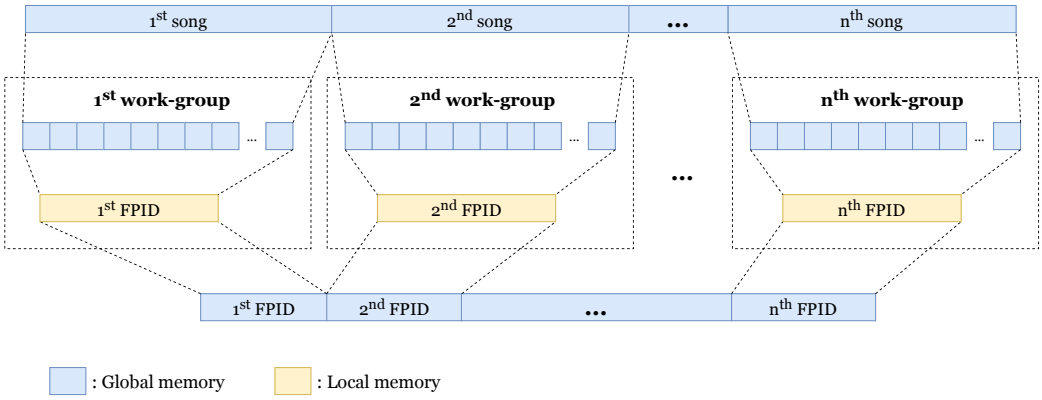

Figure 4.7: Processing multiple-songs with multiple work-groups

4.5 Chapter Summary

In this chapter, we show two approaches to implement HiFP2.0 by using High-level synthesis.

The Single-task kernel seems to be more comfortable in development because most parts are the same as typical programs executed in a host by CPU.

The ND-range kernel style is almost the same as the CUDA programming model; they all use the Single Instruction and Multiple Data model (SIMD) to solve a problem. This model seems to be more challenging than the Single-task because the algorithm's code-base must be changed and required a deep understanding of the original algorithm.

Algorithm 3: ND-range kernel implementation (multiple songs)

Define : N is the number of songs
Input : Wave data array: WAVE_SAMPLES (131,072*N samples)
Work-group size: GROUP_SIZE
Output: Fingerprint ID: FPID (4096*N frames)

```
1  lid := get_local_id();
2  group_id := get_group_id();
3  chunk_size := 4096 / GROUP_SIZE;
4  fpid_offset := lid * chunk_size;
5
6  wave_global_offset := group_id * 131072;
7  fpid_global_offset := group_id * 4096;
8
9  local dwt_wave[4097];
10 local sub_fpid[4096];
11
12 for i:=0 to chunk_size step 1 do
13 |   /* Run in parallel */
14 |   dwt_wave ← Conduct HAAR-wavelet transform for one song
   |     corresponding to wave_global_offset;
15 end
16
17 /* The last wave sample always be 0 */
18 dwt_wave[4097] := 0;
19
20 barrier();
21
22 for i:=0 to chunk_size step 1 do
23 |   /* Run in parallel */
24 |   sub_fpid ← Conduct Feature extraction on dwt_wave
   |     corresponding to fpid_global_offset;
25 end
26
27 FPID ← Merge sub_fpid into global FPID corresponding to
     fpid_global_offset;
28
29 return FPID;
```

Chapter 5

Evaluation

5.1 Introduction

This chapter tends to present the experimental results of our proposed methods. Firstly, we show the productivity comparison between VHDL and OpenCL in term of number of code lines. Secondly, we show the performance of OpenCL, the kernel we optimize and use for evaluation is ND-range kernel. Performance aspects will be shown are:

- Execution-time & Throughput
- Total-time
- Speedup ratio
- Resource Usage

The comparison between our method and the original method will also be discussed.

5.2 Productivity comparison with VHDL

To study the productivity aspect of OpenCL, we implement the core part of a VHDL program to run HiFP2.0 and simulate it with ModelSim[1] (from Mentor Graphics) and then compare the number of code lines between them, VHDL program and OpenCL kernel. The OpenCL implementation used for the comparison is Single-task kernel. The Single-task kernel implementation and the VHDL implementation all have the same procedures to execute

[1] `https://www.mentor.com/company/higher_ed/modelsim-student-edition`

HiFP2.0 algorithm. A source file contains the code for logical instructions and the comments and break lines that make the code understandable.

In this case, we use two versions of source file:

1. Original source file that contains: logical instructions, comments, line breaks.

2. Archival source file that contains only logical instructions.

The result of our comparison is shown in table 5.1. The detailed source code for the comparison is shown in the Listing section.

To make sure our VHDL implementation works well, some test benches are developed and tested. Our result shows that by using OpenCL, the number of code lines can be reduced from 5 to 5.54 times.

No.	**File name**	**Number of code lines**	
		normal	**archive***
1	dwt.vhd	39	30
2	dwt_all.vhd	37	31
3	feature_extraction.vhd	22	18
4	hifp.vhd	52	43
	VHDL total	**150**	**122**

1	single_task_hifp.cl	30	22

Table 5.1: Number of code lines in our implementation.
** : Archive version has neither line breaks nor comments*

5.3 Theoretical Performance Analysis

5.3.1 Theoretical speedup of an optimal HiFP2.0 implementation on an FPGA

Table 5.2 shows the comparison between CPU & FPGA in a specific number of cycles used when conducting the HiFP2.0 algorithm.

In the HAAR-wavelet transform process, each 8-wave samples is reduced into a 1-wave sample, the CPU-version needs 7 cycles to complete, and the FPGA-version needs 3 cycles to complete. Expanding to 4,096 workloads, we have 28,672 (equal to 7*4,096) cycles for CPU-version and 3 cycles for FPGA-version.

We can measure the speedup in latency by the formula 5.1:

$$S = \frac{C_1}{C_2} * \frac{F_2}{F_1} \tag{5.1}$$

Where:

- S is the speedup in latency of architecture 2 concerning architecture 1
- C_1 is the number of cycles of the architecture 1
- F_1 is the clock frequency of the device corresponding to architecture 1
- C_2 is the number of cycles of the architecture 2
- F_2 is the clock frequency of the device corresponding to architecture 2

By applying to the above formula, we have the speedup in latency of FPGA architecture concerning CPU architecture of 32,768/4=8,192. For example, in an ideal environment, where the clock frequency is equal, and the compiler of OpenCL works well, the FPGA implementation should get 8,192 times faster in throughput than the CPU implementation.

	Number of cycles	
	CPU	FPGA
HAAR-wavelet transform	28,672	3
Feature extraction	4,096	1
Total	32,768	4

Table 5.2: Number of clock cycles for HiFP2.0 executed by CPU & FPGA

5.3.2 Theoretical speedup of our HiFP2.0 implementation using OpenCL ND-range kernel

In the previous chapter, algorithm 2 presented ND-range kernel implementation is explained. The algorithm gives 2 parameters as input: WAVE_SAMPLES and GROUP_SIZE. The number of wave samples is constant, then the loop over WAVE_SAMPLES has $O(1)$ complexity. In the first for-loop, it has a range of 0 to *chunk_size*. Because the *chunk_size* is calculated by $4096/GROUP\_SIZE$ and the HAAR-wavelet transform for one song always has a constant number of instructions, then the for loop has $O(\frac{4096}{GROUP\_SIZE})$ complexity. In the second for loop, the feature extraction also has a constant number of

instructions, then the outer for loop has $O(\frac{4096}{GROUP\_SIZE})$ complexity. In total, we have $O = O_1(\frac{4096}{GROUP\_SIZE}) + O_2(\frac{4096}{GROUP\_SIZE})$, which can be literally rewritten as $O(GROUP\_SIZE^{-1})$.

Then, the time complexity of algorithm 2 can be stated by following formula:

$$O(n^{-1}) \quad \text{with } 0 < n < 4096 \tag{5.2}$$

Where:

- n: Number of work-items per work-group (GROUP_SIZE)

For $n = 1$, algorithm 2 becomes a sequential algorithm, which acts the same as the origin.

By applying formula 5.2, we can calculate the theoretical speedup by following formula:

$$\text{Speedup} = \frac{O(n_1^{-1})}{O(n_2^{-1})} = O(\frac{n_1^{-1}}{n_2^{-1}}) = O(\frac{n_2}{n_1}) \tag{5.3}$$

Where:

- n_1, n_2: Number of work-items per work-group.

When $n_1 = 1$, the formular 5.3 becomes $O(n_2)$, which is a linear speedup.

5.4 Experiment

5.4.1 Experimental settings

We use a computer installed with an FPGA and a chip from Intel to conduct experiments.

The execution time of the algorithm running on Device (FPGA) is measured by the profiling API called *clGetEventProfilingInfo*, which is provided along with the acceleration stack from Intel. To calculate the execution time, the starting time and finishing time have to be measured; two tags used are *CL_PROFILING_ COMMAND_START* and *CL_PROFILING_COMMAND_END*, respectively.

The built-in function in C measures the execution time of the algorithm running on Host (CPU) called *clock_gettime* with the tag *CLOCK_MONOTONIC*, this function measures the CPU time when running the algorithm.

The detail information of experimental computer is shown in below:

- FPGA: Intel PAC Arria 10 (DK-ACB-10AX1152AES), max clock frequency: 1000 MHz
- CPU: Intel Core i7-9700K, max clock frequency: 3.6GHz
- Operating system: CentOS 7
- Language/Framework: C, OpenCL 1.0 by Intel (Version 17.1.1)

The detailed information of Intel FPGA Arria 10 is shown in Table 5.3

Parameter name	**Value**
Device name	PAC Arria 10 Platform
Vendor	Intel Corp
Driver version	17.1
Type	Accelerator device
Max compute units	1
Max work-item dimentions	3
Max work-item size	2147483647 / 2147483647 / 2147483647
Max work-group size	2147483647
Max clock frequency	1000 MHz
Address bits	64
Max memory allocated size	8191 MByte
Global memory size	8192 MByte
Error correction support	no
Local memory type	local
Local memory size	16 KByte
Max constant buffer size	2097152 KByte
Queue properties	Profiling enable
Image support	1
Max read image arguments	128
Max write image arguments	128

Table 5.3: OpenCL supports on Intel PAC Arria 10

5.4.2 Experiment design

We design some experiments in order to know the speedup in throughput of our implementation. The main question we intend to answer is how the speedup of OpenCL can give. We have two implementations, which are deeply described in the previous chapter: Single-task kernel & ND-range kernel.

For the implementation in ND-range kernel style, the experiment has two processes:

- **First process**: Conduct an experiment with one work-group per kernel execution and keep changing the size of the work-group (number of work-items per work-group) to find appropriate work-group size parameter in order to achieve the highest throughput.
- **Second process**: Use the work-group size parameter from the previous step and keep changing the number of work-groups per kernel execution until achieving the highest throughput. Our implementation assigns one song to one work-group, then the number of work-groups per kernel execution is equal to the number of songs per kernel execution.

Throughput is calculated by following formula:

$$Throughput \text{ (Gbps)} = n * b * s * \frac{1,000}{e} \tag{5.4}$$

Where:

- n: Number of songs (or work-groups) per kernel execution
- b: Number of bits per wave sample
- s: Number of samples
- e: Execution time (ms)

In our experiment, the number of songs per kernel execution is defined on the host side and changed many times to find the best parameter for the highest throughput. In the previous study, Araki used 16 bits per wave sample, and the number of samples is 131,072. To be fair with the previous work, we use the number of bits per wave samples, and the number of samples per execution is the same as they used. The execution time is retrieved from the provider's API and measured in milliseconds.

To find an appropriate number of work-items per work-group

In the first process of our experiment design, we process one song per kernel execution and keep increasing the number of work-items per work-group to find the appropriate number to give the best throughput. Figure 5.1 shows the throughput changes when changing the number of work-items per work-group. The throughput of the system increases fast when we set the number

of work-items in range 1 to 64, and archives peak when we set the number to 512. Then the appropriate number of work-items per work-group is 512. We will use this value for the second process.

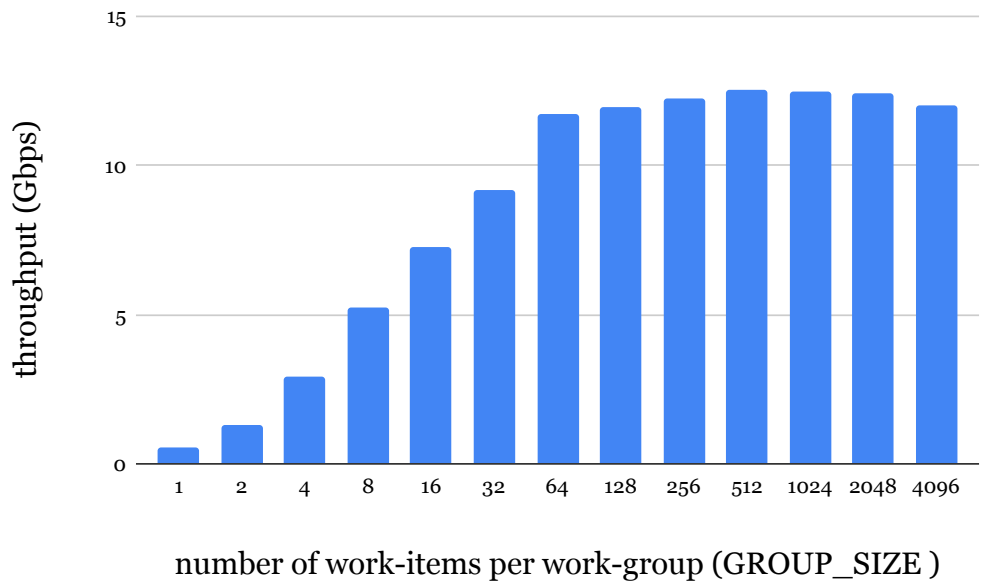

Figure 5.1: Throughput and work-group size when conducting experiment with one song per kernel execution. Maximum throughput is archieved when the number of work-items per work-group is 512

To find an appropriate number of work-groups per kernel execution

In the second process, we use the number of work-items per work-group found in the first process, 512. The algorithm 3 is used for this experiment.

Unlike the number of work-items defined by the kernel's code, the number of work-groups is defined in the host's code and changes anytime after the kernel's compilation. We keep changing the number of work-groups utils the throughput is maximized. The result can be seen in figure 5.2. The number of work-groups per kernel execution leads to the highest throughput of 50, then the appropriate number we choose is 50.

5.4.3 Total-time Decomposition

Our implementation's total time and component time can be viewed in table 5.4. The transfer time oriented to the device is bigger than the host due to

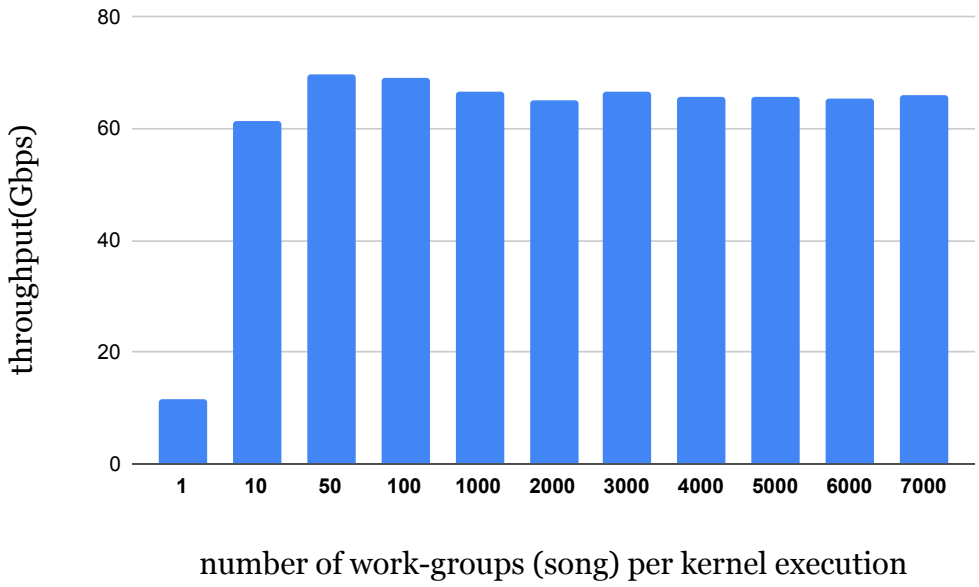

Figure 5.2: Throughput and number of work-groups (songs) per kernel execution when conducting experiment with the number of work-items per work-group is 512. Maximum throughput is archieved when the number of work-groups (songs) per kernel execution is 50

the difference in input data and the output data of kernel execution. The input data is 131,072 wave samples (16 bits per sample), and the output is 4096 FPID frames (16 bits per frame). The execution time is the time for kernel execution in the device. The total time is measure by the host side using function *clockgettime* with tag *CLOCK_MONOTONIC*. The other time is retrieved by calculating the margin between time measured by host and time measured by device, which is: $other = total - transfer\ to\ Host - transfer\ to\ Device - execution$.

Method	**transfer to Device (ms)**	**transfer to Host (ms)**	**execution (ms)**	**other (ms)**	**total (ms)**
Single-task kernel	0.069	0.0080	**0.1196**	0.016	0.21
ND-range kernel (512 work-items, 1 work-group)	0.068	0.0078	**0.1670**	0.015	0.26
ND-range kernel (512 work-items, 50 work-groups)	2.379	0.095	**1.505**	0.037	4.016

Table 5.4: Total-time decomposition of our implementation

5.4.4 Resource Usage

Table 5.5 shows the resource usage for our implementations. HiFP2.0 algorithm uses only integer operators; then, it is not needed to use any digital signal processing (DSP) blocks.

5.5 Speed up by OpenCL ND-range kernel

Figure 5.3 shows the speedup comparison between our experiment and theory when changing the number of work-items per work-group in ND-range kernel implementation. The expected speedup line (red) is regressed on the first-7 data points. Experimental speedup stops increasing when the number of work-items is set on 64. Overhead arises because shared memory is used for communication and synchronization among processing elements.

Method	ALUTs (%)	FFs (%)	RAMs (%)	DSPs (%)	Frequency (MHz)
Single-task kernel	17	16	20	0	156-312
ND-range kernel (512 work-items, 1 work-group)	17	16	21	0	156-312
ND-range kernel (512 work-items, 50 work-groups)	17	17	25	0	156-312

Table 5.5: Resource usage for our implementation
Note: ALUT: Adaptive Look-Up Table, FF: Flip-Flop, RAM: Random-Access Memory, DSP: Digital Signal Processing

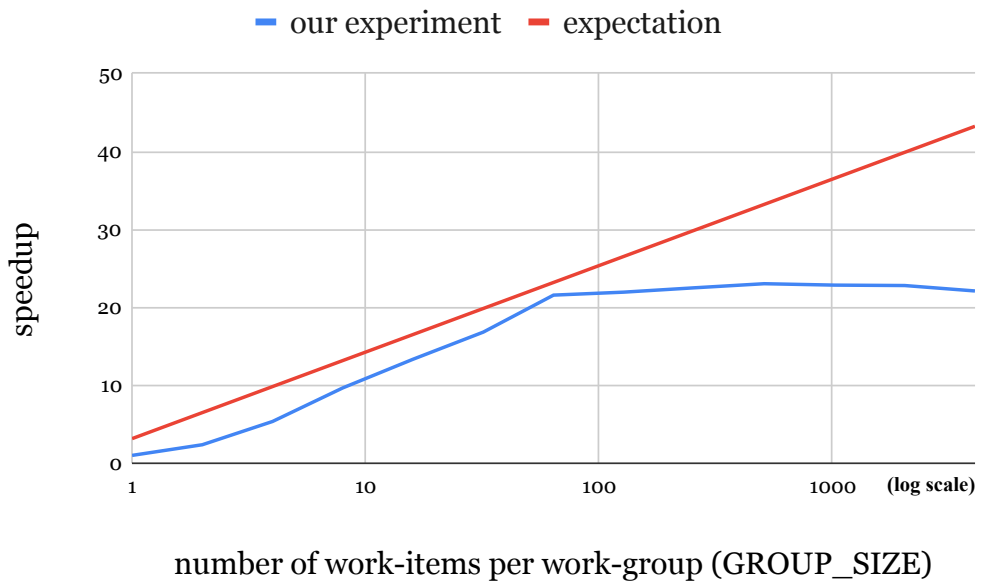

Figure 5.3: Speedup comparison between our experiment and theory. Expected speedup line (red) is regressed on the first-7 data points. Experimental speedup stops increasing when number of work-items is set on 64

5.6 Performance comparison with previous work

We implement the original HiFP2.0 algorithm, which is designed purposely for CPU. In unoptimized implementations, single-task kernel implementation and ND-range kernel implementation (512 work-items, 1 work-group) have lower throughput than the original algorithm by 3.38-5.14 times. The OpenCL implementation's throughput is improved when advanced techniques are applied, which is multiple songs implementation. The throughput of improved implementation is higher than origin by 17.4%.

Method	**Throughput (Gbps)**
C (original algorithm)	59.32
Single-task kernel	17.53
ND-range kernel (512 work-items, 1 work-groups)	11.52
ND-range kernel (512 work-items, 50 work-groups)	69.67

Table 5.6: Throughput comparison among ours and previous work

5.7 Chapter Summary

In this chapter, we discussed our implementations and their performance. To be conclusion, we summarize our results by following:

- By using OpenCL instead of VHDL, the number of code lines can be reduced from 5 to 5.54 times.
- For our implementation, the best parameters led to the highest throughput are 512 work-items per work-group, 50 work-groups per kernel execution.
- The optimized algorithm of OpenCL can achieve more 17.4% performance than the origin, which is intensionally designed for CPUs.

Chapter 6

Conclusion, Limitation & Future work

6.1 Conclusion

In this study, we present the implementation of the HiFP2.0 algorithm on Intel Arria 10 FPGA using OpenCL as a high-level synthesis. Two programming models discussed are Single task kernel and ND-range kernel. Each implementation of the two models has advantages and disadvantages. The decision about which the best kernel to be used is not apparent and requires studying optimization and measurement steps.

To summarize our results, some main points are listed as follows:

- High-level synthesis technology, especially OpenCL, improves productivity of FPGA program development. The number of code lines can be reduced 5 to 5.54 times by using OpenCL instead of VHDL - a register-transfer level language.
- For our implementation, the best parameters led to the highest throughput are 512 work-items per work-group, 50 work-groups per kernel execution.
- The optimized algorithm of OpenCL can achieve more 17.4% performance than the origin, which is intensionally designed for CPUs.

In our experience, to develop an OpenCL kernel efficiently, some appropriate actions can be made are:

- Avoid using more than one kernel, which may lead to low performance: In our first approach to use ND-range kernel, to synchronize two processes of HiFP2.0, two kernels were made. The problem with this

method is the global memory is hit many times. Due to global memory is the lowest memory tier in the OpenCL device, the execution time increases; it is also discussed in our previous work [13]. The problem may be improved by using advanced optimization techniques like pipes/channels between kernels. Liu et al. proposed a source-to-source compiler framework, mainly focused on optimizing the OpenCL application used multiple kernels [11].

- Try to use more than one work-group per kernel execution: An OpenCL device contains many compute units responsible for executing work-groups concurrently and dependently. Depending on the hardware resource, there is a limited number of concurrent compute units executing one kernel call. By dividing a problem to be executed in many work-groups, the program may use full hardware resources.

- Find the best kernel led to the highest performance by "try and error": By using high-level synthesis, software engineer treats compiler as a black box. It is uncertain about the best way to achieve the highest performance from an implementation. The speedup of an implementation always has limitations because of overhead. It is also told by other researchers in [15]. The solution is to share data with other devices.

The future of heterogeneous computing is clear. A program may contain both task-parallel and data-parallel code. The execution of the program can be shared among many devices, based on their ability and availability.

6.2 Limitation

In the time developing OpenCL application, we realize some limitations of OpenCL and our implementations, which may be deeply thought before used:

- For OpenCL application's development, the compile time is very long. Although the functional verification has been speeding up by simulation tools, which is consumed few seconds for compiling, the performance measurement has to be executed in a real device that costs some hours. In our experience, the time costs for compiling kernels usually be 2-3 hours. The time for compiling so long leads to lower productivity of FPGA development compared to other programming models. In CUDA, the time cost for compiling is measured in seconds.

- For our implementation of HiFP2.0, the data transfer is huge and almost not for processing. HiFP2.0 exploits each 4-adjacent wave samples into orientations - or the FPID frames. Because the comparison

is between the first wave sample and the fourth wave sample, it is not needed to use three-wave samples in the middle. By transferring all wave samples of a song, the transfer time increases wastefully.

6.3 Future work

The work for improving an algorithm is huge that impossibly be done in a short time. Some future works listed below are produced from our view and the problems we have:

- To reduce the transfer time: In the first process of our implementation, instead of loading all 131,072 wave samples to memory, we need to load the first wave sample for each four-wave samples. This solution may reduce the number of wave samples by 75%, hence increase the time for pre-processing and transfer time. One disadvantage of this solution is that the loading process needs logical instructions executed by CPU, then the overall execution time may not increase too much. Some parallel loading methods may be investigated to tackle these problems.

- To examine the performance of single-task kernel implementation: In this study, we use single-task kernel implementation for comparison with VHDL implementation in terms of the number of code lines. It is not sure about the performance of single-task kernel compared to ND-range kernel processing multiple songs in one kernel call.

- To compare the performance, device utilization of VHDL and OpenCL executed on an FPGA device: In this study, we implement the core part of HiFP2.0 in VHDL and verify it by a simulation software. To have a bigger picture of comparison between High-level synthesis and Register-transfer level, we should investigate how well a VHDL implementation performs on an FPGA device.

Research Achievements

Conference Proceedings

- M. T. Nguyen, R. Kawano, and Y. Inoguchi. An approach to use high-level synthesis on HiFP2.0: ND-range kernel & Single-task kernel. In *Joint conference of Hokuriku chapters of Electrical and information Societies 2020*, 2020.

Awards

- Excellent presentation award at *Joint conference of Hokuriku chapters of Electrical and information Societies 2020* by IEICE Hokuriku.

Bibliography

[1] K. Araki, Y. Sato, V. K. Jain, and Y. Inoguchi. Performance evaluation of audio fingerprint generation using haar wavelet transform. In *Proceedings of the International Workshop on Nonlinear Circuits, Communications and Signal Processing, Tianjin, China*, pages 1–3, 2011.

[2] P. Cano, E. Batle, T. Kalker, and J. Haitsma. A review of algorithms for audio fingerprinting. In *2002 IEEE Workshop on Multimedia Signal Processing.*, pages 169–173. IEEE, 2002.

[3] P. Cano, E. Batlle, T. Kalker, and J. Haitsma. A review of audio fingerprinting. *Journal of VLSI signal processing systems for signal, image and video technology*, 41(3):271–284, 2005.

[4] V. Chandrasekhar, M. Sharifi, and D. Ross. Survey and evaluation of audio fingerprinting schemes for mobile query-by-example applications. In *12th International Society for Music Information Retrieval Conference (ISMIR)*, 2011.

[5] C. Fäerber, R. Schwemmer, J. Machen, and N. Neufeld. Particle identification on an fpga accelerated compute platform for the lhcb upgrade. *IEEE Transactions on Nuclear Science*, 64(7):1994–1999, 2017.

[6] J. Haitsma and T. Kalker. A highly robust audio fingerprinting system. In *Ismir*, volume 2002, pages 107–115, 2002.

[7] J. Haitsma, T. Kalker, and J. Oostveen. Robust audio hashing for content identification. In *International Workshop on Content-Based Multimedia Indexing*, volume 4, pages 117–124. Citeseer, 2001.

[8] K. Hill, S. Craciun, A. George, and H. Lam. Comparative analysis of opencl vs. hdl with image-processing kernels on stratix-v fpga. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 189–193, 2015.

[9] F. Holm and W. T. Hicken. Audio fingerprinting system and method, Mar. 14 2006. US Patent 7,013,301.

[10] D. H. Jones, A. Powell, C. Bouganis, and P. Y. K. Cheung. Gpu versus fpga for high productivity computing. In *2010 International Conference on Field Programmable Logic and Applications*, pages 119–124, 2010.

[11] J. Liu, A.-A. Kafi, X. Shen, and H. Zhou. Mkpipe: A compiler framework for optimizing multi-kernel workloads in opencl for fpga. *arXiv preprint arXiv:2002.01614*, 2020.

[12] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler. Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pages 1–6, 2017.

[13] M. T. Nguyen, R. Kawano, and Y. Inoguchi. An approach to use high-level synthesis on hifp 2.0: Nd-range kernel & single-task kernel. In *Joint conference of Hokuriku chapters of Electrical and information Societies 2020*, 2020.

[14] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, et al. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 5–14, 2017.

[15] K. Shata, M. K. Elteir, and A. A. El-Zoghabi. Optimized implementation of opencl kernels on fpgas. *Journal of Systems Architecture*, 97:491–505, 2019.

[16] C. B. Weare. System and method for audio fingerprinting, Nov. 8 2005. US Patent 6,963,975.

[17] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. Evaluating and optimizing opencl kernels for high performance computing with fpgas. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 409–420, 2016.

Listing 6.1: dwt.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  package dwt_pkg is
6      type natural_array is array(natural range <>) of
          natural range 0 to 65535;
7  end package;
8
9  ------------------------------------------------
10
11 library ieee;
12 use ieee.std_logic_1164.all;
13 use ieee.numeric_std.all;
14 use work.dwt_pkg.all;
15
16 ENTITY dwt IS
17     PORT (
18         wave: IN natural_array( 0 to 7 );
19         dwt_wave: OUT natural
20     );
21 END ENTITY;
22
23 ARCHITECTURE rtl OF dwt IS
24     signal temp_dwt_1: natural_array( 0 to 3 );
25     signal temp_dwt_2: natural_array( 0 to 1 );
26     signal temp_dwt_3: natural;
27 BEGIN
28     temp_dwt_1(0) <= (wave(0) + wave(1)) / 2;
29     temp_dwt_1(1) <= (wave(2) + wave(3)) / 2;
30     temp_dwt_1(2) <= (wave(4) + wave(5)) / 2;
31     temp_dwt_1(3) <= (wave(6) + wave(7)) / 2;
32
33     temp_dwt_2(0) <= (temp_dwt_1(0) + temp_dwt_1(1)) /
          2;
34     temp_dwt_2(1) <= (temp_dwt_1(2) + temp_dwt_1(3)) /
          2;
35
36     temp_dwt_3 <= (temp_dwt_2(0) + temp_dwt_2(1)) / 2;
37
38     dwt_wave <= temp_dwt_3;
39 END ARCHITECTURE;
```

Listing 6.2: dwt_all.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.dwt_pkg.all;
5
6  entity dwt_all is
7      generic (num_of_fpid_frames : positive := 4096);
8
9      port (
10         wave_all: in natural_array( 0 to
               num_of_fpid_frames*32-1 );
11         dwt_wave_all: out natural_array( 0 to
               num_of_fpid_frames-1 )
12     );
13 end entity;
14
15 architecture rtl of dwt_all is
16     signal tmp_wave_all: natural_array( 0 to
           num_of_fpid_frames*32-1 );
17     signal tmp_dwt_all: natural_array( 0 to
           num_of_fpid_frames-1 );
18
19     component dwt is
20         port (
21             wave: in natural_array;
22             dwt_wave: out natural
23         );
24     end component dwt;
25 begin
26     tmp_wave_all <= wave_all;
27
28     gen_dwt: for i in 0 to num_of_fpid_frames-1 generate
29         dwt_en: dwt
30         port map (
31             tmp_wave_all( i*32 to i*32+7 ),
32             tmp_dwt_all(i)
33         );
34     end generate;
35
36     dwt_wave_all <= tmp_dwt_all;
37 end architecture;
```

Listing 6.3: feature_extraction.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.dwt_pkg.all;
5
6  entity feature_extraction is
7      generic (num_of_fpid_frames: positive := 4096);
8
9      port (
10         dwt_wave_all: in natural_array(0 to
               num_of_fpid_frames-1);
11         fpid_all: out natural_array(0 to
               num_of_fpid_frames-1)
12     );
13 end entity;
14
15 architecture rtl of feature_extraction is
16 begin
17     fpid_all(num_of_fpid_frames-1) <= 0;
18
19     gen: for i in 0 to num_of_fpid_frames-2 generate
20         fpid_all(i) <= 1 when (dwt_wave_all(i) >
               dwt_wave_all(i+1)) else 0;
21     end generate;
22 end architecture;
```

Listing 6.4: hifp.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.dwt_pkg.all;
5
6  entity hifp is
7      generic (num_of_fpid_frames: positive := 4096);
8
9      port (
10         wave_all: in natural_array(0 to
               num_of_fpid_frames*32-1);
11         wave_dwt_all: out natural_array(0 to
               num_of_fpid_frames-1);
12         fpid_all: out natural_array(0 to
               num_of_fpid_frames-1)
13     );
14 end entity;
15
16 architecture rtl of hifp is
17     signal temp_dwt_all: natural_array(0 to
           num_of_fpid_frames-1);
18
19     component dwt_all is
20         generic (num_of_fpid_frames : positive);
21
22         port (
23             wave_all: in natural_array( 0 to
                   num_of_fpid_frames*32-1 );
24             dwt_wave_all: out natural_array( 0 to
                   num_of_fpid_frames-1 )
25         );
26     end component;
27
28     component feature_extraction is
29         generic (num_of_fpid_frames: positive);
30
31         port (
32             dwt_wave_all: in natural_array(0 to
                   num_of_fpid_frames-1);
33             fpid_all: out natural_array(0 to
                   num_of_fpid_frames-1)
34         );
```

```
35      end component;
36  begin
37      en_dwt_all: dwt_all
38      generic map(num_of_fpid_frames => num_of_fpid_frames
            )
39      port map(
40          wave_all,
41          temp_dwt_all
42      );
43
44      en_feature_extraction: feature_extraction
45      generic map(num_of_fpid_frames => num_of_fpid_frames
            )
46      port map(
47          temp_dwt_all,
48          fpid_all
49      );
50
51      wave_dwt_all <= temp_dwt_all;
52  end architecture;
```

Listing 6.5: single_task_hifp.cl

```
1  __kernel void generate_fpid(
2      __global const short int * restrict wave,
3      __global short int * restrict fpid
4  ) {
5      short int dwt_wave[4097];
6
7      // dwt
8      for (int i=0; i<4096; i++) {
9          short int wave_tmp[8];
10
11         for (int j=0; j<8; j++) {
12             wave_tmp[j] = wave[i*32 + j];
13         }
14
15         for (int k=8; k>1; k/=2) {
16             for (int l=0; l<k/2; l++) {
17                 wave_tmp[l] = (wave_tmp[l*2] + wave_tmp[
                       l*2 + 1]) / 2;
18             }
19         }
20
21         dwt_wave[i] = wave_tmp[0];
22     }
23
24     // feature extraction
25     for (int i=0; i<4096; i++) {
26         if (dwt_wave[i] > dwt_wave[i+1]) {
27             fpid[i] = 1;
28         }
29     }
30 }
```

Listings