NANYANG TECHNOLOGICAL UNIVERSITY

# NANYANG TECHNOLOGICAL UNIVERSITY

## IMAGE PROCESSING ON A HETEROGENEOUS COMPUTING PLATFORM

by

SYAM UMA
(G1501681F)

A Dissertation Submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Embedded Systems

Supervised by

Assoc. Prof. Douglas L. Maskell

July 2016

# Contents

# List of Figures

# List of Tables

# Abbreviations

**ACP** Accelerator Coherency Port

**AFI** AXI FIFO Interface

**API** Application Programming Interface

**CU** Compute Unit

**FPGA** Field Programmable Gate Array

**GPIO** General-Purpose Inputs and Outputs

**GPU** Graphics Processing Unit

**HDL** Hardware Description Language

**HLS** High Level Synthesis

**HP** High Performance

**II** Initiation Interval

**OpenCL** Open Computing Language

**OpenMP** Open Multi-Processing

**OS** Operating system

**PE** Processing Element

**PL** Programmable Logic

**PPE** Power Processing Element

**PS** Processing system

**SDSoC** Software Defined System on Chip

**SPE** Synergistic Processing Element

**Abstract**

The strong need for increased computational performance and energy efficiency has led to the use of heterogeneous computing platforms, with graphics processing units (GPUs), massively parallel processor arrays(MPPAs) and other accelerators acting as co-processors for arithmetic-intensive data-parallel workloads. The use of accelerator architectures in heterogeneous computing platforms offers a promising path towards improved performance and energy efficiency. One class of solution includes programmable accelerators such as GPUs and MPPAs. Another class of solution dedicates highly efficient custom-designed application-specific accelerator for computing tasks. This solution used to be preferable due to area, speed and energy efficiency and were deployed as an Application Specific Integrated Circuit (ASIC) block alongside a general purpose processor (GPP). However, developing dedicated ASIC accelerators has become less practical due to the long turnaround time and high cost associated with ASIC development. Field Programmable Gate Arrays (FPGAs), which allow the implementation to be modified post-deployment, are now more commonly used for rapid-prototyping of application specific accelerators in heterogeneous computing platforms. FPGA vendors are introducing tools and programming systems to lower the barriers to entry for software development for their Heterogeneous computing platforms hosting FPGA fabrics. One example is Xilinx Zynq Platform (consisting a GPP and an FPGA fabric) and corresponding SDSoC tool flow which helps to offload compute intensive functions of a C/C++ application to the FPGA fabric in the form of an application specific accelerator in a transparent manner. It also abstracts the runtime management of the accelerator, including data communication to and from the accelerator in the form of software APIs. In this report, we first explore SDSoC tool flow and evaluate the performance of the accelerator created by SDSoC using a series of experiments. The focus is on finding best communication technique between software and hardware and test the impact of the *optimization pragmas* on the system performance. We demonstrate the effective of the SDSoC tool using image processing application.

Next, we move on to a platform portable programming model, OpenCL. Since most high-level languages, like C/C++, are sequential programming languages with no standardized means to describe parallel execution and OpenCL can bridge the gap between the expressiveness of sequential languages and the parallel capabilities of the hardware, we describe image processing applications using OpenCL programming model with the aim of accelerating OpenCL kernels using different types of accelerators without the change in the application source code. OpenCL has the benefit of being portable across architectures, such as FPGAs, GPUs, and other parallel computing resources without changes to algorithm source code.This is a key capability of OpenCL that makes it a promising programming model for heterogeneous platforms. We use the POCL infrastructure on Xilinx Zynq to support OpenCL application execution. We implement OpenCL version and C version of basic gray level transformations used in most of the image processing applications and perform a set of experiments to quantify the overhead of using OpenCL (a portable programming model). In future, we aim to accelerate OpenCL kernels using FPGA based accelerators on Xilinx Zynq platform.

# Acknowledgment

I would first like to thank Assoc. Prof. Dr. Douglas Leslie Maskell for giving me the opportunity for working on this project. I express my sincere appreciation for his support, guidance and encouragement without which this dissertation woud not have been possible.

My deepest gratitude to Abhishek Kumar Jain for his technical guidance and constant monitoring and steering me in the right direction whenever needed. I thank him for his motivation and enthusiasm, which helped me through roadblocks during the course of the dissertation.

Thanks to Mr. Jeremiah Chua in Hardware and Embedded Systems Lab(HESL) for all the facilities and technical support.

I would also like to thank my fellow classmates Rathi Chetan and Ravi Prashant for all their help and support at all times.

Last but not the least, I thank my parents for believing in me and being my pillars of strength during my Masters.

# Chapter 1

# Introduction

## 1.1 Motivation

In 1965, Gordon Moore made a prediction that would set the pace for a modern digital revolution. Moore stated that the number of transistors on an affordable CPU would double every two years. But in the recent years, this exponentially increasing curve is heading to saturation as shown in Fig.1.1. This is mainly because transistors are getting too small to be manufactured efficiently[1].

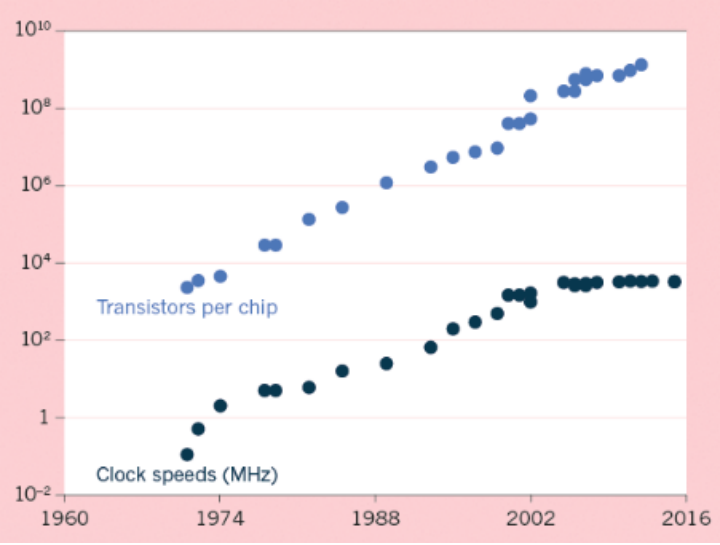



Figure 1.1: Saturation of Moore's Law [2]

In such a scenario, heterogeneous platforms are a potential solution to continue in the direction of energy efficient solutions. The processor capability has been brought down due to limitations in the physical size urging to bring in the use of numerous processors in parallel. The demanding applications, of course, play a major role in driving the need for high performance and power efficient systems. Processing from massive amounts of multiple sensors requires parallel computing. It can be used for video processing applications, audio processing applications like voice recognition, translation of live audio, gauging acoustics in a room and can be extended into security applications. We see that most of these applications require real-time processing and have rigid power constraints. The shortcomings of current hardware architecture and software programs in implementing such algorithms steer us toward heterogeneous platforms and programming.

Venturing into new fields of technology requires that we understand the problem being addressed and the challenges and trade-offs in the solution. One of the first problems is that of power, whose reduction is of increasing priority in all computing segments. There is a demand for improved battery life in gadgets while data centre power requirements and cost of cooling continue to rise. Amidst all this there is a need for constantly improving performance. The need of the hour is an approach that delivers improvement across all domains : power, performance, programmability and portability.

As a solution to this demand, Graphics Processing Unit (GPU) and Field Programmable Gate Array (FPGA) has been introduced into the world of general computing. The heterogeneous system using CPU and GPU is quite common now. FPGAs have also been integrated into heterogeneous systems to achieve speed up greater than GPU. However, it is known that these varied devices require their own programming models. For example, GPU uses CUDA and FPGAs require RTL description. This implies programmers have to be either an expert in all programming models or they should be grouped so as to master only one of these. Either way, it is not resourceful. What we need is to introduce abstractions and new models that help bridge the gap between hardware and software developers.

FPGA vendors are introducing tools and programming systems to lower the barriers to entry for software development for their Heterogeneous computing platforms hosting FPGA fabrics. One example is Xilinx Zynq Platform (consisting a GPP and an FPGA fabric) and corresponding Software Defined System on Chip (SDSoC) tool flow which helps to offload compute intensive functions of a C/C++ application to the FPGA fabric in the form of an application specific accelerator in a transparent manner. In this report, we first explore SDSoC tool flow and evaluate the performance of the accelerator created by SDSoC using a series of experiments. The focus is on finding best communication technique between software and hardware and test the impact of the *optimization pragmas* on the system performance. We demonstrate the effective of the SDSoC tool using image processing application.

## 1.2 Contribution

In our work, we explore SDSoC and Open Computing Language (OpenCL). The contributions include :

***SDSoC:***

- Analyzing performance of hardware-software communication using SDSoC pragmas.
- Determining the most efficient communication pragma, especially for streaming kernels.
- Quantifying the performance gain of using SDSoC for accelerating image processing kernels.

***OpenCL:***

- Understanding the OpenCL programming model to develop efficient host and kernel codes.
- Comparison of performance of OpenCL kernels with respect to C implementation of image processing kernels.
- Quantifying the performance difference between OpenCL and C implementation of image processing kernels.

## 1.3 Organization

The remainder of the report is organized as follows: Chapter 2 gives the background information required to understand the problem being addressed and the solution presented. Chapter 3 talks about one of the SoC programming tools by Xilinx for CPU-FPGA platforms, SDSoC. In chapter 4, we implement OpenCL version and C version of basic gray level transformations used in most of the image processing applications and perform a set of experiments to quantify the overhead of using OpenCL (a portable programming model). We conclude in chapter 5 and discuss future work.

# Chapter 2

# Background

## 2.1 Heterogenous Computing Platforms

Heterogeneous computing platforms refer to systems that use more than one kind of processor [3]. It involves not just multi-core processors but various types of specialised processing units aiding in increasing performance [4]. For example, they typically use CPU and GPU, usually on the same silicon die with the intention to exploit the advantages of both the processor types, GPU for its graphics rendering and also the mathematically intensive computations on very large data sets, and CPUs to run the operating system and perform traditional serial tasks [4]. The rise of the need for heterogeneous platforms can be understood by studying the changing trends over the last few years. There was a shift from single-core to multi-core processors by the end of 2010. Not just dual cores, even quad core processors were becoming mainstream and affordable [5]. Yet there were challenges presented by multi-core processing too. The processor size and power consumption were on the rise to accommodate for the cache memory and extra cores needed for instruction pipelines [4].

Meanwhile, GPUs, which were turning more complex and sophisticated, underwent interesting developments that were a result of advances in semiconductor technology. GPUs with their vector processing capabilities were able to realize parallel operations on very large sets of data at much lower power consumption when compared to similar processing on CPUs [6]. Though GPUs were initially built to help

with graphics processing, they became increasingly attractive for more general purposes, such as addressing data parallel programming tasks [7].

Soon the world of computing realised the potential in combining the best of CPUs and GPUs to achieve faster and more powerful designs. And the need for such a heterogeneous system was further driven by restraint on power and scalability in multi-core CPU development and the promising new vector architecture of GPUs. Vector processors have up to thousands of individual compute cores, which can operate simultaneously. This makes GPUs ideally suited for computing tasks that deal with a combination of very large data sets and intensive numerical computation requirements [4].

While the idea of CPU and GPU contributing to the heterogeneous system architecture was gaining popularity, the idea of integrating custom logic (application specific accelerator) into the architecture emerged. While it is known that customised circuit for any application gives the highest performance, it is not flexible. This shifted the attention to FPGA which can be reconfigured based on the application while outperforming any general purpose processor. And adding such a component to a heterogeneous computing platform, seemed a promising venture.

But of course, vector processing using GPU or accelerators using FPGA is not the answer all the time. There is always an overhead associated with setting up vector processing or data transfer to FPGA, which can easily outweigh the time saved. This behavior becomes evident when vector processing or FPGA acceleration is used on small datasets. Hence there still are problems for which CPUs scalar approach would be the best. This is all the more a reason to retain both CPU, GPU and FPGA (if needed) and harness all their features that are advantageous to us instead of using just one device which cannot guarantee best performance under all situations.

## 2.2 Programming Models

When we talk about systems comprising of CPUs, GPUs and probably even an FPGA, then the next big question is how do we program them to work with each other in synchronization so as to attain the performance gain we are expecting. We discuss some of the popular programming models as follows:

### 2.2.1 OpenMP

Initially, when CPUs moved from single cores to multi-cores with multi-thread handling capabilities, increased performance was a guarantee. Utilizing these features competently was, of course, the condition to be fulfilled to achieve expected performance and that implied using parallel programming. With an aim to make the task of programming simple, the Open Multi-Processing (OpenMP) model was introduced.

OpenMP is an Application Programming Interface (API) that can be considered as an implementation of multi-threading [8]. The run-time environment allocates threads to different processors and the threads run synchronously. OpenMP uses compiler *pragma* to control the program flow and in case the pragmas are not supported, the program will still behave correctly, but without any parallelism. The default way lets each thread execute independently and tasks can be divided among threads using work-sharing constructs. This helps in achieving both task parallelism and data parallelism using OpenMP. It is a portable, scalable model that is a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer [8].

### 2.2.2 CUDA

When GPUs were introduced, they were designed as graphic accelerators. Soon their potential for general purpose parallel computing was uncovered. But GPU programming required using assembly or graphics programming languages like OpenGL, which was not a trivial task. In order to overcome the difficulty of programming, NVIDIA unveiled the CUDA programming model. CUDA enabled the GPU to be programmed

using C, C++ and Fortran [9]. CUDA model uses a host code and a device code which is the compute kernel. The serial code is executed on the host and parallel code on the device. The host code initializes and offloads computations to device code. Decisions regarding thread block size and number of threads per block are done in host side. CUDA provides APIs that help in efficient task and memory management in order to harness maximum power from the GPU. But using CUDA meant that the same code cannot be ported to CPU and one has to re-write with CPU based optimizations for obtaining good performance.

### 2.2.3 OpenCL

From Sections 2.2.1 and 2.2.2, we see that we cannot port OpenMP optimization on GPU nor can we run a CUDA program on CPU. This called for a common programming model across all platforms, especially from the industries. And initiated by Apple, the OpenCL (Open Computing Language) programming model was born which was developed and standardized by Khronos. OpenCL also uses the concept of a host code and device code. The latter is the main computation logic and remains unchanged over varying devices. However, the advantage of platform portability does not extend to performance portability and minor changes in the host code as per the target device might be required for a powerful implementation. But this is just a minor setback and the OpenCL model is already gaining immense popularity. Fig. 2.1 show few of the industries that currently adopt OpenCL model.

## 2.3 FPGA Accelerators and SDSoC

For many years, programmers depended on advancement in processor technology to automatically speed up applications. But in the past few years, power concerns have caused the processor operating frequencies to stagnate. In such a scenario, FPGA is a promising alternative

FPGA consists of an array of logic gates that can be hardware programmed to create customized compute units as per the application. It is also possible to configure

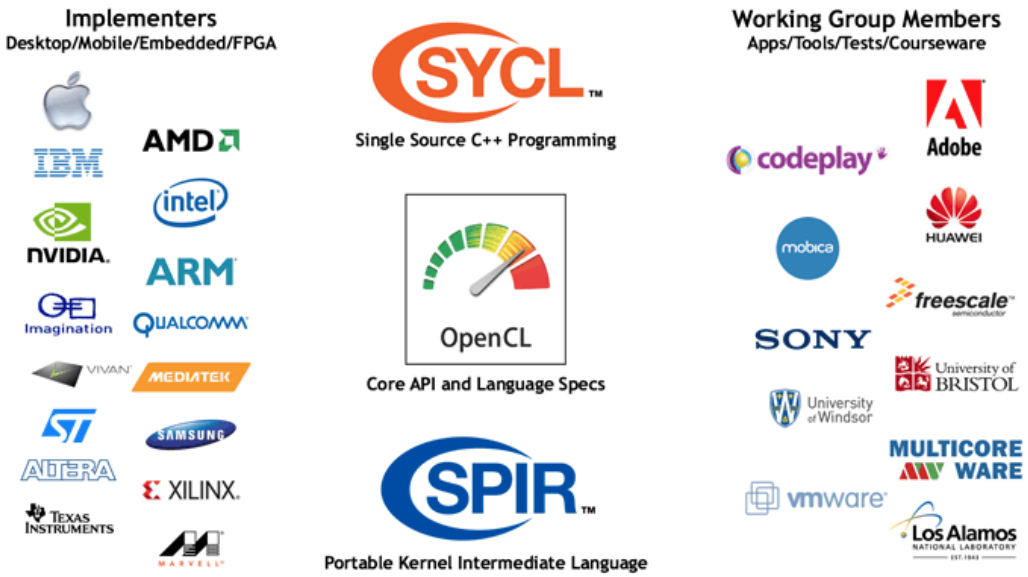



Figure 2.1: Industries using OpenCL

multiple compute units on an FPGA that work in parallel. They are power effective as they operate at low frequencies in the range 100-550 MHz. And they are known to provide up to 100-fold speedups per node over microprocessor-based systems [2].

Tradionally FPGAs are configured by means of a Hardware Description Language (HDL), like VHDL or Verilog. But with a one-to-ten ratio of hardware-software programmers, an HDL based approach for FPGA narrows down their usage as accelerators. Also, the hardware development cycle is much more tedious than software development cycle. In order to overcome these issues, there is a need for raising the programming abstraction to improve the design productivity. SDSoC from Xilinx is one such tool which helps in offloading C implementations to FPGA. Chapter 3 explores SDSoC in detail.

## 2.4 Image Processing

### 2.4.1 Need for Image Processing

Vision forms a very important part of human lives. Every day of our lives we gather information and make decisions based on what we see. Seeing might seem trivial to us, but it is not so. Understanding what we see, distinguishing the features and objects in the world is a complex task involving lot of neural activity. It requires understanding depth, distinguishing foreground from background, recognizing objects presented in a wide range of orientations and many more. If we were to achieve the same using machines, we need to design systems capable of capturing images and algorithms capable of processing the image and retrieving the information we want. Of course one must remember that vision with the speed and accuracy of our brains is difficult to beat as even super computers cannot compete with years of human evolution. But nevertheless, it is possible to achieve high performance in real time, using efficient algorithms and accelerators.

The improvement of pictorial information for human interpretation and processing of a scene data for an autonomous machine perception are two of the principal applications of image processing. Digital image processing therefore has a broad range of applications such as remote sensing, in business applications for storage of data as well as image, in both the medical and forensic sciences, acoustic imaging and industrial automation. Images obtained from satellites can be used to track earth resources, forecasting the weather, geographical mapping and many more applcations requiring wide regions to be surveyed. Space-probe missions return data that are images which require to be analysed to detect objects.In addition to these applications, digital image processing is now being used to solve a wide variety of problems requiring methods capable of enhancing information for human visual interpretation and analysis [10]. The current major area is in solving the problem of machine vision so as to attain good results.

### 2.4.2 Image Enhancement Techniques

The fundamental objective of image enhancement techniques is to process an image so as to enhance it to be a better option than the original for a particular application. Note that the enhancement technique is application specific and what works well for one application need not for another. The two broard categories of enhancement are:

The fundamental objective of image enhancement techniques is to process an image so as to enhance it to be a better option than the original for a particular application. Note that the enhancement technique is application specific and what works well for one application need not for another. The two broad categories of enhancement are:

1. Spatial domain enhancement
2. Frequency domain enhancement

The former techniques refer to processing the image in the image plane (pixels) itself while the latter techniques are based on modifying the transform (Fourier or any other) of an image [10]. Several combinations of both enhancement methods are used in majority of the problems requiring image enhancement. Some examples of enhancement operations are edge enhancement, pseudocoloring, histogram equalization, contrast stretching, noise filtering, un-sharp masking, sharpening, magnifying, etc. These image enhancement operations may be either local or global.

### 2.4.3 Spatial Transformations

We begin with grey level spatial-domain transformations on images. The same can be extended over the Red, Green and Blue components in case of a coloured images. The basic transformation function is given by

$$s = T(r) \tag{2.1}$$

where $r$ is the pixel of the input image and $s$ is the pixel of the output image. $T$ is a transformation function that maps each value of $r$ to each value of $s$ [11] [12]. Fig. 2.2

shows the graphs representing the spatial transforms.The gray level transformations which have been implemented as part of the project are as follows :

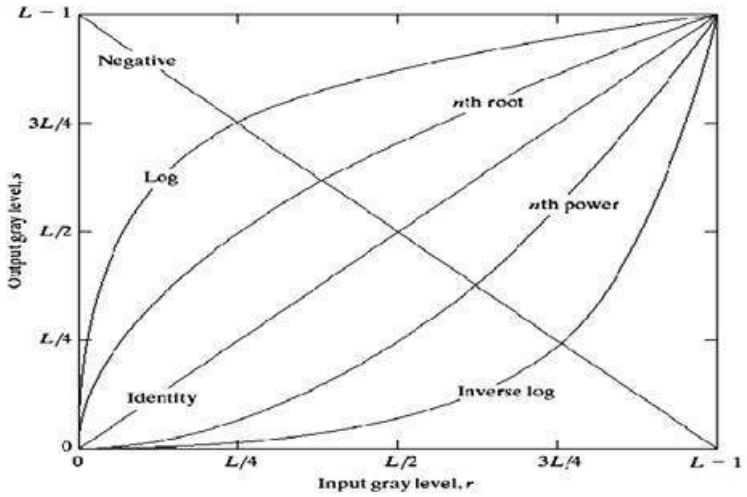



Figure 2.2: Graphical representation of gray level spatial transforms [12]

1. Linear Transform :

   - Identity Transformation: Every input image pixel value is mapped directly to output image which is graphically given by a straight line.
   - Negative Transformation: Each input image pixel value is subtracted from L-1 to produce the corresponding output image pixel value. Here L is the maximum pixel value. The transformation function is given by -

   $$s = (L - 1) - r \tag{2.2}$$

   This results in the lighter pixels becoming dark and the darker picture becoming light and hence an image negative.

2. Power-law Transform : The transformation equation is given by -

   $$s = cr^{\gamma} \tag{2.3}$$

where $\gamma$ is called gamma.

The variation of $\gamma$ value varies the resulting enhanced images. Different display devices / monitors have their own gamma correction, thats why they display their image at different intensity [11] [12]. For example, CRT has a gamma between 1.8 to 2.5, which means the image displayed on CRT is dark. Transforms using convolution

3. Logarithmic Transform : The log transformation is defined by the equation

$$s = c * log(r + 1) \tag{2.4}$$

During log transformation, the dark pixels or lower valued pixels in an image are expanded while the higher pixel values are compressed. This results in evening out the pixel values in general. The value of c in the log transform can be adjusted as per required enhancement.

4. Convolutional Transforms : Convolution can be considered as a local image enhancement techniques using masks to process sub-samples of an image. It helps in achieving effects that the above mentioned grey level transformations cannot achieve, like blurring, smoothening, and sharpening of images etc. Convolution involves a mask or kernel which is slid across the image and at every position the kernel values are multiplied with corresponding image pixel values and summed to obtain a result which will correspond to one pixel of the output image. The general convolution equation is shown below

$$H(x, y) = \sum_{j=1}^{height} \sum_{i=1}^{width} I(i, j)M(x - i, y - j) \tag{2.5}$$

where $I(i, j)$ refers to image pixels and $M(i, j)$ refers to mask pixels.

A mask can be thought of as a filter. Masking is also referred to as spatial filtering. Just varying the mask values yields different filtering techniques making convolution a very powerful tool. The two most common uses of spatial filters are:

- Blurring and noise reduction
- Edge detection and sharpness

### 2.4.4 Spectral Transformations

Unlike in spatial domain that deals directly with pixel values, in the frequency domain, the rate of change of pixel value is what is important. For this purpose we first obtain the frequency distribution of the image.The processing is done on the spectral version and the transformed output is inverted to convert back to image. A signal can be converted from the time domain to the frequency domain using

- Fourier Series
- Fourier transform
- Laplace transform
- Z transform

To be able to decide which transform is apt for an application, refer [11] to learn more.

# Chapter 3

# SDSoC for Programming Heterogeneous Platforms

## 3.1 SDSoC for Zynq

SDSoC is the recent addition to Xilinx SDx development series shown in Fig. 3.1, where "SD" stands for "Software Defined" Since 2014, Xilinx has been introducing a series of SDx development environments like SDNet and SDAccel [13].

SDSoC, the priced new addition, has the ability to create a high-level representation of the system and to then quickly and easily decide which portions are to be implemented in software and which are to be realized in hardware in a way that that does not require software developers to rely on hardware developers. It comprises of a full-system optimizing C/C++ compiler, along with features for system level profiling, automated SW acceleration in programmable logic, automated system connectivity generation, and libraries to speed programming. It also enables developers to rapidly define, integrate, and verify system level solutions [13]. As mentioned earlier, the popularity of SoC is due to the fact that different applications require different architecture for efficient computation. Though processor cores are well-suited for sequential tasks, they are extremely inefficient when it comes to tasks that require massively parallel hardware accelerator cores. Zynq from Xilinx is one such SoC that boasts of a mix of processor cores and programmable logic fabric. It

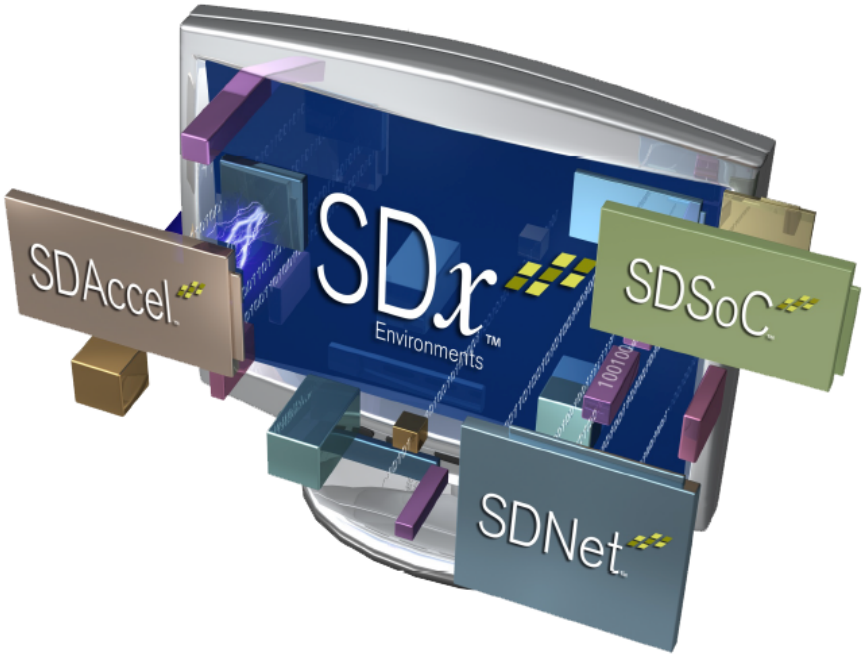



Figure 3.1: SDx Development Seris of Xilinx [13]

also has an on-chip memory, a wide variety of hard core communication and peripheral functions, high-speed data interfaces, and a large number of General-Purpose Inputs and Outputs (GPIO) [14]. The experiments in this report are done on Xilinx Zynq Zedboard. Based on Zynq-7000 SoC operating at 667MHz, it contains dual-core ARM-Cortex A9 based Processing system (PS) and Programmable Logic (PL) fabric in one package. It is composed of Zynq Z7020-clg484 operating at 667 MHz [15]. This new combination of processing subsystems with FPGA fabric command integration between software and hardware engineering, blurring the line between the two.

Software usually is the main contributor in the functionality of most of the system designs. On the hardware side, the FPGA fabric is used for speeding up the computation, which conventionally required RTL representation for not only the function but also interconnect buses and fabrics, memory architecture. This would mean that developing hardware accelerators are reserved for hardware engineers. But SDSoC helps in bridging that gap between software and hardware engineers.

The key contribution of SDSoC would be that it helps in offloading tasks to hardware by just the click of a button. This 'not so trivial' task is made possible with the introduction of High Level Synthesis (HLS) technology. HLS at the back end of SDSoC helps in taking a high-level representation in C/C++ and compiling/synthesizing it into an equivalent RTL representation that can subsequently be used by traditional synthesis technology to generate the ultimate hardware realization [14]. Another impressive (and useful) functionality is an automatic system-level connectivity generator. [14] mentions how this is made possible; by analysing the latency and throughput requirements of each of the communications interface in the design and automatically recommending (and inserting) the optimal type of interconnect. SDSoC also packs the ability to choose to run the application bare-metal or on Linux or FreeRTOS as target Operating system (OS). The inclusion of target OS further simplifies the implementation of Image Processing applications. Last but not the least, the SDSoC profiler can help novice users in deciding which tasks to offload to hardware by identifying the bottlenecks. Thus SDSoC abstracts all the complexities of hardware development and the end user only interacts with the sophisticated software development tools required to implement applications on complex heterogeneous multiprocessing devices.

## 3.2 SDSoC Environment

SDSoC uses an Eclipse based IDE and also supports command line interface for the terminal lovers.At the front end we have compilers and debuggers dedicated to debug embedded software in a parallel heterogeneous multi-processing environment [16]. Based on the target platform and functions to be offloaded onto the hardware, the system compilers (sdscc/sds++) transform C/C++ programs into complete hardware/software systems. To achieve high performance, each hardware function runs as an independent thread [16]. After analysing the dataflow between the software and hardware functions, an application-specific SoC is generated to realize the program. A helpful function in SDSoC is the ability to estimate the performance of an application before offloading it to hardware. The hardware functions are compiled using

the Vivado HLS tool to the PL, and then a complete hardware system based on the selected platform, including DMAs, interconnects, hardware buffers, and other IPs are generated. The Vivado Design Suite tools are then invoked to generate FPGA bitstream. The SDSoC system compilers also generate system-specific software stubs and configuration data, which they compile and link with the application code using a standard GNU toolchain into an application binary [16]. The hardware-specific software configuration codes that are auto-generated by the system compilers manage the data transfers and control the hardware accelerators and they integrate any associated drivers for the IP blocks generated. Since the complete development is from a single source, it is possible to refactor the program so as to iterate over design and architecture changes.

## 3.3 Design flow in SDSoC

To get a better understanding of how SDSoC functions, let us take a look at the design flow showin in Fig. 3.2.

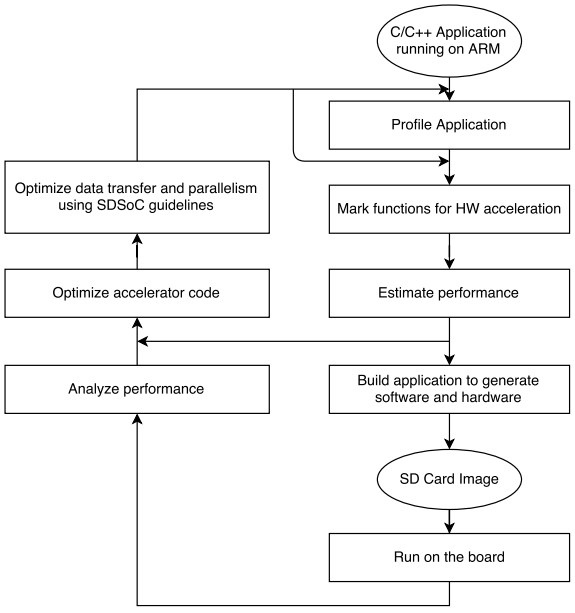



Figure 3.2: Design Flow in SDSoC

The first step is to select a development platform and cross-compile the application for ARM and check for the functional correctness of the appplication. Next, the compute-intensive segments of the code are to be identified so that they can be moved to the programmable logic. One has to note that it is a function that will be offloaded to the hardware, hence it has to be isolated from the rest of the code. In this way, only the required compute function will be accelerated. Next, the SDSoC system compiler is invoked to generate a complete SoC and SD card boot image for the application. The sdscc/sds++ system compiler handles the system generation process. We have the option of optimising the system and hardware functions using pragmas. Once the code is ready, we can estimate the performance using the SDEstimate option. This helps in analysing the speed up that could be achieved by hardware acceleration when compared to software only implementation without actually compiling for hardware. This estimate is based on properties of the generated system and estimates for the hardware functions provided by the IPs when available [16]. The overall design process is iterated until the generated system achieves the performance and cost objectives.

## 3.4 Cross-Compiling for ARM

As mentioned in Section 3.3, the first step is to cross-compile the application code to run on the target platform. Every platform included in the SDSoC environment includes a pre-built SD card image from which we can boot and run cross-compiled application code. When no code is selected to run on hardware, this pre-built image is used. We can always run a software only compile every time we build a new code or make modifications, to check for correctness as the software build is faster. The SDSoC environment includes two distinct toolchains for the ARM CPUs within Zynq architecture devices :

- arm-xilinx-linux-gnueabi - for developing Linux applications
- arm-xilinx-gnueabi - for developing standalone (”bare-metal”) and FreeRTOS applications.

The appropriate tool chain is selected during creation of project and the system compilers handle the invocation of the appropriate compilers. All object code for the ARM CPUs is generated with the GNU toolchains. The sdscc (and sds++) compiler is built upon Clang/LLVM frameworks [16]. An SD card image is generated by the compiler by default in a project subdirectory named sd_card. Note that not all C files are compiled by SDSoC compilers sdscc/sdsc++. It only compiles code that contains a hardware definition, call to a hardware function and uses sds_lib functions. It is also possible to change the compiler settings to compile with gcc instead of sdscc.

## 3.5 Working with SDSoC

Before we move on to exploring optimised coding techniques and acceleration using SDSoC, we go through the basics of getting started with SDSoC.

### 3.5.1 Creating an Application

First, we create a new SDSoC project and specify the target platform and whether we want the application to run bare-metal or use an OS. Fig. 3.3 shows the SDSoC project settings page. In our case, the Zedboard(option zed) is chosen as the target platform to run at the default frequency of 142.86MHz.

During project creation, we can choose to create an empty application or use any of the built in examples as shown in Fig. 3.4. In case a blank project was created, then the required application files are to be added to the src folder. Right click on the src folder and create new source files and, 'header files as per the requirement. Note that the name has to include the extension of .c or .cpp or .h. One can also import an entire project to the workspace using the Import option.

Next, choose the Build Configuration and navigate to 'Set Active' and set the configuration as SDDebug, SDRelease or SDEStimate. To estimate the performance without actually running on the target platform, we build using the SDEstimate option. SDDebug configuration helps in debugging the application as this configuration

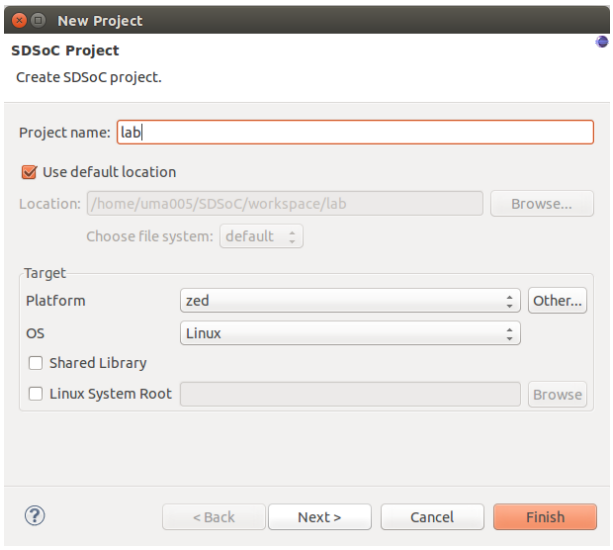



Figure 3.3: Creating a project

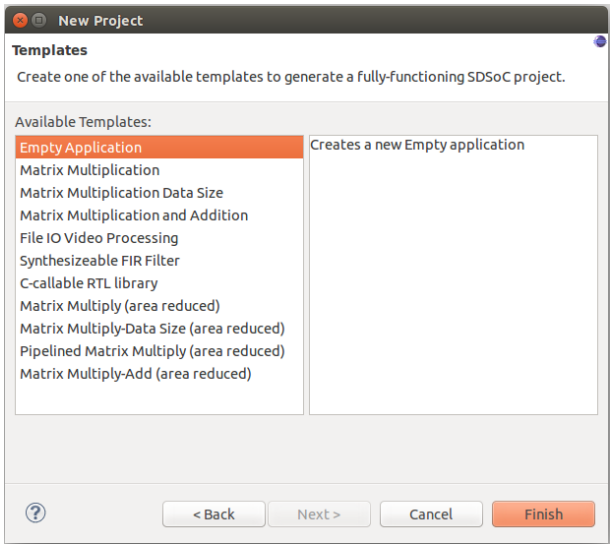



Figure 3.4: Creating application

builds the application with extra information in the ELF (compiled and linked program) that is needed to run the debugger. But note that the debug information in ELF increases the size of the file and makes the application information visible. For best run-time performance, the SDRelease build configuration can be used as it uses higher compiler optimization setting than the SDDebug build configuration.

Once the configuration is selected, build the project. First we use software only compilation due to reasons mentioned in Section 3.4. Next, we can profile the application to identify the bottlenecks and decide the segments to be offloaded to hardware. But on the other hand, if we know which section of application needs acceleration, we can skip the profiling. To offload the function to hardware, there are two methods. One is to expand the .c/.cpp file that has the function to be accelerated and right click on the function name and select toggle[H/S]. A yellow tick against the function name indicates that it has been selected to run on the fabric. Another method is to use the project overview window and click on the 'Add Hardware' icon to specify hardware functions as in Fig. 3.5 Now build the project again. This time, the build will take longer as it has to compile for the hardware and generate the bitstream. Once done, the files required to execute the code will be present in the sd_card folder.

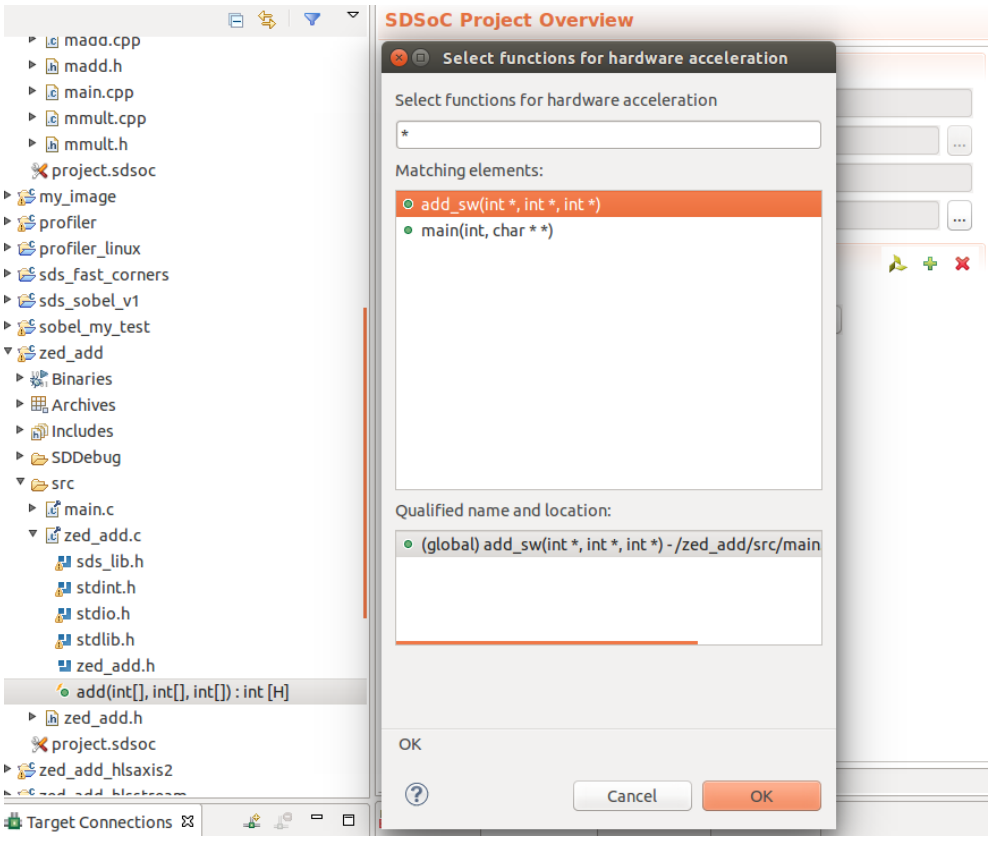



Figure 3.5: Selecting function to offload to hardware

### 3.5.2 Executing Application on the Target platform

To run the application, copy the contents of sd_card directory on to an SD card and insert into the target board. For Linux applications, this directory includes the following files:

- README.TXT- contains brief instructions on how to run the application
- BOOT.BIN - the boot image contains first stage boot loader (FSBL), boot program (u-boot), and the FPGA bitstream
- uImage, devicetree.dtb, uramdisk.image.gz - Linux boot image
- .elf - the application binary executable

First open a serial terminal connection to the target board and power it up. Details can be found in [17]. The user is logged in automatically as Linux boots. A bash shell is the user interface and the SD card is mounted at /mnt. The .elf file of the application can be run from the /mnt directory. For standalone applications, the ELF, bitstream, and board support package (BSP) are contained within BOOT.BIN, which automatically runs the application after the system boots.

## 3.6 SDSoC APIs

We now take a look at few of the SDSOC APIs which will make developing applications using SDSoC and analysing it easier. Refer [16] for more details. Using SDSoC APIs require us to include the library sds_lib. This can be done by adding , #include "sds_lib.h" in the source files.

1. API to map memory spaces, and to wait for asynchronous accelerator calls to complete
    - *void sds_wait(unsigned int id)* : wait for the first accelerator in the queue identified by id, to complete
    - *#pragma SDS wait(id)* : an alternative to the above

2. *void *sds_alloc(size_t size)* : Allocate a physically contiguous array of size bytes

3. *void *sds_alloc_non_cacheable(size_t size)* : Allocate a physically contiguous array of size bytes that is non-cacheable. As the memory allocated in this manner is not cached, the pointer to the memory has to be explicitly passed to the hardware

4. *void sds_free(void *memptr)* : To free an array allocated through sds_alloc()

5. *void *sds_mmap(void *physical_addr, size_t size, void *virtual_addr)* : Create a virtual address mapping to access a memory of size size bytes located at physical address physical_addr.

6. *void *sds_munmap(void *virtual_addr)* : Unmaps a virtual address associated with a physical address created using sds_mmap()

7. *unsigned long long sds_clock_counter(void)* : Returns the value associated with a free-running counter, which is the PS cycles. The global 64-bittimer runs at 1/2 the frequency of the processor [15]. The API samples this register and multiplies it by 2 to get cycles at the PS clock frequency.

## 3.7 Designing Accelerators using SDSoC

### 3.7.1 Factors Affecting Performance

Two of the main factors that affect performance is communication and computation. A well-designed system generally balances the two such that all the hardware components are utilized to the fullest. Extracting maximum performance from a system is highly dependent on the type of application too. Compute intensive or compute-bound applications require throughput to be maximised and latency in hardware accelerators to be minimized. Memory bound applications, on the other hand, might require restructuring of algorithm to increase the temporal and spatial locality in the hardware like adding copy-loops or memcpy to pull blocks of data into hardware instead of randomly accessing external memory.

In SDSoC, we can improve the system performance by controlling the compiler through :

- Improved access to external memory from programmable logic
- Increased concurrency and parallelism in programmable logic

The former can be attained using the various SDSoC communication or data transfer pragmas that are provided. These are explained in detail in Section 3.8. Once the platform and functions to be accelerated are chosen, the hardware/software interface is implicitly defined. The sdscc/sds++ system compilers take care of analysing the program data flow for hardware functions, scheduling each function call, and generating a hardware accelerator and data motion network realizing the hardware functions in programmable logic. Any data movement to and from the accelerator requires a data mover, which consists of a hardware component that moves the data, and an operating system-specific library function. The send/receive calls implemented in hardware are based on program properties like memory allocation of array arguments, function properties such as memory access patterns, latency of the hardware function etc. Table 3.1 shows the supported data movers and their properties.

Table 3.1: SDSoC Data Movers[16]

| SDSoC Data Mover | Vivadi IP Data Mover | Accelerator IP Port Types | Transfer Size | Contiguous Memory Only |
|---|---|---|---|---|
| *axi_lite* | processing_system7 | register,axilite | | |
| *axi_dma_single* | axi_dma | bram,ap_fifo,axis | 8MB | ✓ |
| *axi_dma_sg* | axi_dma | bram,ap_fifo,axis | | |
| *axi_dma_2d* | axi_dma | bram | | ✓ |
| *axi_fifo* | axi_fifo_mm_s | bram,ap_fifo,axis | 300B | |
| *zero_copy* | accelerator IP | aximm master | | ✓ |

- *axi_lite* data mover is usually used to transfer scalar variables over AXI4-Lite bus interface.
- *axi_dma_simple* data mover is used for bulk transfer and is the most efficient for it. The drawback is that it supports only up to 8MB transfers.

- *axi_dma_sg* (scatter-gather DMA) data mover is used for large data transfers, which axi_dma_simple does not support.

- *axi_fifo* data mover is used mostly for payloads up to 300 bytes and does not require as many hardware resources as DMA. The limit on amount of data that can be trasnferred is because of slow transfer rates.

The data mover selection can be overridden by inserting a pragma into program source immediately before the function declaration.

### 3.7.2 Coding the Hardware Function

Having understood how to work with SDSoC and the factors that affect a software-hardware system, we delve deeper into the process of creating applications to be accelerated by the Zynq FPGA. As mentioned in Section 3.3, the first step is to create a C/C++ application code. The guidelines of coding for SDSoC are the same as writing any other C/C+++ code. As an additional feature, we can also use the SDSoC APIs mentioned in Section 3.6. When it comes to functions that are to be offloaded on to the hardware, there are certain rules have to be followed so that the function is actually synthesizable.

Some of the points to remember while developing functions to be offloaded to hardware are as follows -

1. A top-level hardware function must be a global function. Class methods or overloaded functions are not allowed.

2. It is not possible to access global variables within a hardware function.

3. Hardware functions are incapable of exception handling.

4. Every hardware function must have at least one argument.

5. If arrays are being passed as input to the function to be accelerated, then the size has to be known. Pointers are not allowed as function arguments.

6. The data to be processed by the hardware is usually copied to BRAM and hence we are limited by the BRAM size as the maximum possible data size being processed. There are ways to overcome this issue as explained in Section 3.8.1.

7. In case of multiple assignments to an ouput or inout scalar, local variables are to be created.

8. The return value of a hardware function, if present, must be a scalar value within 32 bits.

9. The application must have only a single master thread that controls hardware functions.

10. Predefined macros should be used to guard code with *#ifdef* and *#ifndef* preprocessor statements [16].

11. Certain functions like pow(), printf() are not synthesizable and should not be included.

12. One must use sds_alloc to allocate an array if using zero-copy pragma for the array or if using pragmas to explicitly direct the system compiler to use Simple-DMA or 2D-DMA.

When a code is compiled for hardware, a SDSoC generates a software stub code. This is done by redefining hardware function calls as calls to function stubs that are implemented with low level function calls to a send / receive middleware layer that efficiently transfers data between the platform memory and CPU and hardware accelerators, interfacing as needed to underlying kernel drivers. This can be found in the _sds ⟩swstubs folder of SDSoC. If we take a closer look at the stub code generated, we can see that the function picked by us for offloading to hardware will be marked out by **_p0_** ⟨***name*** ⟩**_0** as shown in the Fig. 3.6.

Last but not the least, as a good coding technique, write the computation to be offloaded to hardware as not just a separate function but preferably as a separate .c/.cpp file

```
#ifdef __cplusplus
extern "C"
{
  #endif
  int _p0_add_0(int a[128], int b[128], int sum[128]);
  #ifdef __cplusplus
}
#endif
int _p0_add_0(int a[128], int b[128], int sum[128])
{ }
```

Figure 3.6: SWStub code of the function 'add' which is to be accelerated

## 3.8 Data Transfer in SDSoC

Once we have defined the functionality of the kernel to be offloaded to the hardware, the focus now shifts to one of the most important aspects of using an accelerator - the movement of data to and from the accelerator. Efficient communication between the host and accelerator is key in achieving significant acceleration of the function offloaded to the hardware. Hence we take a look at all the data movement and data access techniques supported by SDSoC.

### 3.8.1 SDSoC pragmas

A top-level hardware function should not contain any HLS interface pragmas. Instead there are SDSoC environment pragmas which can be used to guide the SDSoC environment to generate the desired HLS interface directives. They are as follows :

1. Data Copy : *#pragma SDS data copy(p[0:])*
   This is the default data transfer mechanism and the *RANDOM* access pattern used by SDSoC. The data is transferred by copying the data, as a consequence of which, an array argument must be either used as an input or produced as an output, but not both. The pragma must be specified immediately preceding a function declaration, or immediately preceding another pragma that is bound to the function.

2. Data zero copy : *#pragma SDS data zero_copy(p[0:])*

This can be used to generate a shared memory interface implemented as an AXI master interface in hardware. In the previous case, we saw that the array argument cannot act as input and output. In case we require an array argument to act as input and output, this pragma can be used as it tells the compiler that the array should be kept in the shared memory and not copied. The shared memory interface is also implemented as an AXI master interface in hardware. An important feature to note when it comes to using the above two pragmas is that they support variable data size transfers to the hardware function. This is possible by using the pragma to generate code whose size is defines by an arithmetic expression -

*#pragma SDS data copy|zero_copy(arg[0:⟨c_size_expression⟩]*

where ⟨c_size_expression⟩must compile in the scope of the function declaration.

3. *SEQUENTIAL* : *#pragma SDS data access_pattern(argument:SEQUENTIAL)* This pragma is used to declare to sdscc that a streaming access is to be allowed for a hardware function. This implies that each element is accessed only once and in index order. The SDSoC environment automatically maps onto a packetized AXI4-Stream channel and streaming is implemented as a FIFO interface in hardware. Streaming interfaces are the fastest means of data transfer especially when there are long streams of data and multiple streams. Since all the data is not copied to BRAM, we are not limited by BRAM size. In case non-sequential access is needed, then the data will have to be stored in local memory. One has to note that a hardware function can have no more than eight input bram or *ap_fifo* arguments.

### 3.8.2 HLS pragmas

For certain cases, the SDSoC directives might not perform as expected. In such cases, SDSoC provides the option to use the HLS pragmas but there are certain guidelines to be followed while coding. As the HLS pragmas cannot be compiled by sdsc/sds++ compilers, they are to be protected using *#ifndef __SDSVHLS__...#endif*. This directs the SDSoC to ignore the segments enclosed within while __SDSVHLS__ directs Vivado

which is used at the back end to compile and synthesize the appropriate hardware. The two commonly used HLS Pragmas, used for streaming of data are as follows :

1. AXI4-Lite : *#pragma HLS INTERFACE s_axilite port=arg*
   This pragma is used to generate a memory mapped control interface in HLS. No FIFOs are present on the command interface or on scalar arguments. Only one explicit AXI4-Lite interface is allowed for a hardware and all the ports must be bundled into a single AXI4-Lite interface.

2. AXI-memory mapped (AXI-MM) master : *#pragma HLS INTERFACE m_axi port=arg* This is used to pass physical addresses over the AXI4-Lite interface and the hardware function acts as its own data mover.

3. AXI4-Stream : *#pragma HLS INTERFACE axis port=arg*
   *#pragma HLS INTERFACE ap_fifo port=arg* this enables in having direct connections between hardware functions with AXI4-Stream interfaces.

A comparison of the DMA Round Trip Time(RTT) for all the data transfer types using SDSoC and HLS is shown in Fig. 3.7.

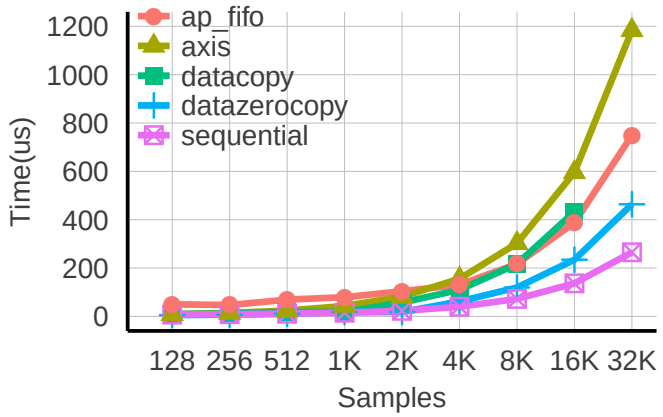



Figure 3.7: Single stream In-Out data transfer time

Each data sample is 4 bytes. The experiment involved a single stream of data transferred to and from the accelerator. It can be seen that *SEQUENTIAL* is the

most efficient while the HLS pragmas perform poorly in comparison to all others. Performance of *SEQUENTIAL* is as expected and though one would expect the other two streaming interfaces to also perform equally good, a completely opposite behavior is seen. *SEQUENTIAL* is 4× faster than *axis* and 3× faster than *ap_fifo*. This could be accounted to the overhead of launching the data transfer. Use of only single stream might mask the performance improvement that could be achieved [16]. Since essentially a streaming form of communication is known to be the fastest, we now delve deeper into the behaviour of these streaming interfaces.

## 3.9 Streaming Interfaces - Case Study

### 3.9.1 Single Stream In-Out

Having seen the relative performance of all the data transfer types, we now take a closer look at the three streaming interface models. We first try to understand the underlying architecture created while using each of these streaming models. The Vivado HLS tool is used to visualise the block designs generated.

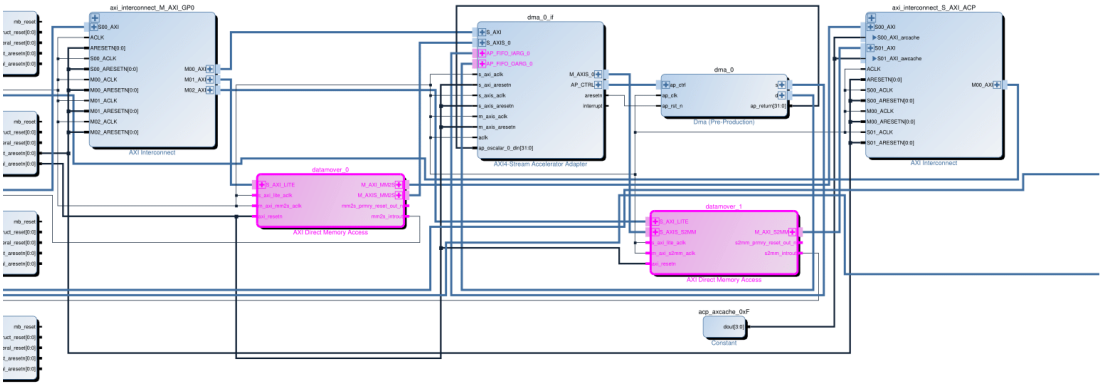


Figure 3.8: Block Diagram for *SEQUENTIAL* DMA

Fig. 3.8 shows the architecture created by SDSoC when the *SEQUENTIAL* pragma is used. We see that AXI interconnects are used and a FIFO is created as part of the adapter which connects to the accelerator. There are two data movers, one for

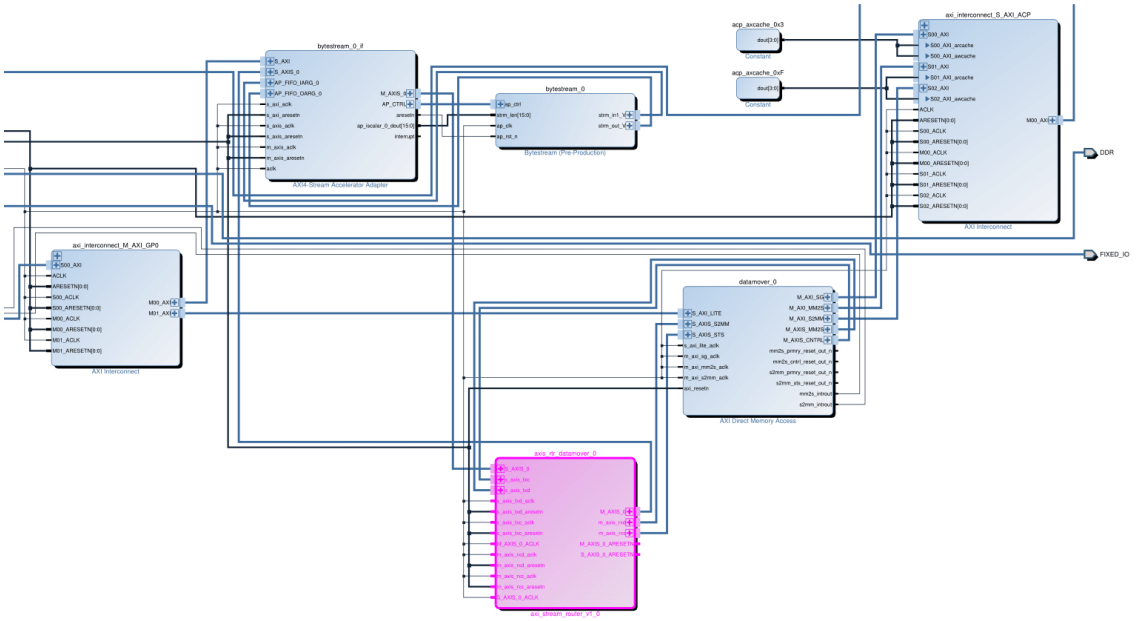


Figure 3.9: Block Diagram for *ap_fifo* port as streaming

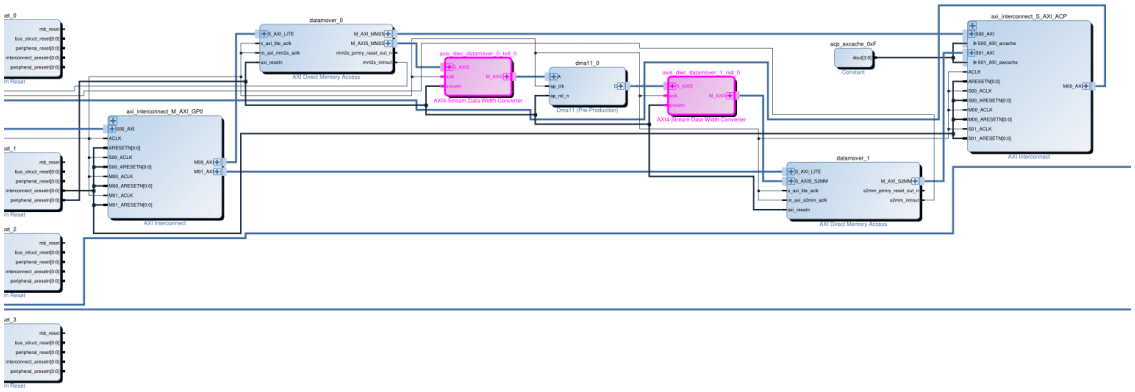


Figure 3.10: Block Diagram for *axis* port as streaming

each input and output streams. Fig. 3.9 shows the streaming architecture built using *ap_fifo* port. Again we notice the auto-generated AXI interconnects and adapters with FIFO. But we now see an " AXI Stream Router" which was not present in Fig. 3.8. Also, there is only one data mover being used. This is because the stream router handles the routing of the input and output streams and we do not require

two data movers. In case of Fig. 3.10, we see that the data movers are back to two but there is no longer an adapter with FIFO interface. Instead, we are provided with a "Data Width Converter".

Note that in all the cases the Accelerator Coherency Port (ACP) port is used, which is the cache coherent interface between programmable logic and external memory [16].One can also explicitly mention the port to be used if required by using the following pragma :

*#pragma SDS data sys_port(ArrayName : port)*

The port options involve ACP, High Performance (HP) ports or the non-cache coherent accessAXI FIFO Interface (AFI)port. The default case is decided by compilers using information regarding the array size, data movers, etc. This pragma can overwrite the default port usage.

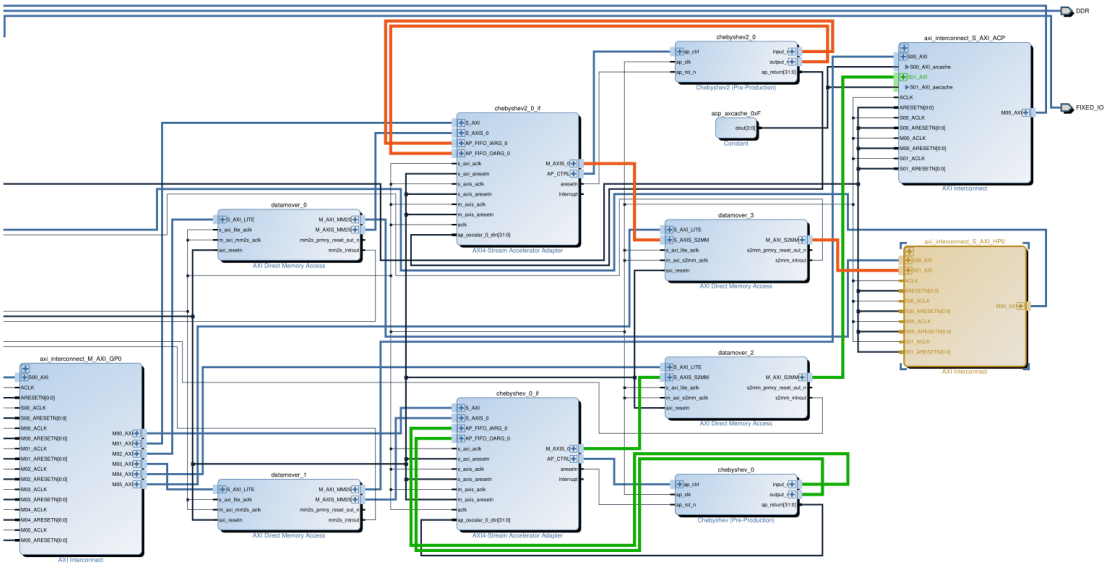


Figure 3.11: Block Diagram showing use of ACP and HP ports

The Fig. 3.11 shows the Vivado generated block diagram when *#pragma SDS data sys_port(ArrayName :* ***AFI****)* was used. To get a clear understanding of the difference in ACP and AFI, two accelerators using single stream were created and one uses ACP (the path is highlighted in green) and the other uses HP port (highlighted in orange). This is an advantage when an application requires multiple accelerators and we do

not want to share the ports.

The following experiments aim to evaluate the effect on communication time for various sreaming techniques using ACP ports. The case of DMA with no computation has been illustrated in Fig. 3.7. Next, we attach a compute unit or an accelerator to the stream. For our experiments, we consider a Chebyshev kernel which processes single incoming stream and produces single output stream and has extensive computations. Fig. 3.12 shows the Chebyshev implementation using all three modes of data transfer.

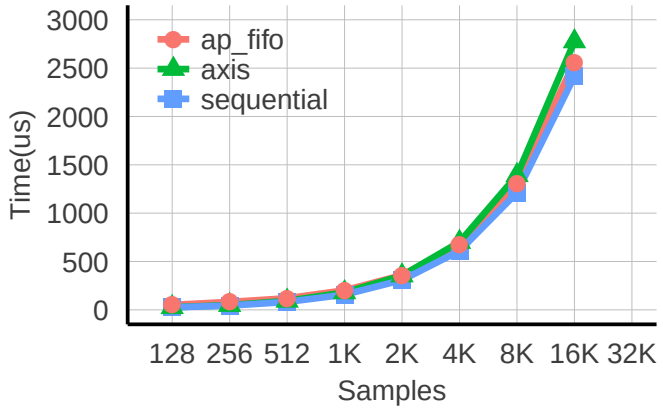



Figure 3.12: Chebyshev kernel without compute optimizations

Table 3.2: Chebyshev Kernel with and without pipelining

| Number of Samples | Time(us) | |
|---|---|---|
| | *SEQUENTIAL* without *pipeline* | *SEQUENTIAL* with *pipeline* |
| 128 | 25.415 | 7.873 |
| 256 | 44.179 | 9.034 |
| 512 | 82.007 | 11.236 |
| 1024 | 157.084 | 15.335 |
| 2048 | 307.712 | 23.28 |
| 4096 | 609.167 | 39.804 |
| 8192 | 1211.849 | 72.472 |
| 16384 | 2416.434 | 136.838 |
| 32768 | 4825.291 | 265.579 |

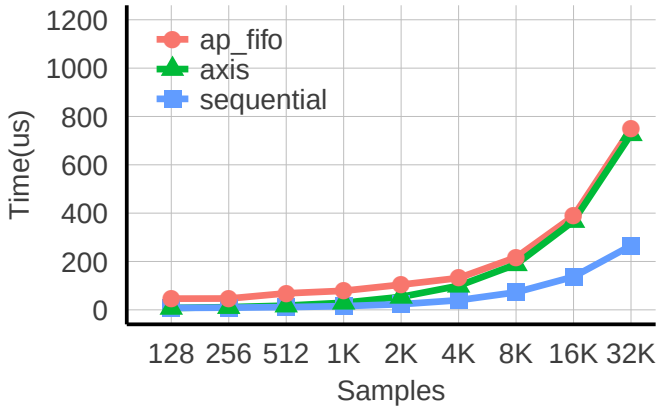



Figure 3.13: Chebyshev kernel with pipelining

Unlike our expectation of curves similar to those in Fig. 3.7, we see that all are performing the same and poorly. This is because we are limited by the extensive computation which is increasing the Initiation Interval (II) of the accelerator. This makes the RTT in all cases approximately the same with a performance of $\approx$ 6MSPS. Hence the advantage of a streaming architecture is lost on the implementation. To bring back the visibility of the effect of streaming, we need to optimise the computation which can be done by pipelining.

Using the *pipeline* pragma to set the II to 1 for the Chebyshev filter computations, the accelerator is able to process one sample every clock cycle. This in turn shifts the performance deciding factor to the communication technique used. The results obtained are shown in Fig. 3.13. Table 3.2 consolidates the profiling results of the experiments mentioned. And again we see that for larger data samples, *SEQUENTIAL* has the least RTT and is 2$\times$ better than other two cases.

Last but not the least, we compare the speed up achieved using accelerator in comparison to the same implementation in software. The example considered is the Chebyshev kernel. The result is tabulated in Table 3.3 and Fig. 3.14 shows the plot. The experiment compares the pipelined *SEQUENTIAL* data transfer implementation with the Chebyshev running on ARM. We see that the Chebyshev kernel is performing

Table 3.3: Chebyshev kernel profiling : Hardwre vs. Software

| Number of Samples | Time(us) | |
|---|---|---|
| | Hardware | Software |
| 128 | 7.873 | 10.836 |
| 256 | 9.034 | 21.266 |
| 512 | 11.236 | 41.911 |
| 1024 | 15.335 | 83.586 |
| 2048 | 23.28 | 166.253 |
| 4096 | 39.804 | 333.631 |
| 8192 | 72.472 | 685.229 |
| 16384 | 136.838 | 1378.818 |
| 32768 | 265.579 | 2757.138 |

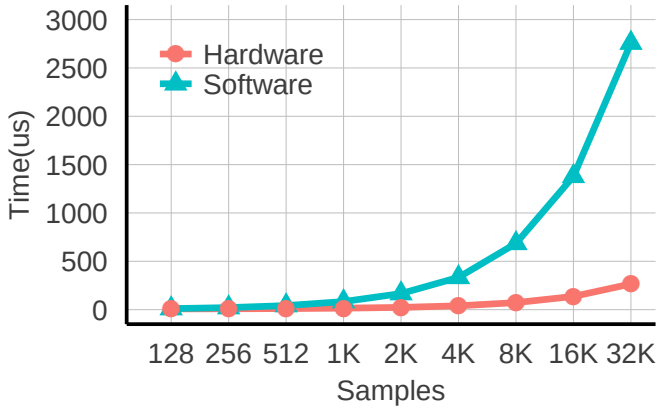



Figure 3.14: Chebyshev kernel hardware vs. software

10× faster on hardware than the software, giving ≈ 120MSPS. This serves as a strong justification for offloading tasks to hardware accelerators.

### 3.9.2 Multiple Stream In - Single Stream Out

Having analysed the effect of single stream in and out, we now take a look at multiple stream in single out data transfers. Experiments have been carried out for 2 In-1 Out, 3 In-1 Out and 4 In-1 Out for all three streaming architectures.

As expected, as the number of streams increase, the RTT also increases in all

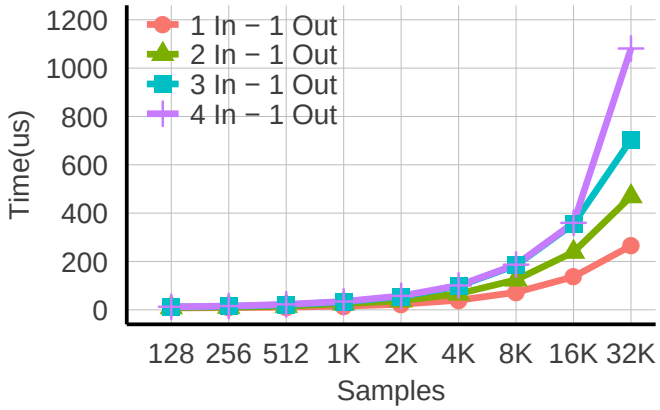



Figure 3.15: Performance of MISO streaming using *SEQUENTIAL*

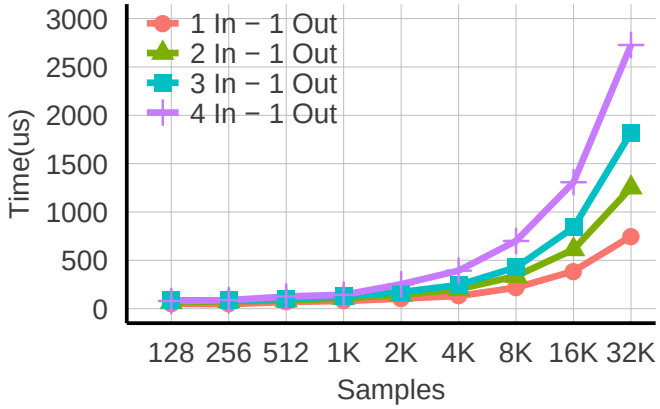



Figure 3.16: Performance of MISO streaming using *ap_fifo* port

cases. This is because all the streams share the same ACP port for streaming in and hence the streaming is of various arrays is in a queued fashion. From the timing values in Fig. 3.15, we can see that *SEQUENTIAL* outperforms all others up to four input streams. There is $\approx 2.5\times$ speed up using *SEQUENTIAL* in 4 In-1 Out compared to *ap_fifo* streaming whose results are shown in Fig. 3.16. In case of *axis* streaming, the BRAM runs out of resources to store arrays of 32K samples for four input streams.

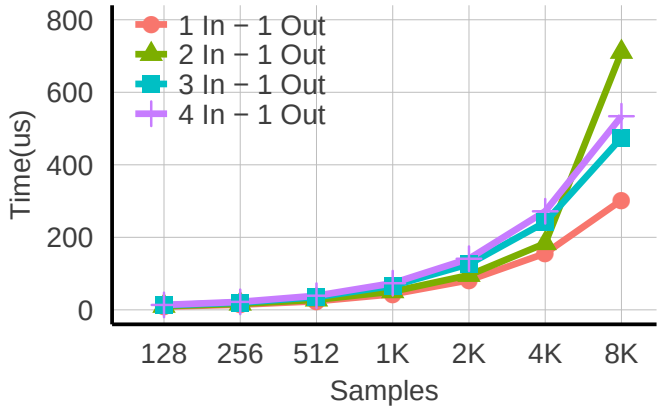



Figure 3.17: Performance of MISO streaming using *axis* port

Hence the Fig. 3.17 only plots timing till 8K samples. But even at sample count of 8K we see that *SEQUENTIAL* is better than *axis* streaming.

## 3.10 Image Processing on Zedboard using SDSoC

The need for image processing was explained in Section 2.4.1. In the real world, the applications deal with video data rather than still images. But video is nothing but streaming frames of still images. Hence we perform experiments using still images, which can later be extended to video stream. The experiments focus on the spatial grey level transforms which have been described briefly in Section 2.4.2.

We first consider a simple application of determining the negative or inverse of an image. The image used is grey scale in *.pgm* format. A header is used to read the image and store the pixel values in a buffer that is allocated using the API *sds_alloc* as shown in Fig. 3.18.

The processing is done in terms of float. Two functions implement computing the negative - one on software or ARM and the other on hardware or FPGA. The compute function that is being accelerated is shown in Fig. 3.19. Adhering to the coding guidelines mentioned in Section 3.7.2, the function to be offloaded to hardware

```
pgm_t ipgm;
/* Image file input */
readPGM(&ipgm, "/mnt/lena.pgm");
width = ipgm.width;
height = ipgm.height;

in_image = (float *)sds_alloc(width * height * sizeof(float));
out_image = (float *)sds_alloc(width * height * sizeof(float));

for( i = 0; i < width; i++ )
{
   for( j = 0; j < height; j++ )
   {
        ((float*)in_image)[(width*j) + i] = (float)ipgm.buf[width*j + i];
   }
}
```

Figure 3.18: Reading the image and allocating memory

is in a separate .c file and accessed from main using its header file.

```
for( i = 0; i < width; i++ )
{
    for( j = 0; j < height; j++ )
    {
        ((float*)out_image)[(width*j) + i] = 255 - ((float*)in_image)[(width*j) + i];
    }
}
```

Figure 3.19: Compute function to be accelerated

Both software and hardware computations can be timed using the SDSoC API *sds_clock_counter(void)* as shown in Fig. 3.20.

We can also check if the output of hardware computation matches the expected software computation. Finally, the processed image bytes are written back to form the output image using the *pgm* format headers as shown in Fig. 3.21. Also, note that the objects allocated using *sds_alloc* and the *pgm* type objects are to be freed at the end of the program as shown in Fig. 3.21.

Once the compilation is successful, the contents of *sd_card* folder along with the image to be processed is copied on to the SD Card. To execute the application,

```
#define sw_sds_clk_start() { sw_sds_counter = sds_clock_counter();
    sw_sds_counter_num_calls++; }
#define hw_sds_clk_start() { hw_sds_counter = sds_clock_counter();
    hw_sds_counter_num_calls++; }
#define sw_sds_clk_stop() { unsigned long long tmp = sds_clock_counter(); \
  sw_sds_counter_total += ((tmp < sw_sds_counter) ? (sw_sds_counter - tmp): (tmp -
      sw_sds_counter)); }
#define hw_sds_clk_stop() { unsigned long long tmp = sds_clock_counter(); \
  hw_sds_counter_total += ((tmp < hw_sds_counter) ? (hw_sds_counter - tmp): (tmp -
      hw_sds_counter)); }
#define sw_avg_cpu_cycles() (sw_sds_counter_total / sw_sds_counter_num_calls)
#define hw_avg_cpu_cycles() (hw_sds_counter_total / hw_sds_counter_num_calls)
.
.
hw_sds_clk_start();
negative(in_image, out_image, width, height);
hw_sds_clk_stop();
sw_sds_clk_start();
negative_sw(in_image, out_image_sw, width, height);
sw_sds_clk_stop();
.
.
printf("\n␣Average␣SW␣cycles␣:␣%llu␣\n", sw_avg_cpu_cycles());
printf("\n␣Average␣HW␣cycles␣:␣%llu␣\n", hw_avg_cpu_cycles());
```

Figure 3.20: Profiling the hardware and software compute times

```
opgm.width = height ;
opgm.height = width ;
/* Image file output */
normalizeF2PGM(&opgm, out_image);
writePGM(&opgm, "output_hw.pgm");
destroyPGM(&ipgm);
sds_free(in_image);
sds_free(out_image);
```

Figure 3.21: Create output image and free memory objects

navigate to the folder containing the *.elf* and the output image in *.pgm* format will be created. The output can be viewed by transferring to a workstation. The input image used for the experiment and the corresponding output is shown in Fig. 3.22.

To evaluate the effect of different data transfer techniques provided by SDSoC, we can mention them as pragmas in the header file just before the function declaration

(a) (b)

Figure 3.22: 512x512 input image and its negative

as shown in Fig. 3.23.

```
#ifndef NEGATIVE_H_
#define NEGATIVE_H_

#pragma SDS data access_pattern(in_image:SEQUENTIAL, out_image:SEQUENTIAL)
int negative(float in_image[512*512], float out_image[512*512], int width, int
    height);

#endif /* NEGATIVE_H_ */
```

Figure 3.23: Using Data Transfer pargmas

Various data packing and transfer techniques have been explored to determine the best performance without using compute optimisations. Initially the image data was of 32-bit floating point precision. Taking into account the fact the pixels of an image have values ranging from 0-255 and can be represented by 8 bits, and the lack of any complex computation, 8-bit representation was used. Furthermore, four 8-bit data was packed to form 32 bits and the resulting packed bits were transferred and the performance measured. The results of the mentioned experiments is tabulated below.

From the Table 3.4, we can see that the computation time for each data transfer reduces as we move from 32-bit to 8-bit precision and finally use packing of data to be transferred. In case of 128x128 image size, we see that software is always giving

Table 3.4: Profiling computation of negative of an image

| Image Size | Data Type | Software(ms) | Hardware | | |
|---|---|---|---|---|---|
| | | | *Data Copy*(ms) | *Data Zero Copy*(ms) | *SEQUENTIAL*(ms) |
| 128x128 | float | 0.47 | 1.576 | 7.952 | 1.271 |
| | unsigned char | 0.112 | 0.398 | 6.913 | 0.239 |
| | 8-bit packed | 0.06 | 0.247 | 1.617 | 0.196 |
| 512x512 | float | 29.55 | NA | 172.75 | 23.93 |
| | unsigned char | 9.784 | NA | 114.88 | 3.68 |
| | 8-bit packed | 4.43 | NA | 25.82 | 3.49 |

the best performance. It could be because the size of data involved in computation is not large enough to mask the cost of DMA transfers. But in case of 512x512 size image, we see that *SEQUENTIAL* is faster than software. This further supports our claim of needing large data sizes to be transferred to hardware so as to achieve the benefit of hardware for computation. The Data Copy pragma is mentioned as NA for 512x512 image size as it exceeds the BRAM capacity and hence cannot be copied to local memory. Data zero copy timing being the highest could be attributed to the use of shared memory for data communication between software and hardware. *SEQUENTIAL* turns out to be the most efficient among other data transfer types as expected. It is to be noted that in all the cases the accuracy of the function was not compromised while trying to achieve faster data transfer.

Thus, from the results presented we can say that using SDSoC for defining functions to be accelerated does provide us considerable speed up compared to software implementation. The fact that such performance is achieved even with the high levels of abstraction is appreciable. Adding to the advantage is the flexibility that helps a pure C implementation to be offloaded to hardware with minimal changes(so as to follow hardware function coding guidelines) which makes SDSoC an easy to use and effcient tool.

# Chapter 4

# OpenCL for programming Heterogeneous platforms

In Chapter3, we explored how to program a CPU-FPGA SoC using the Xilinx high-level synthesis tool SDSoC. SDSoC required us to develop a C/C++ application and everything that follows to offload required computation to hardware was abstracted from us. The only way to interact and set certain hardware performance related properties was through pragmas. We now explore the OpenCL programming model for Zedboard. We have used the OpenCl version 1.2.

## 4.1 Why OpenCL?

The introduction of new architecture brings along with it the challenge of programming them. Most of the compute units that make up a heterogeneous system use a vendor dependent development environment.

Consider the GPU ,which was designed for graphics processing, but its parallel architecture made it suited for data parallel processes. The release of NVIDIAs CUDA programming language finally opened up the possibility of GPGPU (General Purpose GPU). But the CPU cannot be completely replaced either. The CPU manages execution of code, managing the file system and user interface as GPU has no Operating System running on it. 2.2.2 touched upon briefly the CUDA programming model.Now

look at the accelerator Cell/B.E., known for its use on the PLAYSTATION3. The Cell/B.E. is programmed using the "Cell SDK" released by IBM [18]. The Cell/B.E. requires specific compilers for its Power Processing Element (PPE)s as control units and 8 Synergistic Processing Element (SPE)s as compute units. Although it does not extend a language like the CUDA, the SPE must be controlled by the PPE, using specific library functions provided by the Cell SDK [18]. We observe that both the heterogeneous systems have same structure where CPU/PPE is used for control and GPU/SPE does the compute. There are other systems using similar models, for example, DSps managed by CPUS in embedded electronics for signal processing. But in all these cases, inspite of similar procedures being used, the user has to work with APIS that are very different from one another.

Summarizing the above, the common characteristics in the above mentioned systems is that:

- They use SIMD hardware to perform vector-ized operation
- They use a processor to control numerous compute units
- The systems have multi-tasking capabilities

But since the software development method for each combination is unique, software development and services related to each the platform has to start from scratch. This is quite inconvenient, requiring programmers to learn a new set of APIs and languages. And with every new release, there is a possibility that what was learnt might be outdated. This is especially more common in heterogeneous platforms, as programming methods can be quite distinct from each other [19]. Add to it the fact that the software development process might have different levels of diffculty, choosing hardware platforms might not be very easy. The solution to this problem is "OpenCL".

## 4.2 OpenCL for Heterogeneous Platforms

OpenCL is "a framework suited for parallel programming of heterogeneous systems" using a standardized coding method independent of the processor types or vendors [20]. It is standardized by the Khronos Group, consisting of members from the processor or multi-core software vendors like AMD, Apple, IBM, Intel, NVIDIA, Texas Instruments, Sony, Toshiba [19]. The standardization aims at enabling use of one language to programs multiple devices like CPU, GPU, Cell/B.E., DSP, etc. For example, an OpenCL supported multi-core CPU and GPU can be used in sync where CPU with its SSE supporting cores and entire GPU can aid in parallel processing. With heterogeneous models being used largely from Embedded Systems to Desktops, OpenCL looks like the clever choice to consider.

One of the basic concepts that we need to know about OpenCL is that of Host and Device. OpenCL defines the control processors and compute processors as follows:

- Host-Environment comprising of CPU and its memory to control the device through software.
- OpenCL Devices-Environment made up of devices like GPU, Cell/B.E., DSP which executes the compute logic.

Conventionally an OpenCL device packs multiple Processing Element (PE) which combine together to form multiple Compute Unit (CU). For example, a GPU can be described in OpenCL as follows:

- OpenCL Device - GPU
- Compute Unit - Streaming Multiprocessor (SM)
- Processing Element - Scalar Processor (SP)

The code executing on CPU is called the 'Host Code' and on the device is called 'Kernel". The OpenCL kernel language is based on C but the kernels can be called from host using languages such as C, C++, Java, Python, JavaScript, Haskell, Perl, Ruby, etc. This enables portability and reuse of existing OpenCL kernels and results

in flexible support for GPGPU computing across a number of development environments. Developing OpenCL programs require the following two:

- OpenCL Compiler
- OpenCL Runtime Library

OpenCL compilers are designed for a particular environment. They are used to compile the source code that is executed on the device. The controlling processor manages the allocation of memory and loading of binary. The execution process that is common to all heterogeneous combinations is coded by the programmer. The set of commands that are used by the host for compilation is contained in the "OpenCL Runtime Library," which is designed to be used for that particular environment [19]. Once the host is programmed using OpenCL runtime API, it is linked to the OpenCL runtime library implemented for the host-device combination.

Another important aspect is the memory and its management. OpenCL allows the kernel to access the following 4 types of memory:

1. Global Memory or the device's main memory that can be read from all work items.
2. Constant Memory which also can be read from all work items but its utilization efficiency can overtake global memory if constant memory cache is supported by hardware.
3. Local Memory which is the physically shared memory on each compute unit that can be read from work items within a work group.
4. Private Memory which is physically the registers used by each processing element and can only be used within each work item.

Among these, the host is capable of reading and writing to the global, constant, and the host memory. The device memory can only be accessed by the kernel. There are no rules regarding how the host and the OpenCL device are connected. In the case of CPU + GPU, PCI Express is used most often. For CPU servers, the CPUs can be connected over Ethernet, and use TCP/IP for data transfer [19].

Now that we have an overview of OpenCL, we can explore the basic program flow.

## 4.3 OpenCL APIs

The Kernel code represents the function to be executed on the device. The kernel access of global, constant, local, and private memory, is specified by __global, __constant, __local, __private, respectively. If this is not specified, it will assume the address space to be __private, which is the device-side register. The host program tells the device to execute the kernel using the OpenCL runtime APIs which have been explained briefly below. For more details refer [21].

1. ret = *clGetPlatformIDs*(1, &platform_id, &ret_num_platforms);
   This allows the host program to discover OpenCL devices, which is returned as a list to the pointer platform_id of type cl_platform_id.

2. ret = *clGetDeviceIDs*(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &ret_num_devices);
   This selects the device to be used. CL_DEVICE_TYPE_DEFAULT selects the platforms default device. It can be replaced by CL_DEVICE_TYPE_GPU or CL_DEVICE_TYPE_CPU depending on the desired target device being CPU or GPU respectively.

3. context = *clCreateContext*(NULL, 1, &device_id, NULL, NULL, &ret);
   This creates an OpenCL context for the desired devices.

4. command_queue = *clCreateCommandQueue*(context, device_id, 0, &ret);
   The command queue serves as a connection between host and device and is used to control the device. For each device, one or more command queue objects must be created.

5. memobj = *clCreateBuffer*(context, CL_MEM_READ_WRITE, MEM_SIZE * sizeof(char), NULL, &ret);
   This creates a memory object which allows the host to access the device memory.

6. Read Kernel File :
   Since the kernel can only be executed via the host-side program, the host program must first read the kernel program. Using a standard fread(), the kernel, which is in the form of an executable binary, or a source code which must be compiled using an OpenCL compiler, can be read.

7. program = *clCreateProgramWithSource*(context, 1, (const char **)&source_str, (const size_t *)&source_size, &ret);
   This creates a program object which mentions in which context the read source code is to be executed. *clCreateProgramWithBinary()* is an alternative [21].

8. ret = *clBuildProgram*(program, 1, &device_id, NULL, NULL, NULL);
   The program has to be built to create a binary for a particular target device. It is also possible to specify compiler option string as part of this API. This step is unnecessary if the program is created from binary using *clCreateProgramWithBinary()*.

9. kernel = *clCreateKernel*(program, "hello", &ret);
   This creates the kernel object corresponding to each kernel function using the kernel name. Multiple kernel functions require multiple *clCreateKernel()* calls.

10. ret = *clSetKernelArg*(kernel, 0, sizeof(cl_mem), (void *) &memobj);
    This sets the kernel arguments using the kernel object, the argument number of the kernel that is being referred to, the pointer to the argument and the argument size.

11. ret = *clEnqueueTask*(command_queue, kernel, 0, NULL, NULL);
    This performs the task parallel kernel execution which is asynchronous by default. Synchronization is possible using 'event objects' to wait for completion of a kernel execution.

    OpenCl also supports data-parallel kernel execution using the following API :
    ret = *clEnqueueNDRangeKernel*(command_queue, kernel, 1, NULL, &global_item_size, &local_item_size, 0, NULL, NULL);

The dimension of the data we are dealing with, the total number of work items that can execute a particular kernel in parallel and the number of work items to be grouped together in a work group can be specified in the API. Choosing the work group values based on the device is key in efficient optimization. Refer 4.4 for details data parallel execution. execution.

12. ret = *clEnqueueReadBuffer*(command_queue, memobj, CL_TRUE, 0, MEM_SIZE * sizeof(char), string, 0, NULL, NULL);
    This copies the results from device to host. The data copy instruction is placed in the command queue before it is processed. The function can be made synchronous by using the "CL_TRUE" argument , which forces the host to wait for data copy to finish before executing next command."CL_FALSE" on the other hand makes the copy asynchronous.

13. Free Objects :
    This fress all the OpenCL objects created.
    ret = *clReleaseKernel*(kernel);
    ret = *clReleaseProgram*(program);
    ret = *clReleaseMemObject*(memobj);
    ret = *clReleaseCommandQueue*(command_queue);
    ret = *clReleaseContext*(context);

14. Profiling Kernels :
    The OpenCL kernels can be timed most effectively using the OpenCL profiling API provided by Khronos. Firstly, the profiling action must be enabled while creating the command queue by setting the profiling related argument as CL_QUEUE_PROFILING_ENABLE. The 'event' flags are used to determine the start and end of kernel execution. The API for profiling is as shown below :
    *clGetEventProfilingInfo*(event, CL_PROFILING_COMMAND_START, sizeof(time_start), &time_start, NULL);
    *clGetEventProfilingInfo*(event, CL_PROFILING_COMMAND_END, sizeof(time_end), &time_end, NULL);

Compute the difference in values of time_start and time_end to determine the time taken for kernel execution. Note that the time is returned in nano-seconds. Accurate timing requires that OpenCL devices correctly track time across changes in device frequency and power states. The CL_DEVICE_PROFILING _TIMER_RESOLUTION can be used to determine the resolution of the timer being used to query.

In real-life applications, the flow of the OpenCL application development is an iteration over setting kernel arguments, executing the kernel and finally device-to-host copy. This suggests that we do not have to create/destro objects for every iteration.But be aware that creating too many objects without freeing can lead to the exhaustion of memory space at the host-side.

OpenCL has the capability of performing vectorized SIMD operations, data parallel processing, task parallel processing, and memory transfers. The use of kernels efficiently depending on the nature of the task and the target architecture makes OpenCL not only suitable for programming software accelerators, but also for defining the implementation of a custom hardware accelerator. There is no need for modifying main logic of the applications which is part of the kernel code for porting across platforms. The changes from one platform to another would be the available resources in terms of memory and compute units which determine the definition of work group sizes. These modifications can be easily done at the host code to adapt efficiently to the new platform and have maximum hardware utilization.

## 4.4 Data Parallelism in OpenCL

As mentioned in 3.6, OpenCL allows for both data parallel and task parallel operations. In OpenCL, the difference between the two is whether the same kernel or different kernels are executed in parallel [19]. Since our focus is mainly on Image processing, it implies that the computations are of single instruction multiple data(SIMD) type. And a parallel architecture is the best for such computation. Hence we now take a closer look at data parallel kernel execution in OpenCL.

Multiple processors can run the same kernel using *clEnqueueNDRangeKernel()*. Each compute unit runs a group of kernel and each group is assigned an ID. Another ID for each kernel within each compute unit, which is run on each processing element. The ID for the compute unit is called the Workgroup ID, and the ID for the processing element is called the Work Item ID. The OpenCL runtime API assigns IDs for each item once the total number of work items (global item) and the number of work items to be run on each compute unit (local item) is input by the user. This ID is used to access the index-space. The number of workgroups can be computed by dividing the number of global items by the number of local items.The relationship between the global work-item ID, local work-item ID, and the work-group ID are shown below in Fig. 4.1.

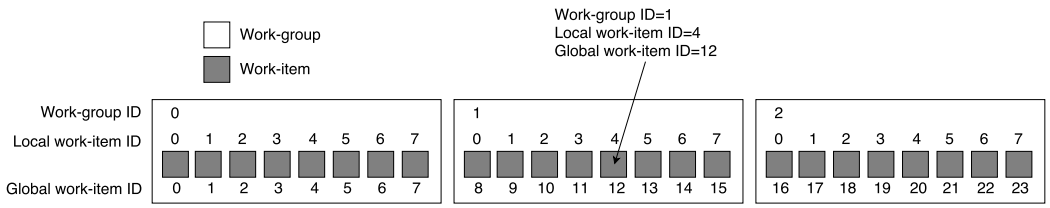



Figure 4.1: Concept of work groups and work items

Data parallel processing follows two steps :

- Get work-item ID
- Process the subset of data corresponding to the work-item ID

One must note that a work-group must consist of at least 1 work-item, and the maximum is dependent on the platform [19]. Ynchronisation and sharing of local memory is allowed between the work-items within a work-group. The number of work-groups and work items has to be passed as the API arguments, as suggested earlier. The data to process can have up to 3 dimensions. The number of work-items per work-group is consistent throughout every work-group. The number of work-items should be divisble evenly among the work-groups, lack of which the fails the call to *clEnqueueNDRangeKernel()* and returns the error value CL_INVALID_WORK_GROUP_SIZE.

## 4.5 Online and offline compilation

Last but not the least is the feature to compile online and offline using OpenCL. Fig. 4.2 gives an idea about the concept.

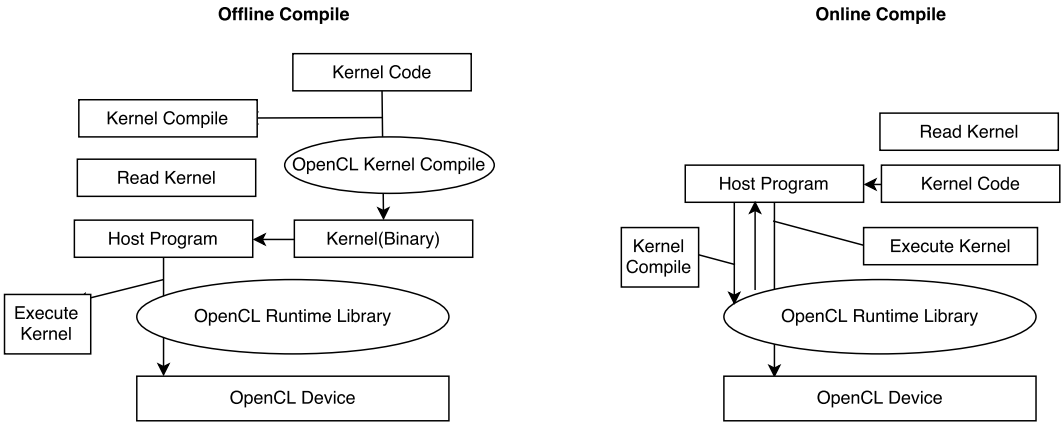



Figure 4.2: Concept of online and offline compilation

In "offline-compilation", the binary generated using a pre-built OpenCL kernel is loaded using the OpenCL API. This reduces the time between starting the host code and executing the kernel. The only problem in this case is that the size of executable kernel increases if we intend to run it on various platforms as we need to include kernels for multiple binaries.

In "online-compilation", the OpenCL runtime library builds the kernel from the source.the kernel is built from source during runtime using the OpenCL runtime library. It has an advantage that we gain a device-independent binary at the host side. Also, we need not compile the kernel every time. However, this is not suited for embedded systems that require real-time processing.

One can choose either of the compilation methods depending on the need.

## 4.6 Image Processing on Zedboard using OpenCL

We again explore the performance of the grey level spatial transforms like in 3.10 and few spectral transform using OpenCL. We implement the applications on Zedboard

and compare the performance of pure C implementation with OpenCl on Zedboard ARM. We also compare the execution times of an OpenCL code with respect to a pure c implementation.

Before we can start executing OpenCL kernels on Zedboard or any other Linux targets, there are a set of pre-requisites that must be fulfilled. Firstly we require an Operating System on the Zedboard. The OS chosen is Xilinux , a linux version for Zedboard which is of Ubuntu 12.04. It can be used to integrate the linux running on the ARM directly to the FPGA through /dev/. The process of installing Xilinux on Zedboard with the base Xillybus bitstream can be seen in detail in [22]. Once the OS is in place, we need to set up the Zedboard so as to be able to identify and execute OpenCL code. For this purpose, we need to first install a portable and open source version of OpenCL that is compatible with ARM. For wide support, POCL was chosen. Refer [23] to know about POCL in detail. Installing POCL requires a set of dependencies to be installed first whose details can be seen in [24].

```
cl_platform_id platform_id = NULL; //platform object
cl_device_id device_id = NULL;  //device object
cl_context context = NULL;  //context object
cl_command_queue queue = NULL;  //command queue
cl_program program = NULL;  //program object

cl_mem xmobj = NULL;   //memory objects
cl_mem rmobj = NULL;
cl_kernel trns = NULL;   //kernel object

cl_uint ret_num_devices;  //ret types for various API calls
cl_uint ret_num_platforms;
cl_int ret;
```

Figure 4.3: Creating OpenCL objects

For profiling of applications, we use the OpenCL API indexed in Section 4.3 for determining the kernel execution time and PAPI to profile the time for every other API function call. PAPI or Performance API is an API that can be used for effective profiling in ARM as well as intel targets. In both cases, the PAPI needs to be compiled from source to support the profiling being used. The installation of PAPI is given in

detail in [24]. Now we are all set to execute and profile OpenCL applications on both our target platforms Zedboard ARM and x86 processors.

Again we first consider a simple application of determining the negative of an image. The host code will deal with reading the image and passing it onto the kernel which then performs the computation and returns the processed values to the host code where the output image is created. Consider the host code which initially creates all the required OpenCl objects as shown in Fig. 4.3.

Note that the OpenCL version of standard data types like *uint* is prefixed by 'cl_'. Once the OpenCL objects and host-end objects are created, we move on to reading the kernel code which is a '.cl' file as shown in Fig. 4.4.

```
FILE *fp;
const char fileName[] = "./negative.cl";

/* Read kernel source code */
fp = fopen(fileName, "r");
if (!fp)
{
    fprintf(stderr, "Failed␣to␣load␣kernel.\n");
    exit(1);
}
source_str = (char *)malloc(MAX_SOURCE_SIZE);
source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp );
fclose( fp );
```

Figure 4.4: Read and store the kernel code

Once the image is read and stored in host-side buffer, we move on to initialising the target device and creating the communication interface through a series of API calls mentioned in Section3.6 as shown in Fig. 4.5.

Next the kernel program is built, arguments set and offloaded for data parallel execution using *clEnqueueNDRangeKernel()* as shown in Fig. 4.6. The kernel code that computes the negative is shown in Fig. 4.7.

Note that there are no loops going over the width and height of the image as one would expect in normal C code. This is because we set the global worksize as the size of the image and the OpneCL runtime automatically handles iterating over the entire global work space thereby covering every pixel of the image.

```
/* Get platform and device information */
ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &
    ret_num_devices);

/* OpenCL creating context*/
context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);

/* Createing command queue */
queue = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE, &ret);

/*Create memory buffer */
xmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, width*height*sizeof(cl_float),
    NULL, &ret);
rmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, width*height*sizeof(cl_float),
    NULL, &ret);
```

Figure 4.5: Initialisation of OpenCL device

```
/* Create kernel program from read source */
program = clCreateProgramWithSource(context, 1, (const char **)&source_str, (const
    size_t *)&source_size, &ret);

/* Build kernel program */
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

/* OpenCL create kernel */
trns = clCreateKernel(program, "negative",&ret);

/* set kernel arguments */
ret = clSetKernelArg(trns, 0, sizeof(cl_mem), (void *)&rmobj);
ret = clSetKernelArg(trns, 1, sizeof(cl_mem), (void *)&xmobj);
ret = clSetKernelArg(trns, 2, sizeof(cl_int), (void *)&width);

/* set global and local work size*/
gws[0] = width;
gws[1] = height;

/*Enque task for parallel execution*/
ret = clEnqueueNDRangeKernel(queue, trns, 2, NULL, gws, NULL, 0, NULL, &event);
```

Figure 4.6: Offloading task to target device

Profiling the kernel execution can be done as shown in Fig. 4.8.

Finally we read the processed data back to host from device memory and create

```
__kernel void negative(__global float *dst, __global float* src, int n)
{
    unsigned int xgid = get_global_id(0);
    unsigned int ygid = get_global_id(1);
    unsigned int iid = ygid * n + xgid;
    unsigned int oid = xgid * n + ygid;
    dst[iid] = 255 - src[iid];
}
```

Figure 4.7: Kernel code computing negative of input image

```
clWaitForEvents(1, &event); //wait for kernel execution to finish
clFinish(queue);  //check that queued kernels are all executed
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(time_start), &
    time_start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(time_end), &time_end
    , NULL);
```

Figure 4.8: Profiling the kernel

the output as shown in Fig. 4.9.

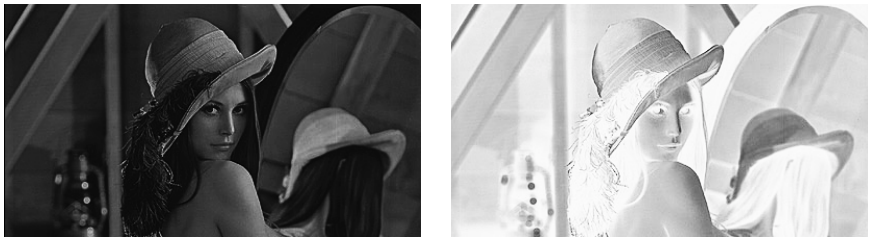
```
/* Read from memory buffer */
ret = clEnqueueReadBuffer(queue, rmobj, CL_TRUE, 0, width*height*sizeof(cl_float),
    rm, 0, NULL, NULL);

/* Output image*/
normalizeF2PGM(&opgm,rm);
writePGM(&opgm, "output.pgm");
```

Figure 4.9: Read from device to host memory

The output images are showin in Fig. 4.10

Following similar coding style and reusing upto 90% of the host code, we can develop several applications. Few of the other grey level spatial transform results has been shown below.

- Transpose : A 90° rotate left operation has been used on a rectangular image whose results are shown in Fig. 4.11.
- Power Law Transform : A $\gamma$ value of 2 has been used to give the output image



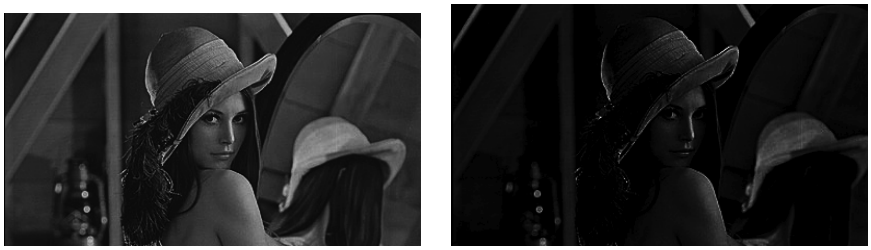
(a) (b)

Figure 4.10: 400x225 input image and its negative

(a) (b)

Figure 4.11: Input image and transposed output

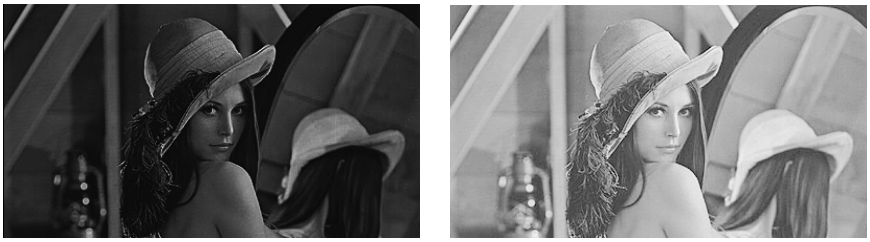
as shown in Fig. 4.12



(a) (b)

Figure 4.12: Input image and power law transformed output

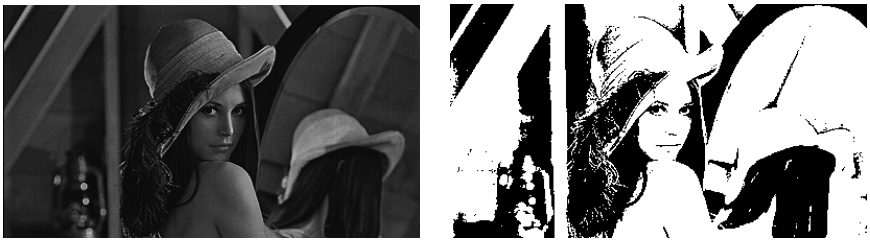
- Log Transform : The log trasnformed image is shown in Fig. 4.13. We can see that the range of pixel values have been spread out to yield a lighter image compared to input.



(a) (b)

Figure 4.13: Input image and log transformed output

- Thresholding : The two level image output for a threshold value of 50 is shown in Fig. 4.14. The value can be shifted up or down for differnt results.



(a) (b)

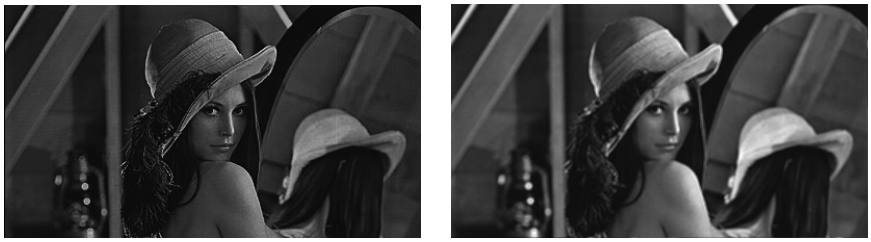
Figure 4.14: Input image and thresholded output

- Convolutional transforms : In this case the mask used is also transferred from host to device along with the input image. The convolutional kernel usually has four loops - two iterating over image and other two over the mask. But in case of OpenCL kernel, we have only the loops iterating over the masks which reduces the code to that shown in Fig. 4.15.

  Blurring can be achieved using convolution using more than one kind of mask. The result of using a Box filter is shown in Fig. 4.16.

```
__kernel void gaussianblur(const __global float * const input,
                           __constant float * const mask,
                           __global float * const output,
                           const int inputWidth,
                           const int maskWidth)
{
    const int x = get_global_id(0);
    const int y = get_global_id(1);
    float sum = 0;
    for (int r = 0; r < maskWidth; r++)
    {
        const int idxIntmp = (y + r) * inputWidth + x;
        for (int c = 0; c < maskWidth; c++)
           sum += ((mask[(r * maskWidth) + c])/9) * input[idxIntmp + c];
    }
    output[y * get_global_size(0) + x] = sum;
}
```

Figure 4.15: Read from device to host memory

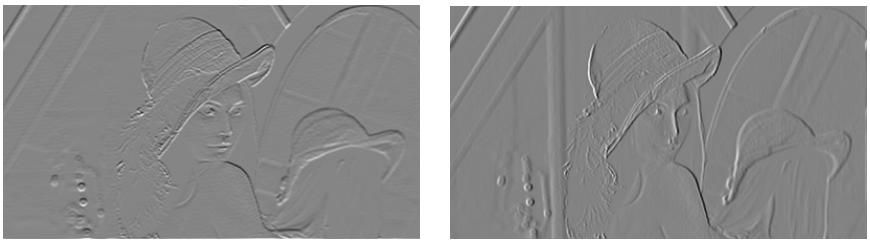


(a) (b)

Figure 4.16: Blurring of image

Edge detection is another application that uses convolution. The result of using a Sobel edge detector is shown in Fig. 4.17(a) and 4.17(b). The output of horizontal and vertical edge detection can also be combined to detect all the edges in the image by summing the magnitudes of former.

Now that we have explored the correctness of OpenCL based still image processing applications, we try to quantify the performance by measuring the kernel execution time and comparing with a pure C implementation. We have used the profiling API mentioned in Section 4.3 to measure the performance in case of OpenCL and PAPI in case of C implementation. The results are tabulated and plotted as shown below.



(a) Horizontal Edges using Sobel (b) Vertical Edges using Sobel

Figure 4.17: Sobel filter for edge detection

Table 4.1: OpenCl kernel and C function execution time

| Application | Time(ms) | |
|---|---|---|
| | *OpenCL kernel* | *C function* |
| Blurring | 4609.718 | 20.792 |
| Log | 4393.950 | 48.016 |
| Negative | 4387.646 | 2.802 |
| Power Law | 4365.141 | 59.749 |
| Thresholding | 4363.409 | 48.016 |
| Transpose | 4404.033 | 1.659 |

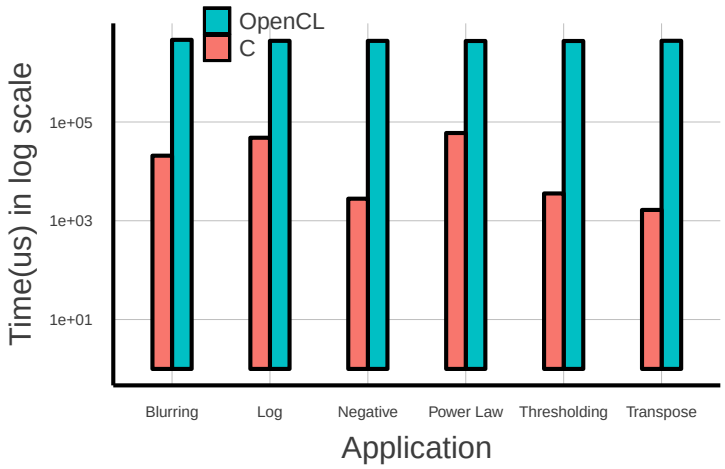



Figure 4.18: Comparing OpenCL and C execution time

From the results in Fig. 4.18, we see that the OpenCL execution time is higher than the corresponding C implementation. This can be attributed to the overheads

of using POCL. In fact this overhead cost might even explain why there isn't much difference in the execution time of different kernels as can be seen from Table 4.1. Though this might make us wonder why use OpenCL at all, we must recall that the main aim of using OpenCL is to achieve portability across platforms. Of course, performance portability at a high cost of performance degradation is not acceptable. But from the results it is evident that the gap in performance is only in 10s of milliseconds. Hence the use of OpenCL can be justified for platform portable SoC applications.

# Chapter 5

# Conclusions and Future Work

This chapter concludes and summarizes this report. Furthermore, in this chapter we discuss future research directions in detail.

## 5.1 Conclusions

The main focus of the thesis was to explore ways to carry out computations using a heterogeneous platform, specifically Xilinx Zynq The challenge and time involved in programming the platform (Xilinx Zynq) using its own programming model brought up the need to use tools (Xilinx SDSoC) which provided us with programming abstraction. The experiments were designed to evaluate the two main performance deciding factors when using FPGA as an accelerator, namely communication and computation. With respect to communication we observed that offloading a task to hardware is worth the effort only when the size of data being transferred is large so as to mask the overhead of launching DMA transfers. We also observed that streaming architecture option provided by SDSoC, *SEQUENTIAL*, is the fastest. With respect to computation, we observed that just offloading a function to hardware is not suffiecient to extract maximum performance. Techniques like pipelining, especially when data is streamed, increases the performance efficiency 20×. We see that despite the high levels of abstraction, it is possible to obtain considerably fast implementation

using SDSoC. Even if it is not maximum achievable performance (considering manually designed custom hardware modules), the speed ups are encouraging enough and makes SDSoC a perfect choice for non-hardware engineers who wish to use FPGA for acceleration.

We use OpenCL to aim source code portability across various platforms. The experiments focused on first understanding the OpenCL programming model and learning how to develop an OpenCL application depending on the target device and the type of computation required. Though OpenCL includes even higher level of abstraction, it still requires that the user has the knowledge of underlying architecture so as to extract maximum performance. Our experiments used generic codes and we observed that the OpenCl kernel execution times were higher than C implementation. This could be attributed to the overheads of using POCL. But nevertheless using OpenCL eliminates the need to know various programming models corresponding to each device being used. Thus, easing the effort and saving the time spent on porting applications to different platform, OpenCL stands out a strong choice for developing applications for heterogeneous platforms.

## 5.2 Future work

With respect to SDSoC, there are a few areas that are yet to be explored, like

- **Real Time Video Processing**: Processing video in real time requires highly efficient and optimised application development. This would lead to exploring various communication ports other than ACP and also other compute optimisations like loop unrolling and array partitioning. We also plan to explore generation of high speed streaming interfaces using SDSoC so that custom stream processors can be integrated with the SDSoC generated streaming interfaces.
- **Machine Learning**: Machine learning is one of the most popular applications requiring extensive parallel computations with a need for high accuracy and speed. We plan to use SDSoC to offload machine learning functions to the FPGA fabric. It is extremely easy to explore the design space using SDSoC due to the high level of programming abstractions.

With respect to OpenCL we can examine the following areas :

- **Memory handling in OpenCL**: One of the most important contributor to time spent for kernel execution is data access. for small datasets it is simple as all the data can be copied to local memory. But in case of large data sets, they have to be efficiently handled and shared so as to minimize the data access overheads.
- **OpenCL for FPGA**: Currently only Altera has developed its version of OpenCL called AOCL that can offload functions to FPGA fabrics in Altera SoCs. There is no open source feature of detecting FPGA as a device for other SoCs. We plan to develop OpenCL drivers for SDSoC generated hardware accelerators.
- **Machine Learning**: Applications like convolutional neural networks (CNN) can be accelerated by running the convolution kernels on target device while all the convolution layer maps can be handled at the host side.

Thus, working towards achieving the set milestones will aid us in bringing to light the techniques that will help extract maximum performance using SDSoC and OpenCL. This in turn will help in popularizing SDSoC and OpenCL as good choices for heterogeneous platform computing.

# Bibliography


[1] End of moore's law. [Online]. Available: http://www.telegraph.co.uk/technology/2016/02/25/end-of-moores-law-whats-next-could-be-more-exciting/

[2] M. M. Waldrop, "The chips are down for moores law," *Nature News*, vol. 530, no. 7589, p. 144, 2016.

[3] Wiki on heterogeneous computing. [Online]. Available: https://en.wikipedia.org/wiki/Heterogeneous_computing?

[4] Amd's "what is heterogeneous computing?". [Online]. Available: http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-computing/

[5] B. S.Z.Ahmad, S.R.Xu and S.F.Ahmad, "Research and implementation of hybrid parallel computing for force field calculation," *Proceedings of the International Conference EITI 2014, Shenzhen, 16-17 August 2014*, pp. 21–24, 2014.

[6] V. H. Naik and C. S. Kusur, "Analysis of performance enhancement on graphic processor based heterogeneous architecture: A cuda and matlab experiment," in *Parallel Computing Technologies (PARCOMPTECH), 2015 National Conference on*, Feb 2015, pp. 1–5.

[7] H. Yazdanpanah, A. Shouraki, and N. Jamali, "Evaluation performance of task scheduling algorithms in heterogeneous environments," *Evaluation*, vol. 138, no. 8, 2016.

[8] Introduction to openmp. [Online]. Available: https://en.wikipedia.org/wiki/OpenMP

[9] Cuda programming model by nvidia. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[10] F. S. Amit Kumar, *IMAGE PROCESSING IN DIABETIC RELATED CAUSES*, 1st ed., ser. Springer Briefs in Forensic and Medical Bioinformatics. Springer, 1 2016, vol. 4.

[11] R. E. W. Rafael C Gonzalez, *Digital Image Processing*, 3rd ed., ser. Prentice-Hall. Inc. Upper Saddle River, NJ, USA: Pearson Education, 1 2008, vol. 4.

[12] Gray level transformation. [Online]. Available: http://www.tutorialspoint.com/

[13] S. E. Gianelli, "Xilinx announces sdsoc development environment for all programmable socs and mpsocs," *Embedded Vision Alliance*, 2015.

[14] Xilinx targets software developers with sdsoc. [Online]. Available: http://www.embedded.com/electronics-blogs/max-unleashed-and-unfettered/4438849/Xilinx-Targets-Embedded-Software-Developers-with-SDSoC

[15] *Zynq-7000 All Programmable SoC Technical Reference Manual.*

[16] *SDSoC Environment User guide.*

[17] *SDSoC Environment User guide.*

[18] Cell sdk by ibm. [Online]. Available: http://www.ibm.com/developerworks/power/cell/index/

[19] Opencl programming book. [Online]. Available: https://www.fixstars.com/

[20] J. Tompson and K. Schlachter, "An introduction to the opencl programming model," *Person Education*, 2012.

[21] *The OpenCL Specification.*

[22] *Getting started with Xillinux for Zynq-7000 EPP.*

[23] Portable computing language. [Online]. Available: http://pocl.sourceforge.net

[24] Installing pocl and papi. [Online]. Available: https://github.com/umaurmi/OPENCL_EXAMPLES_ZEDBOARD