Enabling Development of OpenCL Applications on FPGA platforms

Kavya Shagrithaya
Bradley Dept. of ECE
Virginia Tech
Blacksburg, VA, USA

Krzysztof Kępa
Bradley Dept. of ECE
Virginia Tech
Blacksburg, VA, USA
kepa@vt.edu

Peter Athanas
Bradley Dept. of ECE
Virginia Tech
Blacksburg, VA, USA
athanas@vt.edu

High-level FPGA synthesis tools aim 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. In this paper, an automated compilation flow to generate customized application-specific hardware descriptions from OpenCL computation kernels is reported. The flow uses Xilinx AutoESL tool to obtain the design specification for OpenCL kernel cores. The provided architecture integrates generated cores with memory and OpenCL host application interfaces. The host program in the OpenCL 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.

Keywords— source-to-source translation; OpenCL; AutoESL; Vivado; FPGA; Convey; HPC

I. INTRODUCTION

An increasing number of applications 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 possesses 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 economically viable high productivity systems [4]. DARPA proposed a "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.

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. Chase et al implemented a tensor-based real time optical flow algorithm on FPGA and GPU platforms [31]. 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. Comparable results were achieved for performance on the FPGA and the GPU, while the power consumption was significantly lower on the FPGA. However, the FPGA implementation took more than 12x longer to develop, as compared to the GPU implementation.

There has been significant research and industry related work in providing high level programming solutions for FPGA to reduce these design efforts. None of it yet 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 provides a standard framework that enhances portability across devices essentially rendering it platform independent. It enables the developer to focus on the algorithm while abstracting the implementation details. Being

able to develop applications in OpenCL for FPGA platforms would result in faster implementation times and time-to-market.

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 work reports 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. Also, existing tools are leveraged as required. 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.

The rest of the paper is organized as follows. Section 2 presents a background on high-level synthesis for FPGA platforms and overviews the OpenCL specifications. Section 3 introduces the source-to-source translation problem and details the compilation processes. The mapping of the execution model on the target hardware is also discussed. The results obtained in this work and comparisons with other implementations are discussed in Section 4. Section 5 concludes the paper and proposes future work.

II. Ease of Use

In this section, the existing high-level synthesis technologies for addressing FPGA productivity concerns are discussed. The OpenCL architecture and programming language are introduced.

A. 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 the general purpose processor. FPGAs 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.

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. Table 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.

TABLE I. HIGH-LEVEL SYNTHESIS TOOLS AND THEIR SOURCE INPUTS

Source Inputs	Table Column Head
C/C++	C-to-Verilog, Impulse C, Handel C, Xilinx Vivado(AutoESL), Mitrion C, FPGAC, Synphony C, ROCCC, LegUp
System C	Bluespec, Xilinx Vivado(AutoESL)
Java	JHDL, MaxCompiler
Schematics	Altium Designer, LabViewFPGA, Simulink
Python	MyHDL
C#	Kiwi

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 heterogeneous 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 hardware design concepts 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.

B. *Overview of OpenCL*

OpenCL is a 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].

OpenCL ideas are described using four models, which define platform, execution model, memory access and programming model.

1) 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 1. 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.

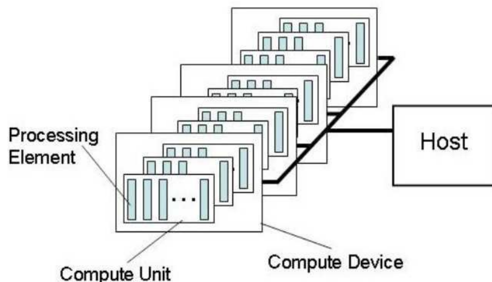

Fig. 1. The platform model for OpenCL architecture [12]

2) Execution Model

The kernel code in an OpenCL application contains the core computational part of the implementation. Kernel execution is mapped onto a *N*-dimensional index space where $1<N<4$. The size of this index space, is known as the *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 executes the same code but the execution flow of the code and the data operated on can vary. Work-items are organized in work-groups and 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.

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.

3) Memory Model

Figure 2 illustrates OpenCL memory regions and their relation to the execution model. Kernel instances (work-items) have R/W access to four distinct memory regions, i.e. Global Memory, Constant Memory, Local Memory and Private Memory.

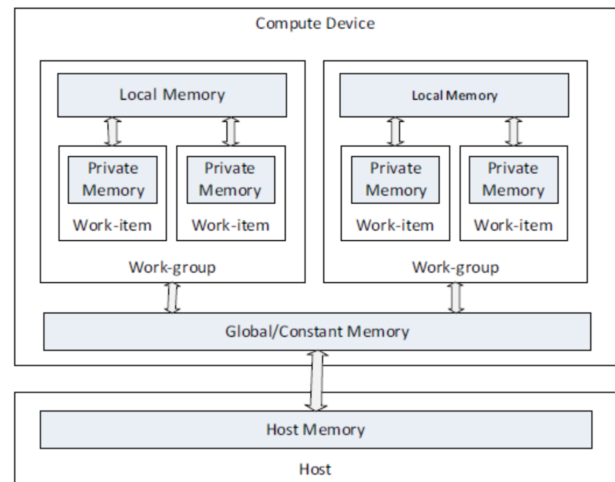

Fig. 2. OpenCL memory regions and their relation to the execution model

4) Programming Model

The OpenCL 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*. 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.

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, e.g. address space qualifiers, synchronization objects, built-in functions etc.

C. Related Work

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 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 Compute Unified Device Architecture (CUDA) [16. 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 framework 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.

Silicon-OpenCL (SOpenCL) [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 distributed control logic, a parameterized data path 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 Low Density Parity Check (LDPC) 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. Results have shown 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 project 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.

D. Summary

This section 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 including the related work in the field of OpenCL for FPGAs.

III. APPROACH AND IMPLEMENTATION

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

A. Approach and introduction to AutoESL

The OpenCL application exists in two parts – an OpenCL C kernel that define the algorithm 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].

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 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 data path 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.

B. *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. 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.

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 AST graph 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 and can be used by various tools for either 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.

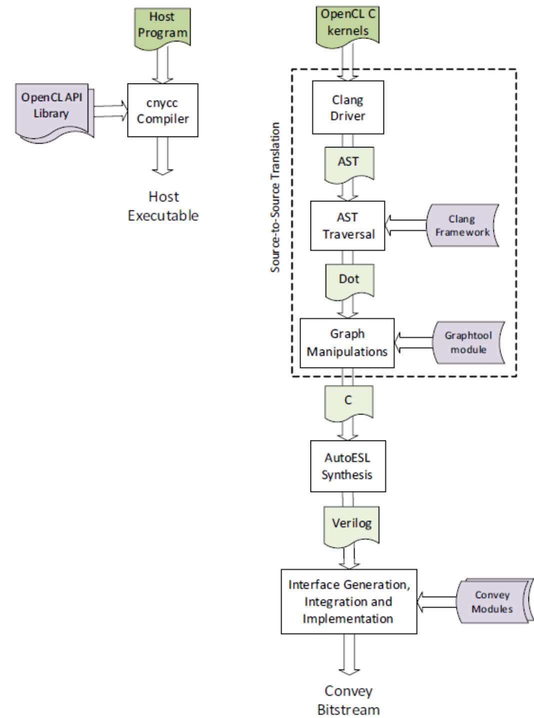

Fig. 3. Overview of the design flow

Figure 4 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 5. The *CompoundStmt* node indicates the beginning of a function body or the body of a statement.

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 are 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. 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.

```
__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];}
```

Fig. 4. OpenCL C file with the kernel function for vector addition

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.

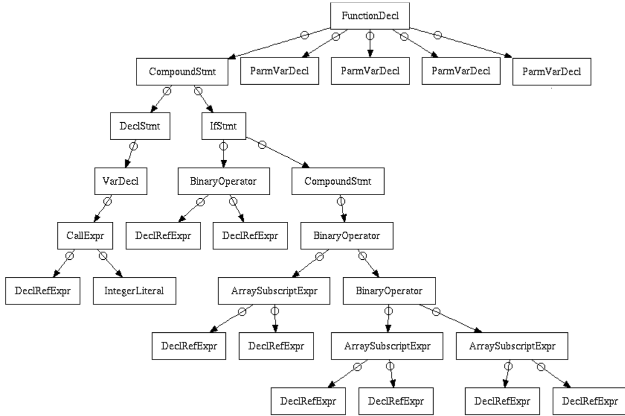

Fig. 5. Dot format visualization for the vector addition kernel

The resulting graph after performing all required transformations is reparsed to generate an AutoESL C code shown in Figure 6.

2) AutoESL Synthesis

AutoESL synthesizes the C program from the translator to generate customized FSM with data path RTL implementation that corresponds to a single processing element (PE) onto which work-items are mapped (see Figure 7). 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 IO ports for the kernel 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) 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.

```
#include "VectorAdd.h"
void VectorAdd(
    const long * a,
    const long * b,
    long * c,
    int iNumElements,
    int get_global_id_0){
#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 = get_global_id_0;
    if ( tGID < iNumElements )
       c[tGID] = a[tGID] + b[tGID];}
```

Fig. 6. AutoESL C output generated by the translator

The Convey coprocessor is considered as an OpenCL compute device with each of the AEs corresponding to a compute unit as shown in Figure 7. 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 up to sixteen parallel memory accesses. The generation of the interface modules and their integration are automated and does not involve any user intervention. The final design is implemented using Xilinx ISE tools to generate the bitstream for the compute FPGAs on Convey.

The 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.

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 and 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. In the current implementation the number of physical cores is same the work-group size.

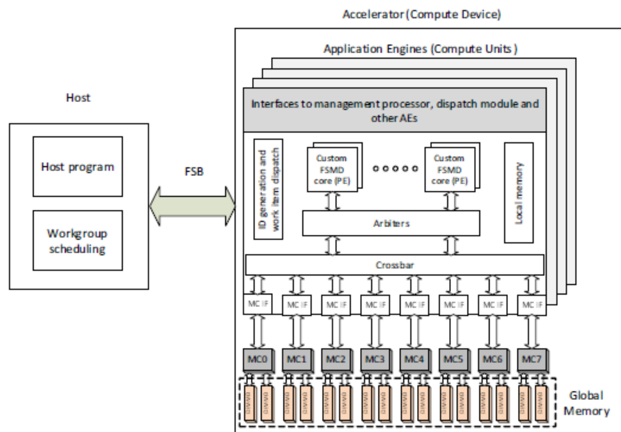

Fig. 7. System architecture on Convey

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.

C. *Summary*

In this section 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.

IV. 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 section, a simple vector addition application is studied. The performance and resource utilization numbers for different input parameters are presented and explained.

A. *Case Study: Vector Addition*

The execution flow for the host program is shown in Figure 8. Initialization includes allocation of host memory for the input vectors and assigning initial values. 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 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.

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.

1) *Performance and Resource Utilization*

Table 2 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.

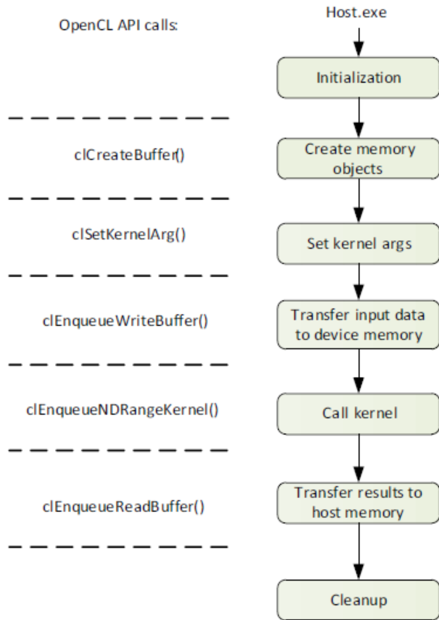

Fig. 8. Execution flow for the vector addition application

TABLE II. PERFORMANCE RESULTS FOR VECTOR ADDITION APPLICATION FOR VECTORS OF SIZE 1024

	Convey Example	OpenCL implementations		
		64 cores	*132 cores*	*192 cores*
Execution time (in ms) /w memory transfer	0.064	1.385	0.67	0.463
Execution time (in ms) /wo memory transfer		1.677	0.968	0.755

The bitstreams for the OpenCL accelerator devices are generated for three different values 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 is 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 Figure 9.

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 3.

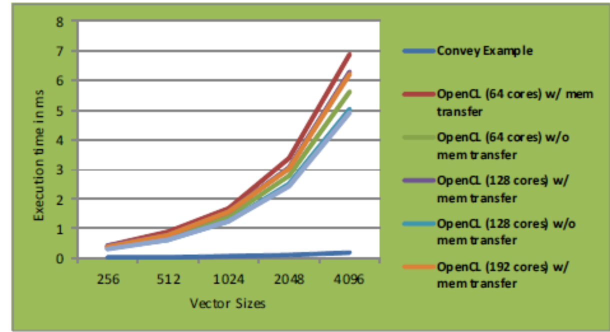

Fig. 9. Performance results for different vector sizes

TABLE III. AREA ESTIMATES FOR A SINGLE CORE FROM AUTOESL REPORT

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

The total device utilization for each of the implementations is shown in Figure 10. The numbers represent the resources for a single compute FPGA and are expressed as a percentage 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.

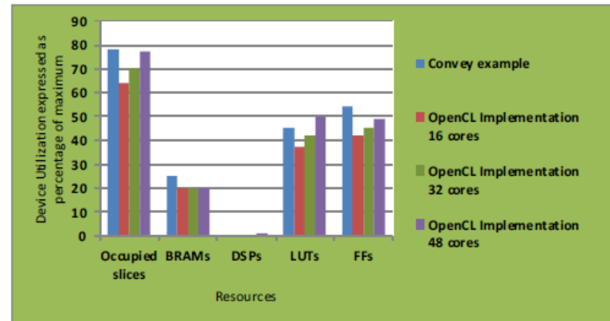

Fig. 10. Resource Utilization for vector addition application for a single AE

B. Case Study: Matrix multiplication

The matrix multiplication computation is used in many applied math and scientific applications. The execution flow of the host program for this application is similar to the flow

discussed for vector addition in the previous section. Figure 11 shows the OpenCL kernel code.

Figure 12 shows the equivalent AutoESL code. The index space for the matrix multiplication application is 2-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.

```
__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 ;}
```

Fig. 11. OpenCL C file with the kernel function for matrix multiplication

C. Comparison of Methodologies

Table 4 shows a comparison of the compilation flow and architecture presented in this work 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 data path and a streaming unit. In the flow presented in this paper, 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.

```
#include ”core_header.h”
void matrixMul(long *C, long *B, long *A, uint wA,
uint wB, int get_global_id_0, int get_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 = get_global_id_0;
    int ty = get_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;}
```

Fig. 12. AutoESL C output for matrix multiplication generated by the translator

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 project, 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 paper 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.

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 5 presents 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 this paper is specific 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.

TABLE IV. COMPARISON WITH SOPENCL AND OPENRCL IMPLEMENTATIONS

Table Head	Presented flow	OpenRCL	SOpenCL
Processing elements	Customized FSMD	Parametrized MIPS	FSMD
Fine-grain parallelism	Yes	Yes	No
Support for multiple FPGAs	Yes	No	No
Design of compiler	No	Yes	Yes
Verilog simulation	Yes	No	No

D. 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 is the ability to use different bit widths for the data. An example application that utilizes this is genome sequencing in bioinformatics. This advantage is nullified OpenCL coding, as the developer is limited to the numerical data types provided in the language.

E. Summary

This section 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 work against other related implementations.

TABLE V. COMPARISON WITH ALTERA'S OPENCL FOR FPGAS

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

V. CONCLUSIONS

High-performance applications often require high design efforts for FPGA implementations. This project aims 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 descriptions was discussed. It also presented the design and implementation of 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.

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.

• 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.

• In the system architecture, modifications can be done to the memory access patterns to improve the memory bandwidth utilization. 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.

REFERENCES

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

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

[3] Falcao, G. et al., 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] Kepner, J., HPC Productivity: An Overarching View. *International Journal of HPC Applications*. vol. 18, 2004, pp. 393–397.

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

[6] Impulse Accelerated Tech. Impulse CoDeveloper C-to-FPGA Tools. http://www.impulseaccelerated.com/ products universal.htm.

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

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

[9] Calypto, Catapult, http://calypto.com.

[10] Mohl, S. The Mitrion-C Programming Language. Mitrionics Inc., 2005.

[11] Rotem, N. C to Verilog. http://www.c-to-verilog.com/.

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

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

[14] Altera Corp. Implementing FPGA Design with the OpenCL Standard. Whitepaper, 2011.

[15] Altera Corp. Altera's OpenCL for FPGAs Program Delivers Dramatic Reductions in Development Times for Early Customers. www.altera.com/corporate/newsroom/releases/2012/products/nr-opencl-gohdr.html.

[16] Nvidia. CUDA. http://www.nvidia.com/object/cuda home new.html.

[17] Zhang, Z. et al. AutoPilot: A Platform-Based ESL Synthesis System. in *High-Level Synthesis,* 2008.

[18] Papakonstantinou, A. et al., Multilevel Granularity Parallelism Synthesis on FPGAs. in *Proc. of FCCM*, 2011.

[19] Lin, M. Lebedev, I. and Wawrzynek, J. OpenRCL: Low-Power High-Performance Computing with Reconfigurable Devices," in *Proc of FPL*. 2010.

[20] Lattner, C. and Adve, V., LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. in Proc. of the ISPCGaO, 2004.

[21] Lebedev, I. et al., MARC: A Many-Core Approach to Reconfigurable Computing. in Proc. of ICRC, 2010.

[22] Owaid, M. Bellas, N. Daloukas, K. and Antonopoulos, C. D. Synthesis of Platform Architectures from OpenCL Programs. in *Proc. of FCCM*., 2011.

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

[24] Convey, "Conveys hybrid-core technology: the HC-1 and the HC-1ex." http://www.conveycomputer.com/Resources/Convey_HC1_ Family.pdf.

[25] LLVM, llvm.org/Users.html.

[26] Quinlan, D. ROSE: Compiler Support for Object Oriented Frameworks. in Proc. of CPC, 2000.

[27] Dave, C. et al., Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Journal Computer*. Vol 42, Issue 12, 2009.

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

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

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

[31] Chase, J. Nelson, B. Bodily, J. Wei, Z. and Lee, D.-J. Real-Time Optical Flow Calculations on FPGA and GPU Architectures: A Comparison Study. in *Proc. of FCCM*, 2008.