FROM OPENCL TO HIGH-PERFORMANCE HARDWARE ON FPGAS

Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman,
Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras and Deshanand P. Singh
Altera Corporation, Toronto Technology Centre
151 Bloor Street West, Suite 200
email:(tczajkow, uaydonat, ddenisen, jfreeman, mkinsner, dneto, jawong, pyiannac, dsingh)@altera.com

ABSTRACT

We present an OpenCL compilation framework to generate high-performance hardware for FPGAs. For an OpenCL application comprising a host program and a set of kernels, it compiles the host program, generates Verilog HDL for each kernel, compiles the circuit using Altera Complete Design Suite 12.0, and downloads the compiled design onto an FPGA. We can then run the application by executing the host program on a Windows(tm)-based machine, which communicates with kernels on an FPGA using a PCIe interface.

We implement four applications on an Altera Stratix IV and present the throughput and area results for each application. We show that we can achieve a clock frequency in excess of 160MHz on our benchmarks, and that OpenCL computing paradigm is a viable design entry method for high-performance computing applications on FPGAs.

1. INTRODUCTION

Field-Programmable Gate Array (FPGA) devices are some of the largest integrated circuits available on the market (3.9B transistors in the largest Altera Stratix V FPGA), and have become a solution of choice for many high-performance applications (medical imaging [1], molecular dynamics [2]). Historically, achieving high-performance required a designer to describe a circuit using Verilog or VHDL, which can have a significant impact on time-to-market. Over the last 20 years, researchers have been investigating the use High-Level Synthesis (HLS) approaches to produce logic circuits and avoiding the use of Verilog or VHDL when possible.

Traditional HLS tools convert a software-like description to a circuit comprising a datapath and a control logic. Parallelism is achieved through scheduling independent instructions in the same clock cycle; however, this is not the best use of an FPGA. It is possible many such circuits are needed to simultaneously process data to increase the throughput of an application. Such concurrency is enabled by pipelining; however, pipelining is not explicit in programming languages such as C, and thus it is not always fully taken advantage of. Also, traditional HLS approaches produce a circuit for a small portion of an application, often leaving a designer with an arduous task of putting together a complete system.

Describing an application in OpenCL [3] addresses the aforementioned problems. An OpenCL application comprises a host and kernels, where the host sets up computation and kernels execute it. Because of the explicit kernel declaration, and that each set of elements processed are known to be independent, each kernel can be implemented as a high-performance hardware circuit. Finally, OpenCL defines the behaviour of the entire system.

We present an OpenCL compilation framework comprising: a kernel compiler, a host library, and system integration. The kernel compiler, based on an open-source LLVM compiler infrastructure [4], implements kernels from an OpenCL description into hardware circuits. The host library enables remote access and control for kernels and abstracts the communication between the host and the kernels. Finally, we put the kernels together on an FPGA and surround them with memory and host interfaces. We then compile it using ACDS 12.0 to produce a programming file that can be downloaded onto an FPGA and run the OpenCL application by executing the host program.

2. BACKGROUND AND RELATED WORK

Open Computing Language (OpenCL) was defined to enable acceleration of parallel computing, targeting a wide variety of platforms [3]. An OpenCL application comprises two parts: host and kernel. The *host* program is responsible setting up data for processing. It can launch a set of threads on a *kernel*, which represents computation to be performed by each thread. The execution of a thread usually comprises reading arguments, loading data from global memory, processing it, and storing the results in global memory. Results are computed for a group of threads at a time. Threads may be grouped into workgroups to allow data sharing. Further details about capabilities of OpenCL are described in [3].

A recent work [5] presented an OpenCL-to-Silicon framework. At a high-level, it uses an FSM with datapath model to create a logic structure suitable to execution of an OpenCL kernel. It is used together with a streaming unit to allow faster access to global data. In [6], an OpenCL-based flow using Transport-Triggered Architectures, where a VLIW-style processor implements instruction-level parallelism. In

978-1-4673-2256-0/12/$31.00

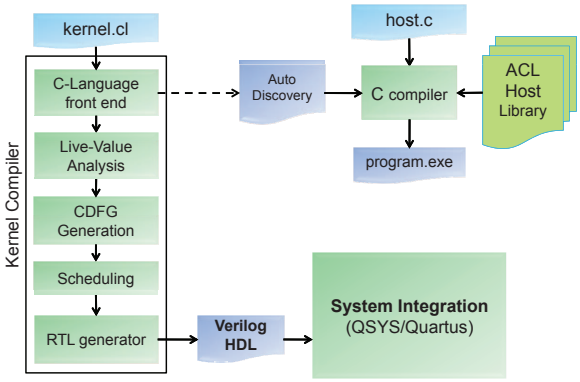

Fig. 1. OpenCL-to-FPGA framework.

these works, the underlying architecture is still a processor that shares functional units and a register file, creating an inherently inefficient design. A recent tool, FCUDA [7], transforms a CUDA program into a C program suited for processing by AutoPilot (developed by AutoESL) tool. At its core, the design is based on an FSM and datapath design. The circuit created by this tool comprises many cores each of which is essentially designed for a single thread.

In our approach, we create pipelined circuits to implement kernels to achieve high performance. When a thread enters a kernel it is followed by another thread, effectively allowing each pipeline stage to function as a separate core.

3. COMPILATION FRAMEWORK

Figure 1 presents the flow of our compilation framework. The input is an OpenCL application comprising a set of kernels (.cl files) and a host program (.c file). The kernels are compiled into a hardware circuit, starting with a C-language parser thath produces an intermediate representation for each kernel. The intermediate representation (LLVM IR) is in the form of instructions and dependencies between them. This representation is then optimized to target an FPGA platform. An optimized LLVM IR is then converted into a Control-Data Flow Graph (CDFG), which can be optimized to improve area and performance of the system, prior to RTL generation that produces a Verilog HDL for a kernel.

The compiled kernels are instantiated in a system with interfaces to the host and off-chip memory. The host interface allows the host program to access each kernel to specify workspace parameters and kernel arguments. The off-chip memory serves as global memory for an OpenCL kernel. This memory can also be accessed via the host interface, allowing the host program to set data for kernels to process and retrieve computation results. The complete system can then be synthesized, placed and routed on an FPGA.

Finally, we compile the host program using a C/C++ compiler. There are two elements in the compilation of the host program. One is the Altera OpenCL Host Library, which implements OpenCL function calls that allow the host

```
__kernel void triangle(__global int *x, __global int *y)
    int i, t = get_global_id(0), sum=0;
    for (i=0; i     t; i++) sum += x[i];
    y[id] = sum;
```

Fig. 2. Example OpenCL Kernel Program

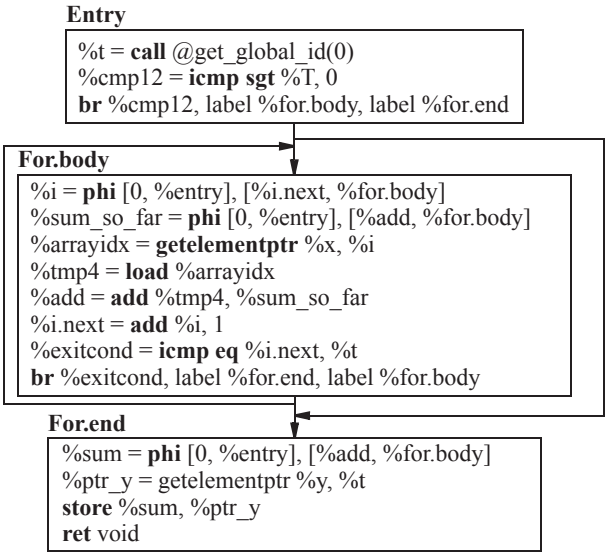

Fig. 3. Intermediate Representation Example

program to exchange information with kernels on an FPGA. The second is the Auto-Discovery module which allows a host program to detect the types of kernels on an FPGA. The Auto-Discovery module is embedded in the system by the kernel compiler, and stores the information pertaining to the kernels in a given design. Due to space limitations, we only discuss the kernel compiler in this paper.

To compile OpenCL kernels into a hardware circuit, we extended the LLVM Open-Source compiler [4] to target an FPGA platform as shown in Figure 1. The LLVM compiler represents a program as a sequence of instructions, such as load, add, subtract, store. Each instruction has associated inputs and produces a resulting value that can be used in computation downstream. A group of instructions in a contiguous sequence constitutes a *basic block*. At the end of a basic block there is always a terminal instruction that either ends the program or redirects execution to another basic block. The compiler uses this representation to create a hardware implementation of each basic block, which are then put together to form the complete kernel circuit.

The first step in the conversion of a high-level description to a hardware circuit is to produce an intermediate representation. To illustrate the data representation, consider a program in Figure 2. In this example, each thread reads its ID using the *get_global_id(0)* function and stores it in variable *t*. It then sums up all elements of array *x* beginning at the first and ending at *t-1* and stores result in array *y*.

C-Language front-end parses a kernel description and creates an LLVM Intermediate Representation (IR), which

is based on static single assignment [8]. It comprises basic blocks connected by control-flow edges as shown in Figure 3. We convert each basic block into a hardware module. To determine the data each basic block consumes and produces, we perform Live Variable Analysis. In our example, the **For.body** basic block includes all kernel arguments as well as the three kernel arguments. It then produces *y*, *t*, *i.next* and *add* as output live variables. Notice that *i.next* and *add* effectively replace *i* and *sum* when the basic block loops back to itself, allowing the loop to function correctly. Once each basic block is analyzed, we create a Control-Data Flow Graph (CDFG) to represent the operations inside it. Each basic block module takes inputs either from kernel arguments or another basic block, based on the results of Live Variable Analysis. Each basic block then processes the data according to the instructions contained within it and produces output that can be read by other basic blocks.

A basic block module consists of three types of nodes. The first node is the *merge* node, which is responsible for aggregating data from previously executed basic blocks. This ensures that for each thread, its *id* as well as all other live variables are valid when the execution of the basic block begins. In addition, in cases such as loops, the merge node facilitates *phi* instructions that take inputs from predecessor basic blocks and select the appropriate one for computation within the basic block, based on which predecessor basic block the thread arrived from.

Operational nodes represent instructions that need to execute, such as load, store, or add. They are linked by edges to other nodes to show where their inputs come from and where their outputs are used. Each operational node can be independently stalled when the successor node is unable to accept data, or not all inputs of the successor node are ready. This resembles the idea of elastic circuits [9, 10], however our implementation is much smaller and simpler because each operation has fixed latency. The last node in a basic block module is a *branch* node. It decides which of the successor basic blocks a thread should proceed to.

Once each basic block is represented as a CDFG, scheduling each node using the SDC scheduling algorithm [11]. The SDC scheduler uses a system of linear equations to schedule operations, while minimizing a cost function. A secondary objective is the reduction of area, and in particular the amount of pipeline balancing registers required. To minimize the impact on area, we minimize a cost function that reduces the number of bits required by the pipeline.

Finally, to generate a hardware circuit for a kernel we put the basic blocks together by linking the stall, valid and data signals as specified by the control edge. We then generate a wrapper around a kernel to provide a standard interface to the rest of the system. Once each kernel has been described as a hardware circuit, we create a design comprising the kernels, memories and an interface to the host platform.

Table 1. Results

Circuit Name	Fmax (MHz)	Util (%)	Num. copies	Throughput
MCBS	192.2	71%	12	2181 MSims/sec
MatMult	175	80%	1	88.4 GFLOPS
FD	163.5	55%	4	647.6 Mpoints/sec
Particles	179.2	69%	1	62 FPS

We utilize a templated design, where sections that do not change from one application to another remain locked down on an FPGA. These sections include memory interfaces and a host interface facilitated by a PCIe core. The sections that change, shown at the bottom of the figure, are attached at compile-time, synthesized placed and routed. They include the kernels as well as any on-chip memory they require.

4. EXPERIMENTAL RESULTS

To evaluate our OpenCL compiler, we implemented several OpenCL applications on a Terasic DE4 board, with the host program running on a Windows XP64-based. Each application was compiled, to generate both the host program and the kernels. The kernels were then synthesized, placed and routed using Quartus 12.0 and downloaded onto the DE4 board. We then ran the host programs to obtain performance results.

The applications we implemented are: Monte Carlo Black-Scholes (MCBS), matrix multiplication (SGEMM) , finite differences (FD), and particle simulation (Particles). MCBS is an option pricing approximation. Each simulation requires a randomly distributed number generator, a Mersenne Twister, as well as floating-point computations, including exponent, logarithm and square root functions. SGEMM is an application that exhibits easy-to-visualise parallelism. FD is an application used in oil and gas industries to analyze sensor data and detect the presence of desired natural resources. It requires a large amount of memory bandwidth to run efficiently. Particles computes the motion and collisions of particles in a cube. It is a broad test of language coverage. The area and performance results for each application are listed in Table 1.

MCBS uses a random number generator that is easy in Verilog, but more challenging in a multi-threaded environment, because each thread must obtain a specific output value from a random number sequence and in turn generate the next random number for a subsequent simulation. While it implies a dependency between threads, we break that dependency by using barriers. When we compare the circuit generated by our compiler to a hand-coded implementation [12], the compiler performed very well. We achieved a throughput of 2181 millions of simulations per second (Msims/sec) in comparison to a hand-crafted design that achieved a 1800 Msims/sec [12].

SGEMM (1024x1024 floating point) uses on-chip memory to store a part of a row and column. Each thread reads this data, performs a multiply-and-add operation and keeps track of the current sum. Once a thread finishes computing a matrix entry, it stores the result in global memory. To improve throughput, the inner loop is unrolled (64 times) and the kernel vectorized (by 4). This implementation permits a maximum theoretical throughput of 89.6 GFLOPS. We achieve 88.4 GFLOPS with some losses due to communication with the host. In comparison to [5], our compiler produces a faster circuit (16ms [5] to compute 64x64 matrix multiplication). A recent work using double-precision floating point on a Virtex5 device showed a performance of 29.8 GFLOPs [13]. Further improvement is possible when we have higher off-chip memory bandwidth to allow a wider datapath to be fully utilized.

The FD application is similar to SGEMM. It comprises floating-point multiplication and addition. However, the data access pattern is irregular, thus lower bandwidth to global memory is achieved. Also, after each iteration of a loop, more preprocessing is required than in the case of SGEMM. While an FSM-based HLS compiler would attempt to apply loop pipelining in this case, it is not a necessary step in our flow. This is because the design is fully pipelined and many threads occupy the loop body simulateneously, fully occupying the loop pipeline.

The particles benchmark requires a lot of communication between the host and the kernels, as well as a GPU in the host machine to display the resulting image. At a high-level, the application works as follows. First, the host program initializes by creating 16384 particles in a cube. The particles locations are specified and velocities set to in 3D space, and the data is copied over to the global memory on an FPGA board. The host then initiates a set of kernels in sequence to perform simulation, starting with the application of velocity and gravity to each particle. When velocity and gravity are applied, particles may find themselves colliding and a collision detection kernel is invoked to adjust their position and velocity. In this application, the constant exchange of data between the host, the FPGA and the GPU slows down the processing. The processing of a single frame takes only 9ms, while the remaining time is spent copying data from the FPGA to the GPU for rendering.

5. CONCLUSION

We presented an OpenCL compilation framework and showed that the OpenCL computing paradigm is well-suited to automatic generation of high-performance circuits for FPGAs. Our framework generated circuits that operate at a clock frequency exceeding 160MHz and has been shown to have a wide coverage of the OpenCL language, including synchronization using barriers, support for local memory, as well as floating-point operations. In addition, the framework includes a host library to communicate with kernels over PCIe interface. The applications implemented using our compiler are very different from what a GPU-based implementation would look like. While at the high-level, the system components are similar, their architecture at the low level accounts for the difference. While on GPUs each processing core performs operations in a SIMD fashion, in our architecture each thread executes a distinct operation on a distinct set of data. Our results are very encouraging and suggest that OpenCL is a viable application description paradigm for FPGAs, able to provide high-quality circuits in a much shorter time than it would take an HDL designer to produce by hand.

6. REFERENCES

[1] M. Leeser, S. Corid, E. Miller, H. Yu, and M. Trepanier, "Parallel-beam backprojection: An FPGA implementation optimized for medical imaging," *Journal of VLSI Signal Processing*, vol. 39, no. 3, pp. 295–311, 2005.

[2] H. Guo, L. Su, Y. Wang, and Z. Long, "FPGA-accelerated molecular dynamics simulations system," *Inter. Conf. on Embedded Computing*, pp. 360–365, 2009.

[3] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.1.48*, June 2009. [Online]. Available: http://www.khronos.org/registry/ cl/specs/opencl-1.0.pdf

[4] LLVM, *The LLVM Compiler Infrastructure Project*, 2010. [Online]. Available: http://www.llvm.org

[5] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos, "Synthesis of platform architectures from OpenCL programs," in *FCCM*, may 2011, pp. 186 –193.

[6] P. O. Jääskeläinen, C. S. de La Lama, P. Huerta, and J. H. Takala, "OpenCL-based design methodology for application-specific processors," in *SAMOS X: Embedded Computer Systems: Architectures, Modeling, and Simulation*, July 2010.

[7] A. Papakonstantinou, G. Karthik, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "FCUDA: Enabling efficient compilation of cuda kernels onto fpgas," in *Proc. of the 7th Symp. on ASPs*, jul 2009, pp. 35–42.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. on Prog. Lang. and Systems*, vol. 13, no. 4, pp. 451–490, Oct 1991.

[9] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *Design Automation Conference*, July 2006, pp. 657–662.

[10] J. Cortadella and M. Kishinevsky, "Synchronous elastic circuits with early evaluation and token counterflow," in *Design Automation Conference*, June 2007, pp. 416–419.

[11] J. Cong and Z. Zhang, "Activity estimation for field-programmable gate arrays," in *IEEE/ACM Design Automation Conference*, 2006, pp. 433–438.

[12] N. A. Woods, "FPGA acceleration of european options pricing," *XtremeData Inc. - White Paper*, 2008.

[13] V. Kumar, S. Joshi, S. Patkar, and H. Narayanan, "Fpga based high performance double-precision matrix multiplication," *International Journal of Parallel Programming*, vol. 38, pp. 322–338, 2010.