FPGA-BASED HYBRID COMPUTING FOR ESS LINAC SIMULATOR

ARUN JEEVARAJ
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

FPGA-BASED HYBRID COMPUTING FOR ESS LINAC SIMULATOR

Arun Jeevaraj
jeevarajarun@gmail.com

Department of Electrical and Information Technology
Lund University

Supervisors: Liang Liu, Emanuele Laface

Asst.Supervisors: Maurizio Donna, Fredrik Edman

Examiner: Erik Larsson

November 8, 2017

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

The thesis explores efficient implmentation strategies for the European Spallation Source (ESS) linear accelerator simulator. The target simulator needs to run at real time, requires high computation accuracy, and should be scalable for high density beam scenarios. The high data processing, communication, and storage requirements due to a large data set, along with a strict accuracy requirement, poses a critical implementation challenge for traditional computing platforms.

To tackle these issues, this thesis uses a scalable platform with hybrid computing capabilities and the Open Computing Language (OpenCL) framework for an unified programming model. The hybrid computing platform allows for mapping tasks to the most suitable hardware and explores heterogeneous memory hierarchy to fast data shuffling. The OpenCL framework allows functional portability and scalability across different target devices such as CPU, Graphics Processing Unit (GPU) and OpenCL accelerator devices like with Field Programmable Gate Arrays (FPGA) and Digital Signal Processor (DSP) arrays. The computational intensive tasks of the simulator can be conveniently mapped to the accelerators, where computational parallelism is explored.

The targeted simulator is implemented in a Xilinx hybrid computing platform, consisting of an Intel i7 CPU, an Nvidia 960 GPU, and a Xilinx Kintex Ultra-scale FPGA. Comparing to the benchmark (a C++ based implementation), we are able to accelerate the ESS simulator by more than 80x on the GPU and 25x with FPGA, with the same simulation accuracy (double precision floating point). We identified the implementation bottleneck on the specific platform, which is the memory bandwidth. This leads to our future work. One important future task is to investigate different hybrid computing platforms of different vendors, considering computation capability, memory bandwidth, as well as design software. Moreover, different data types will be examined, including fixed-point, double/single-precision floating point, or custom floating point.

The structure of the thesis report: Due to it's broad nature, the report has focus on the core content in chapters 1-4, while the theory and implementation details are described in appendix A-E.

Popular Science Summary

Accelerating Linac Simulator using heterogeneous hardware architectures

The European Spallation Source is one of Europe's largest research infrastructures to bring new insights into the challenges of science and innovation in fields as diverse as material and life sciences, energy, environmental technology, cultural heritage,solid-state and fundamental physics by the end of the decade.

A 5 MW, long pulse proton accelerator is used to reach this goal. The particles in this beam contains so much of energy, that the loses to the super-conducting structures become marginally critical. An accurate and real-time simulation model to predict the behaviour of the particles is required to minimize the losses.

There are two approaches to model the particle behaviour, one is to use an approximation with the envelop of the beam and the other is to model each particle and track them. The envelop method can run real-time, but has discrepancies to the actual behaviour of the beam at high energy configurations, hence a computationally intensive mult-particle simulation model is required to provide realistic behaviour. This is achieved by supporting non-linear behaviour such as space charge (particle to particle interactions).

The multi-particle simulation algorithm has linear and non-linear attributes, where the linear section can map efficiently to the graphic processor hardware architecture, while the non-linear section can map more efficiently to the flexible FPGA fabrics. To make the multi-particle simulation model run real-time, a computing platform that can support graphic processors and FPGA accelerators are required.

The thesis proposes to use a hybrid computing platform with FPGA, GPU and CPU to provide the efficient mapping of the multi-particle simulation model. A unified development environment and a scalable platform is developed by using a software stack build based on OpenCL framework. This allowed to develop the system in a rapid design phase and maximize the code reuse.

We were able to accelerate the linear part of the simulation model by upto 89x on the given set of hardware. Different ways to further optimize the solution are also explored.

Acknowledgement

Acknowledgement section is difficult to write, as there are many I am thankful for and every bit was crucial. I will take inspiration from the sanskirt verse "Matha Pitha Guru Deivam", which translates to Mother Father Teacher God and I am following that order. I would like to thank my parents for the love and care, and supporting me to study in Sweden. Next I thank my supervisors and teachers, as every bit of skills I know were learnt from them.

Special thanks to my supervisor Liang Liu for giving the chance to work on this thesis and also solidifying my foundations in computer architecture and VLSI. I am also very grateful to Emannuel Laface from ESS for showing me the wonders of physics and help me understand the simulation model. He always looked like a rockstar to me, when he talked physics. Next, I would like to thank Fredrik Edman for the wonderful supervision and many of the DSP techniques I applied can be pointed to his DSP course lectures. I am also grateful to Maurizio Donna for sharing his know-how on the FPGA hardware design techniques and encouraging me through the tough times. If the thesis report is as good as it is, it is due to the effort from Liang and Fredrik despite their busy schedule.

Table of Contents

List of Figures

List of Tables

Acronyms

APIs Application Programming Interfaces. 49

AXI Advance eXtensible Interface. 44

DMA Direct Memory Access. 44

DSP Digital Signal Processor. i

DUT Design under Test. 44

ELS ESS Linac Simulator. 2

ESS European Spallation Source. i, 1

FPGA Field Programmable Gate Arrays. i

GPU Graphics Processing Unit. i

ICD Installable Client Drivers. 51

IDE Integrated Development Environment. 49

Linacs linear accelerators. 1

OpenCL Open Computing Language. i

PIC Particle in Cell. 2

Chapter 1
Introduction

In this section, the background of the European Spallation Source (ESS) Linear Accelerator simulator is presented. The design challenges associated with the simulator and a hybrid computing platform to tackle these challenges is proposed. Related work done in the field is explored and discussed. The design methodology and tools used in the thesis are also introduced.

1.1 Background

The ESS is a scientific laboratory constructed in cooperation with 17 European Countries to produce the brightest beam to date for use in materials research. At ESS as shown in Fig. 1.1, a stream of neutrons will be created using a linear proton accelerator and a fixed tungsten target. The proton beam has a peak power of 125 MW with a duty cycle of 4% for an integrated power of 5MW on the target. Such beam requires precise and accurate control to minimize losses in the superconducting structures, therefore an accurate multi-particle on-line simulator is needed to provide more realistic beam information and tuning guidance. Existing on-line simulators for proton linear accelerators (Linacs) are mostly developed based on an envelope model. For a majority of applications, such an approximation brings in lower computation load and hence a real time execution model. But for high power beam applications, these approximations of the beam produces marginal discrepancies in the simulator results. It also doesn't support the capability to simulate using real beam distributions and take into account space charge (inter particle interactions).

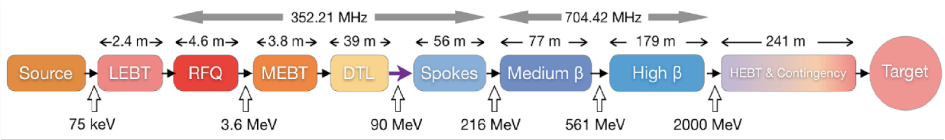

Figure 1.1: ESS Linear Accelerator

The ESS proton Linac will be commissioned and operated with the use of an on-line model. This is a simulator, which uses live readouts from the con-

trol system to provide the most accurate description of the running accelerator as possible, given the rapid response required during operation. Today the available simulators are divided in two groups: (1) beam envelope simulators, providing fast results but only of average quantities of the beam, and (2) multi-particle tracking simulators, providing accurate results but in a time not suitable for live operations.

The multi-particle tracking simulation model can provide an accurate simulation result and support for space charge. However, these are arrived at the expense of increased computation complexity, that reaches Tera matrix multiplications for the required test configurations. This scale of computation requires a high density parallelism and can be taxing on the traditional computational methodology to provide efficient mapping to hardware. When the linear tasks are coupled with non-linear behaviour modelling (space-charge), a suitable hardware with flexible architecture is required. A heterogeneous architecture based on CPUs, GPUs and FPGA accelerators is proposed as a solution.

This thesis project will explore the possibility to accelerate a multi-particle tracking simulator with the goal to bring this family of simulators in the on-line environment of a modern accelerator. The simulator code used for this project will be ESS Linac Simulator (ELS) developed and integrated in the ESS control infrastructure. The core of the simulators can also be used in other applications, such as astronomy to model the interaction between the galaxies and stars.

1.2 Related Work

The envelop based model can be run real-time without the need for further optimizations. Due to the simplifications inherent to the model, it cannot account for the non linear behavior that is needed at higher energy configurations to provide realistic beam predictions. Whereas, the Multi-particle model due to the (heavy) computation requirements requires hardware equal to a super computer for obtaining the results within an acceptable execution time frame.

The work[2] focuses on the use of split-operator methods to integrate single-particle magnetic optics techniques with parallel Particle in Cell (PIC) techniques. By choosing a splitting scheme that separates the self-fields from the complicated externally applied fields, they are able to utilize a large step size and still retain high accuracy. Similarly, Trace 3D [7] simulator uses envelope approach to predict the beam behavior. The work [1] points to a complete implementation of the on-line simulation system, which acquires real-time information from the linear accelerator, then the on-line simulator performs the simulation and provides important statistical information to the control room. Reducing the simulation time will shorten the duration for retuning the accelerator, and there is a need for retuning before different scientific experiments. The disadvantages of these methods are that, the optimizations are applied over the algorithm level, by using numerical approximations and smart application of physics that can only converge for the test configurations that they were designed for. This lead to discrepancies with the actual results beyond an acceptable margin, when the model is scaled for

beam configurations required at ESS (beam with larger particle density moving with high energy).

HPSim[8] is an implementation done with Nvidia CUDA[23] platform at it's core for accelerating the Linac simulation. A python based API to the implementation is available, and results of the acceleration with and without space charge simulation are provided with the publication. It performs the implementation in the state of the art(at time of publication) NVIDIA Tesla K20 GPU and Intel Xeon E5520 2.27GHz processor cluster. The implementation benchmarked a result of about a second with (4960 RF gaps + 206 quads + 460 drift spaces) for 32K particles. Since CUDA is proprietary, it is locked to Nvidia and doesn't support acceleration over other vendor architectures. Although, it takes it in a similar direction as this thesis work, it doesn't consider any hardware level optimizations or steps in utilizing a better data precision. This also doesn't support a flexible hardware configuration, limiting the capabilities needed for mapping non-linear tasks to FPGA, while utilizing the GPU architecture for linear tasks of the simulation model.

The non linear behavior of the simulation, doesn't efficiently map to a GPU architecture due to thread divergence and reduction attributes [4]. Therefore, a flexible hardware platform is required for efficient mapping of the hardware to the optimal task. This can provide maximum processing and power efficiency. [19] is a study on how well, a FPGA performs within the context of the OpenCL implementation against GPU and CPU. However, at the time of writing we couldn't find a similar work on the multi-particle model that utilizes a CPU, GPU and FPGA based hybrid computing platform.

1.3 Multi-particle Simulation Model

The multi-particle simulation model is based on the principles of Hamiltonian to represent the behavior of the particle moving in a given space. Each particle is represented by a vector in phase space of dimension 6 as a function of distance. Given the initial conditions the particle state after moving a distance L in drift space can be modeled with a transport matrix as shown in equations 1.1, 1.2 and 1.3. The β is the velocity of the particle scaled to the velocity of light.

$$\bar{x_1} = R_{drift} \times \bar{x_0} \tag{1.1}$$

$$R_{drift} = \begin{pmatrix} 1 & L & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & L & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{L}{\beta_0^2 \gamma_0^2} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.2}$$

$$\gamma_0 = \frac{1}{\sqrt{1 - \beta_0^2}} \tag{1.3}$$

A FODO(FOcus and DefOcus) cell is a basic block used in any Linac and provides an oscillatory behavior to the envelop of the beam. For the test-case configuration, the FODO cells are iterated over to match the required complexity of an actual test configuration. The transport matrix is modified, when the space-charge and other non-linear behavior is included in the simulation. The FODO cell constitute of quadrupole magnet spaces focusing the beam on X and Y axises separated by drift spaces as shown in Fig. 1.2. The transport matrices for quadrupole spaces are derived based on [12]. Fig. 1.3 shows the envelope of the beam oscillating in the x axis with reference to distance(expressed in units of L).

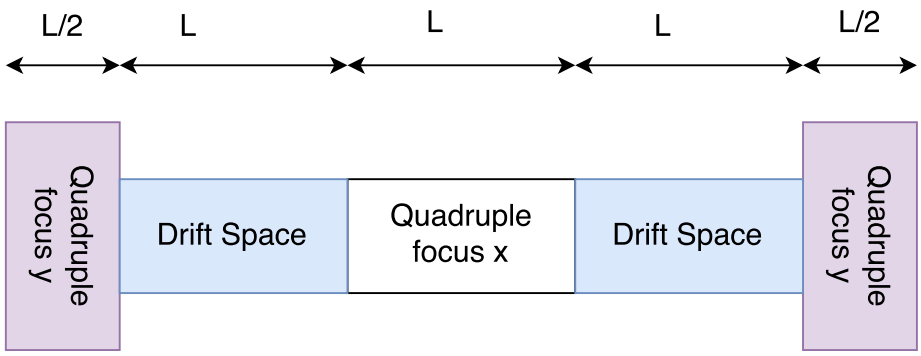

Figure 1.2: The Fodo cell structure.

A more elaborate description on the quadrupole and drift space matrices in context to Hamiltonian principles, and related physics theory is discussed in the Appendix A.

1.4 Design Challenge

The multi-particle simulation model, is intensive in computational and memory requirements. Due to high power beam used at ESS, a test scenario will have a particle count of million particles and their states computed over a million frames. The computation complexity of 10^{12} matrix multiplications needs to be mapped to the hardware. With the double precision data, 48 Mega bytes of memory is required at each frame. This demands a proper memory hierarchy with large and slow memory like off-chip memory interfaced with small and fast on-chip memory. Thus, the shuffling and arrangement of data also becomes a critical task to enhance the memory access efficiency, while meeting the bandwidth requirement for massive parallelism in hardware. The non-linear part of the simulator, such as space charge behavior and obtaining the statistical information at each frame will be also be mapped to hardware, and this can be straining for GPU hardware architecture.

This demands a flexible hardware configuration, to off-load different work loads to different hardware architecture based on their efficiency. A unified hardware platform is required for the flexible hardware configuration and also to provide maximum code re-use in a small development time-frame.

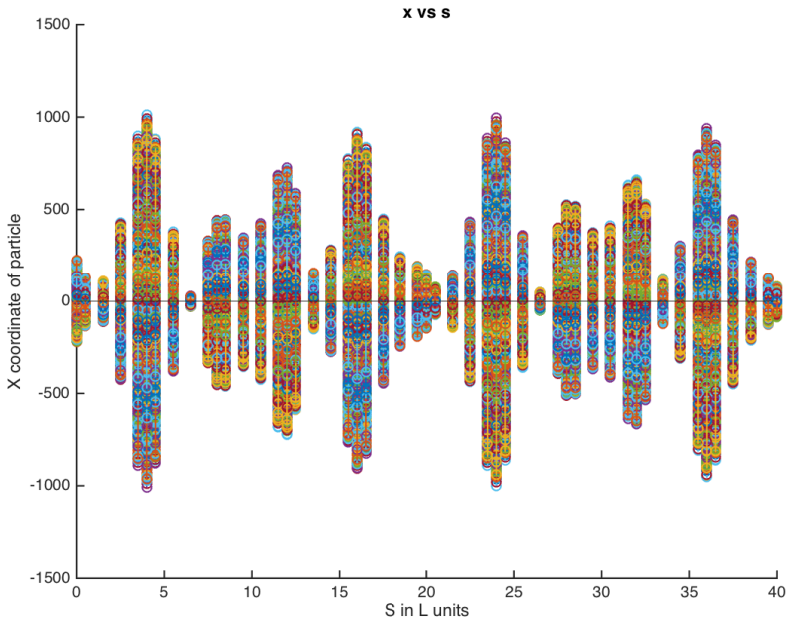

Figure 1.3: Beam state parameter X versus unit distance S.

The memory traffic and memory access rate is directly related to the data type used and hence the feasibility of fixed point implementation and it's throughput versus accuracy needs to be studied.

Depending on the demand and usage scenario, such as to be used on-line or off-line brings in the requirement for a scalable platform. The target solution should be able to meet the real-time performance requirement of the on-line simulation scenario, where as for off-line simulation the cost of hardware will be an added deciding factor. A scalable and functionally portable solution provides improvement to performance by scaling, adding or upgrading the hardware platform without the need for modifying the software.

1.5 Hybrid Computation

In this thesis, we explore the concept of hybrid computation to tackle the aforementioned design challenges by mapping the target algorithm in an efficient way to different hardware platforms (CPU, GPU and FPGA).

1.5.1 System Architecture

This allows task level parallelism and also enables assigning the task best suited for a particular hardware architecture. The mapping of the multi-particle simulation model into a hybrid computation platform can have the FPGA assigned with non-linear tasks, while GPU performing linear tasks. The hardware top-view as in Fig.

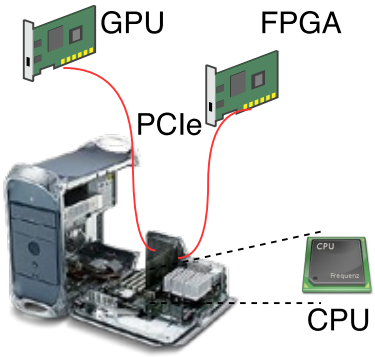

Figure 1.4: Hybrid computing hardware overview.

1.4 of the hybrid platform is to have the host processor connected to the devices using PCIe link for control and data transfer

1.5.2 OpenCL Framework and Resource Mapping

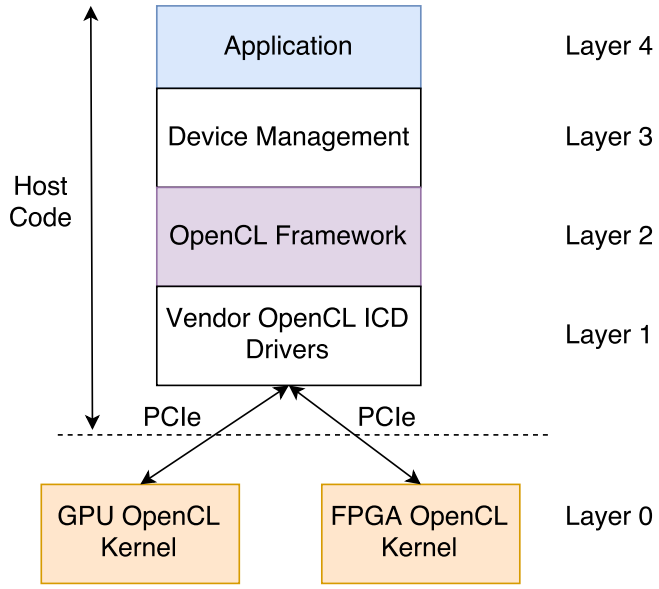

Figure 1.5: Software stack and task mapping.

To provide a unified development environment for different hardware, we designed a software stack based on OpenCL framework. The Fig.1.5 shows the layered approach, that allows scheduling tasks with system level synchronization of devices. A given set of tasks can be assigned to CPU, FPGA and GPU based on the execution efficiency. This also expands the flexibility of the system further by

allowing either to enhance the entire system performance by simultaneous multi-device execution of the same task [6][19] or by mapping non-linear tasks to the FPGA , while routing linear tasks to the GPU. This allows to adapt the simulator performance based on the test configurations. The task behavior are expressed using OpenCL kernels which are loaded to the device during the run-time.

1.6 Design Tools and Methodology

Since the thesis focuses predominantly at the system level, we had the requirement to use different set of design tools, while still aiming to provide a unified development environment. The use of QT framework for GUI and QMake tools enable cross-platform compilation of the host binary. The Matlab and Vivado HLS were used to build the hardware emulation and use it to study different data types and their trade-off. SDAccel from Xilinx provides the development environment for analysing and implementation of FPGA based OpenCL kernel. The Fig. 1.6 shows the main design tools used such as Matlab, Vivado HLS, QT creator, SDAccel and other system tools. All the block diagrams were made using draw.io.

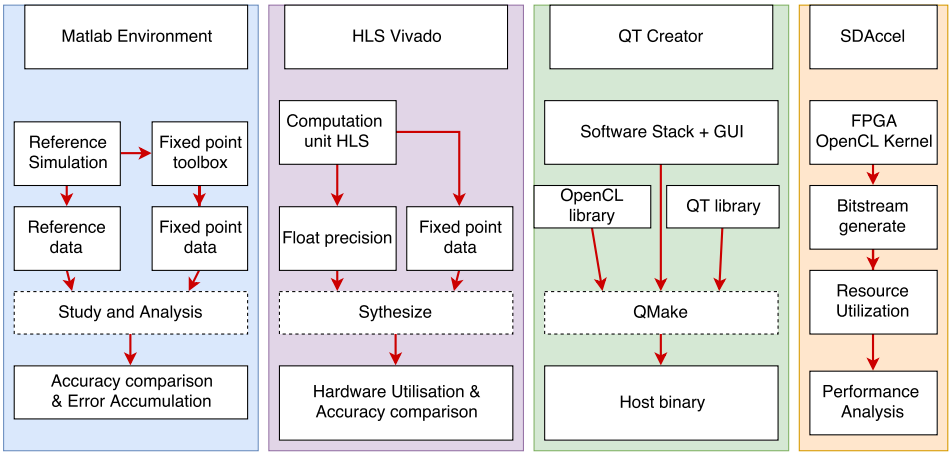

Figure 1.6: Design Tools used in the thesis.

The plots and golden reference data used for verification in other design phases were generated from the Matlab simulation model. By having a cross-platfrom and device independent implementation, we were able to provide maximum code re-use and develop a hybrid computing platform within the time frame of the thesis project.

Chapter 2

Hybrid Computing Platform

This section introduces the hybrid computing platform, and how it provides a unified development environment. OpenCL framework is used to implement the platform, and the software stack used is exposed. The GUI and scalability features are also discussed.

For supporting a scalable and flexible hardware configuration, a hybrid computing platform is required. To enable these features while, providing a unified development environment we make use of OpenCL framework and the layered software stack. As the inherent feature support for a hybrid computing platform, the system can handle task level parallelism and allows to assign sub-tasks to different hardware architectures and to provide maximum throughput for a given configuration as shown in the Fig. 2.2. Since the multi-particle simulation model consist

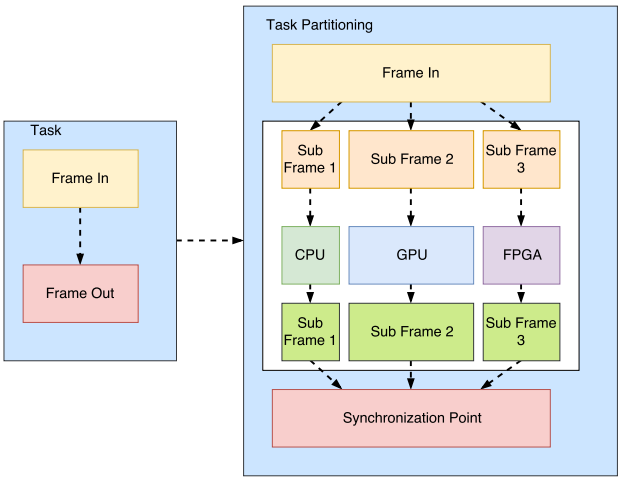

Figure 2.1: The Task Partition in Hybrid Computing.

of linear and non-linear behaviour, the hybrid computing platform can map the linear part of the task to the graphic processor architecture, while the non-linear to the FPGA architecture providing the optimal throughput with efficient power utilization at the system level. The hardware and software architecture of the

hybrid computing platform is explored next.

2.1 System Hardware Architecture

The hardware architecture of the hybrid computing platform consist of a host processor, connected to FPGA and GPU accelerator cards by PCIe interface. By using the OpenCL framework, different configurations of device can be utilized to scale up the hybrid computing platform. The functional portability coupled with layered architecture allows hot plug'n play support to any OpenCL complaint devices.

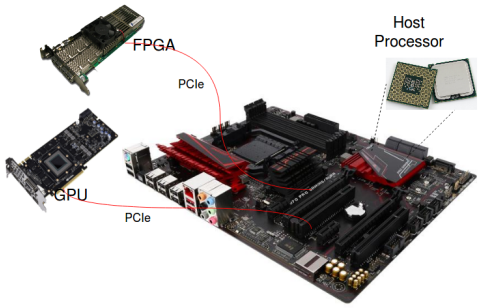

Figure 2.2: The hardware architecture.

2.1.1 Host Processor

For the host processor, we used an Intel I7 6700 HQ which is a quad core processor based on the Skylake architecture. In addition to the four cores with Hyper threading clocked at 2.6 - 3.5 GHz, the chip also integrates an HD 530 GPU and a dual channel DDR4 2133 memory controller. The host processor runs the software stack and manages the device execution.

2.1.2 OpenCL Devices

The OpenCL maps the algorithm efficiently into different hardware architecture, and we implemented GPU and FPGA based accelerators configuration. The FPGA card from Alpha Data, ADM-PCIE-KU3 is used for testing the OpenCL implementation on the FPGA. There are two DDR 3 RAM channels which provides a data bandwidth of 72 bits per sec (with 8 bit reserved for Data Correction) at the rate 1600 Million samples per second. The PCIE link to the host processor is PCI Express® Gen3 x8 . The FPGA hosted on the accelerator is Xilinx Kintex Ultrascale XCKU060 - FFVA1156. The Fig. 2.3 shows the other peripherals present on the target accelerator card.

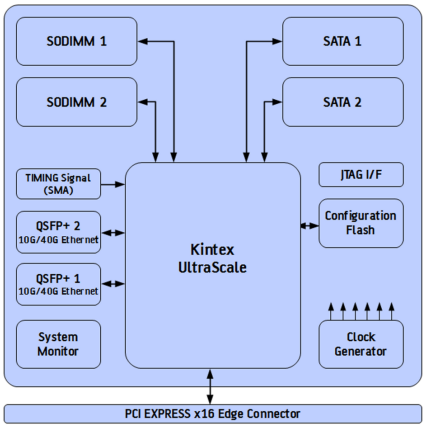

Figure 2.3: The FPGA accelerator card ADM-PCIE-KU3.

The graphics processor, we used in the implementation is a Nvidia 960M; an upper mid-range, DirectX 11-compatible graphics card for laptops unveiled in March 2015. It is based on Nvidia's Maxwell architecture (GM107 chip) and manufactured in 28 nm. The GTX 960M offers 640 shader units clocked at 1097 - 1202 MHz (Boost) as well as fast GDDR5 memory (128 bit, 5000 MHz effective, 80 GB/s).

2.2 System Software Stack

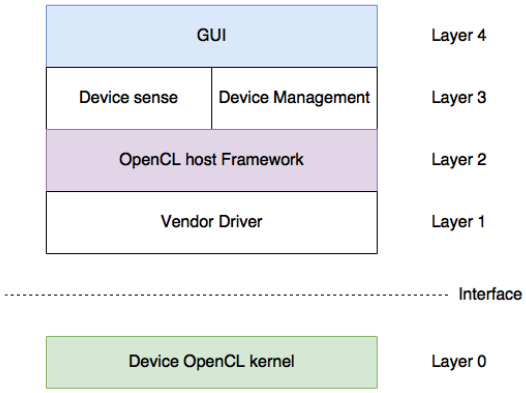

Figure 2.4: The software stack overview.

A layered approach is used to design the host application, to support scalability and maximum code re-use. The top layer is graphic user interface, which provides

access to test configuration and device selection. The device management layer maps tasks to different hardware, while the OpenCL host frame work and Vendor driver establish a device independent application development process. The device layer is the OpenCL kernel, which maps the device behaviour to the algorithm. A detailed overview of each layer is provided next.

2.2.1 Graphic User Interface

The Layer 4 handles the graphical user interface and provides with a basic test configuration and device selection as shown in the Fig. 2.5. QT framework is used to build the graphic elements and the interactions between them.

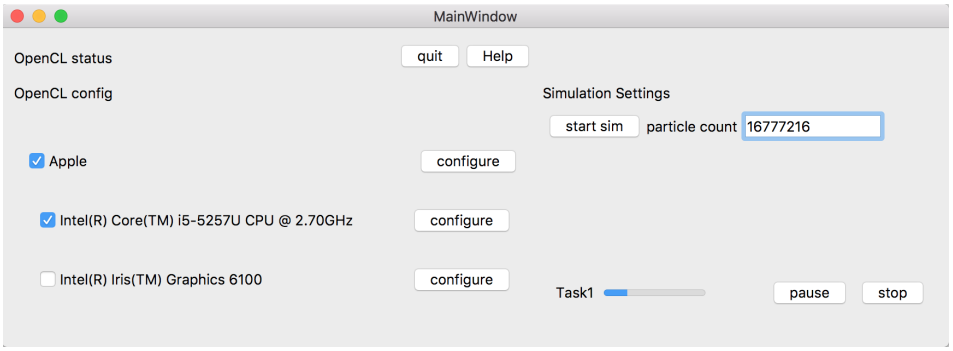

Figure 2.5: The QT based Graphic Interface.

The GUI handling is allocated to a thread, while the layer 3 is executed in a different thread as shown in the Fig. 2.6

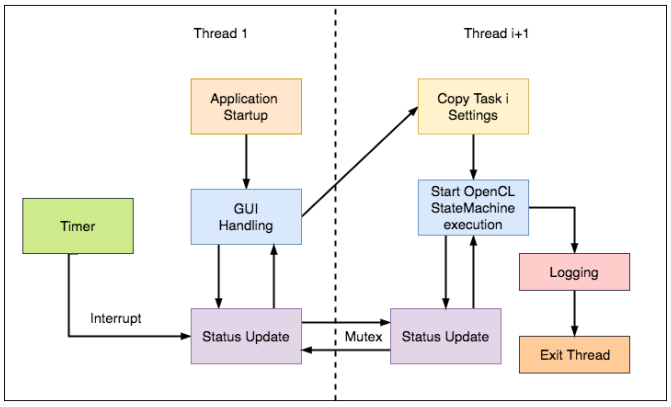

Figure 2.6: The layer 4 and layer 3 communication.

The timer is used to periodically update the GUI elements based on the device status. A mutex (binary semaphore) is used to share the resources between the

graphic user thread and the device management thread.

2.2.2 Device Management

The device management layer is critical in managing the device execution and routing the tasks to different hardware devices.

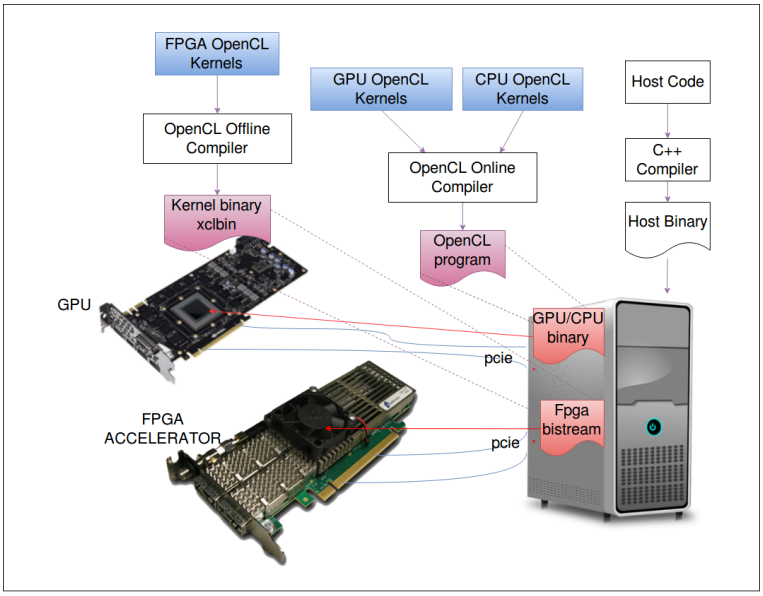

Figure 2.7: The OpenCL program flow

As shown in the Fig 2.7 the device kernel needs to be loaded to the appropriate hardware devices. For GPU and CPU, the device kernels can be compiled on-line and hence can be re-used for different GPU and CPU devices. Since the generation of binary for FPGA takes hours to complete, an off-line compilation process is used instead.

The device management layer also handles the execution range of the device, and manages the synchronization between the devices. A detailed overview of this layer is presented in the Appendix D.

2.2.3 Intermediate Layer 2 and 1

The Fig. 2.8 shows the layer 2 and layer 1 and how they interface the application layer to the hardware. The Installable Client drivers maps the underlying vendor implementation to the OpenCL API, thus separating the application layer from the hardware specific drivers. This also enables connection of new OpenCL compliant devices.

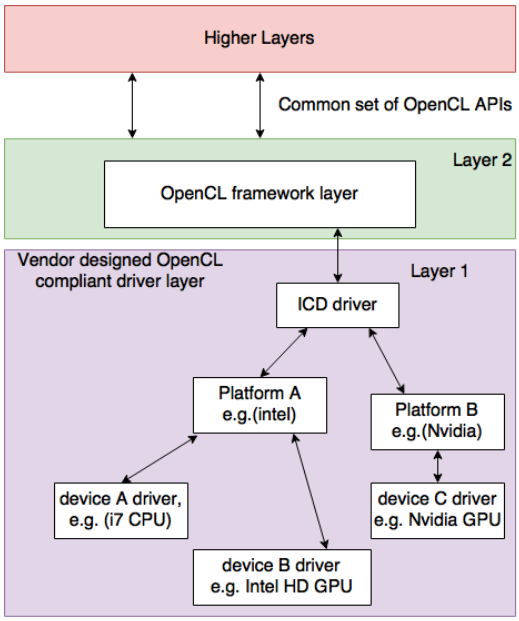

Figure 2.8: Layer 2 and 1.

2.3 Cross-Platform Build

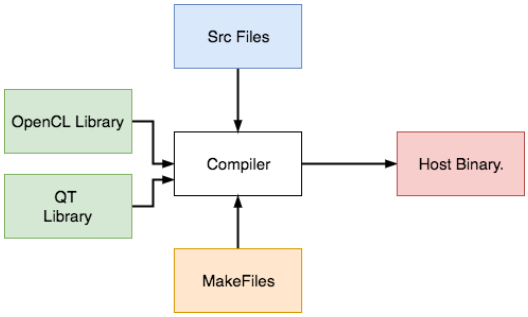

Figure 2.9: The Host code Compilation.

The Qmake is used to link the libraries(OpenCL and QT framework) together and is configured to have build support for Linux, Windows and Mac operating system. The Fig 2.9 represents the compilation flow of the library to produce the host binary files. The required Makefiles are generated and can be used to build the source code.

Chapter 3

OpenCL Device Implementation

In this chapter, the mapping of algorithm to different hardware architectures is explored. Layer 0 in the software stack (OpenCL kernel) for CPU, GPU and FPGA is implemented and then optimized for the architecture.

3.1 OpenCL Kernel

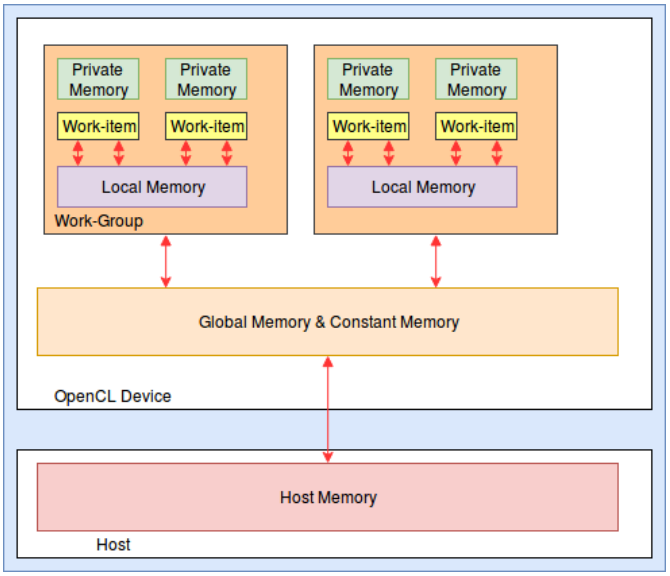

Figure 3.1: OpenCL Abstractions in Processing and Memory levels.

The OpenCL has a well defined Memory and Kernel programming model, which allows to express the parallel concurrency into the target hardware as shown in the Fig 3.1. This allows describing the behavior of a thread, and their synchronization within a computation unit.

The unit of work performed by a thread is termed work item, and a collection of work items forms a work group in OpenCL. By using these abstraction with

their execution space (NDRange space), allows scheduling these work units across parallel frames of data.

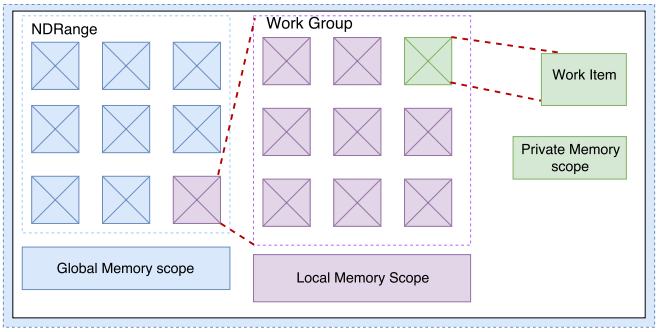

Figure 3.2: The OpenCL execution model with NDRange and Memory scope.

The Fig 3.2 shows a 2 dimensional NDrange (execution space for the work-items) with a set of work groups, with each work group representing a set of work items. The memory scope of global memory, local memory and private memory also follows the same hierarchy. The global memory is shared between all of the work groups, while the local memory is shared within a work group and the private memory has it's scope limited to a work item. By using these abstractions, the transport matrix computation is mapped to the target hardware providing maximum throughput by efficient utilizing of the available parallelism. More detailed information can be found in the OpenCL text[18] and other vendor specific documents.

3.1.1 Frame Partition and Subframe Scheduling

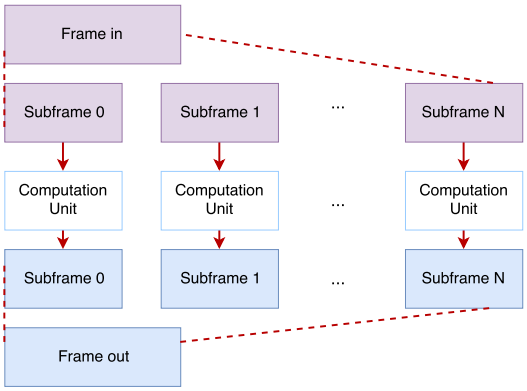

Figure 3.3: The subframe scheduling and partition

After implementing the OpenCL kernel, two parameters can be used to fine-tune the kernel to the hardware architecture. As shown in the Fig. 3.3, by assigning a proper value for work group size, the OpenCL allows to determine the size of the sub-frame and by declaring the NDRange space, the number of sub-frame divisions to be scheduled across the available computation units within a hardware device can also be computed. If there are too many subframes, there is an overhead associated with scheduling them to the available computation units, and if there are too little, the amount of available parallel resources in the hardware cannot be fully utilized.

3.2 OpenCL Implementation on CPU

The Host DDR memory holds the frame data and transport matrix. First, the transport matrix is moved to the cache of each SIMD core, while subframes are moved from DDR memory to cache, executed across the available SIMD cores and moved back to the DDR memory. AVx instruction sets provides vectorization at each SIMD core.

Input: $frame_{in}, T_{matrix}$
Output: $frame_{out}$
for $i < Depth$ **do**
| $localFrame_{in}$[i] = $framein[i]$
end
for $i < Depth$ **do**
| $localFrame_{out}$[i] = $localFrame_{in}$[i] $\times T_{matrix}$
end
for $i < Depth$ **do**
| $frame_{out}$[i] = $localFrame_{out}$[i]
end

Algorithm 1: Pseudo code for CPU work-item.

For the implementation in CPU, the NDRange is declared as the total number of particle in a frame divided by the depth of the kernel(size of the subframe). Here the work item and work group points to the same hardware SIMD core, and the depth represent the size of subframe moved to their local cache. An efficient burst transfer is implied between the cache and the off-chip DDR memory by moving the data as a chunk.

Target Implementation	Execution Time	Gain
c++ benchmark	3.277 s	-
opencl depth 16	208.781ms	15.71
opencl depth 32	217.781ms	14.87
opencl depth 64	194.781ms	16.63
opencl depth 128	228.781ms	14.15

Table 3.1: OpenCL on CPU results.

A test configuration of 1024 × 1024 particles for 15 iterations were used to

fine-tune the depth of the kernel for the CPU used. As per the table 3.1 a kernel depth of 64 was optimal for the I7 6700 HQ processor for a similar work load.

These training workloads needed to be performed for fine-tuning the implementation for a different set of CPU which may have more hardware resources available.

3.3 OpenCL Implementation on GPU

The GPU architecture is designed for handling parallelism in a massive scale and in order to map the algorithm to the it's architecture, a different set of parameters and behaviour description is required. The entire frame data and transport matrix is moved the GDDR5 RAM in the GPU accelerator card. The frames are divided into sub-frames and allocated across different computation units. The Transfer matrix is stored into the local cache of the computation unit. Each subframe is transported to output subframe and the results moved back to the GDDR5 memory.

For mapping these processes to OpenCL kernel, each work item associates with a processing element or thread in computation unit, and a computation unit is mapped to a work group. Thus, the size of the subframe which is allocated to a computation unit can be decided by using the parameter local work group size. The NDRange space should be the total number of particle count from the test configuration. Synchronization between the work items in a work group needs to be provided and are done at the end of these three sub-tasks using a local memory barrier. The implementation as shown in the Fig. 3.4 consist of these sub-tasks: Reading to local memory of computation unit, computation unit execute and write from local memory of the computation unit.

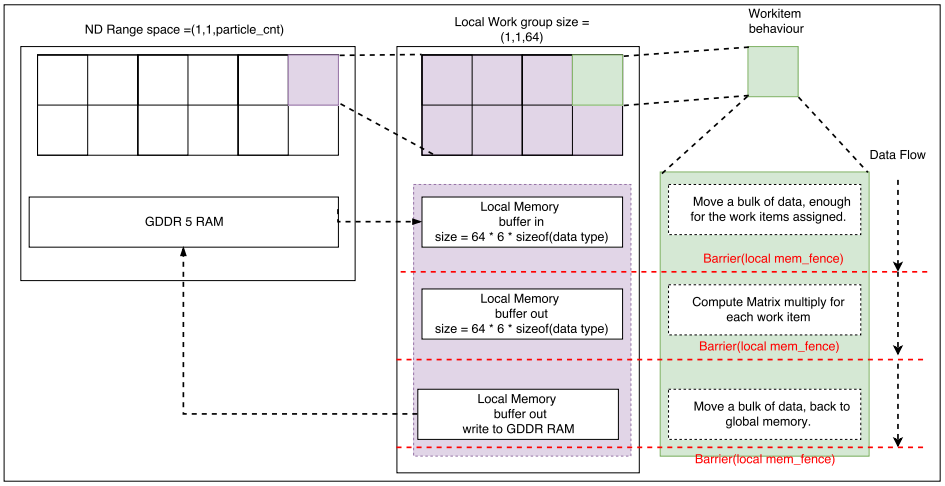

Figure 3.4: The GPU OpenCL implementation.

Similar to the CPU implementation, a training test configuration is used to

fine-tune the kernel implementation for the required work-load. As shown in the

Target Implementation	Execution Time	Gain
c++ benchmark	3.277 s	-
opencl work group size 32	45.781ms	72.82
opencl work group size 64	37.93ms	86.46
opencl work group size 128	not feasible	-

Table 3.2: OpenCL on GPU results.

table3.2 work-group size of 64 provided the optimal execution time for this case.

3.4 OpenCL Implementation on FPGA

By taking advantage of the fact that the FPGA starts off as a blank computational canvas, the user can decide the level of device customization that is appropriate to support a single application or a class of applications. In determining the level of customization in a device, the programmer can take advantage of the fact that kernel compute units are not placed in isolation within the FPGA fabric. FPGA devices capable of supporting OpenCL programs can include, but are not limited to, the following components:

- DMA engines.
- I/O peripherals such as PCIe and Ethernet.
- Memory controllers
- Custom interconnects
- OpenCL compute units
- RTL-based accelerators

Since we are using Xilinx based FPGA accelerator card, the implementation details are related to the infrastructure local to Xilinx OpenCL implementation. As shown in the Fig. 3.5, the FPGA is partitioned into two regions: the static region and the OpenCL region. The Static region provides the control and communication to the host computer, while the compute units generated from user kernel functions are placed in OpenCL region. These compute units (Computation units) are highly specialized to execute a single kernel function and internally contain parallel execution resources to exploit work-group level parallelism. By placing multiple compute units of the same type in the OCL Region, developers can easily scale the performance of single kernels across larger NDRange sizes. By placing multiple compute units of different types in the OCL Region, developers can leverage task parallelism between disparate kernels. In this way, the massive amounts of parallelism available in the FPGA device can be customized and harnessed by the SDAccel developer. This is different from CPU and GPU implementations of OpenCL which contain a fixed set of general purpose resources

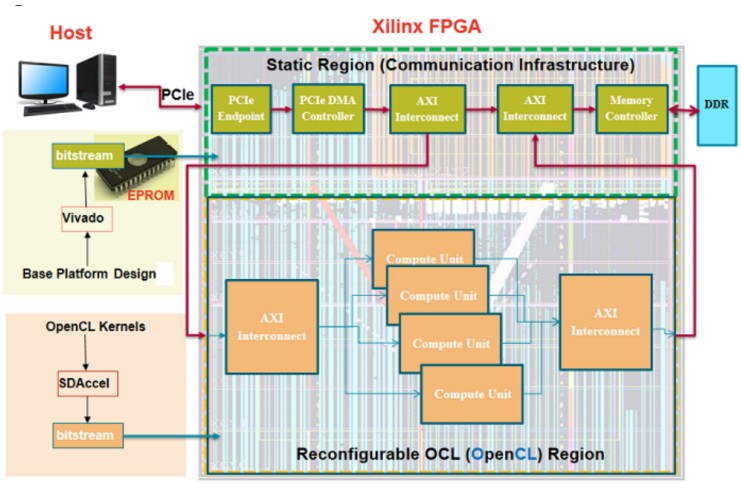

Figure 3.5: The FPGA OpenCL implementation.

3.4.1 Implementation

In this section, we describe the implementation of the multi-particle simulation model on the FPGA. The initial frame data is stored into the FPGA Off-chip memory using PCIe from the host CPU, while the transport matrix is stored in the register banks of each computation units. A computation unit also consist of FIFO structures with highly parallel execution units. The transported frame is also stored in the off-chip DDR memory after execution.

We explored memory hierarchy and parallelism at two levels to provide the optimal throughput for the memory bandwidth. As in the Fig 3.6, we have multiple instances of computation units interfaced to the off-chip DDR memory and each computation units have independent execution of vector arrays with dedicated local memory. Next, we discuss the memory level and processing level optimization applied at each computation units

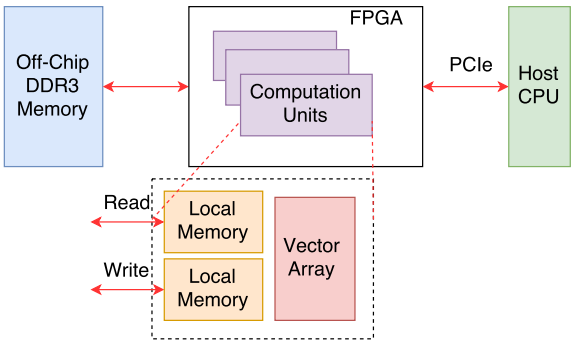

Figure 3.6: FPGA architecture top-view.

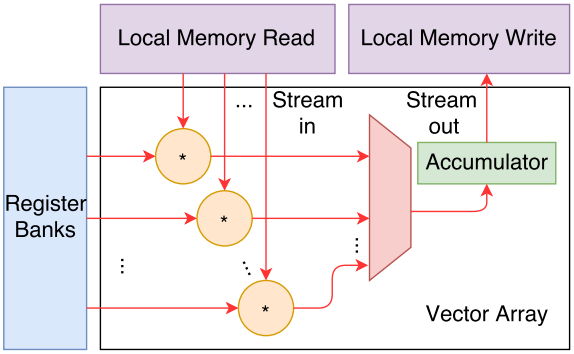

Figure 3.7: Vector array top view.

3.4.2 Memory Optimizations

Off-chip memory access and on-chip memory bandwidth to match the consumption rate of parallel processing elements is critical for a high throughput design.

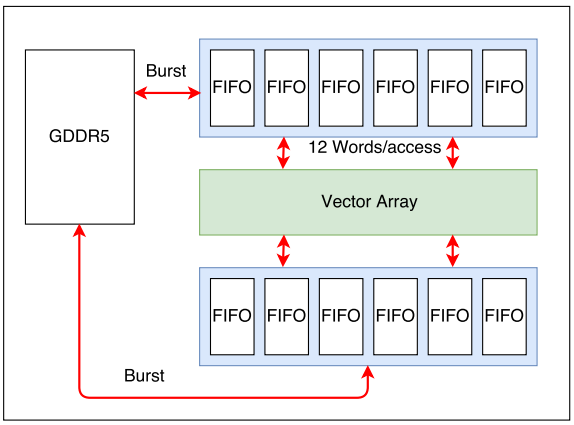

Figure 3.8: Memory optimizations

In order to efficiently transfer data from the off-chip (DDR) memory, burst transfer is inferred into the local memory of each computation unit as shown in the Fig. 3.8. The local memory is split into six FIFOs where, the memory mapped data from the off-chip memory is transformed into a streaming data, processed and transferred back. 12 words per clock bandwidth can be achieved by the six instances of FIFOs to match the consumption rate of the vector arrays. The transfer matrix column data is also stored in 6 different register banks to provide efficient vectorization.

3.4.3 Processing Optimizations

Fig. 3.7 shows the block diagram of the computation units (vector array). By storing the transport matrices in register banks, our architecture allows access to

6 column elements and executes the vector operations in (A.4) onto the streaming data from the local memory. These are pipelined and the results are accumulated in vector register. There are two instances of these vector execution blocks in a computation unit and each of these parameter are scalable. By subdividing the read, execution and write sub blocks, a data-flow optimization is also applied at this stage, to reduce the effective latency of the entire block to max latency of the sub-blocks.

3.4.4 Performance Analysis

By increasing the number of computation units to 8, we were able to maximize the throughput for the DDR 3 off-memory bandwidth. Fig. 3.9 shows efficient utilization of the memory bandwidth.

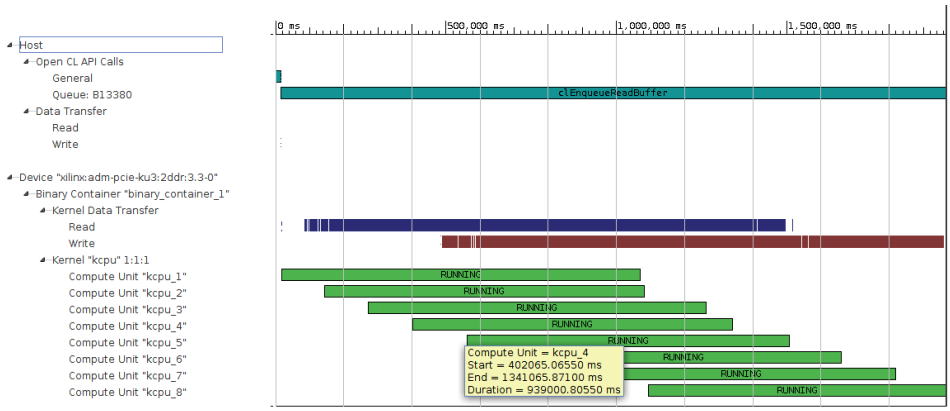

Figure 3.9: The OpenCL Computation units of 8 allows more consistent memory bandwidth utilization.

The HW emulation mode in SDAccel was used to generate a quick analysis of the kernel performance. For a test configuration with 4096 particle count, the kernel performance is listed in the table 3.3.

Target Implementation	Kernel Depth	Computation units	Execution Time	Gain
FPGA Kernel w/o opt	16	1	3.06ms	-
FPGA Kernel w/o opt	512	1	2.010ms	1.52
FPGA Kernel w opt	512	1	.259ms	11.815
FPGA Kernel w opt	512	4	.085ms	36
FPGA Kernel w opt	256	8	0.075ms	40.8

Table 3.3: kernel performance with different configurations.

Chapter 4
Further Optimizations for FPGA

Since the memory bandwidth and storage is directly related to the data type representation, a study is performed for feasibility of different word-lengths. A hardware utilization for float precision and fixed precision is also studied.

4.1 Fixed Point Feasibility

Whether Fixed point data type is feasible with the beam simulation model, can decided by analysing the dynamic range of the beam data and then by analysing the amount of noise injected due to quantization. A suitable word-length then needs to be selected to provide an optimal utilization of memory.

4.1.1 Dynamic Range

For studying the beam behaviour, a test configuration of 1024 particles iterated over 5001 FODO cells is set. It was found that, the beam state parameters always lie within a fixed dynamic range as shown in the Fig 4.1 It is also evident that, the

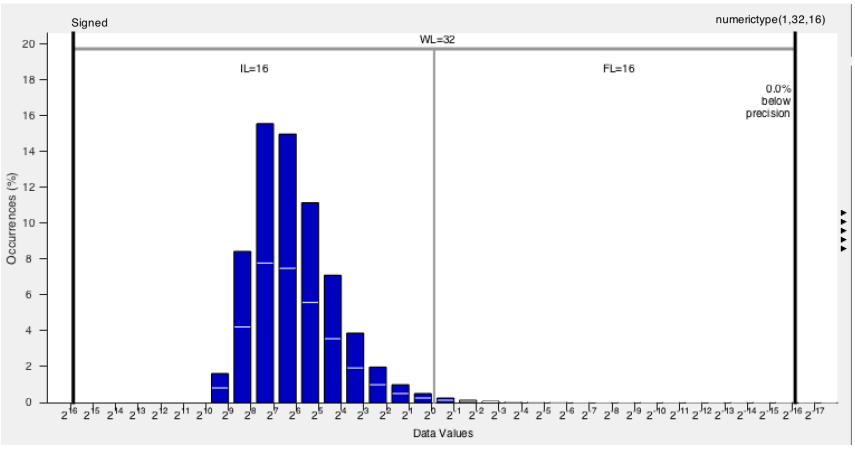

Figure 4.1: The dynamic Range of the beam with binary weights of the data.

occurrence of certain bits are much more frequent than others. This property can be used for lossless compression of the frame and hence provide a lesser memory storage.

4.1.2 Quantization Error

The quantization error is due to assigning the real value numbers to the fixed domain space defined by the word length. There are three major sections were the quantization error is injected into the implementation model. Assuming a different word length is used for frame and for transport matrix:

- Quantization error at converting the initial frame and transport matrix to fixed domain.
- Quantization error at computation stage
- Quantization error due to truncation the results down to the frame word length.

To model these behaviour into simulation , a hardware emulation model is developed in Matlab with FI toolbox. More details of the emulation model is presented in Appendix B and C. The behaviour of the beam frame is studied for extended iterations as shown in the Fig. 4.2. The word length of 40 bits provides an optimal level of accuracy against the double precision.

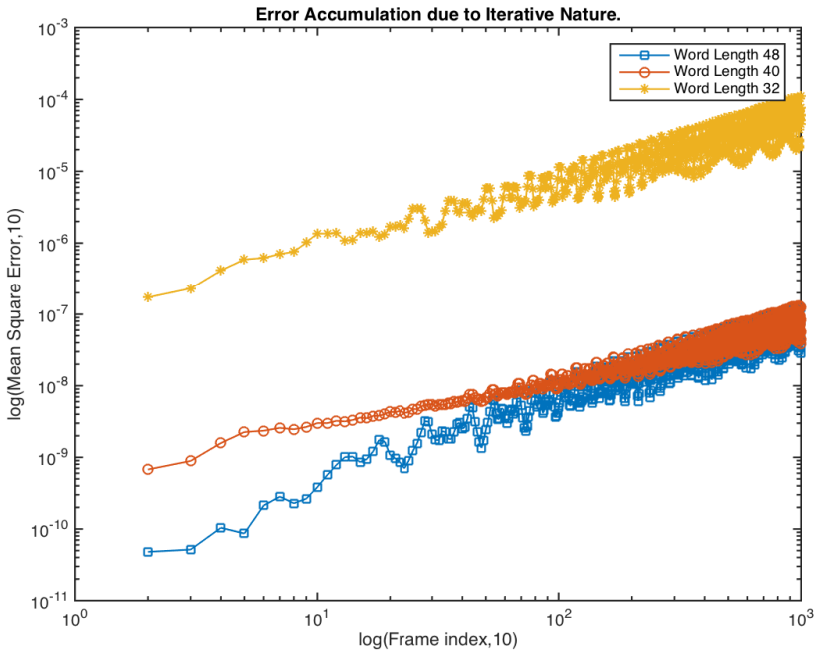

Figure 4.2: The accumulated error for different word-lengths

4.2 Data-type Trade-off

Next, fixed point vs float precision implementation is studied in the context of hardware utilization against the memory utilization. As shown in the Fig. C.1 the hardware model of the compute unit using Vivado HLS is synthesized. This provides a good context to study the FPGA utilization in hardware and their accuracy.

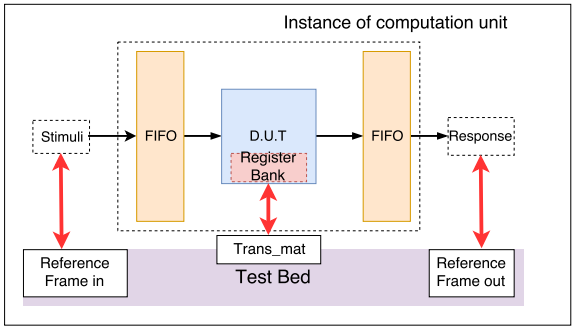

Figure 4.3: The HLS test bed and instance of computation unit.

The data bandwidth into the model and out of the model is set to 32 bits/transfer with the depth of the FIFO set as 256×6 words(for 256 particles). In table 4.1, the worst case latency is shown as the sum of read, execution (Multiply Frame) and write stage. From the analysis on the model, we see that, the min net latency is 1538 clock ticks (memory transfer), due to the data-flow optimization.

Instance	Latency Min(ticks)	Latency Max(ticks)
Read Buffer	1538	1538
Multiply Frame	778	778
Write Buffer	1538	1538
Net	1538	3856

Table 4.1: Performance results of the design.

The data flow optimization interleaves the sub-blocks to reduce the effective latency of the model to that of the latency of the critical block. It can be seen that, the latency for computation is less compared to the memory transfer. Thus for this design, memory bandwidth is the main bottle-neck.

4.2.1 Hardware utilization

FPGA resources utilizations varies with the data types as it has a direct impact on the bus, memory and the overall architecture due to the flexible nature of the FPGA fabric. The table 5.2 shows hardware utilization for fixed point against float precision. For the same word length, the float utilizes less DSP slices when

compared to the fixed precision due to the wider multiplier instances needed for fixed point. However, the LUT and FF resource utilization is higher for float.

DataType	LUT	FF	DSP48E	BRAM
fixed point 32 bits	4738	4581	144	24
fixedpoint 40 bits	6250	5261	180	48
Float single precision	7961	10943	60	24
Float double precision	16431	21794	168	48

Table 4.2: Hardware utilization across different data types.

4.2.2 Accuracy Comparison

The deviation from double precision is measured for the frame generated from the HLS model. The table 4.3 shows the accuracy for different data types. It is noteworthy to mention, for the same word-length, the fixed precision has lower error compared to the float precision.

DataType	Mean Square Error
fixedpoint 32 bits	$1.39e^{-06}$
fixedpoint 40 bits	$5.26e^{-09}$
Float single precision	$4.446e^{-06}$
Float double precision	-

Table 4.3: Hardware accuracy across different data types.

Chapter 5
Results and Analysis

The results of the implementation for different device type is showcased here. Afterwards, the results for different data types, comparing the accuracy with hardware utilization is also discussed.

Implementation	Device	Memory Type	Memory Bandwidth	Execution speed.
C++ CPU	i7 6700HQ @3.3 GHz	DDR4	128 bit @ 2133 MHz	1x
OpenCL CPU	i7 6700HQ @3.2 GHz	DDR4	128 bit @ 2133 MHz	16x
OpenCL GPU	Nvidia 960M @1.2 GHz	GDDR5	128 bit @ 5000 MHz.	89x
OpenCL FPGA	AlphaDataKU3 @350 MHz	DDR3	128 bit @ 1600 MHz	28x

Table 5.1: Comparison of linear model execution speed across different devices

DataType	LUT	FF	DSP48E	Memory bandwidth reduction	M.S.E
FP-32	1.4%	0.6%	5 %	2x	$2.34e^{-07}$
FP-40	1.8%	0.7%	6.5 %	1.67x	$5.26e^{-09}$
Float-single	2.4%	1.6%	2%	2x	$4.46e^{-06}$
Float-double	4.9%	3.3%	6%	-	-
Total	331k	663k	2.7k	-	-

Table 5.2: Hardware utilization and processing accuracy for different data types

Table 5.1 shows the implementation results of the linear model across the test hardware, when double precision data is used. In general, OpenCL framework

provides significant performance improvement over C++ framwork, due to the efficient mapping of resources. Table 5.1 also provides a general trend towards GPU favoring linear part of the simulation model, while FPGA performs better than a CPU. It is worthwhile to be emphasized here that the memory bandwidth is the implementation bottleneck for the simulator and the memory access capabilities of different devices, to some extent, decides the performance.

Table 5.2 lists the simulation accuracy and hardware utilization (of FPGA) of different data types. It is seen that the processing occupies very little FPGA resources, confirming the analysis that memory bandwidth is the limiting factor. It also is evident that there is more parallelism available to exploit provided the incoming data rate can be enhanced. To increase the memory bandwidth, one approach is to improve the number of memory banks or upgrade the memory technology, while the other approach is by utilizing a smaller word length for data computations. It is interesting to see that, for the same word-length, the fixed point provides better precision than a single float precision. Moreover, by using a 40 bit word length, a bandwidth reduction of 1.67× to the memory access can be achieved compared to a double precision data type.

Chapter 6
Future Work

The thesis has explored hybrid computing platform with a device configuration of CPU, GPU and FPGA. We have mapped efficiently the linear behaviour of the multi-particle simulation model to the hardware architectures. However, the key areas that can improve the functionality and efficiency further are listed below.

- Implement space charge task and other non linear models into the hybrid computing platform
- Improve memory bandwidth utilization by using compression, increased I/O channels and custom data-types
- Explore other high-performance hybrid computing configurations

In addition to these key areas, the accuracy with data types, improvement in the software stack, and usage of compression techniques to improve the memory storage and bandwidth are also critical.

The accuracy of the simulation model is a direction to explore further, the thesis used the double precision as the reference and this leaves with the question, which is a better representation than double precision? A real beam data from the accelerator is needed to make further steps at optimizing the accuracy of the model.

The software stack requires the development of an autonomous task partition layer, that manages the execution and task allocation across different devices with an adaptive control feedback. Similar works [6] and [22] utilizes the task scheduler with multi-device support and adaptive task partition based on the performance of the device.

Considering that there is more hardware processing left for parallelism, methods to reduce the off-chip Memory access frequency will increase the performance further. Compression methods to reduce the size of the frame allocated in the DDR memory and to decompress them during the execution is a novel method to tap in more parallelism for a bandwidth constrained scenario. This allows to utilize more FPGA on-chip memory which is faster and also reduces the access frequency to the off-chip memory.

References

[1] Christopher E. Peters , Clayton Dickerson, Francisco Garcia, & Maria A. Power, Integration of the track beam dynamic model to decrease Linac tuning times, *ICALEPCS 2015* Melbourne, C15-10-17, 2015

[2] Qiang, Ryne, Habib,& Decyk, An Object-oriented parallel Particle-in-CELL code for beam dynamics Simulator in Linear Accelerators, *IEEE conference on Supercomputing* Article No. 55, 1999

[3] Implementing FPGA Designs with the OpenCL standard, ALTERA whitepaper, November 2013

[4] OpenCL optimizations case study: Simple reductions, AMD white-paper, 2013

[5] Ji Qiang, Fast 3D Poisson solver with longitudinal periodic and transverse open boundary condition for space-charge simulations, *Comput.Phys.Commun.* 219 255-260, 2017

[6] Kai Maa, Yunhao Baia, Xiaorui Wanga, Wei Chena, Xue Lib, Energy conservation for GPUCPU architecture with dynamic workload division and frequency scaling, *Science Direct* Volume 12, Pages 21–33, December 2016

[7] K. R. Crandall & D. P. Rusthoi, TRACE 3D Documentation manual

[8] X. Pang, L.J. Rybarcyk, S.A. Baily, A multi-particle online beam dynamic simulator for high power ion Linac operations *54th ICFA Advanced Beam Dynamics Workshop*, MOPAB30, 2015

[9] Xiaoying Pang, Advances in Proton Linac online modeling, *IPAC'15*, pp 2423-2427, 2015

[10] Paul Grigoras¸ Pavel Burovskiy, Wayne Luk, & Spencer Sherwin, Optimizing sparse matrix vector multiplication for large scale FEM problems on FPGA, *(FPL'15) IEEE*, 1–6, Aug 2016

[11] Wenqi Bao, Jiang Jiang, Yuzhuo Fu, Qing Sun, A reconfigurable macro-pipelined systolic accelerator architecture *Field-Programmable Technology (FPT) International Conference* 12-14, Dec. 2011

[12] Andrzej Wolski, Beam Dynamics in High energy particle accelerators

[13] Haitao Wei, Guang R. Gao, Elkin Garcia, Energy efficient Multi-level tiling for dense matrix multiplication on many-core architecture *Green Computing Conference and Sustainable Computing Conference (IGSC), 2015*, 14-16, Dec 2015

[14] khronos group homepage for OpenCL, webpage, 2017 `https://www.khronos.org/opencl/`

[15] The OpenCL reference guide, 2017

[16] Megumi Ito and Moriyoshi Ohara, A power-efficient FPGA accelerator: Systolic array with cache-coherent interface for PARI-HMM algorithm *IEEE Symposium in Low-Power and High-Speed Chips*, pp. 1-3, 2016

[17] Danyu Bi, & eric-sardella Intel out of order OpenCL execution MODEL, Intel Whitepaper June, 2017

[18] David R. Kaeli, Perhaad Mistry, Dana Schaa, & Dong Ping Zhang, Heterogeneous Computing with OpenCL 2.0 3rd edition

[19] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, & Satoshi Matsuoka Evaluating and Optimizing OpenCL kernels for high performance computing with FPGAs, *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 13-18, Nov. 2016

[20] SDX Development Environment user-guide, June, 2017

[21] SDX OpenCL Optimizing guide, 2017

[22] C.-K. Luk, S. Hong, H. Kim, & Qilin, Exploiting parallelism on Heterogeneous Multi-processor with adaptive mapping, *Microarchitecture, IEEE/ACM International Symposium*, 12-16 ,Dec. 2009

[23] John Cheng, Max Grossman, and Ty McKercher, Professional CUDA C Programming, 2014

Appendices

Appendix A

Beam Simulation Theory

In this chapter, the theory and physics used to implement the multi-particle simulation model is discussed.

A.1 Theory

In order to model the beam dynamics, while providing a realistic behaviour and support for non-linear particle interaction in the beam, a multi-particle based simulation model is utilized. [9] provides with some of the existing simulation models used in the linear accelerators around the world.

The dynamics of the particles in a beam is fundamentally described based on two steps;

- By representing all the equations of motions affecting the particles
- Then, to solve them to provide the solution for their behaviour.

. Newtons second law as in equationA.1 represents the of motion of particle once, the forces are known.

$$\frac{dp}{dt} = F, \tag{A.1}$$

Though this method can be used, most of the times the electric field and Magnetic fields in an accelerated beam are specified as a function of location rather than time. Hence, a Hamiltonian mechanics based methodology is followed, as it allows for representing the particles as a function of distance along the beam line.

Approaching this problem with Hamiltonian method, makes it simpler to represent the moving particles through a potentially complex sequence of electric and magnetic fields. [12] provides with derivations of the equations and explains how the Hamilton's equation generalized as in equation A.2 where x_i are the co-ordinates of the particle($i = 1...N$ in an N dimensional co-ordinate space), p_i are the components of momentum and H is the Hamiltonian.

$$\frac{dx_i}{dt} = \frac{\partial H}{\partial p_i}, \frac{dp_i}{dt} = \frac{\partial H}{\partial x_i}, \tag{A.2}$$

A.2 can be used to derive for a relativistic particle moving in an electromagnetic field, with the Hamiltonian defined as

$$H = c\sqrt{(p - qA)^2 + (mc)^2 + q\phi} \tag{A.3}$$

Substituting the electromagnetic potential in a particular region of space into Hamiltonian allows to derive the equation of motion of the charged particles moving through the region.

By employing this method, it is possible to find solutions to these equations for commonly used structures/components in the linear accelerators.

A.2 Linear Accelerator Structures

There are different structures, for controlling the beam behaviour and these structures can be modelled as transport matrices. The quadrupole and drift space structures that we used for the thesis, is described below.

A.2.1 Drift Space.

The region of space ,where there are no electric or magnetic fields present is considered as the drift space, where particles are drifting. The derivations on how the equations are solved can be referred from the text, [12] and is not reproduced here as such.

The linear transfer map in matrix form can be expressed for a particle as shown in equation A.4,

$$\bar{x_1} = R_{drift} \times \bar{x_0} \tag{A.4}$$

,where x_i is the phase space vector with components constructed from the dynamic variables, such as the position and momentum in 3-dimensional space.

$$\bar{x} = \begin{pmatrix} x \\ px \\ y \\ py \\ z \\ \partial \end{pmatrix} \tag{A.5}$$

and the R_drift can be represented as shown in equation A.6

$$R_{drift} = \begin{pmatrix} 1 & L & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & L & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{L}{\beta_0^2\gamma_0^2} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{A.6}$$

here, β_0 is the velocity of the reference particle scaled by the speed of light, L the unit distance under consideration and γ_0 is given by the equation A.7.

$$\gamma_0 = \frac{1}{\sqrt{1 - \beta_0^2}} \tag{A.7}$$

A.2.2 Quadrupole Structures

Multipole magnets are one of the important components used in a linear accelerator. They constitute a family where, the dipoles are used for steering, the quadrupole are used for focusing and sextupoles provides with corrections in the chromatic aberrations. In an accelerator beam line, the purpose of quadrupole magnets is to control the beam size by providing transverse focusing. Similar to the drift space, the Matrix to map the behavior of the particle across the quadrupole region of space can be represented as in the equation A.8.

$$R_{quad} = \begin{pmatrix} cos(\omega L) & \frac{sin(\omega L}{\omega} & 0 & 0 & 0 & 0 \\ -sin(\omega L) & cos(\omega L) & 0 & 0 & 0 & 0 \\ 0 & 0 & cos(\omega L) & \frac{sin(\omega L}{\omega} & 0 & 0 \\ 0 & 0 & \omega sin(\omega L) & cos(\omega L) & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{L}{\beta_0^2 \gamma_0^2} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{A.8}$$

The equation A.8 can be manipulated to provide for focusing in the x axis or in the y axis by changing the parameters. More details are presented in the Matlab functional model section as, it is performed for the Fodo cells.

A.3 Matlab Functional Model

The functional model is verify the functionality and act as a reference for other tests performed. Analysis of the simulation model is also performed. The initial state for the simulation model is generated with the configurations set from ESS.

A.3.1 Initial particle state

The initial state of particle are generated by the given set of parameters provided from ESS. The particle beam are uniformly distributed in a circular region of space. These are then transformed into an ellipsoidal space by multiplying with the transformation matrix.

$$\bar{y_0} = \begin{pmatrix} \sqrt{\frac{\beta_x}{\sigma_x}} & 0 & 0 & 0 & 0 & 0 \\ -\frac{\alpha_x}{\sqrt{\sigma_x \beta_x}} & \frac{1}{\sqrt{\sigma_x \beta_x}} & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{\frac{\beta_y}{\sigma_y}} & 0 & 0 & 0 \\ 0 & 0 & -\frac{\alpha_y}{\sqrt{\sigma_y \beta_y}} & \frac{1}{\sqrt{\sigma_y \beta_y}} & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{\frac{\beta_x}{\sigma_x}} & 0 \\ 0 & 0 & 0 & 0 & -\frac{\alpha_z}{\sqrt{\sigma_z \beta_z}} & \frac{1}{\sqrt{\sigma_z \beta_z}} \end{pmatrix} \times \bar{x_0} \tag{A.9}$$

Equation A.9 shows for $\bar{x_0}$ being transformed into ellipsoid space $\bar{y_0}$ as per the alpha and beta parameters provided.

A.3.2 Fodo cell simulation

The pseudo code for the Fodo cell is given below as Algorithm 2. The $beam_{init}$ and the transfer matrices are generated by the initialization script. The variable $beam_{behav}$ is a 3 dimensional variable storing the phase space vector of each particle for the 5 states within a cell for the max number of cells provided.

Data: $beam_{init}, R_{drift}, R_{foc}, R_{defoc}$
Result: Generate $beam_{behav}$
initialization;
$cell_n = cell_{max}$
$particle_{cnt} = particle_{max}$
while $cell_n < cell_{max}$ **do**
  while $particle_{cnt} < particle_{max}$ **do**
    if $cell_n == 0$ **then**
      $beam_{behav} = R_{foc} \times beam_{init}$;
      $beam_{behav} = R_{drift} \times beam_{init}$;
      $beam_{behav} = R_{defoc} \times beam_{init}$;
      $beam_{behav} = R_{drift} \times beam_{init}$;
      $beam_{behav} = R_{foc} \times beam_{init}$;
    else
      $beam_{behav} = R_{foc} \times beam_{behav}$;
      $beam_{behav} = R_{drift} \times beam_{behav}$;
      $beam_{behav} = R_{defoc} \times beam_{behav}$;
      $beam_{behav} = R_{drift} \times beam_{behav}$;
      $beam_{behav} = R_{foc} \times beam_{behav}$;
    end
  end
end

Algorithm 2: Pseudo code for Fodo cell simulation.

A.4 Application Benchmark and Verification

To compare the performance of the OpenCL implementation, a C/C++ based benchmark program, is implemented. It computes the beam behaviour and the time to execute, but with traditional object oriented programming methodology on a single thread. First, the behavior is cross checked with the Matlab model to ensure functional coherence. By using the C application, the discrepancy for the benchmark across different systems can be mitigated, as the test is carried out in different host systems.(with different operating systems housing different hardware)

Appendix B

Matlab Hardware Emulation Model

This section discusses methodology used to develop the hardware emulation model using Matlab fixed point toolbox. This model is used for studying error accumulation and for analysing the beam behaviour.

B.1 Matlab Fixed Point Emulation

Hardware like functionality is emulated in Matlab and used to run the beam simulation. The data is collected and analysed and can be a very powerful tool to understand the algorithm before the hardware development phase. The initial frame data and the result generated from the functional model, is reused in this script for studying the effects such as quantization of the initial beam, and the transfer matrix coefficients, and at the output stage where the output frame is truncated to the frame word-length.

B.2 Quantization at The Input Frame and Transfer Matrices

For analysing, the effect of quantization error at the input and the transfer matrix coefficients, the data is first scaled up by scaling factor to spread the data across the dynamic range(defined by the word-length). This maximises the precision as it is converted into fixed point domain as shown in equation B.1, where *fi* is the function that maps the data into finite fixed point space.

$$beam_{fixed} = fi(beam_i \times scalingfactor) \tag{B.1}$$

The fixed point toolbox is configured with truncation of data and, the overflow is managed by swing around the max range. The result, i.e. the beam frame in fixed point is stored as double precision and then scaled down scaling factor.

The quantization error for the initial beam state is show in fig B.1 for the word length of 32 bits. The mean square error is found to study the induced error due to the conversion as in eqn B.2 for the range of word lengths under consideration.

$$M.S.E = \sqrt{\frac{\sum(beam_{fixed} - beam_{double})^2}{N}} \tag{B.2}$$

The results of comparison are shown in the table B.1.

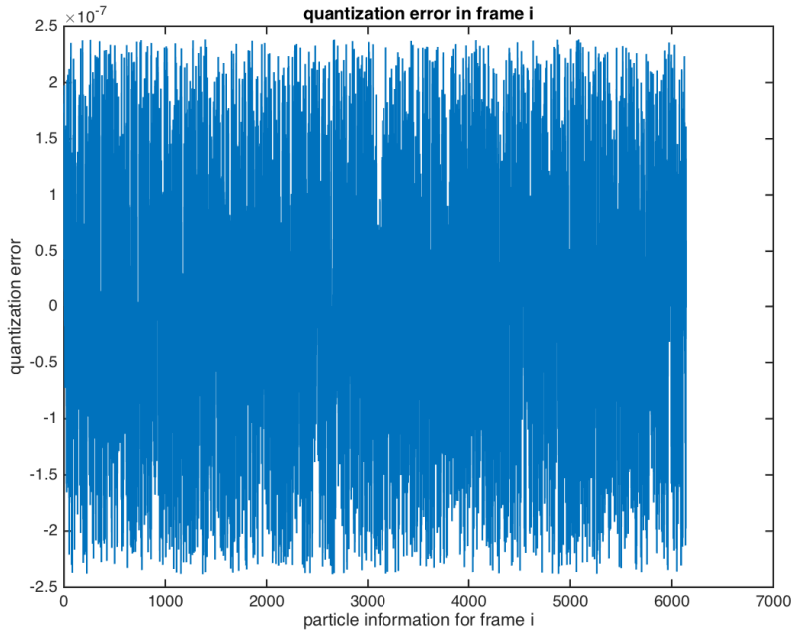

Figure B.1: The quantization error at the input frame for 32 bit signed fixed point space.

Word length	Scaling factor	M.S.E
32	21	1.1e-07
24	13	6.1e-05
16	5	7.5e-03

Table B.1: Mean Square Error for the initial beam state.

similarly, the coefficients of the transport matrices are converted to fixed point space. Since the coefficients has values ranging between 1 and -1, the mean square error are minimal for the conversion, when compared to the frame data conversion. For the word lengths 8, 16, 24 and 32 the error due to quantization is shown in the table B.2.

B.3 Matrix Multiplication in Fixed Point

For computing the matrix multiplication, the word length of the frame data and the transport matrix are set as parameters. Let the beam initial state be W_i word length long and the transfer matrices be of length W_{tm}. The beam frame out data is also quantized back into the word length W_i. The transfer matrices is a square matrix with dimension 6, translates each matrix multiplication to

Transfer Matrix	Word length	Scaling factor	M.S.E
R_{focus}	8	6	1.9e-03
$R_{defocus}$	8	6	2e-03
R_{focus}	16	14	7.15e-06
$R_{defocus}$	16	14	1.004e-05
R_{focus}	24	22	2.7157e-08
$R_{defocus}$	24	22	1.8738e-08
R_{focus}	32	30	1.2347e-10
$R_{defocus}$	32	30	1.3780e-10

Table B.2: Mean Square Error for transfer matrices.

36 multiplications 30 addition operations. A matrix operation is performed per particle in a frame.

For the matrix operation, equation B.3, C can be computed as in equation B.4.

$$C = M \times A. \tag{B.3}$$

$$C_i = \sum_{j=1:6} M_{i,j} \times A_j \tag{B.4}$$

But due to streaming behaviour model used, the columns of the transport matrix is stored into registers and the implementation can be rewritten as in equations B.5 and B.6.

$$col(B_i) = \sum_{j=1:6} M_{j,i} \times A_i \tag{B.5}$$

$$C_i = \sum_{j=1:6} B_{i,j} \tag{B.6}$$

The elements of matrix B need to have word length $W_i + W_{tm} - 1$. The -1 is due to the sign bit fixed point space. For the elements of C matrix, a word length $W_i + W_{tm} - 1 + 3$ is required, to represent a sum of 6 elements.

W_A	W_M	W_B	M.S.E
32	32	63	1.73e-07
32	24	55	6.27e-06
32	16	47	1.5e-03

Table B.3: Mean Square Error for the output frame of the beam.

The elements of C matrices, which are $W_i + W_{tm} - 1 + 3$ long is truncated to a length of W_i bits. This is performed by scaling it down by a downscale factor, same as the scaling factor used for up-scaling the transfer matrix.

The result after the multiplication operation is compared to the double precision reference data. The table B.3 shows the quantization error introduced where

W_x represents the word length of the elements of Matrix x, where x can be either matrix A, B or M. The scaling factor as in the tables B.1 and B.2 are used for this comparison.

The mean square error in the table B.3 is amalgamation of all the quantization errors previously mentioned, i.e the quantization of the input frame, the transfer matrix and then due to storing a partial result back to the finite storage element.

Appendix C

HLS Model

C.1 HLS based Simulation Model

Using Vivado HLS, it is possible to perform a rapid design exploration for the Xilinx FPGA accelerator card (ALPHA DATA ADM-PCIE-KU3). The accelerator card hosts a Xilinx Kintex Ultrascale XCKU060-FFVA1156 FPGA.

In order to provide with a reasonable execution time for the emulation model, the test case is simplified with initial frame of beam getting transported by one iteration with a test configuration of 256 particles. The 256 particle size is a suitable sub-frame size, that allows to distribute a single frame to multiple instances of this computation unit. So this study is consistent with providing the required information of hardware utilization for different data types. The double precision data results from Matlab will be used as the golden reference data to verify the results.

C.2 Matrix Multiplication Architecture

Since the performance of the implementation is affected by how the behavior is described to the FPGA fabrics, a study about the related work is done. Optimizations on the hardware implementation can be fine tuned for a FPGA hardware unlike CPU and GPU, which has a fixed architecture.

The transport matrix being small compared to the frame vector, there is no considerable storage benefits in compressing the transport matrix. If it had a fixed coefficients, some of the sparse matrix optimizations on the processing could have been applied. This would impact the latency incurred per matrix multiplication, however the transport matrix changes with the region of space it is representing. Adding the non linear behavior needs to modify the matrix due reflect the particle-particle interaction(space charge). Hence, the optimizations and work in these direction are not explored due to the above mentioned constraints.

Matrix multiplication by dividing the large matrix into tiles rely on the data locality and cache re-use concept. Since one entire frame cannot fit into the cache memory, the cache re-use concept cannot be utilized fully, however the transport matrix needs to be stored in the cache to provide the maximum efficiency. The tiling method reduces the frequency of access to the same data or elements of the operand matrices while computing the resultant matrix. There are lot of

work showing better power efficiency with higher throughput such as [13]. For the algorithm under consideration as mention before, each particle in the frame is multiplied by the same transport matrix and, each computation unit is explicitly loaded with the transport matrix into their local cache, hence the concept of tiling is done inherently within the hardware implementation.

The works [16] and [11] shows the matrix implementation with systolic array based approach. It mainly rely on the spatial locality of the data, and can provide considerable acceleration to the throughput if can be utilized with the design. A better mapping into hardware by utilizing streaming is considered over the systolic approach as it provides a more simplified solution with the same efficiency.

If we move our focus to the iterative nature of frame to frame computation, the transfer of frame from the device memory to the host memory can also turn into a non trivial issue. But this is handled on the later chapter on OpenCL device implementation.

C.2.1 Hardware Architecture

For the test case, Design under Test (DUT) is processing component of the computation block, where data is streamed from the FIFOs. The transport matrix(transfer matrix) is stored into the register banks. The results are compared with the Matlab reference frame data and mean square error is reported. Figure C.1 shows the test bed environment used for the study.

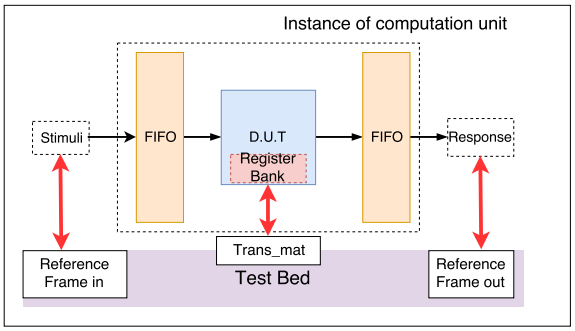

Figure C.1: The HLS test bed and instance of computation unit.

Instead of a memory mapped architecture, by using streaming of data, the need for address decoding is not required. This produces lesser latency and lower hardware utilization at the cost of optimizing the DUT for streaming. The HLS design can be used as a co-processor with a Zynq FPGA or microblaze, by using Advance eXtensible Interface (AXI) interfaces with Direct Memory Access (DMA): stream for Frame in and Frame out, while axi lite for DUT configuration and loading transport matrix.

There are two purposes for this test bench: (1) to study the accuracy offered by different data types and (2) to compare the hardware utilization for the design choices. The first verification of the DUT block is by using double precision

float and as expected it was consistent with the Matlab results, then the design is switched to fixed point and other data types. The optimizations are applied to synthesize the target hardware architecture. After the optimal design is implemented, the comparisons study between latency and hardware utilization are made. The HLS design flow makes the change between fixed point and floating point seamlessly.

C.2.2 HLS Function Overview

For an HLS based design flow, the default implementation implies rolled hardware, and may not be optimal for high throughput applications. Therefore design cues needs to be provided for functions, variables and, the for loops to produce the required throughput in the implementation. The c source code needs to be assimilated into smaller blocks and sometimes a additional variables needs to be introduced. The functional behavior doesn't change, but these additions help to expose the source code allowing the HLS compiler to synthesize the required hardware architecture.

Top Function.

Top function is the function that will be synthesized into HDL with the arguments as interfaces and can be modified as one intend with the help of directives provided in the Vivado HLS tool. Here the top function includes the FIFO structures for data moving, and facilitate parallel data access for DUT and the function that then accesses these Memory structures to produce the result. An overview of the function is shown in the figure C.2, and represents how the data flow, from input to the output.

C.2.3 DUT Optimizations

For the HLS based design flow, the optimization are applied onto different solutions, and these solutions can be easily compared against. The optimal design can be arrived at using different ideologies, one of them is to start from the inner loops of each stages in the design hierarchy, and to work the way up, noticing the bottlenecks at each steps, and overcoming them.

Pipeline

To allows for accessing off-chip memory via burst mode, the internal loops of the memory accesses to/from the FIFO are applied with the directive pipeline. For matching the throughput to the available bandwidth optimizations are also applied on the processing elements. The P_cnt_Loop_mul is pipelined, which exposes all the inner loops to be unrolled. This implies multiple hardware instances of the inner loop (sub-blocks) will be instantiated providing a highly pipelined and efficient hardware architecture.

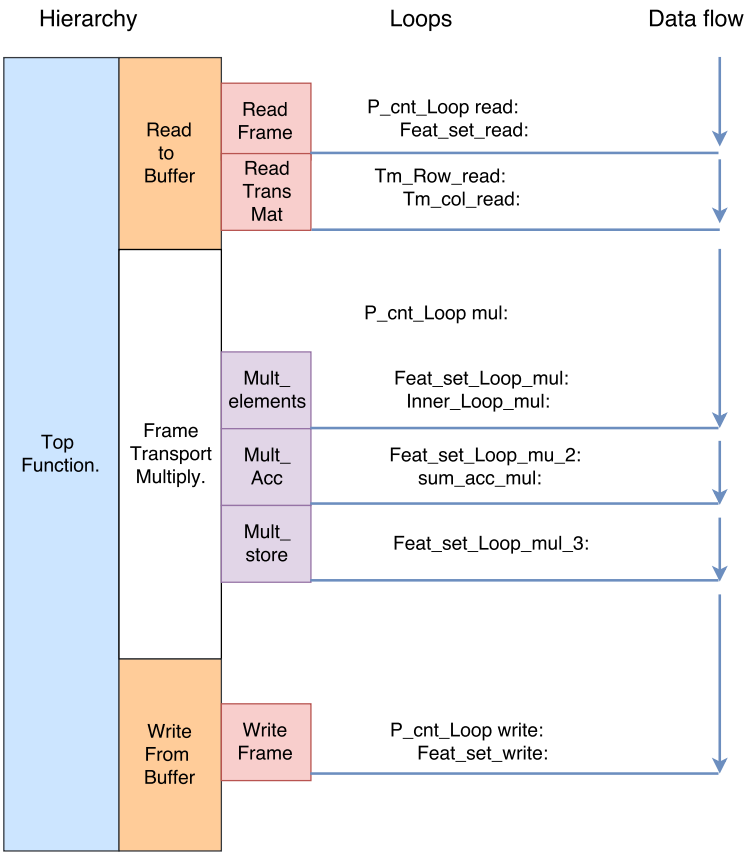

Figure C.2: The design overview in hls.

Array Partition

To allow for the proper unrolling of the loops mentioned above, the memory bandwidth needs to be optimized by increasing the number of access ports to the memory and by moving the critical data access points to registers. To provide the required memory bandwidth the internal buffers are partitioned into 6 block rams, with two ports each. The C source code is re-arranged to expose the critical data access points and these are mapped into hardware as registers.

Data flow

First the stimuli, the Frame data in and the transfer matrices are read into the internal buffers, the DUT streams the data in and processes them and streams the data out to the buffer out structure. The test bench then reads the data out and compares it with reference results.

The read data in, the DUT process and read data out can be processed as in the figure C.3 so that, if for a longer iteration or consecutive execution, the

latencies incurred by the Data read and Data write can be overlapped by DUT, or by which has the longest execution time.

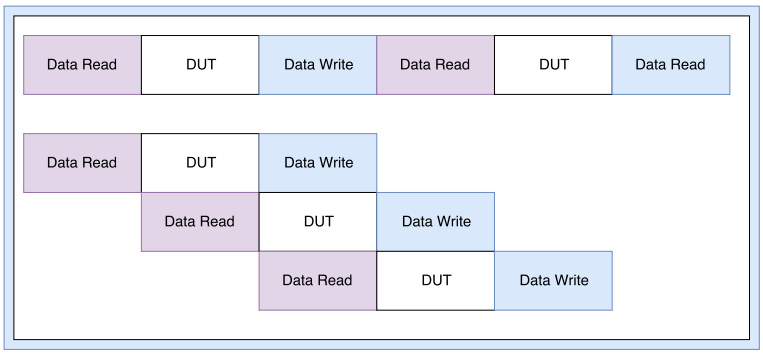

Figure C.3: The data flow optimization.

C.2.4 Optimization Results

After the optimization applied for the fixed point data type, the hardware utilization are tabulated in the table C.1 and the performance of the design in the table C.2.. The performance due to the data flow optimization is a variable, which at

Instance	LUT	FF	DSP48E	BRAM
Read Buffer	207	253	0	12
Multiply Frame	3724	2912	144	0
Write Buffer	240	382	0	12
Net	4738	4581	144	24

Table C.1: Hardware utilization of the design.

worst will be the sum of the latency of the individual sub-blocks and at it's best the latency of the critical block.

Instance	Latency Min	Latency Max
Read Buffer	1538	1538
Multiply Frame	778	778
Write Buffer	1538	1538
Net	1538	3856

Table C.2: Performance results of the design.

It can be seen that the critical block here, are the Read and Write Buffers and is due to the access of the Memory to the internal buffers .For the HLS design flow,

the Top function arguments are by default assumed to be RAM blocks and the latency is attributed due to the I/O bottleneck inherent to the Memory blocks.

One of the optimization that can be applied here, is to widen the memory access, i.e doubling the number of memory thread to the DUT, thereby improving the best case latency to 778 clocks.

C.2.5 Interface and Control

The RTL generated from the HLS flow, has control signals for the FSM and the input and output signals to the DUT can be altered with directives. The control signals are designed with AXI slave lite interface. And, the Input and output signals are interfaced with AXI stream interface. An internal FIFO for the AXI stream can be also added here if, the design needs to work in a different clock domain.

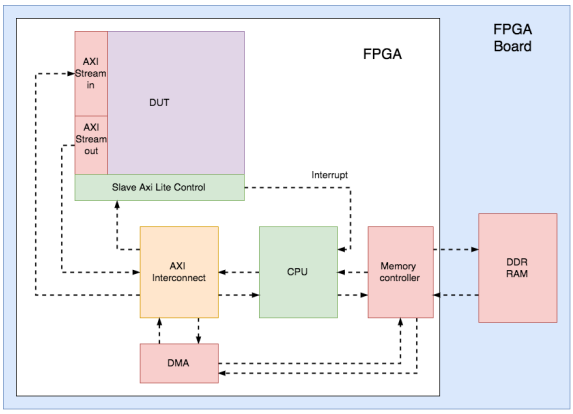

Figure C.4: Using the HLS design as an accelerator IP in FPGA.

The FIFO used in the AXI stream can be used as a second stage for memory hierarchy, from the data flow optimization can be applied to the internal design. More than one of these units can be placed into the design, depending on the memory bandwidth provided by the memory.

An RTL co-simulation is carried out and the testbench verified the results. The testing with FPGA and integration with a Microblaze (soft-core processor) or ARM processor is proposed as a future work.

Appendix D

OpenCL Framework and Software stack

This chapter introduces some of the critical structures in the OpenCL host framework. Some of the implementation details of the software stack is also discussed in this chapter.

D.1 OpenCL Framework

OpenCL is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs, and other processors. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high performance compute servers, desktop computer systems, and handheld devices[14].

In other words, the OpenCL framework is a collection of library aimed for parallel processing, targeting heterogeneous computing applications. It is maintained by Khronos group, and went over several revisions, and at the time of writing this document, the latest release is OpenCL 2.2. It is written over C/C++ programming language. [14] shows the list of changes over each iterations till now.

The OpenCL framework has fundamentally two parts:

- Host code: The host binary that runs on host processor and manages the devices.
- Device Kernel: The binary that maps the behaviour to the device accelerators.

The Framework provides lots of Application Programming Interfaces (APIs), which can facilitate the communication between the two and also make the application layer independent of the hardware configuration. By this virtue, OpenCL framework enables functional portability across different hardware platforms and supports task level parallelism. The list of supported APIs can be found at [15].

The Key functionalities provided by the OpenCL framework are as shown in the Fig. D.1

D.1.1 Functional Portability

Having functional portability across different operating systems and target devices allows for having an unified Integrated Development Environment (IDE) and re-

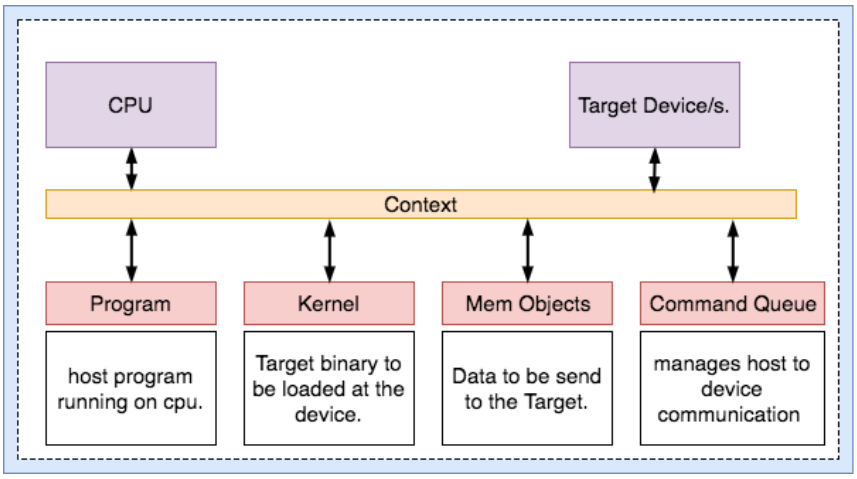

Figure D.1: OpenCL framework.

usability across OpenCL complaint devices. The source of OpenCL framework is based on C and is cross-platform compatible.

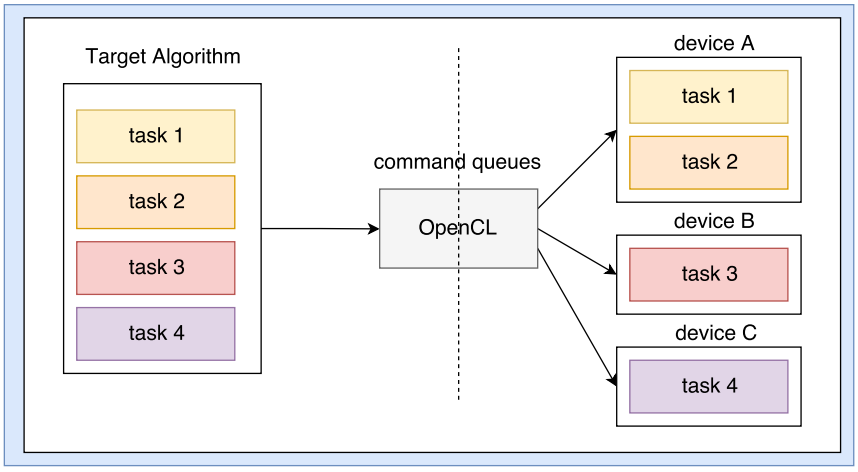

Figure D.2: OpenCL Task Level parallelism.

D.1.2 Task Level Parallelism

OpenCL provides inherently task level parallelism by providing device specific command queue and scheduling. This level of central control can help with synchronizing the tasks between multiple devices. The Fig. D.2 illustrates the command queue structures used for communication between host and OpenCL compatible devices.

D.2 OpenCL Host structure

The host code, is an important part in the OpenCL framework, which is run on the host processor. The key functionalities in the OpenCL application can be summarized as in the figure D.3.

Figure D.3: The OpenCL host code functionality overview.

The OpenCL host environment needs to fulfil some runtime requirements, as listed below

- Vendor OpenCL drivers needs to be installed.
- OpenCL dynaimc library needs to be installed.
- QT framework library needs to be installed.

D.2.1 Device Detection

The OpenCL Installable Client Drivers (ICD) Loader Library allows multiple implementations of OpenCL to co-exist on the same system. Applications may choose a platform from the list of installed platforms and hence dispatch OpenCL API calls to the correct underlying implementation. Each device is identified by their platform ID and device ID. The framework allows to query device details using these IDs. The platform ID is specific to the vendor(manufacture) and the device id unique to target class. With this it is possible for the application to identify new target device with plug n'play support.

D.2.2 Command Queues for Device

Once the device IDSs and platform IDSs are listed, the selected devices can be registered under a context and each context can be assigned with a command queue structure. The command queue is that, facilitates communication between the host and the device

- transfer of memory objects
- loading of kernel program
- kernel execution
- performance profiling

This allows synchronizing the events and controlling the devices. There are two types of command queue structures: the In-Order command queues and Out- Of-Order command queues.[17] is an example case which showcases the differences in the scheduling behavior between the two. The figure D.4 is an example presented in the [17].

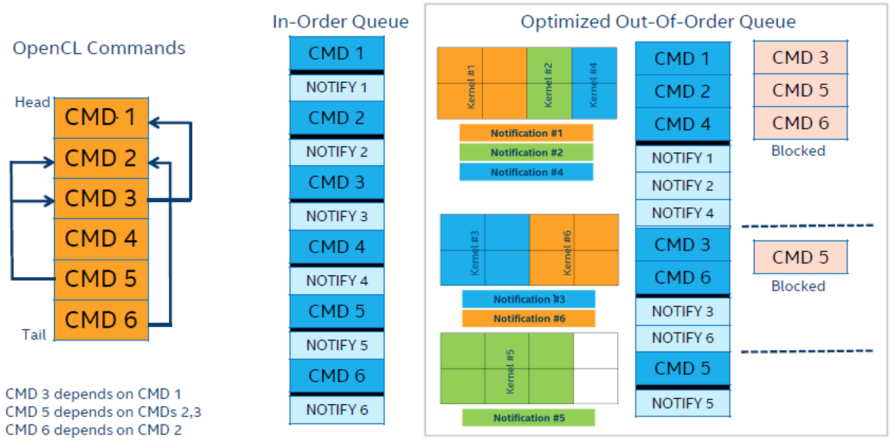

Figure D.4: The In-Order and Out-Of-Order command queue.

The commands in the In-Order command queue, is executed in the order it was registered. while in certain cases out of order execution provides more efficient resource management. In these cases, the Out-Of-Order queue can be used to manually re-order the command events based on the dependencies. This requires attaching an event object to each command object. The event object provides the the status of the command, whether it is executing or completed.

D.2.3 Device Kernel Loading

Device kernel are the binary targeting the device behaviour. Their source file needs to be loaded, compiled and linked with the memory objects before it can be executed. There are two compilation flows:

- on-line compilation, which compiles the source during the runtime.
- off-line compilation.

This on-line compilation allows functional portability, so the program can be executed on newer devices. In certain device classes, such as FPGA where the compilation time is not feasible for on-line compilation, off-line compilation methodology is used. This allows the binary to be pre-compiled and loaded during the runtime.

D.2.4 OpenCL memory Objects

To transfer the data from the host memory to the device memory, the data array needs to be attached under a memory object. Then the memory object can be linked to a read/write OpenCL command API, which is then attached to the command queue to initiate the transfer of the data. The size of data, usually as

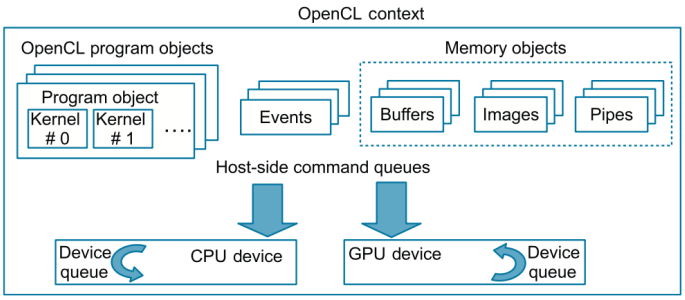

Figure D.5: The OpenCL context: the environment where, the kernel objects, memory objects and command queue are well defined.

counted as the number of bytes needs to provided as the argument for the API. Fig. D.5 shows how the OpenCL objects are defined in a context.

D.2.5 OpenCL Kernel Execution

The OpenCL kernel can be executed by using the OpenCL APIs, which takes in the kernel object(OpenCL object compiled version of the kernel binary), the NDRange structure (the execution space for the command), event structure(synchronization stamp) and a list of events to wait for in case of Out-Of-Order queue. Once it is attached to the command queue, the execution is carried out at the target device, after which the processed data can be read back from the device using the memory objects.

D.3 Host Application

The host application run-time is explained in this section. The test configuration can be initialized with the help of the GUI and then the simulation model can be initiated on the target hardware platform. The state-machine based device management layer is briefly explained in this section.

D.3.1 Application Startup

The Application startup initiates the search for OpenCL complaint devices. The figure D.6 is the startup screen which has detected the Intel i5 CPU and Intel iris 6100 HD graphics processor in a mac book pro 13 system. The devices are grouped under platform Apple(Intel makes customized processor chip for Apple ecosystem). After finding the list of devices, the Qt objects(graphic elements) are dynamically created and the list is populated. A device selection is provided and the simulation model can be initiated on the targeted device.

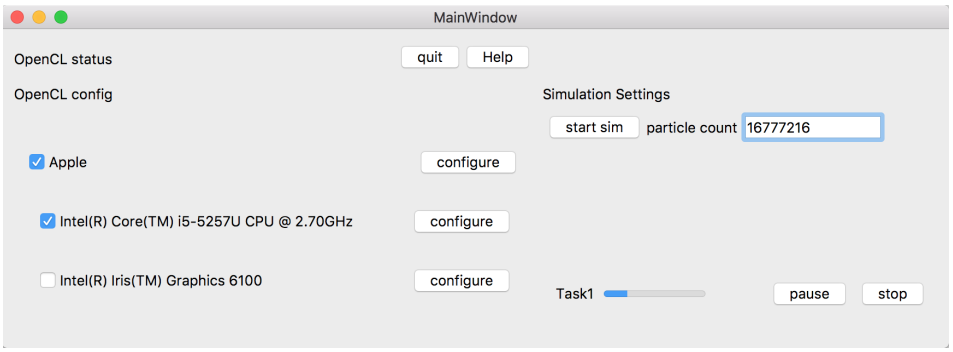

Figure D.6: The QT based Graphic Interface.

D.3.2 StateMachine Execution

The particle count can be specified, and when the start sim button is pressed, the OpenCL task is initiated onto a new thread. Based on the number of devices selected, each OpenCL device will be allocated a free thread for device management, and as shown in the Fig. D.7 the thread synchronizes with the host machine.

It is important that all the devices are processing the same frame and there is a synchronization between them. Since the effect of non-linear behaviours, needs to modelled per frame based on the statistical information collected. The transport matrix will be modified to introduce the space charge effect and can infer latency. By allocating sub-frames based on the performance efficiency of each device, this model of frame-work can be used to extract the maximum throughput at the system level.

The figure D.7 shows the flow chart of the state machine. A mutex(mutual exclusion), that is a binary semaphore is used at the boundary of the threads for data exchange, this helps to avoid dead blocks, where each of the threads races to access common resources.

This work acts a prototype(proof of concept) and supports multi-device execution, however for efficient thread utilization for the device management, a more efficient thread to thread communication protocol needs to be investigated and integrated into the design.

D.4 Unified Testing Environment

The software stack can be used for development of similar work, where a hybrid computing platform can be used for efficient mapping of the algorithm to different hardware architectures providing a rapid development platform. Since it supports OpenCL complaint devices, new hardware configurations can be tested with the same host frame-work.

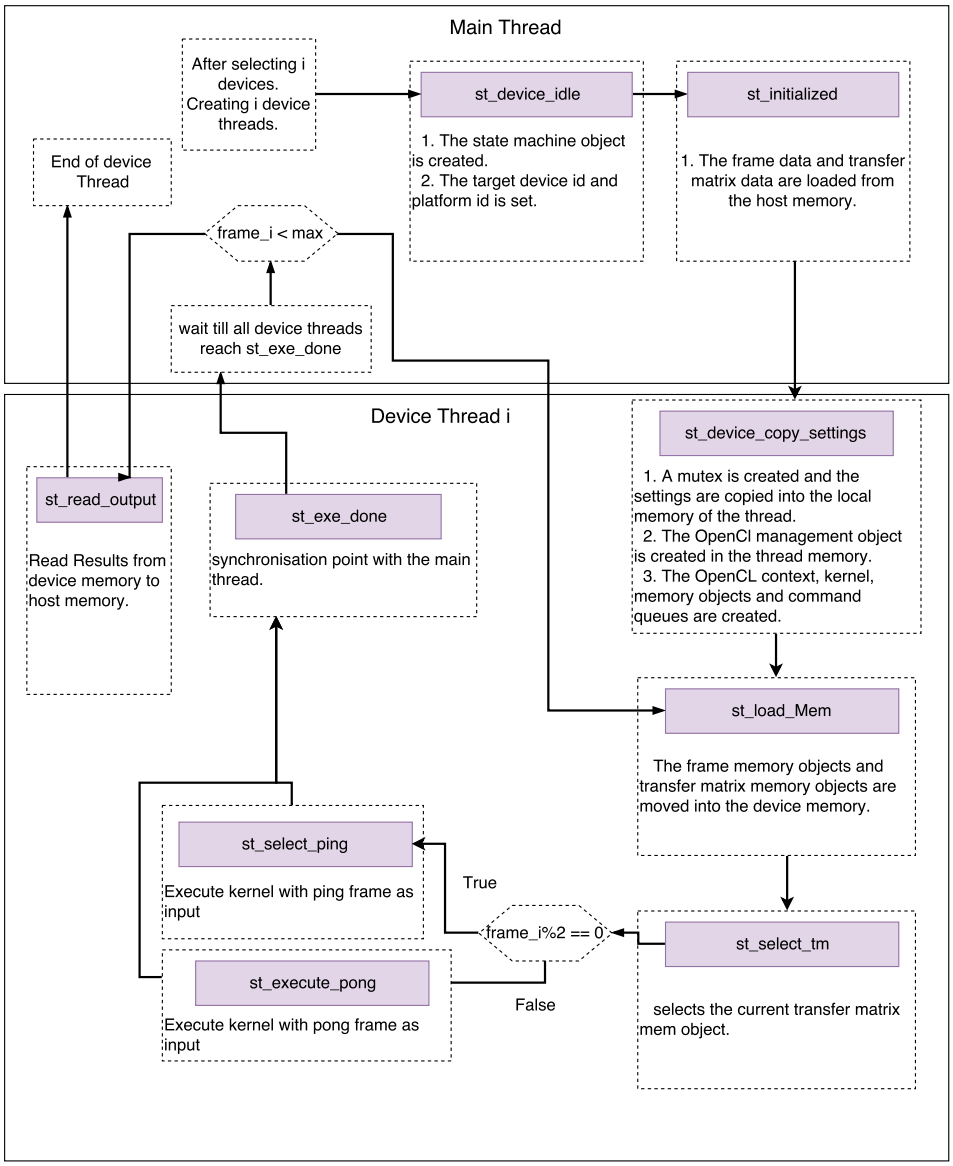

Figure D.7: The device state machine for managing OpenCL execution.

Appendix E

Xilinx OpenCL Design Methodology

This chapter is dedicated to the OpenCL methodology used for the development of the OpenCL kernel using SDAccel in Xilinx based FPGA accelerator card.

E.1 OpenCL on Xilinx FPGA

An SDAccel device contains a customization area called the OpenCL region (OCL Region). Although not defined in the OpenCL standard, the OCL Region is an important concept in SDAccel. The OpenCL static region along with the OCL region provides the OpenCL compliance. The static regions allows to place the computation unit defined by the OpenCL kernel to be placed into the OCL region by using partial reconfiguration technology.

E.2 FPGA Kernel Development Flow

For the FPGA implementation, the OpenCL computation units can be described by using either the OpenCL C description, the HLS C description or the RTL HDL description. For the thesis, only the first method is implemented. The design flow for OpenCL implementation followed an iterative process of development through three compilation flows as shown in the figure E.1. Firstly a check on the functional behavior of the OpenCL kernel is verified with CPU emulation flow. This is fast and is a quick check for functional coherence.

This is followed by using the hw emulation flow, where the cycle accurate model of the kernel is generated and used to analyze the performance and functionality. The compilation time is within half an hour to an hour for this step.This compilation flow can be used to model how fast the implementation is performing and to pinpoint the bottlenecks of the design. Many iterations needs to compiled for fine -tuning the kernel for the requirement throughput and performance constraints. However, testing this model over large data set is time-consuming and is not feasible. So a smaller data frame is used for providing the functional check after the fine-tuning of the implementation.

The OpenCL compilation for FPGA, is very time consuming and can take up hours depending on the design complexity. This can extend further based on the compiler flags set. After the hw emulation step, the OpenCL model is implemented, placed into the real hardware and tested.

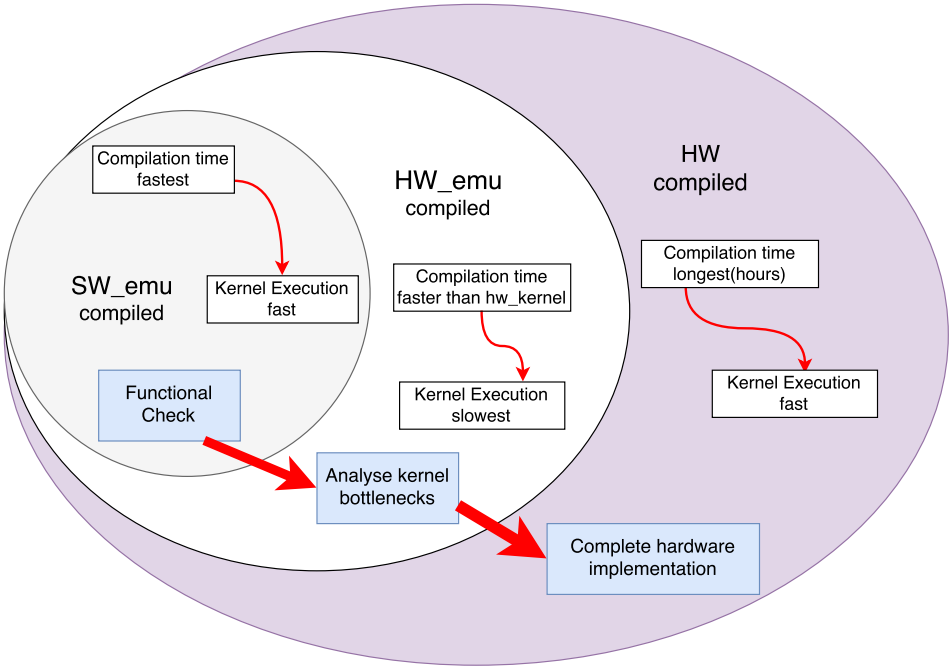

Figure E.1: The SDAccel OpenCL compilation work flow

E.3 OpenCL Implementations for FPGA

Two approaches for kernels were tested on the FPGA based on how the algorithm was mapped to CPU and GPU hardware architecture. They are termed as FPGA_C kernel(CPU based kernel) and FPGA_G kernel(GPU based kernel) henceforth. Optimizations were applied to improve the throughput for these kernels and is discussed below.

E.3.1 Kernel_C Optimizations

To the base FPGA_C kernel, vendor specific directives were used to optimize the throughput. The optimizations used for the implementation is addressed here.

Increasing Kernel Depth

By increasing the depth of the kernel, the overhead due to scheduling the entire work load to the work group is reduced and allows for lesser frequency of off-chip memory access. By increasing the depth from 16 to 512, a performance gain to 1.5223x is observed.

Bulk Memory Access and Memory Partition

The read and write to the global memory to/from local memory (on FPGA memory) is made pipelined to infer bulk memory transfer. The transport matrix is stored in register banks, while the sub-frame data is stored into 6 separate Memory structures with cyclic storage. This allows the high bandwidth required for the internal parallel processing structures.

Reducing Local Memory Accesses

The inner loop in the execution, it was noted that accumulation of the result, had a high frequency access to local memory. A separate variable inferred as registers were used to significantly reduce the frequency of the local memory access from 36 to 6 .

Unrolled and Pipelining.

The Computation loops are unrolled and pipelined, there by exposing the multipliers and adder resources to facilitate parallel execution.

Data Flow

The entire kernel function is applied with data flow optimizations, which interleaved the execution of the sub-blocks such as local memory read, execution sub-blocks and local memory write. This helps with reducing the overall latency of the entire kernel execution.

Increasing Computation Units

Increasing the number of computation units allows for more parallel execution of work groups and thereby expanding parallelism at the kernel level. The kernel functionalities can be instantiated multiple times. 8 instance of the computation units were instantiated to produce the gain of 40.8x.

Kernel_C Performance.

For the Kernel_C, the local work group size attribute is not utilized. Here a work item and a work group are identical, instead the depth of the bulk transfer to the local memory is the parameter used to fine-tune the kernel. This produces a similar impact as work group size would otherwise produce. The table E.1 shows the result of the Hardware emulation test bench for frame with particle count of 4096.

E.3.2 Kernel_G Optimizations

The GPU architecture has a different approach for efficient mapping with OpenCL kernel than the CPU architecture. Parameters such as work-group and work-items needs to utilized and an efficient execution depends on the synchronization between the work-items within a work-group. The following optimizations were applied.

Target Implementation	Kernel Depth	Computation units	Execution Time	Gain
Cpu Kernel on FPGA w/o opt	16	1	3.06ms	-
Cpu Kernel on FPGA w/o opt	512	1	2.010ms	1.52x
Cpu Kernel on FPGA w opt	512	1	.259ms	11.815x
Cpu Kernel on FPGA w opt	512	4	.085ms	36x
Cpu Kernel on FPGA w opt	256	8	0.075ms	40.8x

Table E.1: Kernel_C performance overview

Memory Access Optimizations and Unrolling.

The memory needs to be partitioned to provide more data bandwidth to the execution units(work-items) within a computation unit(work group). The local memory is partitioned by 6 to provide the required bandwidth. Each work item is mapped to a matrix multiplication operation. By assigning private memory tag to the accumulator used in the matrix multiplication stage(which implies it as registers), the access frequency to the local memory can be reduced. The computation for loops are pipelined and unrolled. A performance gain of 6.67x is observed after the above optimizations.

Computation Units

The number of computation units can be increased so that more work groups can be scheduled for execution, thereby increasing parallelism. Increasing the work group size to 1024 and changing the computation unit count to 4, showed a performance improvement of 15.625x. A similar trend in performance gain is observed with increasing the computation units to the maximum limit of 16. But this approach has a penalty of higher hardware utilization.

Kernel_G kernel Performance

The overview about the performance improvements by different optimizations are listed in the table E.2

Target Implementation	workgroup size	Computation units	Execution Time	Gain
Gpu Kernel on FPGA no opt	128	1	16ms	-
Gpu Kernel on FPGA w opt	128	1	2.4ms	6.67
Gpu Kernel on FPGA w opt	1024	4	1.024ms	15.625
Gpu Kernel on FPGA w opt	256	16	0.619ms	25.84

Table E.2: Performance Overview of kernel_G .

E.3.3 FPGA Kernel Performance and Utilization

Kernel	Kernel Work-Group	LUT	FF	DSP48E	BRAM	Performance Gain
Kernel CPU utilization %	512	45908 13	66446 10	504 18	28 1	8.25
Kernel GPU utilization %	1024	8780 2	11627 1	14 0	79 3	1

Table E.3: Hardware utilization across OpenCL kernels.

The two of the efficient approaches used for mapping the simulation model are described and the optimizations applied explained. The OpenCL_C kernel performs faster than the OpenCL_G kernel, due to several factors. A stream based approach has reduced complexity compared to a memory mapped approach. The Hardware utilization for the two Kernels is also compared. The overall performance in an FPGA is also factored by the off-chip memory bandwidth.

Appendix F
Git Repository

A Git based revision control was used during the development of the thesis project. All the source files are uploaded into the Github server and free to access. A clone of the source files can be produced locally by using the git command in a suitable terminal
git clone `git@github.com:arunjeevaraj/hybrid_computing_thesis.git`

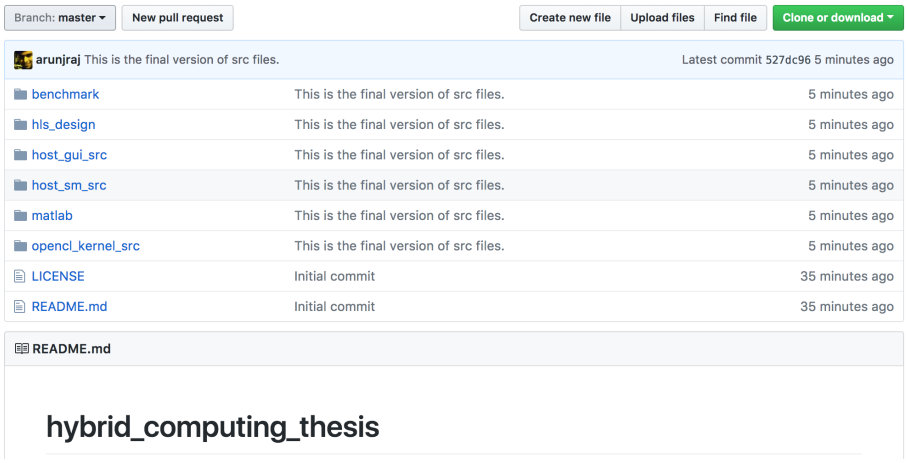

Figure F.1: The git repository for the thesis project

F.1 Folder structure

The source files are stored into their respective folders as shown in the Fig. F.1 and are listed below.

- benchmark: contains the source file for benchmark cpp program.
- hls_design: contains the hls design source and tcl script.
- host_gui_src: contains the source files for the software stack with GUI.

- host_sm_src : contains the state machine for multi-device management and execution. Needs to be integrated with the software stack.
- matlab : contains the Matlab source files for the functional model, the hardware emulation model, generation of plots and reference data etc.
- OpenCL kernel src: contains the OpenCL kernel file for different devices.

Printed by Tryckeriet i E-huset, Lund 2017

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2017-609

http://www.eit.lth.se