Enabling Development of OpenCL Applications on FPGA platforms

Kavya S Shagrithaya

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Peter M. Athanas, Chair
Patrick R. Schaumont
Paul E. Plassmann

August 6, 2012
Blacksburg, Virginia

Keywords: OpenCL, AutoESL, FPGA, Convey, HPC

Enabling Development of OpenCL Applications
on FPGA platforms

Kavya S Shagrithaya

(ABSTRACT)

FPGAs can potentially deliver tremendous acceleration in high-performance server and embedded computing applications. Whether used to augment a processor or as a stand-alone device, these reconfigurable architectures are being deployed in a large number of implementations owing to the massive amounts of parallelism offered. At the same time, a significant challenge encountered in their wide-spread acceptance is the laborious efforts required in programming these devices. The increased development time, level of experience needed by the developers, lower turns per day and difficulty involved in faster iterations over designs affect the time-to-market for many solutions. High-level synthesis aims towards increasing the productivity of FPGAs and bringing them within the reach software developers and domain experts. OpenCL is a specification introduced for parallel programming purposes across platforms. Applications written in OpenCL consist of two parts - a host program for initialization and management, and kernels that define the compute intensive tasks. In this thesis, a compilation flow to generate customized application-specific hardware descriptions from OpenCL computation kernels is presented. The flow uses Xilinx AutoESL tool to obtain the design specification for compute cores. An architecture provided integrates the cores with memory and host interfaces. The host program in the application is compiled and executed to demonstrate a proof-of-concept implementation towards achieving an end-to-end flow that provides abstraction of hardware at the front-end.

Acknowledgements

Firstly, I am sincerely thankful to my advisor, Dr. Peter Athanas for providing me an opportunity to be a part of the Configurable Computing Lab. He has been a source of inspiration throughout and has been very encouraging at all times. I would like to express my deep gratitude for his continual support and invaluable guidance without which I could not have progressed in my research. I have learnt a lot from him, in academics and beyond.

I would like to thank Dr. Patrick Schaumont and Dr. Paul Plassman for serving as members of my committee. The courses that I took under Dr. Schaumont were a great learning experience and I immensely enjoyed his style of teaching.

My parents and my brother have been there for me, through thick and thin. I am very grateful to them for encouraging me towards pursuing graduate studies and supporting me throughout. Special thanks to my father, my role model, for all his words of wisdom.

The environment in the lab has always been conducive to learning things the enjoyable way. I would like to thank my lab mates; Tony, for his advice on all work-related matters; Karl, for helping me ramp up in the intial days; Krzysztof, for his help and guidance on this project; Kaiwen, for his inputs towards this work; and the rest of the CCM family, David, Ritesh, Ramakrishna, Keyrun, Xin Xin, Wen Wei, Andrew, Richard, Shaver, Jason, Ryan, Teresa, Abhay and Jacob for making this a truly enriching experience.

My heartfelt thanks to Kaushik and Abhinav for their support in curricular and non-curricular aspects; Iccha, Vishwas, Vikas, Shivkumar, Navish, Atashi and Mahesh for the fun times off work; and all my other friends at Blacksburg who have made these two years very memorable for me.

I am thankful to God for everything.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An increasing number of application areas today are embracing high performance computing (HPC) solutions for their processing needs. With frequency scaling having run its course, conventional processors have given way to various accelerator technologies to meet the computational demands. These range from Graphic Processing Units (GPU), Field Programmable Gate Arrays (FPGA), heterogeneous multicore processors like Cell to hybrid architectures like Convey HC-1. Different classes of applications employ different targets best suited for the problem at hand. Each of the accelerator technologies possess advantages over the other for variant tasks leading to possibilities of a heterogeneous mix of architectures [1–3]. Reconfigurable systems like FPGAs provide promising opportunities for acceleration in many fields due to their inherent flexibility and massive parallel computation capabilities.

The availability of a multitude of hardware devices require comprehensive approaches to solution which include the knowledge of underlying architecture along with methods of designing the algorithm. This translates to an increase in implementation effort, high learning curves and architecture aware programming at the developer's end. In 2004, DARPA launched a project named High Productivity Computing Systems (HPCS) that aimed at providing eco-

nomically viable high productivity systems[4]. They proposed using "time to solution" as a metric that includes the time to develop a solution as well the time taken to execute it. CPUs and GPUs are programmed in high level languages like C/C++ and CUDA. This level of abstraction enables faster deployment of solutions. In the case of FPGAs, implementations require tedious hardware design and debug which greatly impact the development time.

1.1 Motivation

Enormous acceleration can be delivered by FPGAs owing to the flexibility and parallelism provided by the fine grained architecture. However, being able to fully harness this potential presents challenges in terms of programmability. Implementation of applications on FPGAs involve cumbersome RTL programming and manual optimizations. Besides domain expertise and software design skills, developers are required to understand intricate details of hardware design including timing closure, state machine control and cycle accurate implementations. As a result the effort in drastically reducing the execution time translates to a significant increase in the time to develop the solution. Jeff Chase et al implemented a tensor-based real time optical flow algorithm on FPGA and GPU platforms. A fixed-point version was mapped on a Virtex II Pro XC2VP30 FPGA in the Xilinx XUP V2P board and a single-precision floating-point implementation was mapped on an Nvidia Geforce 880 GTX GPU with appropriate design optimizations for each platform. Comparisons were drawn on both implementations on various parameters including performance, power, and productivity. The work presented comparable results for performance on the FPGA and the GPU, while the power consumption was significantly lower on the FPGA. The FPGA implementation, however, took more than 12 times longer to develop, as compared to the GPU implementation. There has been significant research and industry related work in providing high level pro-

gramming solutions for FPGA to reduce these design efforts. They however do not succeed in abstracting all implementation details as the knowledge of hardware concepts is required, to an extent, to appropriately design the software solution for maximum performance.

OpenCL (Open Computing Language) is a cross platform standard for building parallel applications. The OpenCL programming model, albeit skewed towards GPU architectures, provides a standard framework that enhances portability across devices essentially rendering it platform independent. It enables the developer to focus on design of the application while abstracting the fine details of implementation. Being able to develop applications in OpenCL for FPGA platforms would result in faster implementation times and time-to-market.

1.2 Contributions

Supporting OpenCL applications on a new platform requires a compiler to translate the kernels to the device specific executable, a library for the OpenCL API and a driver that manages the communications between the host and the devices. This thesis presents a proof of concept system that enables OpenCL application development on FPGAs. An efficient method for compiling OpenCL kernel tasks to hardware is demonstrated, without having to build a compiler from scratch. Existing tools are leveraged, as required, in the process. A system architecture is presented for the target device with an automated generation of interfaces for the kernel. Also, a library that supports a subset of the OpenCL host API is provided.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 presents a background on high-level synthesis for FPGA platforms and a perspective on why OpenCL would be a good option. It also provides an overview of the OpenCL specifications, introducing the architecture framework and the programming language. Current on-going efforts on supporting OpenCL applications on FPGA fabric is discussed in the related work section. Chapter 3 introduces the approach to the problem statement and proceeds to provide details on the source-to-source translation and compilation processes, using a sample application. It also discusses the mapping of the execution model on the target hardware. The results obtained in this work and comparisons with other implementations are discussed in Chapter 4. The conclusions and future work for the project is included in Chapter 5.

Chapter 2

Background

In this chapter, the existing high level synthesis technologies for addressing FPGA productivity concerns are briefly discussed. The following sections introduce the various aspects of OpenCL architecture, the programming language and an insight into the ongoing efforts on enhancing FPGA productivity using OpenCL.

2.1 High-Level Synthesis

FPGAs are being used in embedded systems and high performance computing solutions to deploy complete applications or act as a coprocessor to general purpose processors. They provide significant performance and power advantages owing to their massively parallel architecture. However, application development on FPGA in the conventional manner requires dealing with the nuts and bolts of hardware design. This leads to longer design cycles, time consuming methods to iterate over different alternatives and tedious debugging procedures resulting in lower turns per day.

Source Inputs	Languages and Compilers
C/C++	C-to-Verilog, ImpulseC, Handel C, Xilinx AutoESL, Mitrion C, FPGAC, Synphony C, ROCCC, LegUp
System C	Bluespec, Xilinx AutoESL
Java	JHDL, MaxCompiler
Schematics	Altium Designer, LabViewFPGA, Simulink
Python	MyHDL
C#	Kiwi

Figure 2.1: Few high-level synthesis tools and their source inputs

High-level synthesis for FPGAs to enhance the portability, scalability and productivity while maintaining the optimized performance achievable, has been a classic area of interest for decades. Several commercial tools and academic research projects have risen to provide this abstraction in FPGAs and significantly reduce the design efforts. These follow either a high level programming approach or a graphical design capture for development. Figure 2.1 shows few such tools categorized according to their source inputs.

In the text-based approach, the most common source input is the C/C++ derivative with restrictions on certain aspects of the language like recursions and pointer manipulations. Interpretation of algorithms described at this level and conversion to hardware has been extensively explored. The tools aim towards facilitating faster prototype, implementation and debugging through the familiar C environment. Advanced compiler techniques are used to optimize the design for the target. The generated HDL from these tools, in many cases, have shown to outperform hand-crafted optimized HDL implementations in larger designs. Handel-C [5] is a high level language aimed towards synchronous hardware design and development. Parallel constructs were used to express parallelism in an otherwise sequential

program written in conventional C. Communication or synchronization between the parallel processes was achieved through channel type. Impulse-C [6], by Impulse Accelerated Technologies, is yet another C-based language and is derived from Streams-C [7]. Based on ANSI-C, it is combined with a C compatible function library for parallel programming. The streaming programming model of the tool targets dataflow-oriented applications. Impulse-C Co-developer tools include a C-to-FPGA compiler and platform support packages for a wide range of FPGA based embedded systems and high performance computing platforms. Similar to these, there are many other tools like C2H [8], Catapult-C [9], Mitrion-C [10], C-to-Verilog [11], to name a few, that explore into different techniques and mechanisms for efficient compilation of traditional C into optimized hardware.

While the input specifications into these tools are close to the C language, the code is often annotated with additional constructs that control the specifics of the circuit implementation. The program is written at a coarse grain level and the tools extract instruction-level parallelism to achieve concurrency. Fine-grained control is provided through options like loop unrolling and pipelining. To be able to generate optimized implementations, the developer has to follow a hardware aware programming approach thus necessitating the knowledge of basic circuit design.

OpenCL is a platform independent framework introduced by the Khronos group for parallel programming[12]. Being functionally portable across platforms, it has opened up various research avenues in heterogenous computing. Though OpenCL architecture and specifications are skewed towards GPU programming, its framework can be efficiently mapped onto various accelerator technologies. Commercial compilers from companies such as AMD, NVIDIA, Intel and Apple enable development of OpenCL applications on CPUs and GPUs. For years, CUDA, which is available only for NVIDIA cards, has been used to develop many GPU-accelerated implementations. CUDA to OpenCL translators like CU2CL [13] were created

in an effort to utilize these designs on other GPUs as well. Providing support for OpenCL on other hardware architectures including, but not limited to, multi-core models, reconfigurable FPGA fabric and heterogeneous environments with different combinations of CPU, GPU and FPGA is another path being tread.

OpenCL computation kernels are described at the finest granularity making it an inherently parallel language. In case of C-to-HDL compilers, the input language is sequential by nature, placing a significant weightage on the capability of the tool to extraction parallelism at the instruction level. Optimizations at a fine-grain level can be achieved through the use of appropriate directives. Due to this, a learning curve is associated with every tool for writing C code in a manner that is efficient for compilation to hardware. The developer needs to be familiar with the basics of hardware design along with the programming model of the tool and the additional options provided in it to tweak the hardware implementation. OpenCL on the other hand has an architecture and a programming model that the developer needs to be familiar with. The manner in which this is mapped onto an accelerator is abstracted from the front end, thus ensuring that the programmer is agnostic of the underlying hardware.

2.2 Overview of OpenCL

OpenCL (Open Computing Language) is an parallel language specification aimed to provide portability across different platforms in a heterogeneous computing system. It consists of an API for coordinating computational and communicational tasks with the accelerators and a language based on C99 for describing the compute core at the finest granularity. Detailed descriptions of OpenCL concepts and architecture can be found in the OpenCL Specification [12].

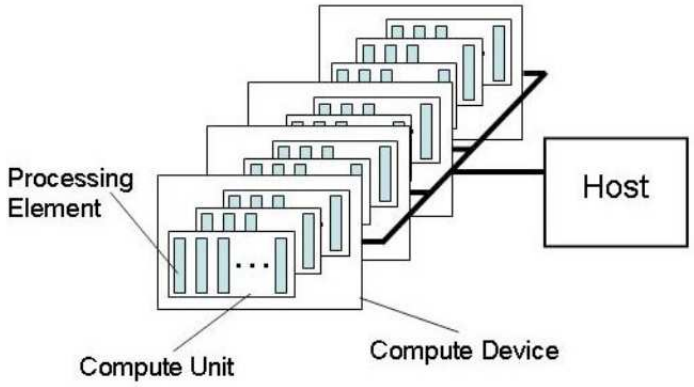

Figure 2.2: The platform model for OpenCL architecture [12]

2.2.1 OpenCL models

OpenCL ideas are described using four models, which will be summarized in this section.

Platform Model

The OpenCL platform model consists of a host device connected to one or more *compute devices*, each of which contain *compute units*, which are internally composed of *processing elements*. This is shown in Figure 2.2. Computations are executed on these processing elements in a single instruction, multiple data (SIMD) or single program, multiple data (SPMD) fashion. The compute devices in a platform can be CPU, GPU, DSP, FPGA or any other accelerator.

Execution Model

The kernel code in an OpenCL application contains the core computational part of the implementation. Its execution is mapped onto a N dimensional index space where N can be *1, 2*, or *3*. The size of this index space expressed as an n dimensional tuple is known as the

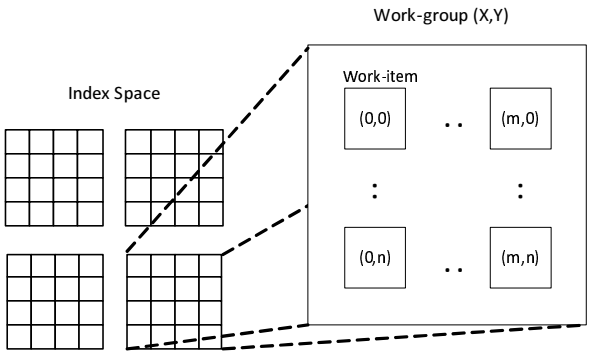

Figure 2.3: An example of an NDRange index space where $N = 2$

global size. Each point in this space, termed as a *work-item*, corresponds to a single instance of the kernel and is assigned a unique *global ID* identified by its coordinates in space. Each of the work-items execute the same code but the execution flow of the code and the data operated on can vary. Work-items are organized in work-groups, the size of which divide the global size evenly in each dimension. Each work-group is assigned a *group ID* and the work items in it have a *local ID* that corresponds to their coordinates within the group. All work items in a group execute concurrently; however, the same cannot be said at the work-group level.

The host program defines the context for kernel execution which includes *kernel function, program objects* and *memory objects*. It also submits commands that control the interaction between the host and the OpenCL devices. These can be *kernel execution commands* which submit the kernel code for execution on the devices, *memory commands* which control all memory transfers and *synchronization commands* which control the order of command execution.

Memory Model

Kernel instances have access to four distinct memory regions:

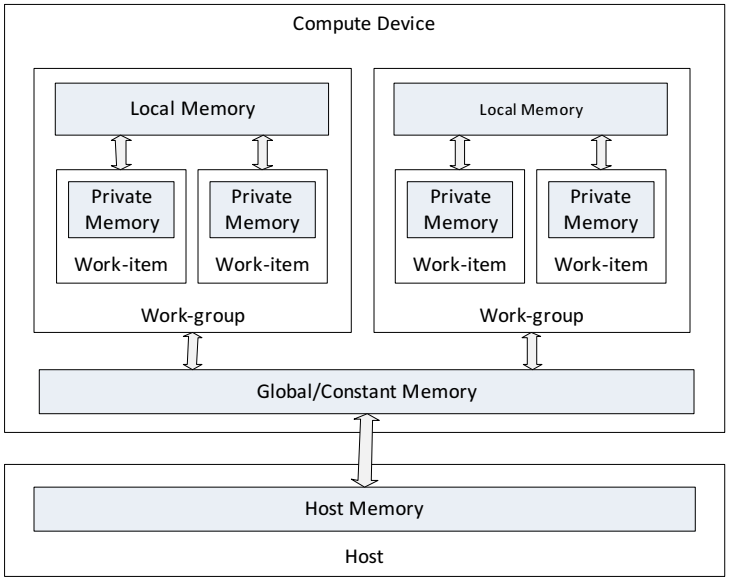

Figure 2.4: OpenCL memory regions and their relation to the execution model

1. ***Global Memory:*** A memory region that is accessible to all work-items in the index space for read/write purposes.

2. ***Constant Memory:*** A region in the global memory that remains constant during a kernel execution.

3. ***Local Memory:*** A region local to a work-group and shared by all work-items within that work-group.

4. ***Private Memory:*** The memory region private to a work-item. Data in this region is not visible to or accessible by other work-items.

Programming Model

The execution model supports *data parallel* and *task parallel* programming models. In data parallel model, each work-item executes on one element or a subset of elements in memory, decided by its global ID. OpenCL supports SIMD and SPMD modes of execution. In task

parallel programming model, a single instance of the kernel executes on the device. This is equivalent to having a single work-item in the index space. Expressing concurrency is then internal to the kernel function.

2.2.2 OpenCL Framework

Platform Layer

This layer implements platform-dependent features that query the OpenCL devices connected to the host machine and create contexts on them.

Runtime Layer

The OpenCL runtime includes application programming interfaces (API) that manage OpenCL objects like command queues, memory objects, kernel program objects etc., using the context created for the device.

2.2.3 OpenCL C Programming Language

The OpenCL programming language is used to create kernel programs that can be executed on one or more target devices. It is based on the C99 specification and supports a subset of the language with restrictions like recursion, function pointers etc. It also consists of various extensions, few of which are listed here :

- Address space qualifiers : The keywords *__global, __local, __private* and *__constant* in variable declarations indicate the region of memory where space needs to be allocated for the variable.
- Synchronization objects : The barrier function is used for synchronizing within a work-

group. All work-items in a work-group must execute the barrier before any can proceed further beyond the barrier.

- Built-in functions : The language provides a set of built-in functions for scalar and vector operations, math functions, geometric functions etc.

2.3 Related work

With the introduction of OpenCL as a parallel programming standard, there has been significant interest, in industry and academia, in combining the OpenCL model with the parallelism capabilities of reconfigurable fabric.

In Novemeber 2011, Altera launched its OpenCL for FPGAs program [14]. Kernel functions are implemented as dedicated pipelined hardware circuits and replicated to further exploit parallelism. On the host side, the OpenCL host program with the standard OpenCL application programming interfaces (API) is compiled using the ACL compiler. goHDR, one of Altera's early customers is recently reported to have achieved a substantial reduction in the development time and a dramatic increase in performance using Altera's OpenCL for FPGA to develop an HDR-enabled television solution[15].

FCUDA explored into programming FPGAs using CUDA (Compute Unified Device Architecture) [16], the parallel programming model developed by NVidia for their GPUs. Its flow includes source-to-source compilation from FCUDA annotated CUDA code to C program for Autopilot. Autopilot is a high-level synthesis tool that converts input specifications in C, C++ or SystemC into RTL descriptions for the target FPGA device [17]. The source-to-source translator coarsens the granularity and extracts parallelism at the level of thread

blocks. ML-GPS (Multi-Level Granularity Parallelism Synthesis) [18] extends this framework to provide flexible parallelism at different levels of granularity.

Lin et. al [19] designed the OpenRCL system for enabling low-power high-performance reconfigurable computing. The target framewok is comprised of parameterized MIPS based processor cores as processing elements. An LLVM based compiler is used for conversion of kernel functions into low level descriptions. LLVM (formerly Low Level Virtual Machine) is an open source compiler infrastructure [20]. Experimental results, using the Parallel Prefix Sum (Scan) application kernel, showed that the performance of FPGA is comparable to that of GPU while the power metric is considerably in favor of FPGAs over the other platforms. This work is generalized in MARC [21] which consists of one control processor and the rest RISC or customized application specific processing algorithmic cores, on its target platform. Using the Bayesian inference application, the performance of a few core variants were observed against a full custom hand-optimized reference implementation on the FPGA. to demonstrate the trade-offs.

SOpenCL (Silicon-OpenCL) [22] is an architecture synthesis tool and follows a template-based hardware generation for the FPGA accelerator. Source-to-source code transformations coarsen the granularity of the kernel functions from a work-item level to that of work-groups. SOpenCL tool flow, which extends the LLVM compiler framework, generates HDL code for these functions, which are then mapped onto the configurable architectural template. This template includes a distributed control logic, a parameterized datapath with the functional units and streaming units for handling all data transfers.

Falcao et. al [3] proposed a multi-platform framework for accelerating simulations in LDPC (Low Density Parity Check) decoding. The OpenCL programming model was used to target a CPU, GPU and an FPGA without any modifications to the input code and using SOpenCL for mapping OpenCL kernels onto FPGA reconfigurable fabric. The LDPC decoding appli-

cation was executed independantly on each of the target architectures and analyses and comparisions were drawn on the performances. The GPU and FPGA outperform the CPU in terms of throughput, while the performance of the FPGA as compared to that of the GPU depends on the size of the design and number of iterations.

Most of the OpenCL to FPGA projects include development of a tool flow that converts the high level specifications from C or OpenCL C to low level RTL descriptions. This thesis aims to build an end-to-end flow leveraging the existing tools for this purpose. The intention is to use current technologies to the best advantage in converting high level algorithms to RTL descriptions so that other aspects like architecture aware optimizations can be concentrated on. Another aspect is that most approaches coarsen the granularity to the work-group level, thereby following a sequential mode of execution for all the work-items within a work-group. This work attempts to increase the concurrency by maintaining the fine-grained parallelism of the language.

2.4 Summary

This chapter presented a brief overview of high-level synthesis for FPGAs and discussed a possible reason for the tools not having gained much popularity. OpenCL was introduced as a viable alternate high-level programming language. The OpenCL architecture, programming models and the language were presented. The chapter also included related work in the field of OpenCL for FPGAs.

Chapter 3

Approach and Implementation

This chapter introduces the approach and discusses the implementation details involved in enabling development of OpenCL application on FPGA platforms.

3.1 Approach

There exists two parts to an OpenCL application - OpenCL C kernels that define the computation for a single instance in the index space on the device and a C/C++ host program that uses OpenCL API for configuring and managing the kernel execution.

In this work, the conversion of the kernel code into hardware circuitry utilizes Xilinx AutoESL C-to-HDL tool[23]. A source-to-source translator is built to convert the kernel program in OpenCL C language to AutoESL C code with appropriate directives, thereby shifting the task of correct directive based programming on the translator as opposed to the developer. Adhering to the specifications of the OpenCL architecture, the granularity is maintained at the level of a work-item. Thus, the HDL core generated from AutoESL represents a single

kernel instance. Multiples of these are instantiated and integrated with memory and dispatch interfaces on the FPGA devices.

A subset of the OpenCL API for the host has been supported to enable testing of applications on the accelerator hardware. The target platform in this implementation is the Convey HC-1 hybrid core computer [24].

3.1.1 An Introduction to AutoESL

AutoESL, a high-level synthesis tool by Xilinx, accepts behavioral level and system level input specifications in C, C++ or System C languages and produces equivalent hardware description files in VHDL, Verilog and System C. It offers myriad directives and design constraints that drive the optimization engine towards desired performance goals and RTL design. The directives specified can either be pertaining to the algorithm or the interfaces. HDL modules are generated as cores with a datapath and finite state machine (FSM) based control logic. The AutoESL tool integrates with the LLVM compiler infrastructure [25] and applies a variety of optimization techniques on the input to reduce code complexity, maximize data locality and extract more parallelism. It also provides a method for functional verification of the generated hardware description using RTL-cosimulation.

3.2 Implementation Specifics

The steps involved in generating the host executable and a full hardware implementation in the form of a bitstream for the FPGA accelerator device is as shown in Figure 3.1. The flow chart shows all the processes and the intermediate output formats for each. The OpenCL host program is compiled and linked with the API library using Convey's *cnycc* compiler.

The right-hand side shows the processes involved in the conversion of a kernel from high level to hardware. The parts of the flow enclosed in dotted lines indicate the source-to-source translation from OpenCL C language to AutoESL C. AutoESL synthesis refers to the C to HDL synthesis performed by the tool. In the interface generation, integration and implementation step, interfaces for the kernel HDL modules are generated so as to integrate them into the convey framework. The entire design is then implemented to generate a bitstream using Xilinx ISE tools.

3.2.1 OpenCL to AutoESL Source-to-Source Translation

Various mechanisms are in use for source-to-source translations between languages at the same abstraction level. One of the methods adopted is to convert the input source to an intermediate representation (IR), perform required transformations on the IR and generate code in the output language. Numerous compilation frameworks [26–28] are available that can be leveraged for this purpose. With an intention of exploring into the feasibility of managing all transformations using simple graphs, Clang framework [29] is used to obtain the Abstract Syntax Tree (AST) of the input code and Graphtool [30] is used for further manipulations.

Clang is an open source compiler front end, designed essentially as an API with libraries for parsing, lexing, analysis and more. This makes it easier to embed into other applications as compared to gcc, which has a monolithic static compiler binary. The Clang driver has an option to emit Clang AST files for the source inputs. Using the ASTConsumer and Visitor classes in the AST libraries, the tree is traversed to generate a simple directed graph for the kernel functions, those declared with the *__kernel* qualifier, in dot format. A dot file is a plain text graph description of the tree, that can be used by various tools for either

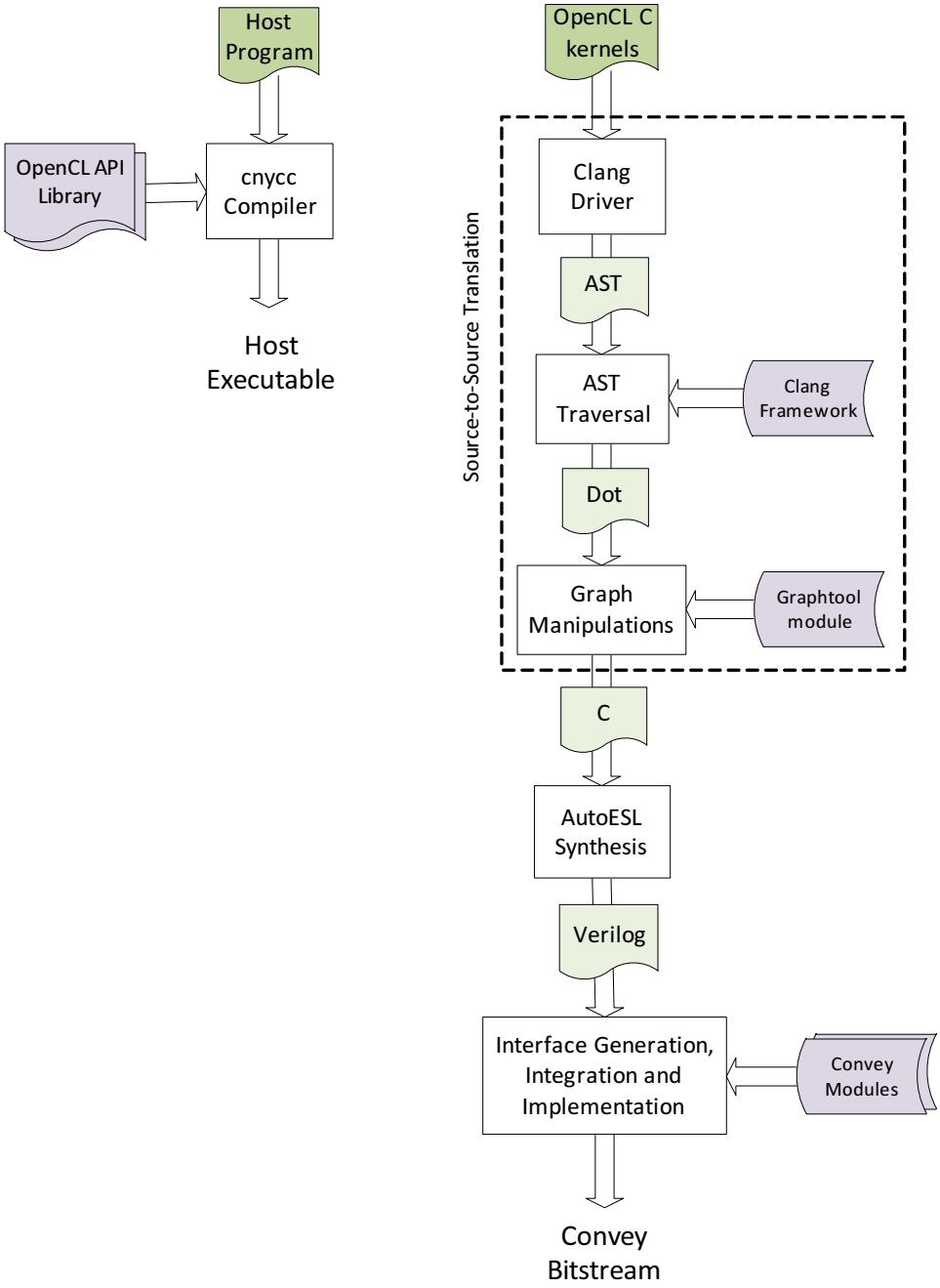

Figure 3.1: Overview of the design flow

graphic rendering or processing. Separate dot files are generated for each kernel in case of multiple kernel tasks defined in the application. While clang includes methods for recursive AST traversal and Graphviz dot file generation, custom methods are created for both in accordance with the requirements. The dot file generated from the clang driver includes limited details about the code, with the information being only about the type of a statement or expression for the purpose of visualization. The custom AST traversal method visits all required statements and includes the variables as well as the operators into the dot file. This acts as input for the graph processing tool called Graphtool.

Listing 3.1 shows the OpenCL C kernel program for a vector addition application. It defines the task for a single instance which adds one element of the first vector to the corresponding element in the second vector and stores the sum in the result vector. The instance identifies the index of this element in the vector using its global ID. The dot file visualization for the abstract syntax tree of vector addition kernel is shown in Figure 3.2. The *CompoundStmt* node indicates the beginning of a function body or the body of a statement. The text description of this file, which contains the name and value for each of the nodes, is available in the Appendix A.

Listing 3.1: OpenCL C file with the kernel function for vector addition

```
__kernel void VectorAdd(__global const long * a, __global const
   long * b, __global long * c, int iNumElements)
{
  int tGID = get_global_id(0);
  if (tGID < iNumElements)
  {
    c[tGID] = a[tGID] + b[tGID];
  }
```

```
}
```

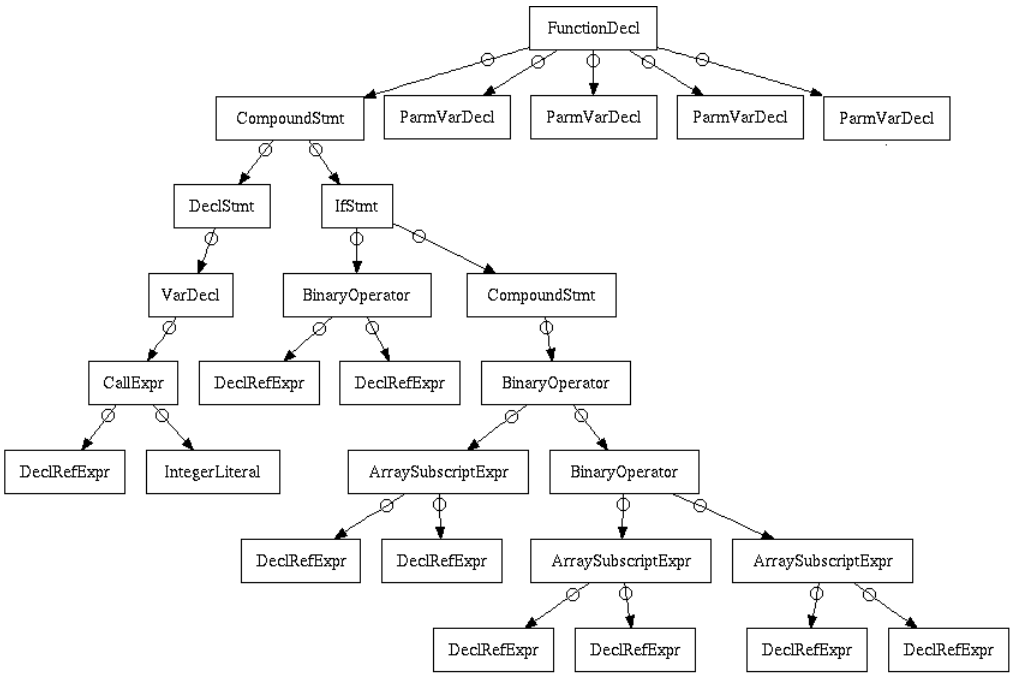

Figure 3.2: Dot format visualization for the vector addition kernel

Graphtool is a python-based module for graph analysis and manipulations, with functions for many graph algorithms. Filtering, purging or addition of nodes and modifications to the tree are relatively easier to handle using this module as compared to manipulating Clang's AST within its framework. The *subgraph_isomorphism* function in the topology subpackage performs structural pattern matching and is used in the translator to identify function calls. The *local ID, global ID* and *group ID* for a work-item are accessed within the kernel code. In the vector addition example *get_global_id(0)* is used to obtain the *global ID* of the instance. The argument specifies the dimension in which the ID is requested. In the hardware implementation these values are to be sent to the core modules as an input. For this reason, they are modified from being called functions to arguments into the kernel function.

The starting addresses for the input and output variables as passed as pointers into the kernel in OpenCL. By default, pointers in function arguments are used to indicate BRAM interfaces. In order to force additional handshake signals for each of the ports in RTL, the pointers in the kernel function arguments are annotated with AutoESL *ap_bus* interface directive. One of the parameters for this directive in AutoESL is width, which is used by the verification engine of the tool while generating test bench for the code. Since this feature is not being used, the global size of the application is assigned to it by default. The arguments with the *__private* access specifier are transformed into variable declaration statements within the function. This is because the private memory is specific to a work-item and is implemented within the core.

Barriers in OpenCL allow for synchronization between threads in a work-group. All work-items within a work-group must execute this before any are allowed to continue beyond the barrier. The translator modifies the barrier functions to *barrier_hit* and *barrier_done* signals at the function interface. In the present implementation, when a core reaches a barrier instruction, it sends a value on the *barrier_hit* port and then waits to receive a high on *barrier_done* before proceeding further.

The resulting graph after performing all required transformations is reparsed to generate an AutoESL C code shown in Listing 3.2.

```
#include "VectorAdd.h"

void VectorAdd(const long * a, const long * b, long * c, int
    iNumElements) {
#pragma AP interface ap_bus depth=1024 port a
#pragma AP interface ap_bus depth=1024 port b
#pragma AP interface ap_bus depth=1024 port c
```

```
  int tGID = global_id_0;
  if (tGID < iNumElements) {
    c[tGID] = a[tGID] + b[tGID];
  }
}
```

Listing 3.2: AutoESL C output generated by the translator

3.2.2 AutoESL Synthesis

AutoESL synthesizes the C program from the translator to generate customized FSM with datapath RTL implementation that corresponds to a single processing element (PE) onto which work-items are mapped. The PEs are shown in Figure 3.3. The required frequency of operation and the target FPGA device are provided as input to the tool along with the annotated C code. Apart from input or output ports for the function arguments, the tool's interface protocol provides *clock, reset, start, done* and *idle* handshake signals for the generated module. Optimized cores from Xilinx libraries, like floating point cores, storage cores and functional unit cores are included into the HDL design, by the tool, as required.

3.2.3 Integration and Mapping on Convey

The target platform in the implementation is the Convey HC-1 hybrid core computer that consists of an Intel Xeon CPU and an FPGA based reconfigurable coprocessor. The coprocessor, connected to the CPU through the front-side bus (FSB), hosts four Xilinx Virtex-5 XC5VLX330 compute FPGAs known as application engines (AE), eight memory controllers

(MC) that support a total of sixteen DDR2 memory channels and an application engine hub (AEH) that implements the interface to the Intel host. Each AE is connected to the rest of the coprocessor through a dispatch interface, memory controller interfaces, AE-AE interface and a management interface. This framework is provided by Convey in the form of Verilog modules.

The Convey coprocessor is considered as an OpenCL compute device with each of the AEs corresponding to a compute unit as shown in Figure 3.3. At any given time a single work-group is mapped onto an application engine. Scheduling of the work-group tasks among the four compute units is done by the host CPU. Each AE contains multiple instances of the kernel cores onto which work-items are mapped. The top-level Verilog module from AutoESL is parsed to generate appropriate wrapper, dispatch and memory access modules that are interfaced with the Convey provided framework. The dispatch unit sends the appropriate IDs and start signals to the cores. Round-robin arbiters control the load/store requests from the cores to the memory controller interfaces. There are two arbiters for every memory controller interface, each connecting to the even and odd ports of the interface, thus facilitating upto sixteen parallel memory accesses. The generation of the interface modules and their integration are automated and does not involve any user intervention. Figure 3.3 shows the architecture of the system on convey. The final design is implemented using Xilinx ISE tools to generate the bitstream for the compute FPGAs on Convey.

Global memory, which can be read/written to by all work-items, is mapped onto the external DDR2 modules on the coprocessor. The latency of this memory is high; however, sixteen channels are available for parallel accesses. Local memory being smaller and faster as compared to the global memory, is implemented using on-chip BRAMs on the AE. Registers within the kernel core modules are used for implementing private memory.

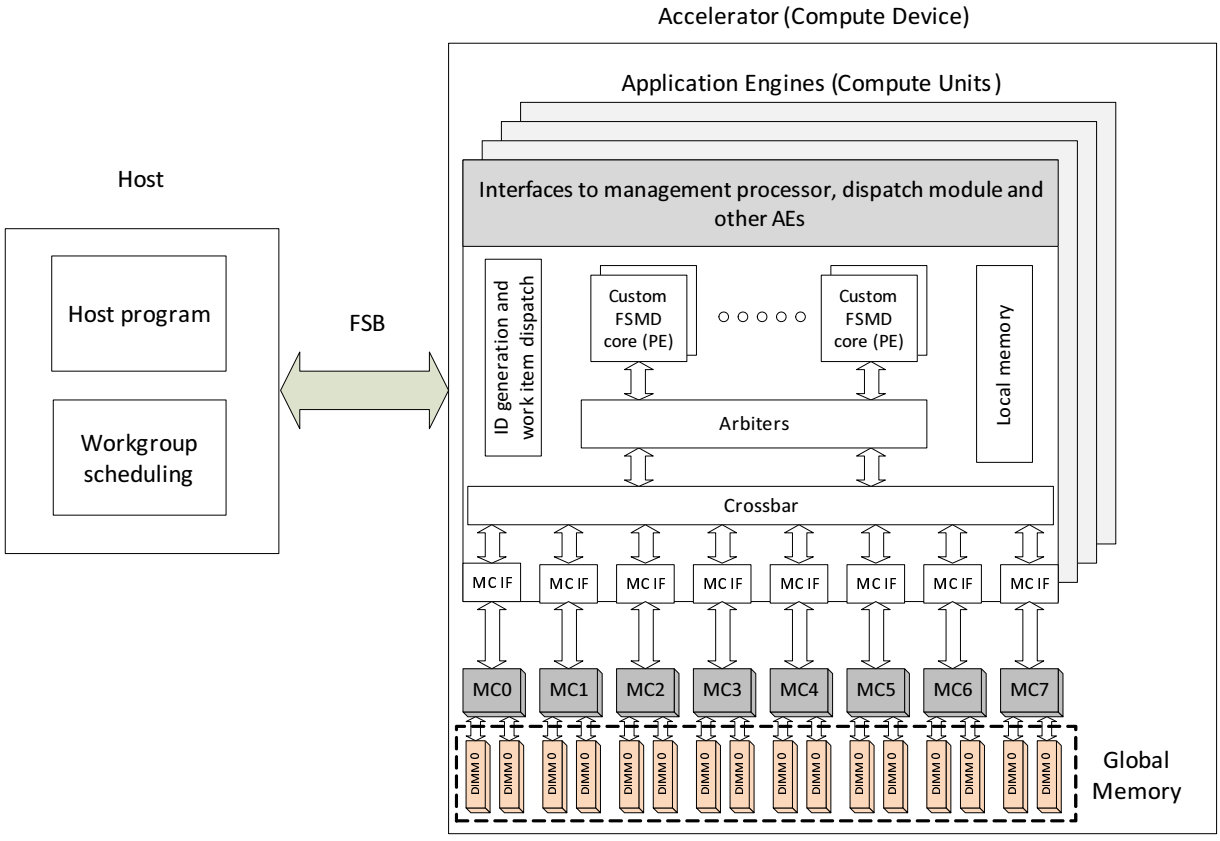

Figure 3.3: System architecture on Convey

3.2.4 Host Library

The *main()* function in an OpenCL host program primarily performs the following operations using OpenCL API:

- Detect the accelerator connected to the host machine,
- Create context and command queue for the accelerator,
- Load kernel file, build a program object and create kernel object,
- Create memory objects for the kernel arguments,
- Enqueue buffers to transfer data from host to device memory,
- Enqueue kernel for execution, and
- Read the results from the memory objects.

The host library in this work contains definitions for a subset of the API required to test the execution of the kernel tasks on the hardware. The definitions are targeted specifically for the Convey platform. A driver for the FPGA device is yet to be implemented and presently all communications between the host and the accelerator are managed through Convey specific assembly routines.

One of the aspects of OpenCL is online compilation where the OpenCL C programs are built at run-time. Since FPGA implementation times on Convey run into hours, a pre-compiled bitstream is loaded onto the hardware. One disadvantage of following an offline compilation model is that the number of dimensions and the size of a work-group in each dimension has to be fixed at compile-time as the hardware implementation varies depending on these numbers. The parameters and their values are declared in a file and passed as input into the

translator. The contents of this file for the vector addition application, with a work-group size of 16, is shown in Listing 3.3. In the current implementation the number of physical cores is same the work-group size.

```
# Number of dimensions in the NDRange index space
dim 1

# Size within a workgroup in each dimension
local_size_0 16

# Total number of physical cores
num_cores 16
```

Listing 3.3: Parameters file for vector addition application

Some of the definitions implemented for the API are as explained here :

- `clSetKernelArg()` function sets the value for a specific argument of a kernel. The AEs on Convey contain application engine registers (AEGs) over which the host has read/write access. The host sends the kernel arguments to the compute devices by writing these values onto the AEGs in each AE. In this work, the AEG registers from *0* to *9* are reserved. Arguments are written over starting at AEG register *10*.

- `clEnqueueWriteBuffer()` and `clEnqueueReadBuffer()` functions transfer data between host memory and the buffer object allocated on the coprocessor memory region.

- `clEnqueueNDRangeKernel()` is implemented as a blocking function that enqueues kernel for execution and waits for its completion. The kernel task is divided into

work-groups and dispatched to the compute devices. At a given instant of time, four work-group tasks are being executed concurrently on the four AEs on Convey. The scheduling of the work-group tasks between the AEs is managed by the host machine, using polling technique. The AEs are constantly polled after dispatch of the tasks to check their state. When any are done, the next work-group task is dispatched onto it. This process is continued until all tasks are completed. The implementation for this function is provided in Appendix B.

3.3 Summary

In this chapter the steps involved in the compilation of kernels into HDL cores and the architecture of the system on the FPGA hardware, was discussed using the example of a vector addition application. It was seen that existing tools can be used to the best advantage without having to build our own compiler. The host library supports functionality for some of the OpenCL API, assuming the FPGA coprocessor to be present and available.

Chapter 4

Results

The main objective of this work is to develop a proof of concept system that enables the development of OpenCL applications on FPGA platforms. In this chapter, a simple vector addition application is studied. The performance and resource utilization numbers for different input parameters are presented and explained.

4.1 Case Study : Vector Addition

The conversion of the OpenCL kernel for vector addition and the obtained AutoESL output was discussed in the Implementation chapter. A pre-compiled bitstream loaded onto the FPGA devices in the Convey coporocessor. The execution flow for the host program is shown in Figure 4.1. Initialization includes allocation of host memory for the input vectors and assigning values to the them. Kernel arguments are sent to the AEG registers on the application engines and the kernel is enqueued for execution. The total computation time for the accelerator in OpenCL code involves the time taken for transfer of input data to device or coprocessor memory, setting of kernel arguments, execution time on hardware and the

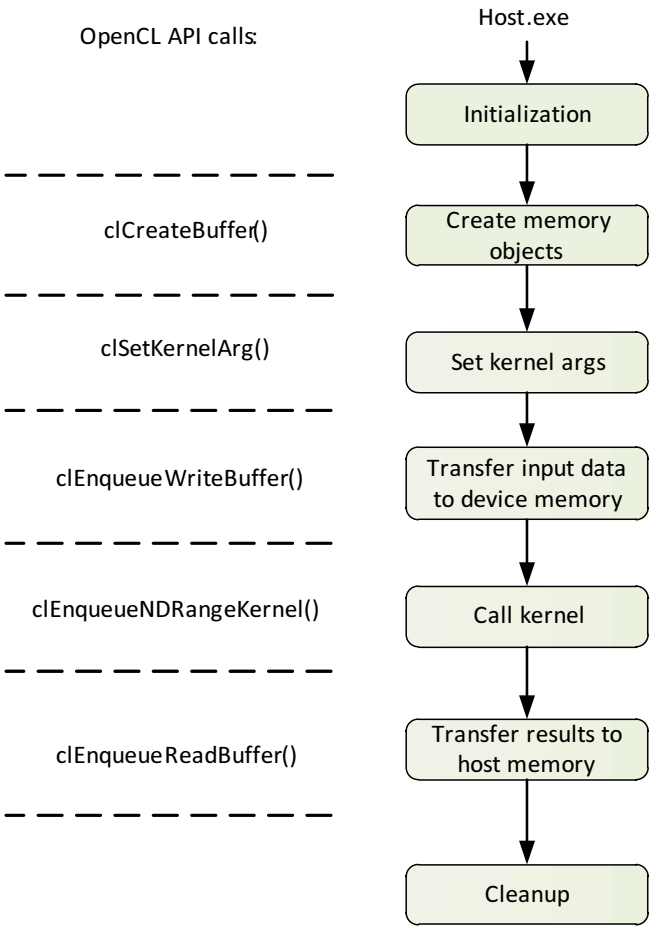

Figure 4.1: Execution flow for the vector addition application

time taken to transfer results back to host memory. Convey provides a memory allocation instruction through which the host can directly allocate space on the coprocessor memory. Using this would avoid the need to transfer data between host and coprocessor memory on Convey.

Since the method of offline compilation of OpenCL kernels is being used, parameters pertaining to the dimensions of the solution index space, the global size for the solution and the local size of a work-group are fixed at compile time. In the current implementation the size of a work-group is the same as the number of physical cores on each AE in the Convey coprocessor. The performance of the application was evaluated for different sizes of the work-group.

	Convey Example	OpenCL implementations		
		64 cores	132 cores	192 cores
Execution time (in ms) including memory transfer	0.064	1.385	0.67	0.463
Execution time (in ms) excluding memory transfer	-	1.677	0.968	0.755

Table 4.1: Performance results for vector addition application for vectors of size 1024

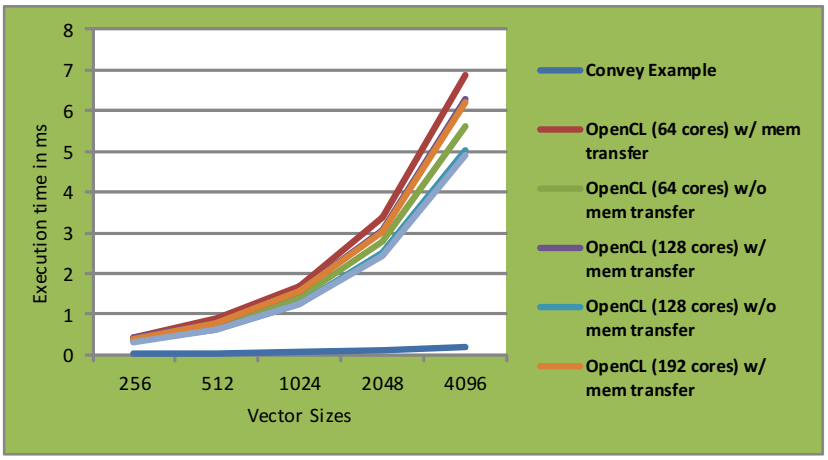

Figure 4.2: Performance results for different vector sizes

After the integration of the kernel modules and the interfaces into the framework, verilog simulation is performed over the entire design using Convey's simulation environment. After ensuring functional correctness, the application was executed on hardware.

4.1.1 Performance and Resource Utilization

Table 4.1 compares the performance results between the vector addition example from Convey and the OpenCL implementation for the same application. All programs are executed over vectors of size 1024. The target devices are four Virtex-5 XC5VLX330 FPGAs operating at a frequency of 150 MHz.

The bitstreams for the OpenCL accelerator devices are generated for three different values

Name	FF	LUT	BRAM	DSP48E	SLICE
Component	-	-	-	-	-
Expression	0	44	-	-	-
FIFO	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	3	-	-	-
Register	99	-	-	-	-
Total	99	47	0	0	0
Available	207360	207360	576	192	51840

Table 4.2: Area Estimates for a single core from AutoESL report

of the work-group size - *16, 32* and *48*. In each case, the number of physical cores on each AE corresponds to the size of the work-group. The total number of cores on the coprocessor device are 64, 128 and 192 respectively. Convey's example design consists of 16 adder modules per FPGA, amounting to a total of 64 modules. These modules access memory in a continuous manner over the entire range as opposed to the OpenCL implementations where batches of tasks are scheduled by the host. The scheduling at the work-group level calls for additional overhead which is prominent in smaller designs as can be seen from the execution times in the table. With the vector size constant, as the work-group size is increased, the number of work-group tasks to be scheduled decreases thus reducing the total execution time. The performance numbers for different vector sizes is shown in Fig 4.2.

AutoESL synthesis provides a report for every generated design which contains the estimated resources for the hardware. On testing other sample AutoESL applications, these numbers have been found to comply well with the actual resource utilization numbers from Xilinx tools after implementation. The area estimates, according to the AutoESL tool for a single vector addition module is as shown in Table 4.2.

The total device utilization for each of the implementations is shown in Table 4.3. The numbers represent the resources for a single compute FPGA and are expressed as a percentage

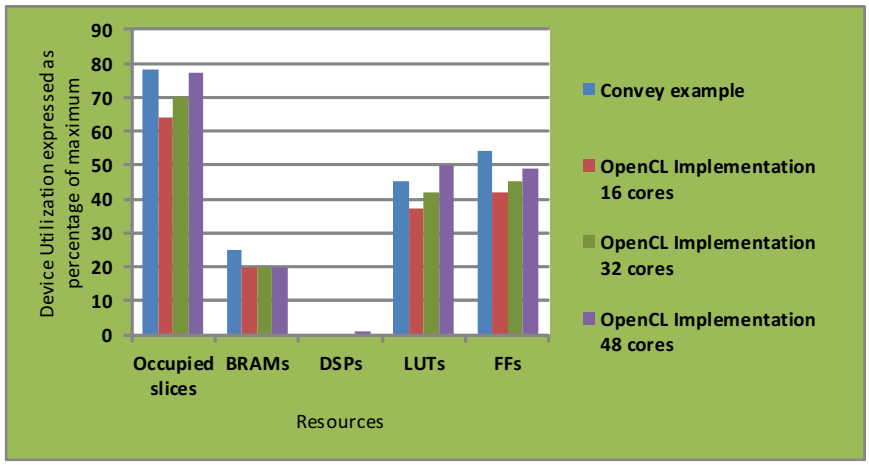

Figure 4.3: Resource Utilization for vector addition application for a single AE

of the maximum device resources available. Modules provided in the Convey framework consume about 11% of the device resources.

The resource utilization for the OpenCL implementations are observed to be much lesser than the Convey example. With enhancements to the memory access patterns and differentiation between the physical and logical number of cores in a work-group, performance improvements over the current implementation can be achieved.

4.2 Case Study : Matrix Multiplication

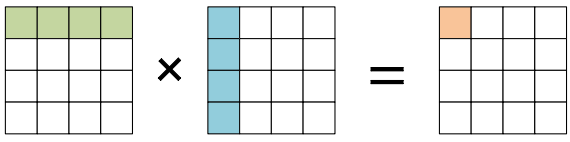

Figure 4.4: Matrix multiplication operation

The matrix multiplication computation is used in many applied math and scientific applications. A quick recall of the method is shown in Fig 4.4. The shaded regions depict the elements involved in calculation of an element in the result matrix.

```
// Matrix Multiplication
__kernel void matrixMul(__global long * C, __global long * B,
   __global long * A, uint wA, uint wB)
{
  int tx = get_global_id(0);
  int ty = get_global_id(1);

  long value = 0;
  for (int k = 0 ; k < wA; k++)
  {
    long As = A[ty * wA + k];
    long Bs = B[k * wB + tx];
    value += As * Bs;
  }
  C[ty * wA + tx] = value;
}
```

Listing 4.1: OpenCL C file with the kernel function for matrix multiplication

The execution flow of the host program for this application is similar to the flow discussed for vector addition in the previous section. Listing 4.1 shows the kernel code in OpenCL C. Listing 4.2 shows the equivalent AutoESL code. The index space for the matrix multiplication application is two dimensional. Each work-item evaluates one element in the result matrix. In the current implementation, a work-group is of size *4* by *4* with a total of *16* work-items. The work-group sizes in each dimension can vary as long as it evenly divides the entire global space. The HDL generated from AutoESL was successfully integrated into

the Convey framework and implemented using Xilinx ISE tools. The functionality of the application was tested in simulation for different matrix sizes, using the host program and the Verilog files for the hardware.

```
#include "core_header.h"

void matrixMul(long * C , long * B , long * A , uint wA , uint wB
    , int global_id_0 , int global_id_1  ) {
#pragma AP interface ap_bus depth=64 port=C
#pragma AP interface ap_bus depth=64 port=B
#pragma AP interface ap_bus depth=64 port=A

  int tx = global_id_0 ;
  int ty = global_id_1 ;
  long value = 0 ;
  int k;
  for ( k  = 0 ;k  < wA ; k++ ) {
    long As = A  [ ty  * wA  + k ];
    long Bs = B  [ k  * wB  + tx ];
    value  += As  * Bs ;
  }
  C  [ ty  * wA  + tx ] = value ;
}
```

Listing 4.2: AutoESL C output for matrix multiplication generated by the translator

4.3 Comparison of Methodologies

Table 4.3 shows a comparison of the compilation flow and architecture presented in this thesis with other implementations, which were discussed in the related work section. The first parameter defines the nature of the processing element (PE) in each framework. OpenRCL implementation includes parameterized MIPS cores over which kernel instances are executed as threads. Though the processor supports variable datapath width and multi-threading, speed-up comparable to a complete hardware circuitry is hard to achieve. SOpenCL uses an architectural template for the generation of HDL. This template is implemented as a combination of a datapath and a streaming unit. In the flow presented in this thesis, the PEs are customized FSMD cores generated by the AutoESL tool, optimized for the application at hand. For complex computations, aggressive optimizations can be achieved by enforcing area and performance constraints into the tool.

The second aspect of comparison is the parallelism in the implementation. OpenCL kernels are defined at the finest granularity. SOpenCL coarsens the granularity to the level of work-groups using thread serialization technique. The computation is enclosed in nested loops, one for each dimension, thus enforcing sequential execution of work-items in all dimensions within a work-group. In this thesis, the fine-grained parallelism of the application is maintained.

Support for multiple FPGA devices is provided in the current work and has been successfully demonstrated. A higher degree of parallelism is achieved on partitioning the execution onto multiple FPGAs. There has not been any explicit mention of this feature in the other implementations.

The compilation flow presented in this thesis avoids the re-invention of a C-to-HDL compiler by using an existing tool for the purpose of conversion. Both OpenRCL and SOpenCL build their compilers using the LLVM framework.

	Presented Flow	OpenRCL	SOpenCL
Processing elements	Customized FSMD	Parameterized MIPS cores	FSMD
Fine-grain parallelism	Yes	Yes	No
Support for mutiple FPGAs	Yes	No	No
Design of compiler	No	Yes	Yes
Verilog simulation	Yes	No	No

Table 4.3: Comparison with SOpenCL and OpenRCL implementations

	Presented Flow	OpenCL for FPGAs
Processing elements	Customized FSMD	Pipelined hardware
Platform dependent	Yes	Yes
OpenCL runtime support	Work in progress	Good

Table 4.4: Comparison with Altera's OpenCL for FPGAs program

Another aspect is verilog simulation. A simluation environment is available in the current implementation using which the verilog files can be simulated along with the host program to check for functional correctness or try out alternatives.

Table 4.4 presents few points of comparisons against the Altera's tool for supporting OpenCL applications on FPGAs. Altera's tool implements the kernel logic as deeply pipelined hardware circuits, which are then replicated to increase parallelism. A common factor is that both implementations are platform dependant. Altera's tool is used for Altera's FPGA families. The implementation presented in thesis is specfic to Xilinx devices due to the use of Xilinx AutoESL tool in the compilation flow. The current OpenCL API library in this work provides a limited support and will be extended in future to include other features as well.

4.4 Challenges in Using OpenCL

OpenCL provides a good abstraction from the low level details of hardware implementation through its virtual architecture. This ensures smaller development times and faster time to market. Also, the applications are portable across different platforms. At the same time, portability is only functional. Various architecture aware optimizations are needed for every hardware device, in order to obtain maximum performance.

One the biggest advantage of FPGAs lie in being able to use different bit widths for the data. An example application that utilizes this is genome sequencing in bioinformatics. This advantage is nullified when programming using OpenCL, as the developer is limited to the numerical data types provided in the language.

4.5 Summary

This chapter discussed the flow of a vector addition application in OpenCL and its execution on FPGA hardware. The performance and resource utilization values for different solution sizes were presented. Also, comparisons were drawn on the features of this thesis work against other related implementations.

Chapter 5

Conclusion

High-performance applications often require high design efforts for FPGA implementations. This thesis aimed towards improving the design productivity of FPGAs using OpenCL as the high-level programming language for development. In this work a method of compilation of OpenCL C kernels into hardware decriptions was discussed. It also presented the design and implementation of an architecture on the reconfigurable fabric to support the execution of the computation kernels by interfacing the cores with host and memory modules. On the host side, the functions in the OpenCL API required to manage kernel execution were supported to test the flow. Conversion of the kernels to device specific executable and execution of the application was successfully demonstrated in simulations and on the Convey HC-1 hybrid computer.

5.1 Future Work

The main aim of this work was to successfully demonstrate the compilation and execution an OpenCL application on FPGA platform. The simulation and the hardware results were

presented for a vector addition program. The work can be extended to provide a more complete and robust flow with improved performance for more complex designs. The following points are the main areas for enhancements.

- In this work, only a subset of the OpenCL host API was supported on the host. Also, the definitions contained Convey specific assembly routines to perform the desired operations, assuming that FPGAs were connected to the host and were available for programming. These can be made generic. A driver can be implemented to detect accelerators attached to the host machine and help create contexts and command queues on them. Additional API function calls, that manage the initial set up with the hardware, will have to be implemented. The driver would then facilitate communication between the hardware device and the host API.

- The compilation flow can be extended to support more features of the OpenCL language. Special data types for images, vectors and built-in math functions are used in many applications. One option would be to support AutoESL implementations for these and link them with the input code into the tool.

- In the system architecture, modifications can be done to the memory access patterns to improve the memory bandwidth utilization. Currently a handshaking mechanism is implemented at the interface of the cores for reading data from and writing data to the memory. For example, if the memory accesses are sequential, then contiguous elements can be pre-fetched from memory in order to reduce the latency in subsequent off-chip requests.

Bibliography

[1] K. H. Tsoi and W. Luk, "Axel: A Heterogeneous Cluster with FPGAs and GPUs," in *FPGA'10, Monterery, California, USA*, 2010.

[2] T. Ahmed, "OpenCL Framework for a CPU, GPU and FPGA Platform," Master's thesis, University of Toronto, 2011.

[3] G. Falcao, M. Owaida, D. Novo, M. Purnaprajna, N. Bellas, C. D. Antonopoulos, G. Karakonstantis, A. Burg, and P. Ienne, "Shortening design time through multiplatform simulations with a portable OpenCL golden-model: the LDPC decoder case," in *IEEE 20th International Symposium Field-Programmable Custom Computing Machines*, 2012.

[4] J. Kepner, "HPC Productivity: An Overarching View," vol. 18, pp. 393–397, November 2011.

[5] "Handel-C Language Reference Manual." `http://www.pa.msu.edu/hep/d0/l2/Handel-C/Handel%20C.PDF`.

[6] "Impulse CoDeveloper C-to-FPGA Tools." `http://www.impulseaccelerated.com/products_universal.htm`.

[7] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-Oriented FPGA Computing in the Streams-C High Level Language," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[8] A. Corp., "Nios II C2H Compiler User Guide," 2009.

[9] I. A. Technologies.

[10] S. Mohl, "The Mitrion-C Programming Language," (Lund, Sweden), Mitrionics Inc., 2005.

[11] C. to Verilog. `http://www.c-to-verilog.com/`.

[12] Khronos Group, "OpenCL specification 1.1."

[13] G. Martinez, M. Gardner, and W.-c. Feng, "CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-core Architectures," in *IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, 2011.

[14] "Implementing FPGA Design with the OpenCL Standard," tech. rep., November 2011.

[15] `www.altera.com/corporate/news_room/releases/2012/products/nr-opencl-gohdr.html`.

[16] NVIDIA, "CUDA." `http://www.nvidia.com/object/cuda_home_new.html`.

[17] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "AutoPilot: A Platform-Based ESL Synthesis System," in *High-Level Synthesis*, 2008.

[18] A. Papakonstantinou, Y. Liang, J. A. Stratton, K. Gururaj, D. Chen, H. W.-M. W., and J. Cong, "Multilevel Granularity Parallelism Synthesis on FPGAs," in *Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2011.

[19] M. Lin, I. Lebedev, and J. Wawrzynek, "OpenRCL: Low-Power High-Performance Computing with Reconfigurable Devices," in *International Conference on Field Programmable Logic and Applications*, 2010.

[20] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[21] I. Lebedev, S. Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, M. Lin, and J. Wawrzynek, "MARC: A Many-Core Approach to Reconfigurable Computing," in *International Conference on Reconfigurable Computing*, 2010.

[22] M. Owaid, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of Platform Architectures from OpenCL Programs," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2011.

[23] X. Inc. `http://www.xilinx.com/products/design-tools/autoesl/index.htm`.

[24] C. computer, "Conveys hybrid-core technology: the HC-1 and the HC-1ex." `http://www.conveycomputer.com/Resources/Convey_HC1_Family.pdf`.

[25] `llvm.org/Users.html`.

[26] D. Quinlan, "ROSE: Compiler Support for Object Oriented Frameworks."

[27] B. e. a. Hansang, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores."

[28] http://gcc.gnu.org/.

[29] "clang: a C language family frontend for LLVM." http://clang.llvm.org/.

[30] http://projects.skewed.de/graph-tool/.

Appendix A: Dot File Description for Vector Addition

```
digraph VectorAdd {
  Node1000 [shape=record, label="FunctionDecl", name = VectorAdd,
     type = void];
  Node1000 -> Node1001;
  Node1001 [shape=record, label="ParmVarDecl", name = a, type =
     "const __global unsigned long *"];
  Node1000 -> Node1002;
  Node1002 [shape=record, label="ParmVarDecl", name = b, type =
     "const __global unsigned long *"];
  Node1000 -> Node1003;
  Node1003 [shape=record, label="ParmVarDecl", name = c, type =
     "__global unsigned long *"];
  Node1000 -> Node1004;
  Node1004 [shape=record, label="ParmVarDecl", name =
     iNumElements, type = "int"];
  Node1000 -> Node1005;
```

```
Node1005 [shape=record, label="CompoundStmt" ];
Node1005 -> Node1006;
Node1006 [shape=record, label ="DeclStmt" ];
Node1006 -> Node1007;
Node1007 [shape=record, label ="VarDecl", type = "int", name =
   tGID, value = 1 ];
Node1007 -> Node1008;
Node1008 [shape=record, label ="CallExpr" ];
Node1008 -> Node1009;
Node1009 [shape=record, label="DeclRefExpr", name =
   get_global_id ];
Node1008 -> Node1010;
Node1010 [shape=record, label ="IntegerLiteral", value = 0 ];
Node1005 -> Node1011;
Node1011 [shape=record, label="IfStmt" ];
Node1011 -> Node1012;
Node1012 [shape=record, label="BinaryOperator", value = "<" ];
Node1012 -> Node1013;
Node1013 [shape=record, label="DeclRefExpr", name = tGID ];
Node1012 -> Node1014;
Node1014 [shape=record, label="DeclRefExpr", name =
   iNumElements ];
Node1011 -> Node1015;
Node1015 [shape=record, label="CompoundStmt" ];
Node1015 -> Node1016;
```

```
  Node1016 [shape=record, label="BinaryOperator", value = "=" ];
  Node1016 -> Node1017;
  Node1017 [shape=record, label="ArraySubscriptExpr" ];
  Node1017 -> Node1018;
  Node1018 [shape=record, label="DeclRefExpr", name = c ];
  Node1017 -> Node1019;
  Node1019 [shape=record, label="DeclRefExpr", name = tGID ];
  Node1016 -> Node1020;
  Node1020 [shape=record, label="BinaryOperator", value = "+" ];
  Node1020 -> Node1021;
  Node1021 [shape=record, label="ArraySubscriptExpr" ];
  Node1021 -> Node1022;
  Node1022 [shape=record, label="DeclRefExpr", name = a ];
  Node1021 -> Node1023;
  Node1023 [shape=record, label="DeclRefExpr", name = tGID ];
  Node1020 -> Node1024;
  Node1024 [shape=record, label="ArraySubscriptExpr" ];
  Node1024 -> Node1025;
  Node1025 [shape=record, label="DeclRefExpr", name = b ];
  Node1024 -> Node1026;
  Node1026 [shape=record, label="DeclRefExpr", name = tGID ];
}
```

Listing 1: Text description of the dot file for vector addition, before manipulations

Appendix B: OpenCL API Implementations for Kernel Execution

- *clSetKernelArgument* function : The aegWr function writes the value of arg_value in the AEG register specified by arg_index, on all the application engines.

```
cl_int
clSetKernelArg (cl_kernel kernel,
                cl_uint arg_index,
                size_t arg_size,
                const void *arg_value)
{
  aegWr(arg_index,(*(uint64*)arg_value));
  return CL_SUCCESS;
}
```

Listing 2: Convey specific definition for clSetKernelArgument function

- *clEnqueueNDRangeKernel* function : The definition contains instructions for dispatch of work-group tasks to the AEs and the polling mechanism for scheduling. The imple-

mentation below is for an application that uses single dimension in the index space.

```
cl_int
clEnqueueNDRangeKernel (cl_command_queue command_queue,
                        cl_kernel kernel,
                        cl_uint work_dim,
                        const size_t *global_work_offset,
                        const size_t *global_work_size,
                        const size_t *local_work_size,
                        cl_uint num_events_in_wait_list,
                        const cl_event *event_wait_list,
                        cl_event *event)
{
  /*calc num of workgroups*/
  size_t workgroups;
  if ((*global_work_size) % (*local_work_size))
  //round up to cover remainder leftovers
    workgroups = (*global_work_size) / (*local_work_size) + 1;
  else
    workgroups = (*global_work_size) / (*local_work_size);

  /*Simple kernel scheduler*/
  int freeAE;
  int workg=0;
  int timeout = TIMEOUT;
  aecWr(-1ll);
```

```
//assing first engines
if (workgroups >3){
  //start first four workgroups at once
  copCall_init();
  workg += 4;
  //calculate remaining ones
  while((workg<workgroups)&&(timeout>0)){
  //busy waiting for any AE available

    //find available AE
    freeAE = aegRd(AE_DONE);
    timeout--;
    while ((freeAE != 0) && (workg<workgroups))
    {
      timeout=TIMEOUT;
      //find first free AE and allocate
      if(freeAE & 0x1){//use first AE
        //set work group
        aegWrAE0(AE_GRID, workg);
        //start kernel
        copCallAE0();
        workg++;//go to next
        freeAE ^= 0x1;//mark as busy
        continue;
      }
      if(freeAE & 0x2){//use second AE
        //set work group
```

```
            aegWrAE1(AE_GRID, workg);
            //start kernel
            copCallAE1();
            workg++;//go to next
            freeAE ^= 0x2;//mark as busy
            continue;
          }
          if(freeAE & 0x4){//use third AE
            //set work group
            aegWrAE2(AE_GRID, workg);
            //start kernel
            copCallAE2();
            workg++;//go to next
            freeAE ^= 0x4;//mark as busy
            continue;
          }
          if(freeAE & 0x8){//use fourth AE
            //set work group
            aegWrAE3(AE_GRID, workg);
            //start kernel
            copCallAE3();
            workg++;//go to next
            freeAE ^= 0x8;//mark as busy
            continue;
          }
        }
```

```
    }
  }
  else{
    //special case for small data
    //set common data
    uint64 done_flag=0;
    switch(workgroups){
      case 3:
            aegWrAE2(AE_GRID,2);
            done_flag|=0x1<<2;
            copCallAE2();
      case 2:
            aegWrAE1(AE_GRID,1);
            done_flag|=0x1<<1;
            copCallAE1();
      case 1:
            aegWrAE0(AE_GRID,0);
            done_flag|=0x1<<0;
            copCallAE0();
      case 0:;
      default:break;
    }
    freeAE = aegRd(AE_DONE);
    timeout=TIMEOUT;
    while(((aegRd(AE_DONE)&done_flag) != done_flag)
      && timeout > 0)
    {
```

```
        timeout--;
      }
    }

    return CL_SUCCESS;
}
```

Listing 3: Convey specific definition for clEnqueueNDRangeKernel function