Hardware Acceleration of Most Apparent Distortion Image Quality Assessment Algorithm on FPGA Using OpenCL

by

Aswin Gunavelu Mohan

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2017 by the
Graduate Supervisory Committee:

Sohum Sohoni, Co-Chair
Fengbo Ren, Co-Chair
Jae-sun Seo

ARIZONA STATE UNIVERSITY

May 2017

ABSTRACT

The information era has brought about many technological advancements in the past few decades, and that has led to an exponential increase in the creation of digital images and videos. Constantly, all digital images go through some image processing algorithm for various reasons like compression, transmission, storage, etc. There is data loss during this process which leaves us with a degraded image. Hence, to ensure minimal degradation of images, the requirement for quality assessment has become mandatory. Image Quality Assessment (IQA) has been researched and developed over the last several decades to predict the quality score in a manner that agrees with human judgments of quality. Modern image quality assessment (IQA) algorithms are quite effective at prediction accuracy, and their development has not focused on improving computational performance. The existing serial implementation requires a relatively large run-time on the order of seconds for a single frame. Hardware acceleration using Field programmable gate arrays (FPGAs) provides reconfigurable computing fabric that can be tailored for a broad range of applications. Usually, programming FPGAs has required expertise in hardware descriptive languages (HDLs) or high-level synthesis (HLS) tool. OpenCL is an open standard for cross-platform, parallel programming of heterogeneous systems along with Altera OpenCL SDK, enabling developers to use FPGA's potential without extensive hardware knowledge. Hence, this thesis focuses on accelerating the computationally intensive part of the most apparent distortion (MAD) algorithm on FPGA using OpenCL. The results are compared with CPU implementation to evaluate performance and efficiency gains.

DEDICATION

I would like to dedicate this thesis to my Mom and Dad. Thanks for your continuous love and support. You've always been behind me and pushed me to be the best that I can be. A special mention to my brother for his moral support and encouragement.

ACKNOWLEDGMENTS

I would like to thank the professors I have worked with on the project that resulted in the work presented in this thesis. First and foremost, I would like to thank Dr. Sohum Sohoni and Dr. Fengbo Ren for giving me this opportunity to work on this project. Their technical advice was essential to the completion of this research and has taught me valuable lessons and insights on the workings of academic research in general. My thanks and appreciation to Vignesh and Mike for persevering with me as my mentor throughout the time it took me to complete this research and write the thesis. I will never forget the all-nighter before my defense. The members of my thesis committee, Dr. Jae-sun Seo, have generously given their expertise to better my work. I thank them for their contribution and their good-natured support. I must acknowledge all my colleagues, students, and teachers who assisted, advised, and supported my research and writing efforts over the years.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

FPGA Based Acceleration

In 1965, Gordon Moore predicted that the number of transistors in a dense integrated circuit would double approximately every eighteen months (Moore, G. E., 1998) Ever since, the semiconductor industry has proved his prediction correct. This has led to an extraordinary improvement in computation power and semiconductors have become increasingly cost effective.

However, in the last few years, reducing the transistor size has become increasingly difficult (Huang, A., 2015). On the other hand, the demand for high performance and power efficient microprocessors is increasing with the emerging applications in various fields such as machine learning, computer graphics, and data-mining. In a few years, simply adding more cores and memory into a processor may no longer effectively increase the performance. Thus, an alternative solution will be necessary to overcome the increasing computational demands. The introduction of the recent generation of Field Programmable Gate Arrays (FPGAs) with built-in floating point DSP blocks enables FPGAs to accelerate computationally intensive problems, and compete with traditional CPU and GPU based computing platforms.

Unlike a CPU or a GPU, an FPGA does not have an instruction set or fixed pipeline built-in. Instead, it has a large amount of reconfigurable logic that could be configured to perform any digital logic function. The advantage of FPGA is that when solving different problems, an FPGA could be customized to efficiently solve each of the problems, and achieve much faster speed and energy efficiency than CPU or GPU. At the same time, comparing to Application Specific Integrated Circuits (ASICs), FPGAs cost far less to develop. The downside of FPGA is that traditionally, FPGA requires hardware description languages (HDLs) to program, which is tedious to develop. The developer should design a highly-detailed description of the hardware architecture before implementing on the FPGA. Thus, FPGA development requires extensive hardware knowledge, and the development time is often far longer than developing software for CPUs or GPUs.

Altera (Intel FPGA) SDK for OpenCL aims to reduce the difficulty of deploying FPGA computing solutions and makes FPGA a more favorable computing platform. OpenCL stands for Open Computing Language, which is an industry-standard parallel programming language for heterogeneous systems. The OpenCL is supported by most CPU and GPU vendors from the past, and the recent introduction of Altera SDK for OpenCL (AOCL) extended its support to FPGA as well (Tang, Q. Y., 2016).

Image Quality Assessment Algorithms

Image quality assessment (IQA) algorithms evaluate the visual quality of an image, to that of the human visual system. Using human observers to assess the quality of an image is time-consuming when compared to automatic evaluation using IQA algorithms. Besides, different individuals can evaluate the same image differently. Hence automatic image quality assessment algorithms gained importance. IQA algorithms are classified into three categories based on the availability of the reference image. They are Full Reference (FR), No Reference (NR), and Reduced Reference (RR). Full reference IQA algorithms require both the reference and the distorted image as input. A simple FR-IQA algorithm calculates the mean squared error (MSE) between the two images using the error value for every pixel location. Full reference algorithms have applications in image compression (Charrier, C et al., 2010), image acquisition (Lee, Y. H., Khalil-Hani, M., Bakhteri, R., & Nambiar, V. P., 2016), and television.

No Reference (NR) IQA algorithms, on the other hand, do not require a reference image, which makes it a complex algorithm. However, it is a closer model to the real world applications, where the reference image is not always available. In photography, we do not have a reference image to evaluate its quality, so we need No Reference (NR) IQA to predict the quality (Kamble, V., & Bhurchandi, K. M., 2015). Reduced reference (RR) IQA algorithms fall between the Full Reference and the No Reference algorithms, as they do not require the full reference image, but rather they need some features from the reference image for evaluating the distorted image. Reduced Reference has its applications in remote sensing (Buiten, H. J., & Van Putten, B., 1997) and satellite imaging.

With the development of many new technologies in the past few decades, there is an exponential increase in the creation of digital images and videos. To maintain the quality of the images and videos, we need to evaluate it constantly. An IQA algorithm with high predictive performance requires an execution time in the order of seconds for a single image (Phan, T., Sohoni, S., Chandler, D. M., & Larson, E. C., 2012). There is a need to evaluate and improve the runtime performance of IQA algorithms to meet the demands of the real-time applications.

In order improve the runtime performance of the IQA algorithms, Jain, R. K.(1992), Zhao et al.(2005) did performance analysis to understand the bottlenecks affecting its performance. Microarchitectural hotspot analysis of image quality assessment algorithms on CPU was performed by Phan et al. (2014), which showed that almost all the algorithms have bottleneck related to memory hierarchy and computation and so, the authors proposed microarchitecture conscious coding techniques to improve its performance.

There are two methods by which we can improve the performance of an algorithm (Kannan, V., 2016): Algorithm based techniques and underlying hardware-based techniques.
Software or algorithmic techniques such as accelerating discrete cosine transform (DCT) by using variations of Fast Fourier Transform (FFT) is one such example (Chen, W. H., Smith, C. H., & Fralick, S. C., 1977). Discrete cosine transforms are the essence to most of the IQA algorithms.
Hardware based acceleration techniques based on GPU (Okarma, K., & Mazurek, P., 2011) and FPGA implementations (Alam, S. R et al., 2007) have yielded excellent speedups. Holloway et al. (2016) implemented a GPU-based acceleration for most apparent distortion (MAD) (Larson, E. C., & Chandler, D. M., 2010) exploiting the massive parallelism of the GPU. The GPU implementation showed a 24x speedup and 33x speedup by using multiple GPUs at the same time.

Related Work

Ayat, S. O., Khalil-Hani, M., & Bakhteri, R.(2015) presented that OpenCL hardware-software co-design is effective in realizing parallel architecture designs in heterogeneous FPGA platforms. The Sobel filter algorithm was implemented using OpenCL, and it was found that increasing the filter size from 3x3 to 5x5 resulted in only 11.3% increase in computation time for FPGA, but the execution time was as high as 23.6% and 85.7% for CPU and GPU, respectively.
Tang, Q. Y. (2016) studied various algorithms like k-means clustering, k-nearest neighbor search, N-body simulation and LU decomposition and found that the OpenCL FPGA implementation of k-means clustering and k-nearest neighbor kernels significantly out-

performed the optimized CPU implementations while achieving better power efficiency than that of GPU.

Suda et al (2016) presented that a systematic design space exploration methodology can be used to get a higher throughput for a Convolution Neural Networks (CNN) model implemented using OpenCL FPGA. The proposed methodology is demonstrated by optimizing two large-scale CNNs, AlexNet and VGG. A peak performance of 117.8 GOPS was achieved for the entire VGG network that performed ImageNet Classification.

Abdelfattah et al (2014) implemented a high-speed lossless data compression (Gzip) on an FPGA using OpenCL. Their work shows how we can make use of a heavily-pipelined custom hardware implementation to achieve a high throughput of approximatly 3 GB/s with more than 2x compression ratio over standard compression benchmarks. The implementation had 12x improvement in performance-per-watt when compared to the highly tuned CPU implementation. Additionally, they had also compared the OpenCL FPGA implementation to a hand-coded commercial implementation of Gzip to quantify the gap between OpenCL and a hardware description language (HDL) like Verilog. When compared to Verilog, OpenCL's performance was 5.3% lower, and there was 2% and 25% increase in logic resources and memory resources available in the FPGA, respectively, demonstrating significant productivity gain.

Although there are not many significant works in this field, current studies show that OpenCL based implementation on FPGA has improved performance and power efficiency over CPU and GPU implementations, and it also has a signficant productivity gain over HDL implmentation. Hence, in this thesis, the hotspot function of MAD algorithm was implemented on FPGA using OpenCL to evaluate its performance.

OpenCL Overview

OpenCL is an open standard targeted for general-purpose parallel programming on different types of processors. The goal of OpenCL is to provide software developers a standard framework for easy access to heterogeneous processing platforms. The OpenCL standard specifies a set of API and a programming language based on C.

The OpenCL specification can be the described by the following four models.

- Platform Model
- Memory Model
- Exection Model
- Programming Model

Platform Model

Platform defines the relation between the host and the device. Mainly there is one processor the host and there are one or more devices capabale of executing OpenCL kernel code. The platform models defines an abstact hardware mode used by the programmers while using OpenCL.

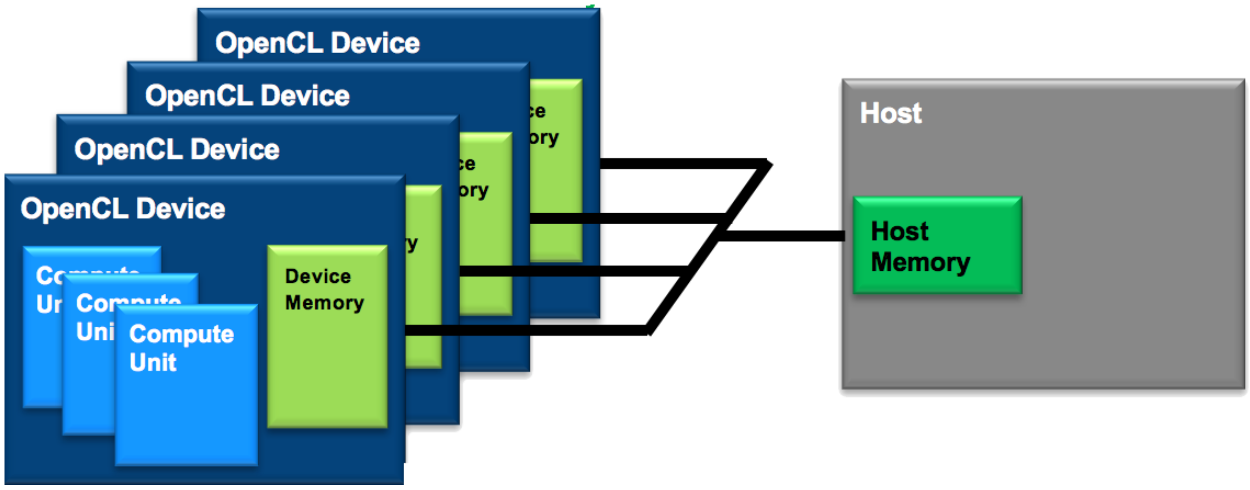


Figure 1 Describes the Platform Model (Intel Best Practice Guide. 2016).

Each OpenCL device has one or more Compute Units (CU), and each CU has one or more Processing Elements (PE). The actual computation is done on the PEs.

Execution Model

The execution model of an OpenCL application has two parts. They are host and kernel; kernel part executes on the devices and the host part that executes on the host. The host part manages the kernels and the memory objects under a context through command queues.

Program and Memory Object

The program object consists of the source code and the binary implementation of the kernels. During the execution of the application, the binary implementation can be generated from the source code in the case of GPU or CPU, or a pre-compiled binary can be loaded to create the program object in the case of FPGAs. A program object can be considered as a library for kernels because one program object can contain multiple kernels. The host application decides which kernel to execute during runtime.

The memory objects are visible to both the host and the kernels and used to transfer data between the host and the device. The host creates memory objects by using the OpenCL API functions to allocates memory on the device for the memory objects. The details of the memory model are described in below section.

Command Queue

Each device in the context has an associated one or more command queue, and kernel execution and memory transfer are coordinated using the command queue. There are three types of commands that can be issued. Memory commands are used to transfer memory between the host and the device. Kernel commands are used to start the execution of kernels on the device. Synchronization commands can be used to control the execution order of the commands.

Context

Contexts are abstract containers that manage host device interaction. This includes keeping track of memory object, compiling programs, extracting kernels, and managing a queue for all action needed to be performed by the device.

Kernel

Kernel are the computation that is executed on the processing elements. They are the functions that are executed on the devices. The kernels are extracted from program objects which are either compiled on run time for GPUs and CPUs or precompiled as in the case of FPGAs. The kernels can be instantiated N number of times or only once based on the requirements for the current application from the host program.

Work Items and Work Groups

An N-dimensional indexed space is defined for the execution of the kernel, and one kernel instance is executed for each value in the indexed space. The value of N can be one, two, or three. Each kernel instance is called a work-item. All the work-items execute the same code. However, they work on a different set of data. Each work-item is assigned a global ID that is unique across the index space. Work-items can be grouped together to form work-groups, with all the work-groups has a local ID that is unique within the work-group. Work-group members also have access to shared local memory.

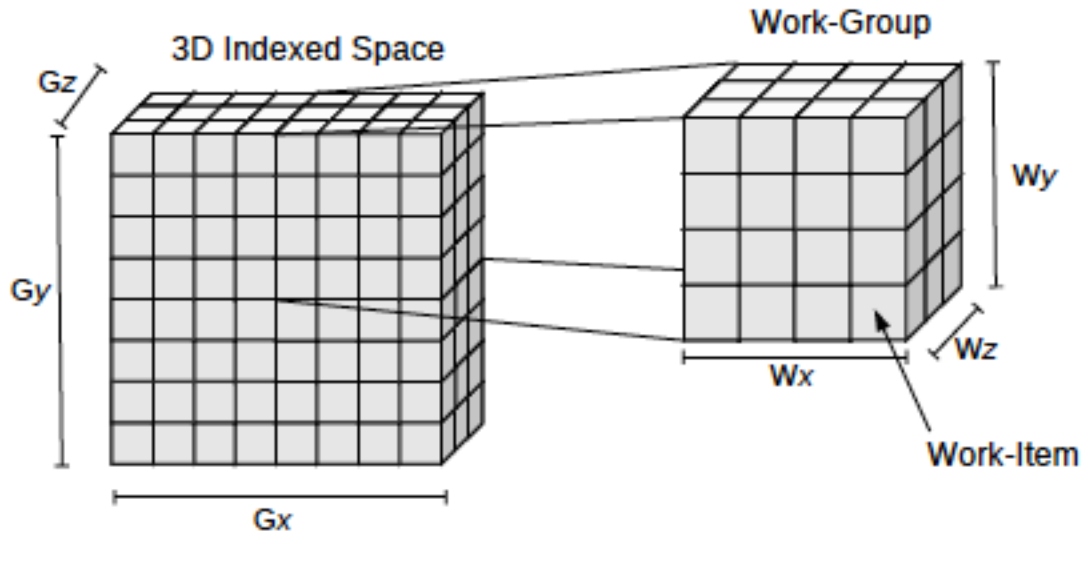


Figure 2 N-dimensional indexed space is defined for the execution of the kernel

Memory Model

OpenCL has four types of memory classified based on their memory access capabilities of the work items.

Global Memory: A region of memory that allows read/write access to all the work-items. Usually, the input data for the kernels are written into this region. Global memory reads and writes may be cached depending on the capability of the device.

Constant Memory: A region of global memory that remains constant during the execution of the kernel and it is accessible to all the work-items. The host program allocates and initializes the memory objects placed in the constant memory.

Local Memory: A region of memory that is local to a work-group. All the work-items in a work-group shares this memory region. This memory allows work-items to share the share the variables within a work-group.

Private Memory: A region of memory that is local to a work-item. Each work-item has its copy of the variable that is defined by using the private memory.

Figure 1 is Figure 3.3 in (OpenCL Specification 1.1, 2011) describes the OpenCL device architecture with all memory regions along with processing elements (PE) and compute units (CU) in the device. We can see the memory access made by the processing elements and compute units on different types of memory regions.

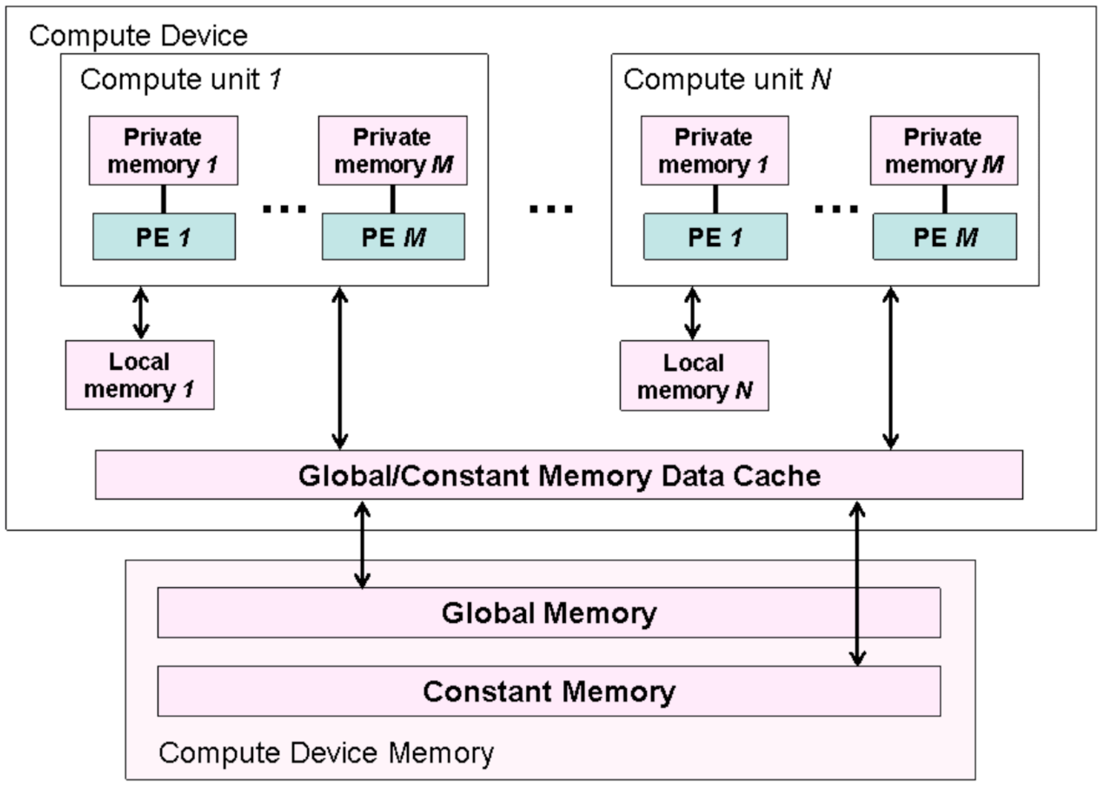
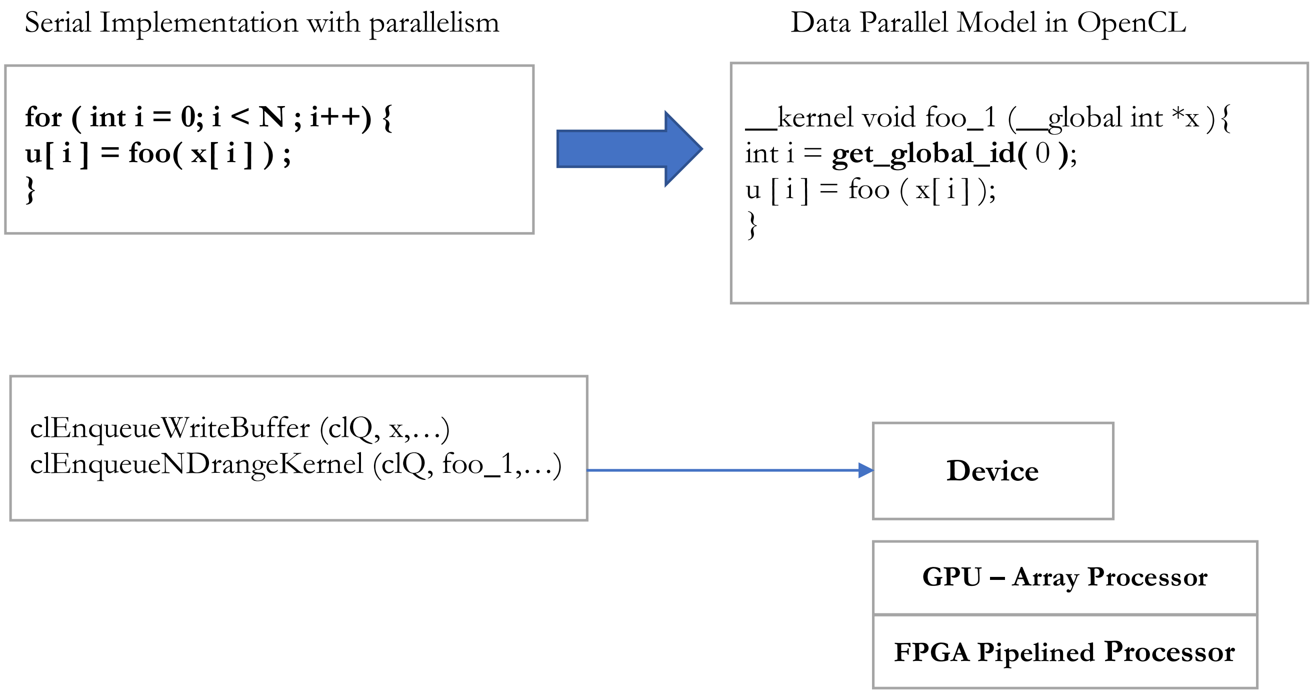


Figure 3 OpenCL device architecture with the memory regions. (OpenCL Specification 1.1, 2011)

Programming Model

In OpenCL programming model, computations can be done in data parallel, task parallel or a hybrid of both models.

Data Parallel Model – Each work-item works on a data item. Executes single kernel with multiple threads like Single Instruction Multiple Data (SIMD).



Serial Implementation with parallelism
Data Parallel Model in OpenCL
Device
GPU – Array Processor
FPGA Pipelined Processor

```
for ( int i = 0; i < N ; i++) {
u[ i ] = foo( x[ i ] ) ;
}
```

```
__kernel void foo_1 (__global int *x ){
int i = get_global_id( 0 );
u [ i ] = foo ( x[ i ] );
}
```

```
clEnqueueWriteBuffer (clQ, x,…)
clEnqueueNDrangeKernel (clQ, foo_1,…)
```

Figure 4 Execution of a data parallel model (Intel Best Practice Guide. 2016).

Task Parallel – In this model we can execute multiple kernels at the same time, where each kernel can have only one work item. This model is a very efficient model for FPGAs. It is also possible to have a hybrid model where multiple kernels can have multiple work-items for execution at the same time.

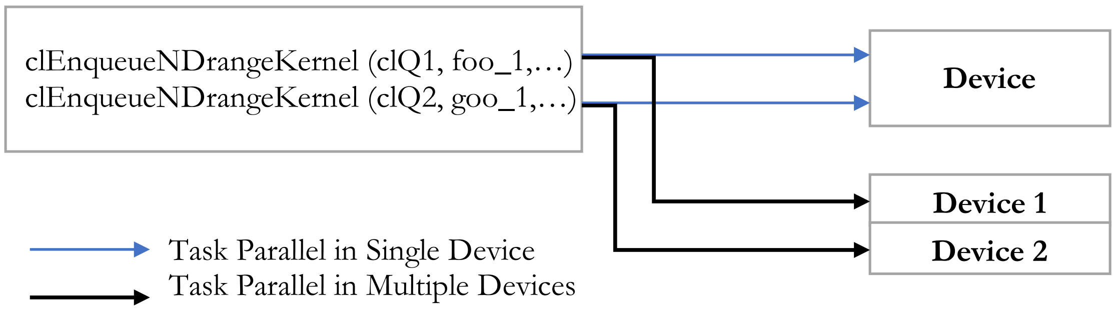

clEnqueueNDrangeKernel (clQ1, foo_1,…)
clEnqueueNDrangeKernel (clQ2, goo_1,…)
Device
Device 1
Device 2
Task Parallel in Single Device
Task Parallel in Multiple Devices

Figure 5 Execution of a task parallel model (Intel Best Practice Guide. 2016).

OpenCL Application Flow

The OpenCL application flow is illustrated in the Figure below. They are given step numbers for reference in the following discussion. The flow can be split into two parts. The platform layer which creates a context based on the available platform, and the runtime layer which creates all the other necessary objects to execute the kernel. The Figure 6 is from Ahmed, T., 2011.

Platform Layer

Step 1. An OpenCL application queries for the OpenCL platforms available.

Step 2. The application selects the one with the desired device type from the platform list. The device types allowed in the OpenCL specification are CL DEVICE TYPE CPU, CL DEVICE TYPE GPU, and CL DEVICE TYPE ACCELERATOR.

Step 3. With the desired number of devices from the available devices, create the context. The devices are made exclusive to the context until the context is released.

Runtime Layer

Step 4. To issue these commands to the devices, a command queue is created for each device selected under the context

Step 5. The memory objects are created to allocate memory on the devices. The read/write permission to these memory objects from the host is set by the application when they are created.

Step 6. The program objects are created by loading the source code or the binary implementation of one or more kernels. Once created, the program objects are then built to generate the device-specific executable.

Step 7. Input data is transferred to the device memory by issuing memory copy commands to the associated memory objects. This is done before executing the kernel.

Step 8. Kernel arguments are set once the input data is transferred to the device.

Step 9. Command queue is used to schedule the kernels for execution.

Step 10. The output memory is transferred to the host from the device once the kernel execution is complete.

Step 11. All the OpenCL objects created in the application are released once the computation is done.

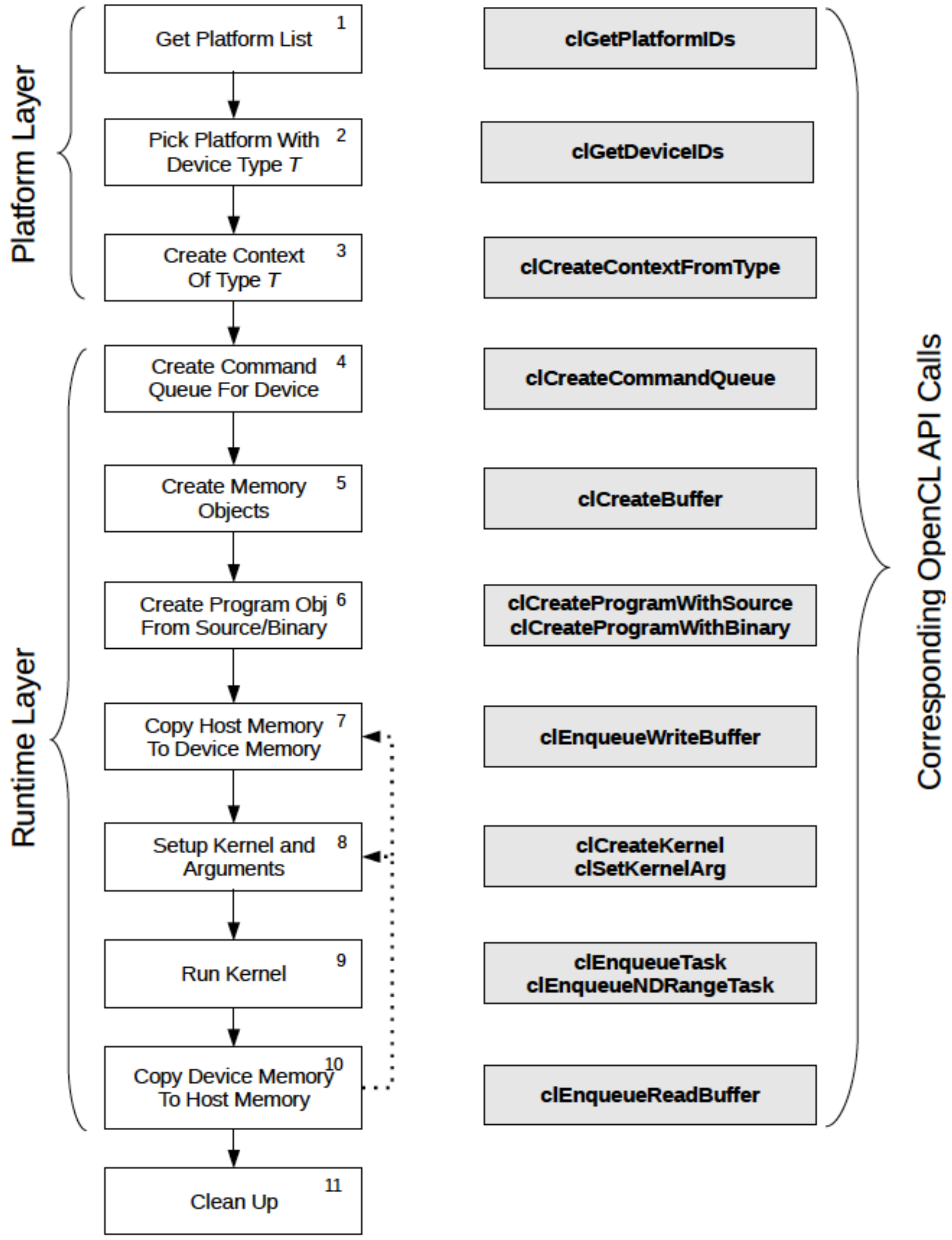

Platform Layer
Runtime Layer
Get Platform List 1
Pick Platform With Device Type T 2
Create Context Of Type T 3
Create Command Queue For Device 4
Create Memory Objects 5
Create Program Obj From Source/Binary 6
Copy Host Memory To Device Memory 7
Setup Kernel and Arguments 8
Run Kernel 9
Copy Device Memory To Host Memory 10
Clean Up 11
clGetPlatformIDs
clGetDeviceIDs
clCreateContextFromType
clCreateCommandQueue
clCreateBuffer
clCreateProgramWithSource
clCreateProgramWithBinary
clEnqueueWriteBuffer
clCreateKernel
clSetKernelArg
clEnqueueTask
clEnqueueNDRangeTask
clEnqueueReadBuffer
Corresponding OpenCL API Calls

Figure 6 Block diagrams showing the OpenCL Application Flow (Ahmed, T., 2011).

CHAPTER 2

ALGORITHM AND ANALYSIS

Most Apparent Distortion (MAD) Algorithm

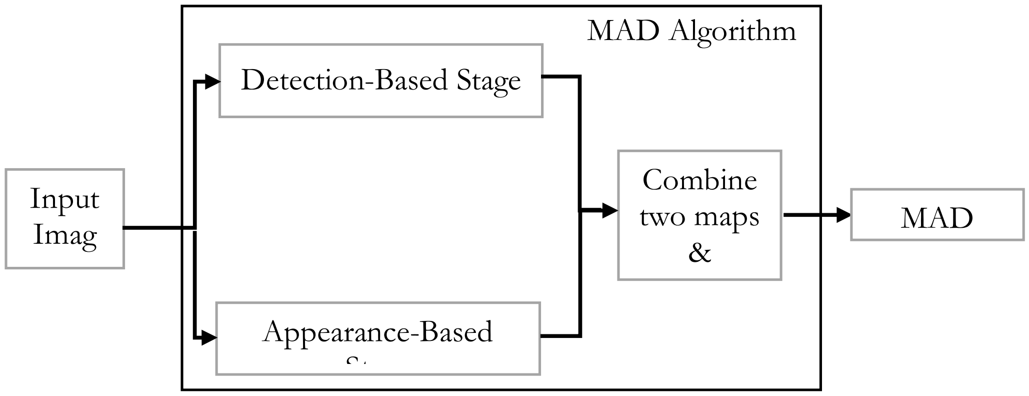


Figure 7 Block diagram of MAD Algorithm

MAD is one of the algorithms which has high estimation accuracy for the visual quality of an image. Most of the IQA algorithms focus on the most dominating strategy used by the Human Visual System (HVS) while MAD uses multiple strategies to determine image quality of an image (Larson, E. C., & Chandler, D. M., 2010). The algorithm uses detection based approach to estimate quality based on the extent to which the distortions are visible, and the appearance-based approach is used to estimate quality based on the extent to which the image is recognizable. The block diagram of the MAD algorithm is shown in Figure 2.1. The algorithm takes two input images: the reference image and the distorted image, the MAD index/score is computed in two main stages, the *detection-based stage* and the *appearance-based stage*. The *detection-based stage* computes the detection-based difference map, the difference between the original and the distorted images. Also, the *appearance-based stage* computes the appearance-based difference map using mean, variance, skewness, and kurtosis for all local blocks of the log-Gabor filtered images. The detection-based difference and appearance-based difference maps are combined to get the high-quality and low-quality indexes. The weighted geometric mean of the indexes is computed. The final MAD index is computed using the weighted mean and a specific weight determined based on the amount of distortion. In this thesis, our focus is on the *appearance-based stage* of the algorithm; we refer readers to (Larson, E. C., & Chandler, D. M., 2010)s for further details on *detection-based stage*.

Appearance-based stage:

The sub-stages of the *appearance-based stage* are shown in the Figure 2.2. The block diagram in Figure 2.2 was drawn with the referenced from (Phan, T., 2012; Holloway, J, et al., 2016). There are four main operations stages. The *Build log-Gabor and Filter Input* images filter the image using log-Gabor filter applied using FFT. The *inverse 2D-FFT* calculates the inverse FFT and the magnitude of the filtered image. The *compute statistical difference map* computes the variance, skewness, and kurtosis for each 16x16 block (with 75% overlap between blocks). The first three sub-stages are executed for each log-Gabor filter. We have filters for five scales and four orientations which are 20 filters for each image. Finally, the *combine difference maps*, combines all the difference maps from each filter to one difference map.

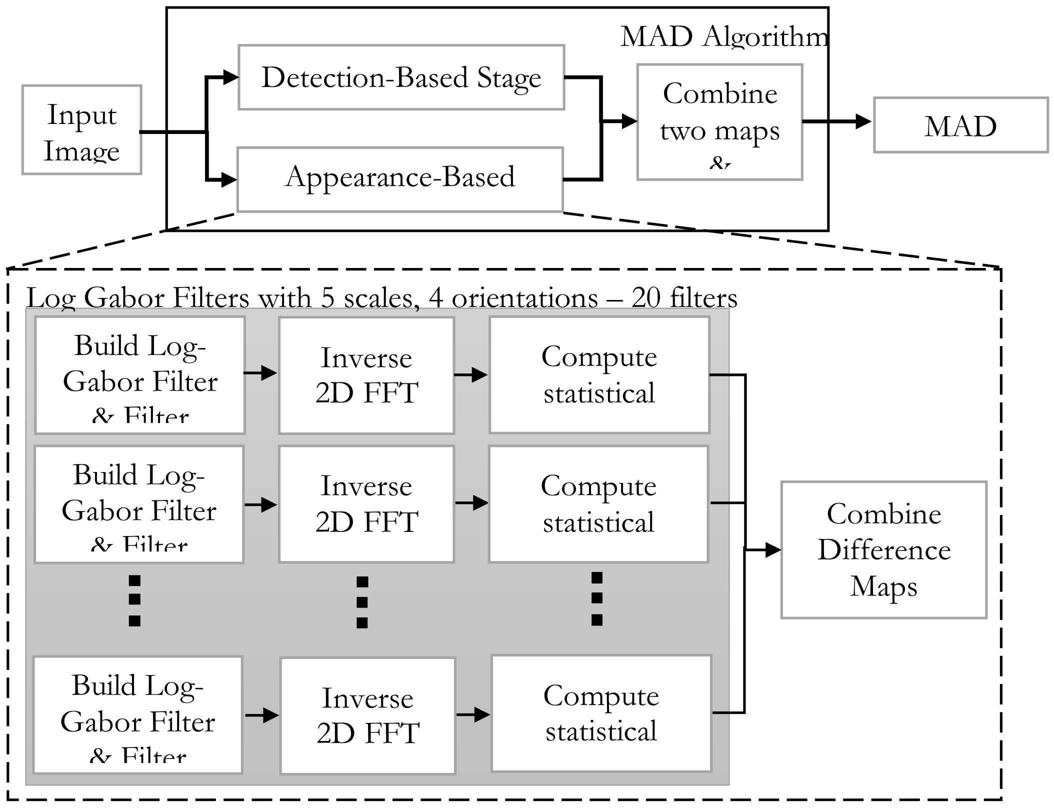


Figure 8 Block diagram showing the sub-stages of the appearance-based stage

The MAD code is ported to C++ using the Matlab version, which is available publicly to download (MAD Matlab Implementation. 2011). The input images are stored in 2-D arrays using GBuffer image library. In the detection-based stage, the images are taken to the luminance domain using a look-up table. The Ooura's mathematical packages (Ooura's Mathematical software packages) are used for calculating Fast Fourier Transform and inverse FFT.

Preliminery analysis of MAD hotspot on a CPU

In this section, we do the preliminary analysis on the C++ implementation on the CPU to determine the top hotspot function, that is to be accelerated using OpenCL FPGA. We discuss the top hotspot and their corresponding execution time for MAD. From the hotspot analysis, we find that MAD has a very high execution time. The average total execution time and average execution time for individual hotspots, along with their % contribution to the total execution time can be found in the Table 1.

Table 1

Average execution time for the Hotspot function, also expressed as percentage of total execution time.

Function/Block	**Average total execution time(ms)**	**% of total execution time**
All	5058	100
Detection-Based Stage	527	10
Appearance-Based Stage	4531	90
• **Statistical Computation**	2824	56
• **Inverse 2D FFT**	917	18
• **Log Gabor Block**	548	11
• **Combine Difference Maps**	242	5

As seen from the table, the top hotspot function *appearance-based stage* takes about 90% of the total execution time. So, to significantly improve the performance of the total execution time, we focus on the *appearance-based stage*. On further analysis of the hotspots in the *appearance-based stage*, the *statistical computation* function takes about 56% of the total execution time. It calculates the statistical difference map using variance, skew and kurtosis of the gabor filtered images (Larson, E. C., & Chandler, D. M., 2010). Then comes *inverse 2d-FFT* which takes about 18% of the total execution time. It calculates the inverse 2d-FFT of the gabor filtered images. Next is the *log gabor block*, which takes about 11% of the total execution time. It calculates the gabor filtered images from the input images using FFT. The final hotspot function is *combine difference maps*, which takes a total of 20 maps for each reference and distorted image, computes the difference and then combines them to one difference map. It takes about 5% of the total execution time.

Based on the hotspot analysis, we plan to implement the top hotspot, the statistical computation which takes about 56% of the total execution time in FPGA using OpenCL and evaluate the performance of the functions in hardware.

CHAPTER 3

IMPLEMENTATION AND RESULTS

Implementation

This chapter describes the implementation of the statistical computation block of the appearance-based stage from the MAD algorithm in FPGA using OpenCL. We apply various methods to improve the performance of the implementation.

In short, *statistical computation* performs the following:

1. Iterates *i* and *j* values using a nested loop; *i* and *j* are incremented by 4 for every iteration. The *i* and *j* values are used to select the 16x16 block for the block-based statistical computation.
2. Calculates the sum of all the elements using a nested loop for the 16x16 block.
3. Calculates the mean using the sum.
4. Calculates standard deviation, kurtosis, and skewness using nested loop for the 16x16 block with the mean value.

OpenCL Methods of Kernel Implementation

- Multi-Threaded Model
- Single Threaded Model

Multi-threaded Model:

To implement a multi-threaded model, the kernels are mapped by replicating hardware in FPGAs, but in the case of GPUs and CPUs, we make use of the available hardware. It is not possible to replicate hardware for each work-item(thread) which would be a waste of resources in the FPGA. Usually, the number of work-items is known only at the runtime, and it is not possible to replicate the required hardware during that time.

We can replicate the hardware to process multiple work-items in parallel while compiling using Altera Offline Compiler's (AOC) preprocessor directives. Still, the number of replications are very limited.

In FPGAs, we can take advantage of the pipelining parallelism by creating a deeply pipelined representation of the kernel. On each clock, an input data from a new thread is sent into the kernel.

For example, let's take vector addition; on a given cycle, each portion of the pipeline is processing different threads. From the figure below, we can see that while the work-item 2 is being loaded, the work-item 1 is being added and the work-item 0 is being stored.

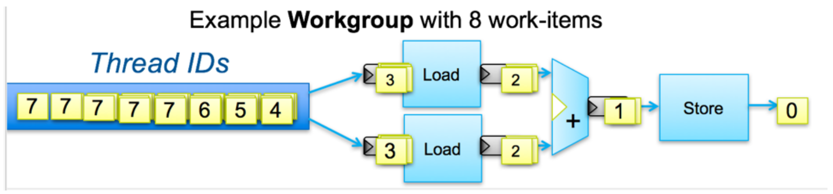


Figure 9 Multi-threaded model using vector add (Intel Best Practice Guide. 2016).

Basic Implementation:

The basic implementation is the kernel function of the statistical computation in Table 2 which was ported to OpenCL with the references from the C++ implementation of MAD (Phan, T., Sohoni, S., Chandler, D. M., & Larson, E. C., 2012) and GPGPU implementation of MAD (Holloway, J, et al., 2016).

Changes made to parallelize the C++ implementation:

From the function description in the section 3.1, we know that computation is based on the *i* and *j* values. The *i* and *j* values can be sent from the host using the Multi-threaded model to the kernel function to do the computation. This removes the outer nested for-loops used for *i* and *j* in the C++ implementation. To index the thread launched by the host, we use the OpenCL function *get_global_id ()* which determines the value of *i* and *j* for the kernel function. The rest of the code is same as that of the sequential implementation.

Table 2

Basic Implementation using Multi-threaded model

Basic Implementation:

```
__kernel void stats_Appearance (int P, __global float *restrict xVal,
                       __global float *restrict Std,
                       __global float *restrict Skw,
                       __global float *restrict Krt)
{
    int index_x = get_global_id(0);
    int index_y = get_global_id(1);
    int index = (index_x * 128) + index_y;
    int i = 4 * index_x;
    int j = 4 * index_y;
    int limit = (P*4)-15;
    int iB,jB;
    float mean_sum=0;
    float mean,tmp_mean,stmp;
    float stdev=0;
    float skw=0;
    float krt=0;
    if (i < limit && j < limit) {
        for (iB = 0; iB < 16; iB++)
        {
            for (jB = 0; jB < 16; jB++)
            {
                mean_sum = mean_sum + xVal[((iB+i) * P * 4) + (jB+j)];
            }
        }
        mean = mean_sum / 256.0f ;
        for (iB = 0; iB < 16; iB++)
        {
            for (jB = 0; jB < 16; jB++)
            {
                tmp_mean = xVal[((iB+i) * P * 4) + (jB+j)] - mean;
                stdev = stdev + (tmp_mean * tmp_mean);
                skw = skw + (tmp_mean * tmp_mean * tmp_mean);
                krt = krt + (tmp_mean * tmp_mean * tmp_mean * tmp_mean);
            }
        }
        stmp = sqrt(stdev / 256.0f);
        stdev = sqrt(stdev / 255.0f);
        if (stmp != 0){
            skw = (skw / 256.0f) / ((stmp)*(stmp)*(stmp));
            krt = (krt / 256.0f) / ((stmp)*(stmp)*(stmp)*(stmp));
        }
        else{
            skw = 0.0f;
            krt = 0.0f;
        }
        Std[index] = stdev;
        Skw[index] = skw;
        Krt[index] = krt;
        }
    else {
        Std[index] = 0.0f;
        Skw[index] = 0.0f;
        Krt[index] = 0.0f;
          }
}
```

Host side Execution:

In the host, the input data is initialized and copied to the device. The kernel is enqueued into the command queue. The kernel starts to execute once its enqueued. The kernel terminates once all the threads launched to the kernel are completed. After this, the results are copied back to the host for verification.

Kernel side Execution:

1. Every thread gives the value for *i* and *j*. The *i* and *j* values are used to select the 16x16 block for the block-based statistical computation.
2. Calculate the sum of all the elements from the global memory using a nested loop.
3. Using the sum, calculate the mean.
4. Using mean value, calculate standard deviation, kurtosis, and skewness using nested loop.
5. Store the calculated values to the global memory.

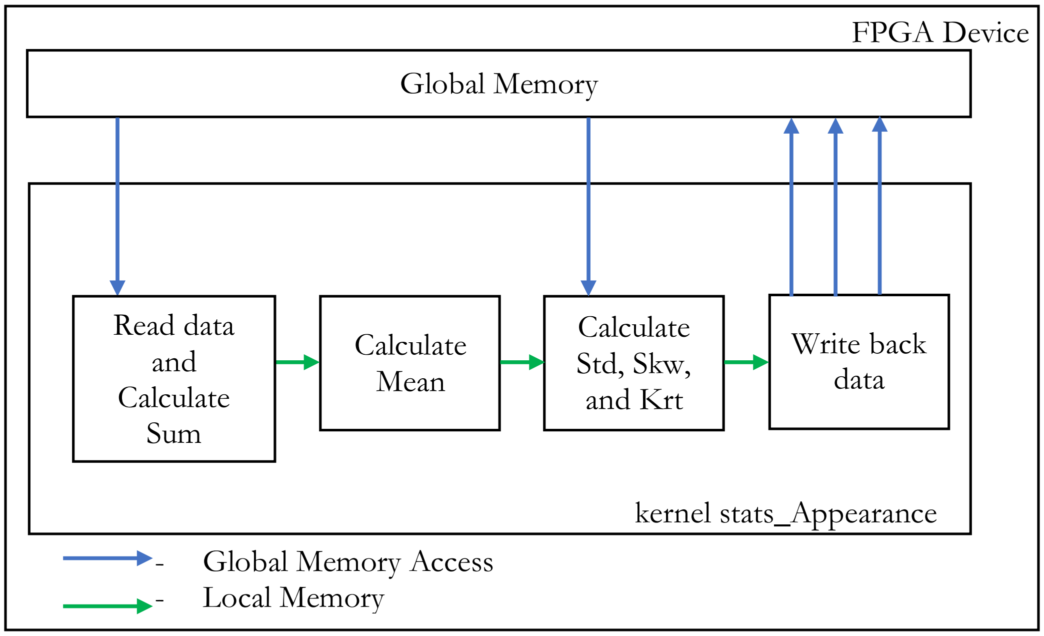


Figure 10 *Block diagram showing device side execution flow – Basic Implementation*

Table 3

Results for the Basic Implementation using Multi-Threaded model

Parameters/Devices	FPGA	CPU
Time	42.33ms	106ms
Frequency	252 MHz	3.5 GHz
Memory Stall	61.25%	NA
Resource Utilization	Logic Utilization = 36%	
	ALUTs = 14%	
	Logic Registers = 22%	
	Memory Blocks = 16%	
	DSPs = 7%	

SIMD Implementation:

To achieve higher throughput, we can vectorize the kernel. Kernel vectorization allows multiple work-items to execute in a single instruction multiple data (SIMD) fashion. We can direct the Intel FPGA SDK for OpenCL Offline Compiler to translate each scalar operation in the kernel, such as addition or multiplication, to an SIMD operation. We can include the num_simd_work_items attribute in the kernel code to direct the offline compiler to perform more additions per work-item without modifying the body of the kernel (Intel Best Practice Guide. 2016).

In the following implementation, we use the num_simd_work_items attribute to 4 and evaluate its performance improvements to that of the basic implementation. By setting the attribute to 4, we inform the compiler to vectorize the kernel to allow four work-items to execute concurrently. The kernel accesses four data from the global memory when compared to the basic implementation. This is indicated by the broad arrows in Figure 11. To accommodate four work-items, the compiler adds the necessary recourses to do so. We can see the increase in the resource utilization in Table 4 when compared to the utilization by the basic implementation from Table 3. There is not any significant improvement in the performance because the kernel is sequential, and using SIMD adds memory access stall to the kernel.

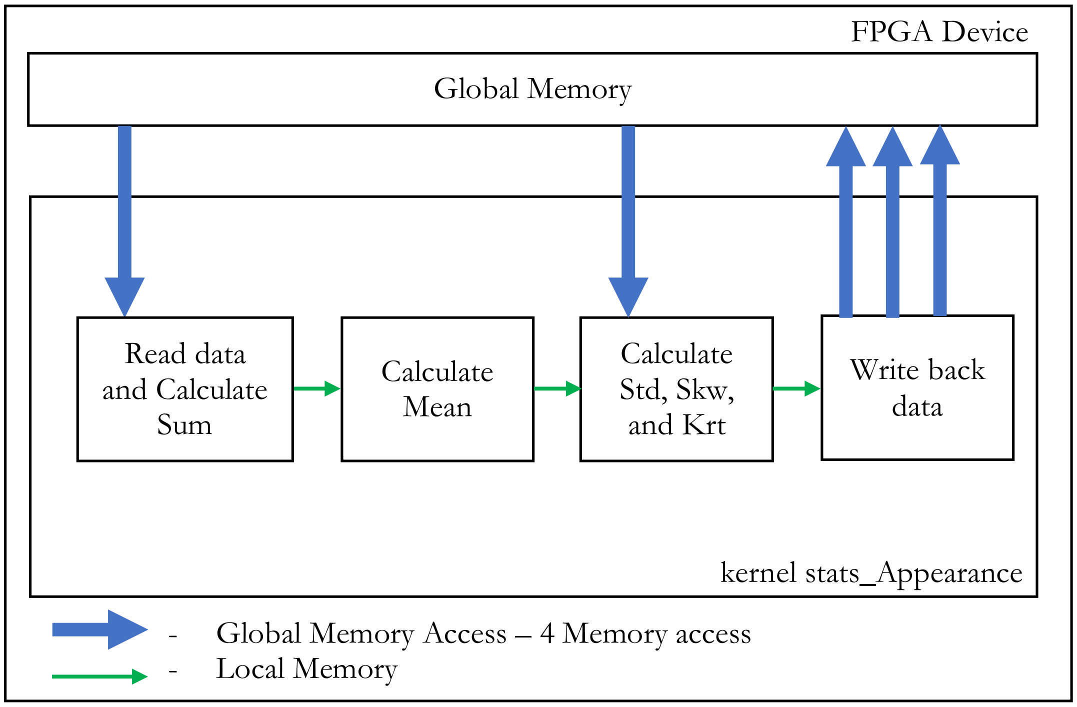

FPGA Device
Global Memory
Read data and Calculate Sum
Calculate Mean
Calculate Std, Skw, and Krt
Write back data
kernel stats_Appearance
- Global Memory Access – 4 Memory access
- Local Memory

Figure 11 Execution flow of SIMD Implementation - Broad arrows shows 4 memory access made by the kernel

Table 4

Results from the SIMD Implementation

Parameters/Devices	FPGA
Time	42.95ms (CPU 106ms)
Frequency	205 MHz
Memory Stall	66.31%
Resource Utilization	Logic Utilization = 46%
	ALUTs = 17%
	Logic Registers = 29%
	Memory Blocks = 30%
	DSPs = 14%

Multiple Compute Units Implementation:

To achieve higher throughput, the Intel FPGA SDK for OpenCL Offline Compiler can generate multiple compute units for each kernel. The hardware scheduler in the FPGA dispatches work-groups to additionally available compute units (Intel Best Practice Guide. 2016)].

In the following implementation, we set the num_compute_units attribute to 4 and evaluate its performance improvements to that of the basic implementation. By setting the attribute to 4, we inform the compiler to create four unique compute units as showed in Figure 12; the work-items in the work group are divided among the compute units.

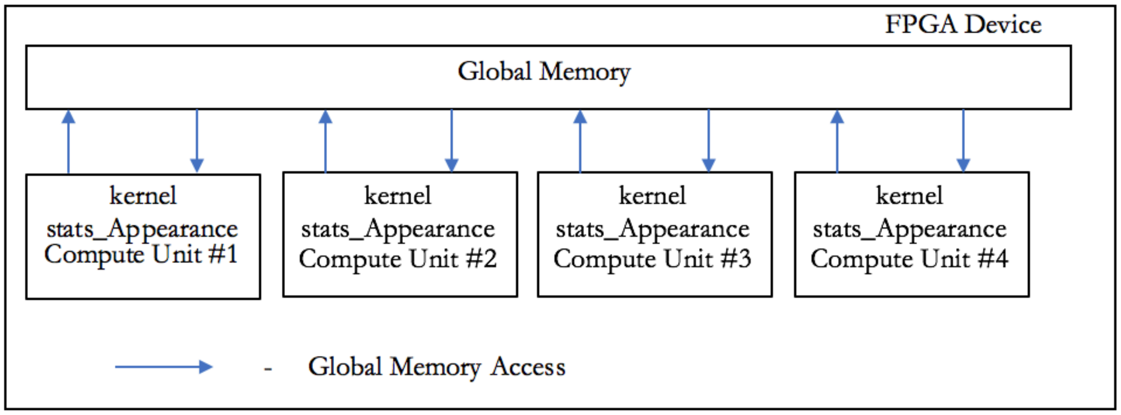


Figure 12 Execution flow Multiple Compute Unit Implementation - Work-items are divided among the 4 CUs

To create four compute units, the compiler adds the necessary recourses to do so. We can see the increase in the resource utilization in Table 5 when compared to utilization of the basic implementation from Table 3 is like that of SIMD implementation. From Table 5, we can see that using compute units has increased the memory stall to 71% which decreases the performance of the kernel.

Table 5

Results for Multiple Compute Unit Implementation

Parameters/Devices	FPGA
Time	46.25ms (CPU 106ms)
Frequency	185MHz
Memory Stall	71.1%
Resource Utilization	Logic Utilization = 48%
	ALUTs = 17%
	Logic Registers = 30%
	Memory Blocks = 30%

Loop Unrolling Implementation:

We can control the way the Intel FPGA SDK for OpenCL Offline Compiler translates OpenCL kernel descriptions to hardware resources. The performance of the loop iterations can be increased by unrolling the loop. Loop unrolling decreases the number of iterations that the offline compiler executes at the expense of increased hardware resource consumption (Intel Best Practice Guide. 2016).

Table 6

Code snippet showing the changes for the Loop Unrolling Implementation - 1

Loop Unrolling Implementation 1

```
__kernel void stats_Appearance (inputs and outputs)
{
Initializations
if (i < limit && j < limit) {
        #pragma unroll 4
        for (iB = 0; iB < 16; iB++)
        {
            #pragma unroll 4
            for (jB = 0; jB < 16; jB++)
            {
                mean_sum = mean_sum + xVal[((iB+i) * P * 4) + (jB+j)];
            }
        }
        #pragma unroll 4
        for (iB = 0; iB < 16; iB++)
        {
            #pragma unroll 4
            for (jB = 0; jB < 16; jB++)
            {
                tmp_mean = xVal[((iB+i) * P * 4) + (jB+j)] - mean;
                stdev = stdev + (tmp_mean * tmp_mean);
                skw = skw + (tmp_mean * tmp_mean * tmp_mean);
                krt = krt + (tmp_mean * tmp_mean * tmp_mean * tmp_mean);
            }
        }
}
```

In Loop Unrolling Implementation 1, as we can see from the code snippet in Table 6, we have unrolled all the loops with a factor of 4. By doing so, from Table 7, we can see that we have got about 4x improvement in performance at the expense of increased hardware utilization.

Table 7

Results for Loop Unrolling Implementation – 1

Parameters/Devices	FPGA
Time	11.778ms (CPU 106ms)
Frequency	208MHz
Memory Stall	65.4%
Resource Utilization	Logic Utilization = 46%
	ALUTs = 17%
	Logic Registers = 29%
	Memory Blocks = 22%
	DSPs = 14%

In Loop Unrolling Implementation 2, as we can see from the code snippet in Table 8, we have changed the loop unroll factor of all the loops to 8. From Table 9, we can see that it took about two times the hardware utilization of the basic implementation to give us about 6x improvement in performance which is very ineffective.

Table 8

Code snippet showing the changes for the Loop Unrolling Implementation - 2

Loop Unrolling Implementation 2

```
__kernel void stats_Appearance (inputs and outputs)
{
Initializations
if (i < limit && j < limit) {
        #pragma unroll 8
        for (iB = 0; iB < 16; iB++)
        {
            #pragma unroll 8
            for (jB = 0; jB < 16; jB++)
            {
                mean_sum = mean_sum + xVal[((iB+i) * P * 4) + (jB+j)];
            }
        }
        #pragma unroll 8
        for (iB = 0; iB < 16; iB++)
        {
            #pragma unroll 8
            for (jB = 0; jB < 16; jB++)
            {
                tmp_mean = xVal[((iB+i) * P * 4) + (jB+j)] - mean;
                stdev = stdev + (tmp_mean * tmp_mean);
                skw = skw + (tmp_mean * tmp_mean * tmp_mean);
                krt = krt + (tmp_mean * tmp_mean * tmp_mean * tmp_mean);
            }
        }
}
```

Table 9

Results for Loop Unrolling Implementation - 2

Parameters/Devices	FPGA
Time	6.7ms (CPU 106ms)
Frequency	204 MHz
Memory Stall	66.2%
Resource Utilization	Logic Utilization = 70%
	ALUTs = 26%
	Logic Registers = 44%
	Memory Blocks = 42%
	DSPs = 36%

In Loop Unrolling Implementation 3, we made changes to code based on the fully pipelined model from the single-threaded implementation. The inner loops were fully unrolled. From Table 11, we can see that we got about 10x improvement in performance by utilizing lesser hardware resources compared to its previous implementations.

Table 10

Code snippet showing the changes for the Loop Unrolling Implementation - 3

Loop Unrolling Implementation 3

```
__kernel void stats_Appearance (inputs and outputs)
{
Initializations
if (i < limit && j < limit) {
        #pragma unroll 1
        for (iB = 0; iB < 16; iB++)
        { mean_sum1=0;
            #pragma unroll 16
            for (jB = 0; jB < 16; jB++)
            {
                mean_sum1 = mean_sum1 + xVal[((iB+i) * P * 4) + (jB+j)];
            }
            mean_sum+=mean_sum1;
        }
        #pragma unroll 1
        for (iB = 0; iB < 16; iB++)
        {
             stdev1=0;
             skw1=0;
             krt1=0;
            #pragma unroll 16
            for (jB = 0; jB < 16; jB++)
            {
                tmp_mean = xVal[((iB+i) * P * 4) + (jB+j)] - mean;
                stdev1 = stdev1 + (tmp_mean * tmp_mean);
                skw1 = skw1 + (tmp_mean * tmp_mean * tmp_mean);
                krt1 = krt1 + (tmp_mean * tmp_mean * tmp_mean * tmp_mean);
            }
            stdev+=stdev1;
            skw+=skw1;
            krt+=krt1;
        }
}
```

Table 11

Results for Loop Unrolling Implementation – 3

Parameters/Devices	FPGA
Time	4.41ms (CPU 106ms)
Frequency	250MHz
Memory Stall	68.1%
Resource Utilization	Logic Utilization = 39%
	ALUTs = 15%
	Logic Registers = 24%
	Memory Blocks = 19%
	DSPs = 15%

Single Threaded Model:

Single Threaded Model is equivalent to launching a multi-threaded model kernel with only one work-item (thread). The single-threaded kernel follows the sequential model similar to C programming. It is used when the algorithm is not data parallel. Some algorithms are inherently sequential and depend on previous results. Similar to the multi-threaded model, the compiler will infer pipelined execution across loop iterations as shown in Figure 13.

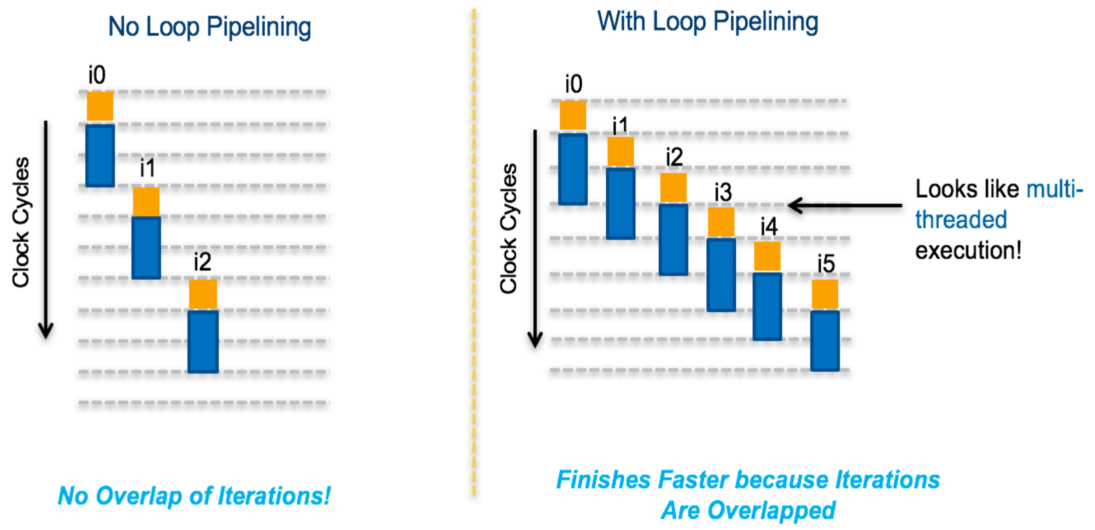

No Loop Pipelining
i0
i1
i2
Clock Cycles
No Overlap of Iterations!
With Loop Pipelining
i0
i1
i2
i3
i4
i5
Clock Cycles
Looks like multi-threaded execution!
Finishes Faster because Iterations Are Overlapped

Figure 13 Execution of iteration based on No Loop Pipelining and With Loop Pipelining (Intel Best Practice Guide. 2016).

To achieve high-throughput for single work-item-based kernel execution on the FPGA, the Intel FPGA SDK for OpenCL Offline Compiler must process multiple pipeline stages in parallel at any given time. The mode of operation is particularly challenging in loops because the offline compiler executes loop iterations sequentially through the pipeline by default (Intel Best Practice Guide. 2016).

Basic Implementation:

In single-threaded model, the basic implementation is similar to that of the C++ implementation.

Table 12

Code snippet of Basic Implementation using Single-threaded model – showing only the loop structures

Basic Implementation:

```
__kernel void stats_Appearance (int P, __global float *restrict xVal,
                       __global float *restrict Std,
                       __global float *restrict Skw,
                       __global float *restrict Krt)
{
   for (int index_x = 0; index_x < P; index_x++)
    {
        for (int index_y = 0; index_y < P; index_y++)
        {
            if (i < limit && j < limit) {
                for (iB = 0; iB < 16; iB++)
                {
                    for (jB = 0; jB < 16; jB++)
                    {
                        mean_sum = mean_sum + xVal[((iB+i) * P * 4) + (jB+j)];
                    }
                }
               mean = mean_sum / 256.0f;
                for (iB = 0; iB < 16; iB++)
                {
                    for (jB = 0; jB < 16; jB++)
                    {
                        tmp_mean = xVal[((iB+i) * P * 4) + (jB+j)] - mean;
                        stdev = stdev + (tmp_mean*tmp_mean);
                        skw = skw + (tmp_mean*tmp_mean*tmp_mean);
                        krt = krt + (tmp_mean*tmp_mean*tmp_mean*tmp_mean);
                    }
                }
        }
    }
}
```

The Basic Implementation in Table 12 has three sets of nested loops. The compiler builds all the loops to iterate sequentially through the pipeline. It also generates an optimization report on how well the kernels are pipelined. By referring the report, we can observe the pipeline information of the kernel. The optimization report in Table 14 shows that the kernel was well pipelined. From Table 13, we see that without any optimization methods, the basic implementation produced a 5x improvement in performance compared to that of the C++ implementation on the CPU.

Table 13

Results for Basic Implementation using Single-threaded model

Parameters/Devices	FPGA
Time	19.424ms (CPU 106ms)
Frequency	224MHz
Memory Stall	0.0%
Resource Utilization	Logic Utilization = 39%
	ALUTs = 15%
	Logic Registers = 24%
	Memory Blocks = 19%
	DSPs = 7%

From Table 14, we can see that the two inner most loops were pipelined successfully and iterations were launched every cycle. The other outer loops were pipelined successfully and iterations were launched every two cycles because they had a subloop inside them. This is tool dependent. This happens in Intel FPGA SDK for OpenCL Offline Compiler version 16.0. In the new version 16.1, this has been fixed to launch iterations every cycle. However, the FPGA board used in this thesis only supports 16.0 version.

Table 14

Optimization Report for Basic Implementation - showing pipeline information about each loop

Optimization Report

```
==================================================================================================
==
Kernel: stats_Appearance
==================================================================================================
==
The kernel is compiled for single work-item execution.

Loop Report:

 + Loop "Block1" (file mad.cl line 4)
 | Pipelined with successive iterations launched every 2 cycles due to:
 |
 |     Pipeline structure: every terminating loop with subloops has iterations launched at least 2
cycles
 |     apart.
 |     Having successive iterations launched every two cycles should still lead to good performance
 |     if the inner loop is pipelined well and has sufficiently high number of iterations.
 |
 |
 |-+ Loop "Block2" (file mad.cl line 6)
   | Pipelined with successive iterations launched every 2 cycles due to:
   |
   |     Pipeline structure: every terminating loop with subloops has iterations launched at least
2 cycles
   |     apart.
   |     Having successive iterations launched every two cycles should still lead to good
performance
   |     if the inner loops are pipelined well and have sufficiently high number of iterations.
   |
   |
   |-+ Loop "Block3" (file mad.cl line 19)
   | | Pipelined with successive iterations launched every 2 cycles due to:
   | |
   | |     Pipeline structure: every terminating loop with subloops has iterations launched at
least 2
   | |     cycles apart.
   | |     Having successive iterations launched every two cycles should still lead to good
performance
   | |     if the inner loop is pipelined well and has sufficiently high number of iterations.
   | |
   | |
   | |-+ Loop "Block4" (file mad.cl line 21)
   |       Pipelined well. Successive iterations are launched every cycle.
   |
   |
   |-+ Loop "Block7" (file mad.cl line 29)
     | Pipelined with successive iterations launched every 2 cycles due to:
     |
     |     Pipeline structure: every terminating loop with subloops has iterations launched at
least 2
     |     cycles apart.
     |     Having successive iterations launched every two cycles should still lead to good
performance
     |     if the inner loop is pipelined well and has sufficiently high number of iterations.
     |
     |
     |-+ Loop "Block8" (file mad.cl line 31)
         Pipelined well. Successive iterations are launched every cycle.


==================================================================================================
======
```

Loop Unrolling Implementation:

Loop unrolling decreases the number of iterations executed by the offline compiler at the expense of increased hardware resource consumption (Intel Best Practice Guide. 2016), which increases the throughput of the kernel. In Loop Unrolling Implementation, as we can see from the code snippet in Table 15, we have unrolled all the loops with a factor of 4, like in the loop unrolling implementation under multi-threaded model. From Table 17, we can see that the performance has dropped significantly even at the expense of increased hardware utilization.

Table 15

Code snippet showing the changes for the Loop Unrolling Implementation

Loop Unrolling Implementation

```
__kernel void stats_Appearance (inputs and outputs)
{
for (int index_x = 0; index_x < P; index_x++)
    {
    for (int index_y = 0; index_y < P; index_y++)
      {
Initializations
if (i < limit && j < limit) {
        #pragma unroll 4
        for (iB = 0; iB < 16; iB++)
        {
            #pragma unroll 4
            for (jB = 0; jB < 16; jB++)
            {
                mean_sum = mean_sum + xVal[((iB+i) * P * 4) + (jB+j)];
            }
        }
        #pragma unroll 4
        for (iB = 0; iB < 16; iB++)
        {
            #pragma unroll 4
            for (jB = 0; jB < 16; jB++)
            {
                tmp_mean = xVal[((iB+i) * P * 4) + (jB+j)] - mean;
                stdev = stdev + (tmp_mean * tmp_mean);
                skw = skw + (tmp_mean * tmp_mean * tmp_mean);
                krt = krt + (tmp_mean * tmp_mean * tmp_mean * tmp_mean);
            }
        }
}
```

From the optimization report in Table 16, we can see that partially unrolling the loop results in lots of dependency issues just with the nested loop that calculates the mean_sum. The compiler still tries to pipeline the kernel despite the dependency issues, which cause the loops in the kernel to launch iterations every 16 clock cycles. Launching iterations every 16 clock cycles will not produce a very high throughput even though the kernel is pipelined.

Table 16

Optimization Report Loop Unrolling Implementation – only nested loop used for calculating mean_sum

Optimization Report

```
===================================================================================================
==
Kernel: stats_Appearance
===================================================================================================
==
The kernel is compiled for single work-item execution.

Loop Report:

|-+ Loop "Block3" (file mad.cl line 19)
   | | Loop was partially unrolled 4 times due to "#pragma unroll" annotation.
   | | NOT pipelined due to:
   | |
   | |   Loop iteration ordering: iterations may get out of order with respect to the listed inner
loops,
   | |   as the number of iterations of the listed inner loops may be different for different
iterations of this loop.
   | |       Loop "Block12" (file mad_s2.cl line 22)
   | |       Loop "Block11" (file mad_s2.cl line 22)
   | |       Loop "Block10" (file mad_s2.cl line 22)
   | |       Loop "Block4" (file mad_s2.cl line 22)
   | |
   | |   To fix this, make sure the listed inner loops have the same number of iterations for each
iteration of this loop.
   | |   See "Out-of-Order Loop Iterations" section of the Best Practices Guide for more
information.
   | |   Not pipelining this loop will most likely lead to poor performance.
   | |-+ Loop "Block4" (file mad_s2.cl line 22)
   | |   Loop was partially unrolled 4 times due to "#pragma unroll" annotation.
   | |   Pipelined with successive iterations launched every 16 cycles due to:
   | |
   | |       Data dependency on variable
   | |       Largest Critical Path Contributors:
   | |           25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
   | |           25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
   | |           25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
   | |           25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
| |-+ Loop "Block10" (file mad_s2.cl line 22)
   | |   Loop was partially unrolled 4 times due to "#pragma unroll" annotation.
   | |   Pipelined with successive iterations launched every 16 cycles due to:
   | |
   | |       Data dependency on variable
   | |       Largest Critical Path Contributors:
   | |           25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
   | |           25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
   | |           25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
   | |           25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
   | |-+ Loop "Block11" (file mad_s2.cl line 22)
   | |   Loop was partially unrolled 4 times due to "#pragma unroll" annotation.
   | |   Pipelined with successive iterations launched every 16 cycles due to:
   | |
   | |       Data dependency on variable
   | |       Largest Critical Path Contributors:
   | |           25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
   | |           25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
   | |           25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
   | |           25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
   | |-+ Loop "Block12" (file mad_s2.cl line 22)
   |     Loop was partially unrolled 4 times due to "#pragma unroll" annotation.
```

```
    |     Pipelined with successive iterations launched every 16 cycles due to:
    |
    |         Data dependency on variable
    |         Largest Critical Path Contributors:
    |             25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
    |             25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
    |             25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
    |             25%: Hardened Floating-Point Add Operation  (file mad_s2.cl line 24)
====================================================================================================
======
```

Table 17

Results for Loop Unrolling Implementation

Parameters/Devices	FPGA
Time	25.044ms (CPU 106ms)
Frequency	203MHz
Memory Stall	0.0%
Resource Utilization	Logic Utilization = 56%
	ALUTs = 21%
	Logic Registers = 34%
	Memory Blocks = 49%
	DSPs = 14%

Building a Fully Pipelined Model:

In this section, we will see how the kernel was fully pipelined, all the loop structures were structured to launch iterations every cycle.

Step 1. **Determining the Loop Unroll factor:**

In the loop unroll implementation section, when we tried to partially unroll the loops, there were dependency issues on the variable *mean_sum* because of which the loop was pipelined to launch at every 16 cycles. To make the kernel a pipelined model to launch every cycle, we need to solve the dependency issues with partial loop unrolling.

The dependency issues on variables can be resolved by creating a new variable to handle the computation inside the inner loop and copying the data into a different variable when exiting the inner loop (Intel Best Practice Guide. 2016). These dependency issues are resolved when the inner loops are fully unrolled or the inner loops are not unrolled.

Case 1: When we try to partially unroll the loop, the dependency issues on the variables are not solved by the Intel FPGA SDK for OpenCL Offline Compiler (AOC).

Table 18

Partial Loop Unroll - factor 2

Partial Loop Unroll - factor 2

```
#pragma unroll 1
        for (iB = 0; iB < 16; iB++)
        {
            #pragma unroll 2
            for (jB = 0; jB < 16; jB++)
            {
                mean_sum = mean_sum + xVal[((iB+i) * P * 4) + (jB+j)];
            }
        }
```

Let's take only the nested loop that is used to calculate the *mean_sum* as shown in Table 19. The inner loop is partially unrolled by factor 2. From Table 19, we can see that there is a dependency issue in the variable *mean_sum*.

Table 19

Optimization Report – Partial Loop Unroll –factor 2

Optimization Report – Partial Loop Unroll –factor 2

```
|-+ Loop "Block3" (file mad_s25_1.cl line 20)
| Pipelined with successive iterations launched every 2 cycles due to:
|
|     Pipeline structure: every terminating loop with subloops has iterations launched at least 2
cycles apart.
|     Having successive iterations launched every two cycles should still lead to good performance
|     if the inner loop is pipelined well and has sufficiently high number of iterations.
|
|
| |-+ Loop "Block4" (file mad_s25_1.cl line 23)
| |   Loop was partially unrolled 2 times due to "#pragma unroll" annotation.
| |    Pipelined with successive iterations launched every 8 cycles due to:
| |
| |       Data dependency on variable
| |       Largest Critical Path Contributors:
| |           50%: Hardened Floating-Point Add Operation  (file mad_s25_1.cl line 25)
| |           50%: Hardened Floating-Point Add Operation  (file mad_s25_1.cl line 25)
```

We can see the iterations of the inner loop being launched every 8 cycles in the timing diagram from Figure 16 when the loop is unrolled by factor 2. In Figure 15, we can see the iterations of inner loop being launched in the timing diagram when the loop is not unrolled and the iterations are launched every cycle. When the loop was not unrolled, the inner loop launched all the iterations in 16 cycles. But when the loop was unrolled by a factor of 2, the inner loop (Block 4) took 57 cycles to launch all the 16 iterations.

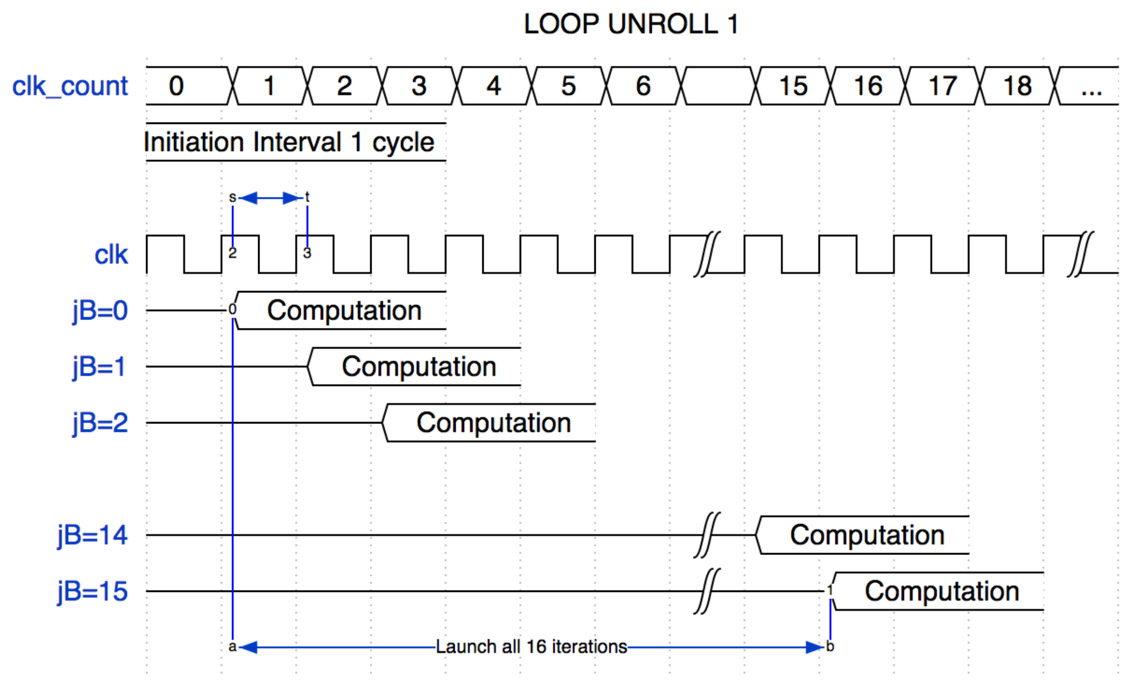


Figure 14 Timing Diagram - Unroll factor - 1 [No Unrolling]

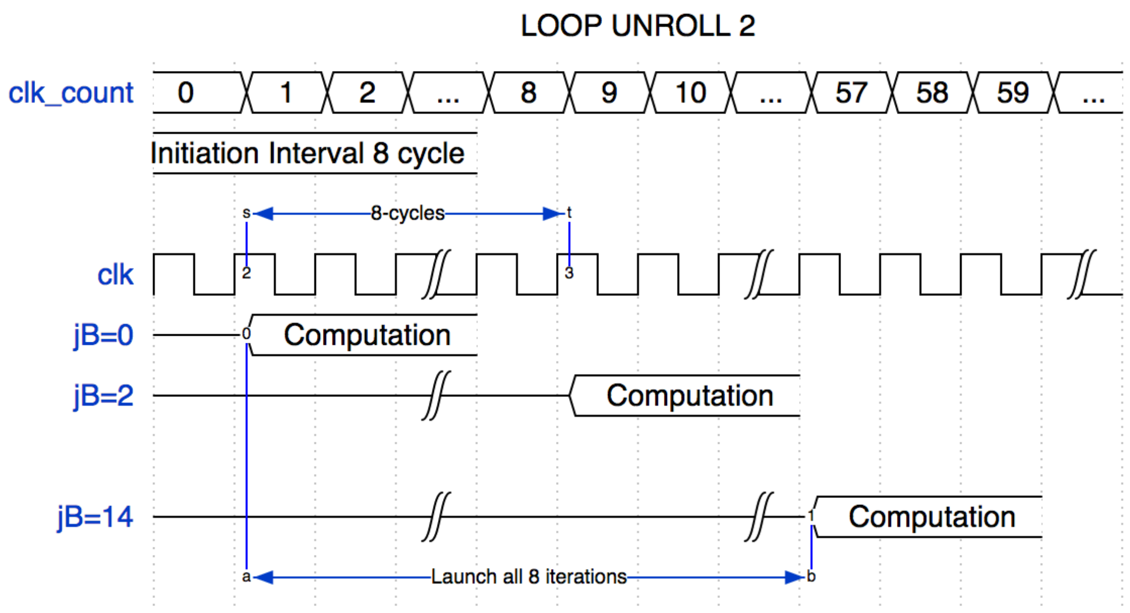


Figure 15 Timing Diagram - Partial Loop Unroll - factor 2

We add a new variable to solve the dependency issues on the variable, *mean_sum* as shown in Table 20. But the dependency issues were not resolved and the optimization report for the code snippet in Table 20 was the same as in Table 19.

Table 20

Partial Loop Unroll - factor 2 with additional variable to solve dependency issues

Partial Loop Unroll - factor 2 with additional variable to solve dependency issues

`#pragma unroll 1`
`        for (iB = 0; iB < 16; iB++)`
`        {`
`            float mean_sum1=0;`
`            #pragma unroll 2`
`            for (jB = 0; jB < 16; jB++)`
`            {`
`                mean_sum1 = mean_sum1 + xVal[((iB+i) * P * 4) + (jB+j)];`
`            }`
`            mean_sum += mean_sum1;`
`        }`

From this, we can infer that partially unrolling the loop and assigning new variables to break down the critical path on the variable to solve the dependency issues did not work.

We need partial loop unrolling because we can't unroll the loops fully every time. We can still do partial loop unrolling, but it should be done manually as shown in Table 21. In this case, we have resolved the data dependency issues such that the inner loop (Block 4) can launch iterations every clock cycle, and it can complete launching all the iterations in 8 cycles (Unrolled manually by factor 2), as can be seen in Table 22.

Table 21

Partial Loop Unroll – factor 2 - done manually

Partial Loop Unroll – factor 2 - done manually

```
#pragma unroll 1
        for (iB = 0; iB < 16; iB++)
        {
            float mean_sum1=0;
            float mean_sum2=0;
            #pragma unroll 1
            for (jB = 0; jB < 16; jB=jB+2) // jB=jB+2 because loop unroll 2
            {
                mean_sum1 = mean_sum1 + xVal[((iB+i) * P * 4) + (jB+j)];
                mean_sum2 = mean_sum2 + xVal[((iB+i) * P * 4) + (jB+j+1)];
            }
            mean_sum += mean_sum1 + mean_sum2;
        }
```

Table 22

Optimization Report – Partial Loop Unroll – factor 2 - done manually

Optimization Report – Partial Loop Unroll – factor 2 - done manually

```
Loop Report:
   |-+ Loop "Block3" (file mad_s25_1.cl line 21)
   | | Pipelined with successive iterations launched every 2 cycles due to:
   | |
   | |     Pipeline structure: every terminating loop with subloops has iterations launched at
least 2 cycles apart.
   | |     Having successive iterations launched every two cycles should still lead to good
performance
   | |     if the inner loop is pipelined well and has sufficiently high number of iterations.
   | |
   | |
   | |-+ Loop "Block4" (file mad_s25_1.cl line 26)
   |     Pipelined well. Successive iterations are launched every cycle.
```

Case 2: When we unroll the inner loop fully (Table 23), the data dependency issue is not resolved because the variable is still required by the outer loop for its next iteration as seen in the optimization report from Table 24 and the timing diagram from Figure 16.

Table 23

Inner Loop Fully Unrolled

Inner Loop Fully Unrolled

`#pragma unroll 1`
`for (iB = 0; iB < 16; iB++)`
`{`
`#pragma unroll 16`
`for (jB = 0; jB < 16; jB++)`
`{`
`mean_sum = mean_sum + xVal[((iB+i) * P * 4) + (jB+j)];`
`}`
`}`

```
#pragma unroll 1
        for (iB = 0; iB < 16; iB++)
        {
            #pragma unroll 16
            for (jB = 0; jB < 16; jB++)
            {
                mean_sum = mean_sum + xVal[((iB+i) * P * 4) + (jB+j)];
            }
        }
```

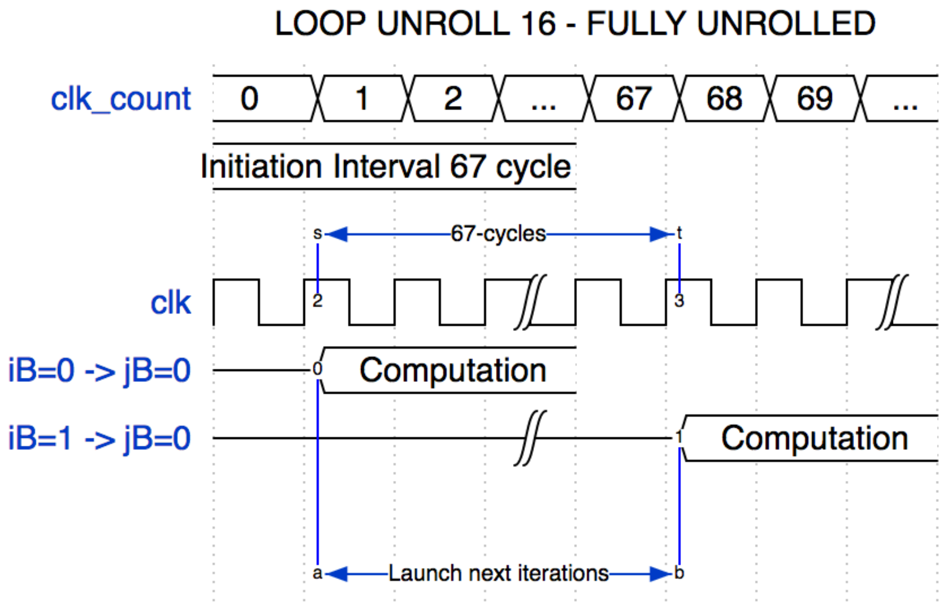


Figure 16 Timing Diagram - Inner Loop Fully Unrolled

Table 24

Optimization Report – Inner Loop Fully Unrolled

Optimization Report – Inner Loop Fully Unrolled

```
|-+ Loop "Block3" (file test_1.cl line 20)
| | Pipelined with successive iterations launched every 67 cycles due to:
| |
| |     Data dependency on variable
| |     Largest Critical Path Contributors:
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
| |          6%: Hardened Floating-Point Add Operation  (file test_1.cl line 26)
```

In this case, simply assigning new variables (Table 25) to break down the critical path on the variable can resolve the dependency issues and make both the inner and outer loops fully pipelined and launch iterations every cycle which can be seen in Table 26.

Table 25

Inner Loop Fully Unrolled with additional variable to solve dependency issues

Inner Loop Fully Unrolled with additional variable to solve dependency issues

```
#pragma unroll 1
        for (iB = 0; iB < 16; iB++)
        {
            float mean_sum1=0;
            #pragma unroll 16
            for (jB = 0; jB < 16; jB++)
            {
                mean_sum1 = mean_sum1 + xVal[((iB+i) * P * 4) + (jB+j)];
            }
            mean_sum += mean_sum1;
        }
```

Table 26

Optimization Report – Inner Loop Fully Unrolled with dependency issues resolved

Optimization Report – Inner Loop Fully Unrolled with dependency issues resolved
(report below)

```
|-+ Loop "Block3" (file mad_s25_1.cl line 25)
| | Pipelined well. Successive iterations are launched every cycle.
| |
| |
| |-+ Fully unrolled loop (file mad_s25_1.cl line 29)
|      Loop was fully unrolled due to "#pragma unroll" annotation.
```

Step 2**. Remove the outer loops 1 and 2, which iterate the value of i and j for the computation.**

The Outer loops 1 and 2 are pipelined to launch every 2 cycles since they have sub loops. The sub loops that calculate the mean and the standard deviation are pipelined to launch every cycle. Even though the 2 cycles are added by the compiler, they still affect the throughput of the kernel.

One such solution is to remove the outer loops 1 and 2 from the computation kernel and build a control kernel which sends the i and j values to the computation kernel through channels. Channels are FIFOs that connect one kernel to the other. Data can be sent from one kernel to the other using channels without using the global memory.

In the control kernel, instead of using nested for loop to generate the i and j values, we used simple math operation using the modulo (%) operation to iterate based on number of times we need to execute the kernel (Table 28).

Fully Pipelined Model – 1:

This is a fully pipelined model in which loop iterations are launched every cycle. From Table 27, we can see that all the loops are pipelined to launch every cycle, which gives us a very high throughput. The block diagram in Figure 16 shows that the kernel *control_loop* uses the channels to control the *stats_Appearance* kernel and *stats_Appearance* kernel, reading and writing data from the global memory.

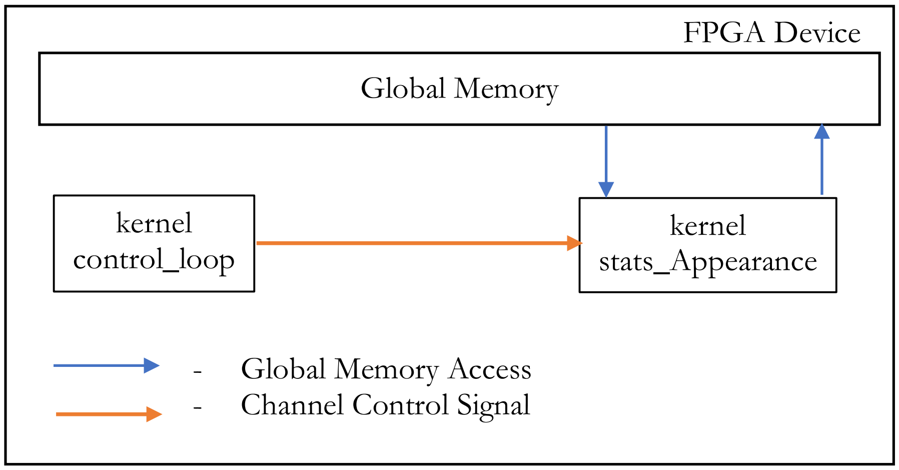


Figure 17 Block diagram showing device side execution flow – Fully Pipelined Model – 1

From Table 28, we see that using this model we can get about 80x improvement in performance. In Table 29, we can see the time taken when the image size was varied. We observe that as the image size became larger there was a slight drop in performance due to increase in the memory stall.

Table 27

Optimization Report - Fully Pipelined Model – 1

Optimization Report - Fully Pipelined Model – 1

```
===================================================================================================
Kernel: control_loop
===================================================================================================
The kernel is compiled for single work-item execution.
Loop Report:
 + Loop "Block1" (file mad_s23_5.cl line 13)
   Pipelined well. Successive iterations are launched every cycle.

===================================================================================================
Kernel: stats_Appearance
===================================================================================================
The kernel is compiled for single work-item execution.
Loop Report:
 + Loop "Block4" (file mad_s23_5.cl line 46)
 | Pipelined well. Successive iterations are launched every cycle.
 |
 |-+ Loop "Block5" (file mad_s23_5.cl line 61)
 | | Pipelined well. Successive iterations are launched every cycle.
 | |
 | |-+ Fully unrolled loop (file mad_s23_5.cl line 65)
 |      Loop was fully unrolled due to "#pragma unroll" annotation.
 |
 |-+ Loop "Block6" (file mad_s23_5.cl line 73)
   | Pipelined well. Successive iterations are launched every cycle.
   |
   |-+ Fully unrolled loop (file mad_s23_5.cl line 79)
        Loop was fully unrolled due to "#pragma unroll" annotation.
===================================================================================================
```

Table 28

Fully Pipelined Model – 1 results

Parameters/Devices	FPGA
Time	1.33ms (CPU 106ms)
Frequency	205MHz
Memory Stall	1.4%
Resource Utilization	Logic Utilization = 41%
	ALUTs = 16%
	Logic Registers = 25%
	Memory Blocks = 22%
	DSPs = 14%

Table 29

Fully Pipelined Model – 1 results on various Image Sizes

Parameters/Image Size	512x512	1024x1024	2048x2048	4096x4096
Time	1.33ms	5.347ms	63.66ms	271.58ms
Avg. time per 512x512 computation	1.33ms	1.33ms	3.96	4.24
Memory Stall	1.4%	2%	66%	68%

Table 30

Code Snippet for Fully Pipelined Model - 1

Fully Pipelined Model - 1

```
__kernel void control_loop(int S){
    int count=0; int row=0;
    int col=0; int stop=0;
    while(count<S*S){
        int i=row;
        int j=col;
        write_channel_altera(x,i);
        write_channel_altera(y,j);
        count++;
        if(count%S==0 && count%(S*S)!=0){
            row++;
            stop=0;
        }
        else if(count%S==0 && count%(S*S)==0){
            row=0;
            stop=1;
        }
        if(count%S==0){
            col=0;
        }
        else {
            col++;
        }
        write_channel_altera (stop_signal, stop);
    }
}
__kernel void stats_Appearance (int P, __global float *restrict xVal, __global float *restrict Std,
                       __global float *restrict Skw,
                       __global float *restrict Krt)
{int stop=0;
while (! stop) {
             int i = 4*read_channel_altera(x);
             int j = 4*read_channel_altera(y);
             stop=read_channel_altera(stop_signal);
             for (iB = 0; iB < 16; iB++)
             {
                  mean_sum1=0;
                  #pragma unroll 16
                  for (jB = 0; jB < 16; jB++)
                  {
                       mean_sum1 = mean_sum1 + xVal[((i+iB) * P * 4) + (j+jB)];
                  }
                  mean_sum += mean_sum1;
              }
             for (iB = 0; iB < 16; iB++)
             {
                 stdev1=0;
                 skw1=0;
                 krt1=0;
                 #pragma unroll 16
                 for (jB = 0; jB < 16; jB++)
                 {
                      tmp_mean = xVal[((i+iB) * P * 4) + (j+jB)] - mean;
                      stdev1 += (tmp_mean*tmp_mean);
                      skw1 +=(tmp_mean*tmp_mean*tmp_mean);
                      krt1 +=(tmp_mean*tmp_mean*tmp_mean*tmp_mean);
                 }
                 stdev += stdev1;
                 skw += skw1;
                 krt += krt1;
              }
          }
}
```

Fully Pipelined Model -2:

To remove the memory stall from the *stats_Appearance* kernel, we sent data along with the control signal from the *control_loop* kernel and created one more kernel, *write_data* to write the data back to the global memory as shown in Figure 17. In this model, since all the data are read from the global memory by the *control_loop* kernel and sent to the *stats_Appearance* kernel, we need to maintain a local memory to hold the 256 elements which are required for the current computation. From Table 31, we can see that throughput dropped because of the addition of a sub loop inside the control_loop to read data which made the outer loop in the control_loop kernel to launch every 2 cycles which is unavoidable.

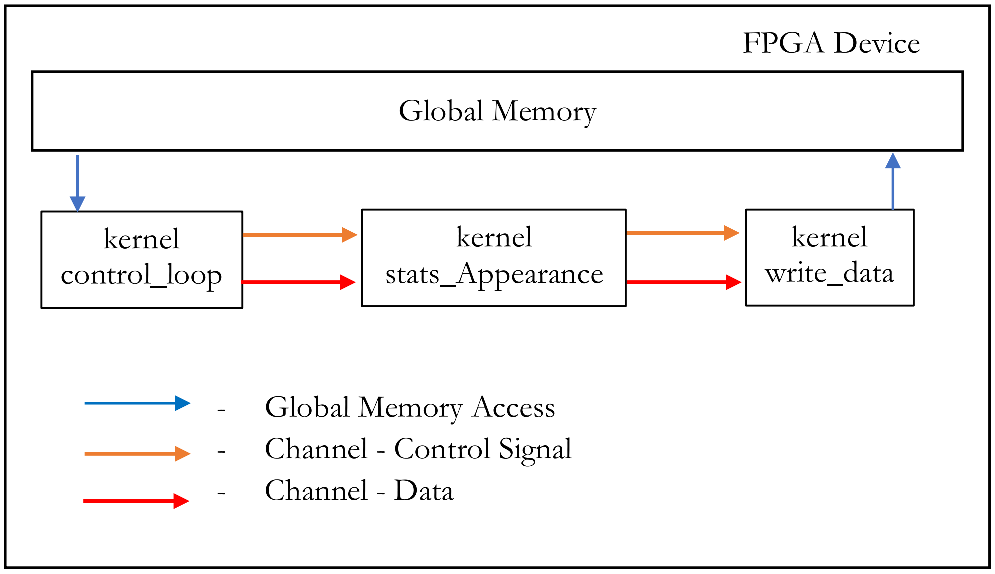


Figure 18 Block diagram showing device side execution flow – Fully Pipelined Model – 2

Table 28

Fully Pipelined Model – 2 results

Parameters/Devices	FPGA
Time	3.53ms (CPU 106ms)
Frequency	200MHz
Memory Stall	0.0%
Resource Utilization	Logic Utilization = 44%
	ALUTs = 17%
	Logic Registers = 26%
	Memory Blocks = 24%
	DSPs = 14%

Fully Pipelined Model – 3:

This model is same as fully pipelined model 2, except for one change in the code. The fully pipelined model 1 and model 2 had a stop signal coming from the control_loop kernel to stop the stats_Appearance kernel when all the data has been operated on. Since the data comes along with the control signal and if there are no *i* and *j* values sent from the control_loop, the stats_Appearance kernel waits till it gets new data on the channel. The channel (FIFO) already acts as a control signal to start and stop the kernel, so we removed the stop signal controlling the stats_Appearance kernel. As we can see from Table 32, the operation frequency improved, which resulted in an improvement in performance when compared to the previous model.

Table 29

Fully Pipelined Model – 3 results

Parameters/Devices	FPGA
Time	3.14ms (CPU 106ms)
Frequency	225MHz
Memory Stall	0.0%
Resource Utilization	Logic Utilization = 43%
	ALUTs = 17%
	Logic Registers = 26%
	Memory Blocks = 23%
	DSPs = 14%

From Table 30, we can see that on varying the image size, the performance the kernel was consistent when compared to that of fully pipelined model 1. There is stall but the data is buffered in the channels, which decreases the memory stall, so the memory stall does not have significant effect on varying image size.

Table 30

Fully Pipelined Model – 3 results on various Image Sizes

Parameters/Image Size	512x512	1024x1024	2048x2048	4096x4096
Time	3.14ms	12.75ms	51.52ms	231.50ms
Avg. time per 512x512 computation	3.14ms	3.18ms	3.22ms	3.61ms
Memory Stall	0.1%	0.2%	56%	66%

Fully Pipelined Model – 4 [2 images]:

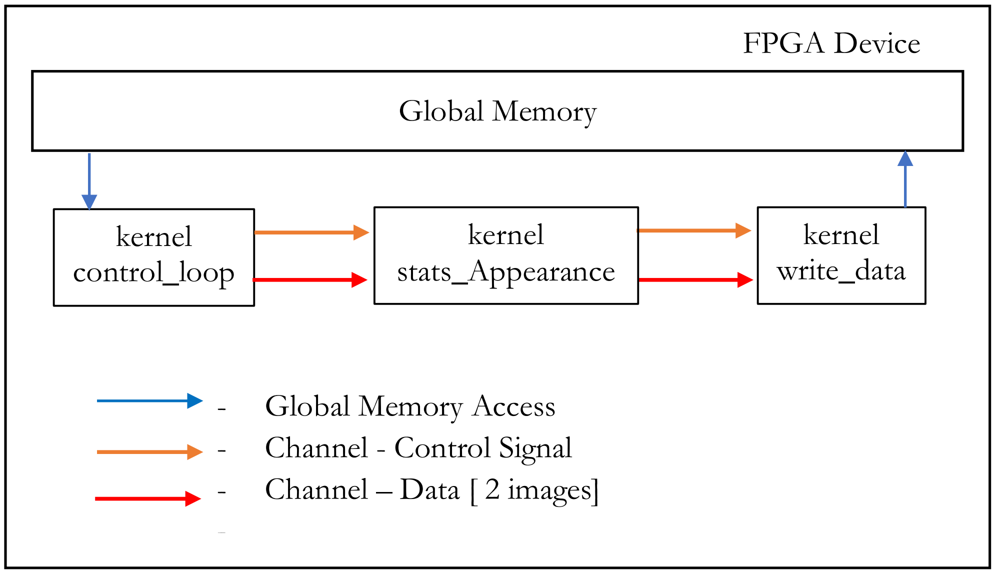


Figure 19 Block diagram showing device side execution flow – Fully Pipelined Model – 4

This model is similar to fully pipelined model 3. The fully pipelined model 4 has process two images in a pipelined fashion. In the previous model, we used channels as a buffer to hide the memory stall in the kernel partially. From Table 32, we can see that with the increase in image size the performance of the fully pipelined model 4 dropped when compared to that of fully pipelined model 3 because we now read two images which adds additional memory stall to the kernel. With increased memory stall we no longer able to hide it by buffering the data using channels. Still, we have higher throughput when compared to the fully pipelined model 3 because we now process two images (Table 31).

Table 31

Fully Pipelined Model – 4 results

Parameters/Devices	FPGA
Time	3.66ms (CPU 106ms) => 1.83ms/image
Frequency	192MHz
Memory Stall	1.0%
Resource Utilization	Logic Utilization = 49%
	ALUTs = 20%
	Logic Registers = 30%
	Memory Blocks = 31%
	DSPs = 23%

Table 32

Fully Pipelined Model – 4 results on various Image Sizes

Parameters/Image Size	512x512	1024x1024	2048x2048	4096x4096
Time	1.83ms	7.44ms	37.52ms	158.78ms
Avg. time per 512x512 computation	1.83ms	1.86ms	2.24ms	2.48ms
Memory Stall	1%	1%	69%	70%

Summary of Results

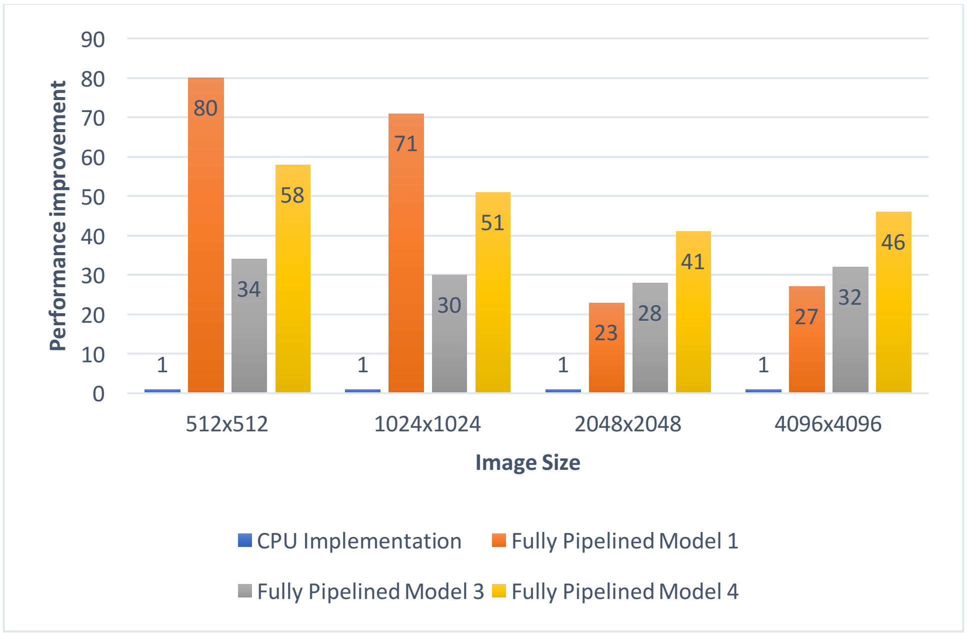


Figure 20 Performance improvement over CPU implementation

- Fully Pipelined Model-1 kernel implementation has provided 80x speedup for the corresponding statistical computation kernel over the CPU implementation.
- With the increase in image size, Fully Pipelined Models 3 and 4 have better performance when compared to that of Fully Pipelined Model 1.
- Fully Pipelined Model 3 has a very small change in performance when compared to other implementations with varying image sizes.
- Fully Pipelined Model 4 has a similar performance like Fully Pipelined Model 1 because of Memory Stall. Fully Pipelined Model 4 has better performance than Fully Pipelined Model 1 because we process two images, which give us a better throughput even with Memory Stall.

All the results were verified using the C++ implementation of the MAD Algorithm.

CHAPTER 4

CONCLUSION

We demonstrated various models in which the computationally complex function in the C++ implementation was implemented, and its performance was compared with that of the CPU implementation. The fully pipelined model -1 was pipelined to launch every cycle, giving a very high throughput, resulting in 80x speedup when compared to the CPU implementation. The performance of the function was also tested across varying image sizes; we observed a drop in the performance with the increase in image size. To make the computation kernel to have consistent performance with varying image size, fully pipelined models 2 and 3 were implemented to use channels to send data to the computation kernel. Since the data to the computing function was buffered through the channel, the overall memory stall with varying image size reduced and the performance drop was less significant. To improve the throughput further, fully pipelined model 4 was implemented to process two images in a pipelined fashion. With fully pipelined model 4, we got increased throughput when compared to fully pipelined model 3.

At each implementation, we evaluated the bottleneck and applied different methods to resolve it. Memory Stall was identified to be the bottleneck in all the implementations when the images sizes were larger than 1024x1024. We could hide the bottleneck to an extent by making use of channels as buffers. We can make use of other I/O interfaces in the FPGA like Ethernet to stream the data directly into the kernel. In this, we do not need any global memory access, which makes memory stall zero. By making the memory stall zero, we could get a kernel with higher throughput for larger images similar to that of small images.

Findings

- For sequential algorithms, we got high throughput and less area utilization by using single-threaded model (Pipeline the algorithm to launch iterations every cycle) when compared to the multi-threaded model.
- Partial loop unrolling dependency issues can be solved by manually unrolling the loops using additional variables.
- Using a new kernel to control the main kernel will help in reducing the number of loops in the main kernel, which helps in implementing a fully pipelined model.
- Using channels to send data across the kernel removes the memory read and write overhead in the main kernel.
- The kernel which uses only channels for input and outputs (no global memory access) and with always true condition (while (1)) results in higher frequency which in turn results in higher throughput.

Future Work

- Memory Stall: We could completely remove memory stall (global memory access) by making use of Ethernet to stream the data to the FPGA from the computer.
- Data Reuse: Statistical computation function has 75% overlap of data with the neighboring pixels. Redesigning the fully pipelined model to reuse the data will result in higher throughput.
- Analyzing other kernels in CPU MAD implementation: Next most computational complex function, Inverse 2D-FFT, can be implemented in FPGA using OpenCL. Currently, Altera's OpenCL example, 2D-FFT, has 15x speedup over the CPU implementation.

REFERENCES

Abdelfattah, M. S., Hagiescu, A., & Singh, D. (2014, May). Gzip on a chip: High performance lossless data compression on fpgas using opencl. In Proceedings of the International Workshop on OpenCL 2013 & 2014 (p. 4). ACM.

Ahmed, T. (2011). OpenCL framework for a CPU, GPU, and FPGA Platform (Doctoral dissertation, University of Toronto).

Alam, S. R., Agarwal, P. K., Smith, M. C., Vetter, J. S., & Caliga, D. (2007). Using FPGA devices to accelerate biomolecular simulations. Computer, 40(3).

Ayat, S. O., Khalil-Hani, M., & Bakhteri, R. (2015, November). OpenCL-based hardware-software co-design methodology for image processing implementation on heterogeneous FPGA platform. In *Control System, Computing and Engineering (ICCSCE), 2015 IEEE International Conference on* (pp. 36-41). IEEE.

Buiten, H. J., & Van Putten, B. (1997). Quality assessment of remote sensing image registration—analysis and testing of control point residuals. ISPRS journal of photogrammetry and remote sensing, 52(2), 57-73.

Bukh, P. N. D., & Jain, R. (1992). The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling.

Charrier, C., Knoblauch, K., Moorthy, A. K., Bovik, A. C., & Maloney, L. T. (2010, January). Comparison of image quality assessment algorithms on compressed images. In IS&T/SPIE Electronic Imaging (pp. 75290B-75290B). International Society for Optics and Photonics.

Chen, W. H., Smith, C. H., & Fralick, S. C. (1977). A fast-computational algorithm for the discrete cosine transform. IEEE Transactions on communications, 25(9), 1004-1009.

Holloway, J., Kannan, V., Chandler, D. M., & Sohoni, S. On the Computational Performance of Single-GPU And Multi-GPU CUDA Implementations of the MAD IQA Algorithm.

Huang, A. (2015). The Death of Moore's Law Will Spur Innovation. IEEE Spectrum.
Intel FPGAs (2016). Intel FPGA Best Practice guide
www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf

Kamble, V., & Bhurchandi, K. M. (2015). No-reference image quality assessment algorithms: A survey. Optik-International Journal for Light and Electron Optics, 126(11), 1090-1097.

Kannan, V. (2016). An Analysis of the Memory Bottleneck and Cache Performance of Most Apparent Distortion Image Quality Assessment Algorithm on GPU (Doctoral dissertation, Arizona State University).

Khronos Group (2011). OpenCL Specification 1.1.
www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

Larson, E. C., & Chandler, D. M. (2010). Most apparent distortion: full-reference image quality assessment and the role of strategy. *Journal of Electronic Imaging*, *19*(1), 011006-011006.

Lee, Y. H., Khalil-Hani, M., Bakhteri, R., & Nambiar, V. P. (2016). A real-time near infrared image acquisition system based on image quality assessment. Journal of Real-Time Image Processing, 1-18.

Moore, G. E. (1998). Cramming more components onto integrated circuits. Proceedings of the IEEE, 86(1), 82-85.

MAD Matlab Implementation (2011).
www.sse.tongji.edu.cn/linzhang/IQA/Evalution_MAD/eva-MAD.htm

Ooura's Mathematical software packages.
www.kurims.kyoto-u.ac.jp/~ooura/

Okarma, K., & Mazurek, P. (2011). GPGPU based estimation of the combined video quality metric. In Image Processing and Communications Challenges 3(pp. 285-292). Springer Berlin Heidelberg.

Phan, T., Sohoni, S., Chandler, D. M., & Larson, E. C. (2012, April). Performance-analysis-based acceleration of image quality assessment. In Image Analysis and Interpretation (SSIAI), 2012 IEEE Southwest Symposium on (pp. 81-84). IEEE.

Phan, T. D., Shah, S. K., Chandler, D. M., & Sohoni, S. (2014). Microarchitectural analysis of image quality assessment algorithms. Journal of Electronic Imaging, 23(1), 013030-013030.

Suda, N., Chandra, V., Dasika, G., Mohanty, A., Ma, Y., Vrudhula, S., ... & Cao, Y. (2016, February). Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 16-25). ACM.

Tang, Q. Y. (2016). FPGA Based Acceleration of Matrix Decomposition and Clustering Algorithm Using High Level Synthesis.

Zhao, L., Iyer, R., Makineni, S., & Bhuyan, L. (2005, March). Anatomy and performance of SSL processing. In Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on (pp. 197-206). IEEE.