OpenCL for FPGAs: Prototyping a Compiler

Tomasz S. Czajkowski, David Neto, Michael Kinsner, Utku Aydonat, Jason Wong,
Dmitry Denisenko, Peter Yiannacouras, John Freeman,
Deshanand P. Singh and Stephen D. Brown

ABSTRACT

Hardware acceleration using FPGAs has shown orders of magnitude reduction in runtime of computationally-intensive applications in comparison to traditional stand-alone computers [1]. This is possible because on an FPGA many computations can be performed at the same time in a truly-parallel fashion. However, parallel computation at a hardware level requires a great deal of expertise, which limits the adoption of FPGA-based acceleration platforms.

A recent interest to enable software programmers to use GPUs for general-purpose computing has spawned an interest in developing languages for this purpose. OpenCL is one such language that enables a programmer to specify parallelism at a high level and put together an application that can take advantage of low-level hardware acceleration.

In this paper, we present a framework to support OpenCL compilation to FPGAs. We begin with two case studies that show how an OpenCL compilation could be done by hand to motivate our work. We discuss how these case studies influenced the inception of an OpenCL compiler for FPGAs. We then present the compilation flow and the results on a set of benchmarks that show the effectiveness of our automated compiler. We compare our work to prior art and show that using OpenCL as a system design language enables large scale design of high-performance computing applications.

1. INTRODUCTION

Modern FPGAs are some of the largest and most complex integrated circuits, and have become a defacto solution for many high-performance applications, such a network packet processing. They have also been successfully used to accelerate computation (medical imaging [1, 2], molecular dynamics [3]) in comparison to workstation computers.

Starting in the early 1990s through the present day, there has been an increasing interest in high-level synthesis (HLS), because describing a circuit at a high level can take a 10^{th} of the lines of code as compared to an equivalent Verilog or VHDL description [4]. If HLS tools could produce a good circuit, then they would significantly improve productivity, and allow designers to test their circuits at a higher level of abstraction, making the process much faster.

Traditional HLS tools implement a circuit from a software-like language, such as C. For a program, they implement a circuit in a single-instruction single-data (SISD) fashion, comprising a datapath that performs computation and a control circuit that schedules the flow of computation. Some parallelism is achieved through scheduling independent instructions at the same clock cycle. This approach, however, does not provide the best possible use of an FPGA. First, an FPGA can be a very large device, and it is possible many such circuits can simultaneously process data, increasing the overall throughput of a design. Second, pipelining is not explicit in programming languages such as C, and thus it is not always possible to generate a circuit that has a comparable performance to a hand-coded Verilog/VHDL design.

Recently, a new computing paradigm called OpenCL (Open Computing Language) [5] has emerged that is suitable for adoption as an FPGA design entry method and addresses the two aforementioned problems associated with HLS. In OpenCL computation is performed using a combination of a host and kernels, where the host is responsible for I/O and setup tasks, and kernels perform computation on independent inputs. Because of the explicit declaration of the kernel, and the fact that each set of elements processed are known to be independent, each kernel can be implemented as a high-performance hardware circuit. Based on the amount of space available on an FPGA, the kernel may be replicated to improve performance of the application.

In this paper, we first prototype two applications to determine how to build an automated compiler to efficiently synthesize applications from an OpenCL description into a complete system on an FPGA. After analyzing the results from case studies, we proceeded with the description of the architecture of the compiler, highlight some of its key features and present the results we obtained by using the compiler on a few key benchmark applications that we implemented on an Altera Stratix-IV based board (DE4).

2. BACKGROUND

In this section, we provide some basic background about OpenCL and our target platform, the DE4 board.

2.1 OpenCL Primer

OpenCL is a language, and a computing paradigm, to enable acceleration of parallel computing, targeting a wide variety of platforms [5]. OpenCL was developed to standardize the

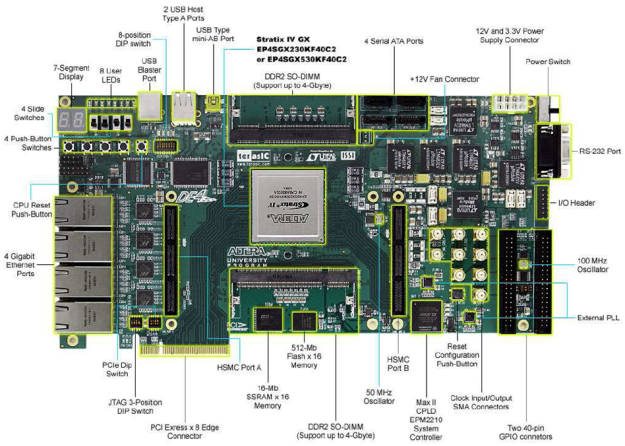

Figure 1: DE4 Research Board

method of parallel computation using GPGPUs, for which proprietary languages such as CUDA already existed.

An OpenCL application comprises a host and kernels. The *host*, is responsible for processing I/O requests and setting up data for parallel processing. When the host is ready to process data, it can launch a *kernel*, which represents a unit of computation to be performed. A kernel executes computation by loading data from global memory as specified by the host, processing it, and then storing the results back into global memory so that they can be read by the host. In OpenCL terminology a kernel and the data it is executing on comprise a *thread*. Results are computed for a group of threads at a time. Threads may be grouped into *work groups*, which allow data to be shared between the threads in a work group; however, no constraints are placed on the order of execution of threads in a work group and they are thus required to be independent.

2.2 DE4

The DE4 FPGA board, manufactured by Terasic Technologies, features an Altera Stratix IV 230/530 GX device. It provides two DDR2 memory SO-DIMM slots, four GigE ethernet ports, four SATA connections, a PCIe header, three USB ports, SD Card slot, and other peripherals. Figure 1 shows a photograph of the board.

3. INITAL CASE STUDIES

The first step towards an OpenCL-to-FPGA compiler was to study two benchmarks that could be implemented in OpenCL. These benchmarks are Black-Scholes option pricing and a Bloom filter. In both cases, we implemented each application using a host/kernel model, to generate a circuit in the OpenCL style. We found that in each case we were able to steer the final implementation towards a pipeline-oriented design, replicating the kernel hardware several times to achieve high performance.

We implemented each design to have a Nios II processor [6] to run the host program and interact with kernels, two DDR2-800 memory controllers to store data, and a set of hardware accelerators that implement the kernels in OpenCL. The host program communicates with the kernels using memory-mapped I/O to set up each kernel, launch it, and monitor its progress. The kernels run independently of the host, loading and storing data in memory using dedicated memory-mapped master interfaces and local memory buffers for temporary data storage.

3.1 Black-Scholes Option Pricing

Black-Scholes equations are used for modeling pricing of European-style options. The equations computed in this model are as follows:

$$\begin{aligned} d_1 =& \frac{\log \frac{S}{X} + T\left(R + 0.5V^2\right)}{VT^{0.5}} \\ d_2 =& d_1 - VT^{0.5} \\ \text{call} =& SN\left(d_1\right) - N\left(d_2\right) Xe^{-RT} \\ \text{put} =& Xe^{-RT}\left(1 - N\left(d_2\right)\right) - S\left(1 - N\left(d_1\right)\right) \end{aligned} \tag{1}$$

where R is the riskless rate of return, V is stock volatility, T represents option years, X represents a strike price and S the current price. The N function is an approximation of a cumulative normal distribution. The outputs *call* and *put*, represent the *call* and *put* prices. Usually, the above computation is performed for a set of options and a fixed volatility and rate of return, which requires an input of three floating-point numbers per computation.

In the OpenCL paradigm, a thread is responsible for producing one or more results. Each thread loads inputs X, S and T from memory and computes the *call* and *put* values that are then stored in the corresponding location in memory.

On a GPU, the threads are executed in a SIMD fashion on a multiprocessor. A similar level of parallelism can be achieved on an FPGA by using pipelining. A pipelined design allows a long and intensive computation to be performed over a series of clock cycles. During each clock cycle, a new set of data can be accepted for computation. Thus, if a computation pipeline has a depth of 200 cycles, results will be produced at every clock cycle after some latency.

3.1.1 An Efficient Pipelined Computation Engine

Building a hardware implementation of this kernel begins with a data-flow graph (DFG). The DFG consists of three parts. The first part performs computation of inputs to a cumulative normal distribution (CND) function approximation, the second part is the CND function itself, and the final part computes the *call* and *put* values. Each part of the computation produces data at each clock cycle, allowing the hardware to produce a new result at every clock cycle. To demonstrate how this is accomplished, we show the implementation of the first part in Figures 2 and 3.

Figure 2 shows the DFG, where circles represent single-precision floating point operations, implemented using hardware cores. The latency of each block varies, as it takes a different number of clock cycles to multiply, invert, or take a square root of a number. To balance the latencies on each path, we insert shift registers as shown in Figure 3. The resulting latency of this computational block is 53 clock cycles. When each part of the computation is similarly implemented, we put them together to create an engine that given a set of inputs produces a result 161 clock cycles later.

3.1.2 Memory Access

To attain a high performance from the above kernel implementation we need to sustain data throughput to and from

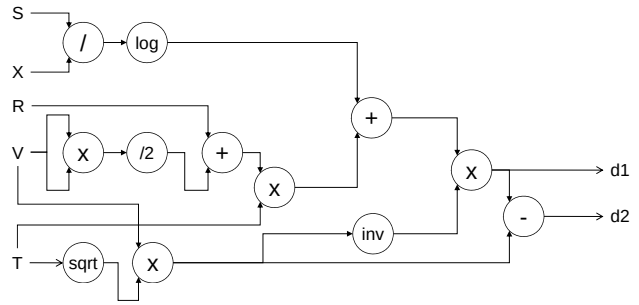

Figure 2: DFG for the first part of the computation

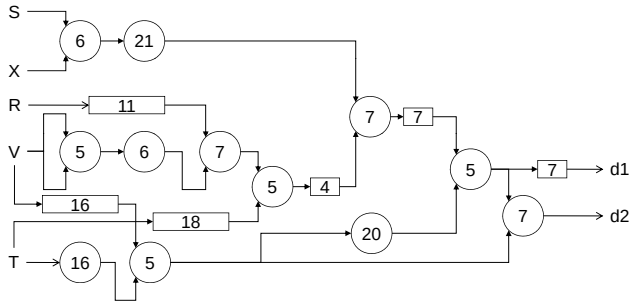

Figure 3: DFG with latency-balancing shift registers

memory. For this kernel, the data consists of three floating-point numbers as input, and two floating-point numbers as output. To facilitate the required memory bandwidth we use two separate high-speed memories (DDR2-800).

A DDR2-800 is a high-speed memory device that can be accessed via a DDR2 memory controller. The controller is capable of high-speed transfers of data in a burst fashion. It exchanges data with the external memory at a clock rate of 400 MHz, 64-bits at each edge of the clock. On the side of the FPGA, the controller facilitates a synchronous interface clocked at a rate of 200 MHz, providing 256-bits of data at the positive edge of the clock. To read and write data through this interface, a kernel needs to access data in 256-bit blocks at each edge of the clock. This is done by combining inputs and outputs into groups, such that each 256-bit read comprises several sets of inputs X, S, and T. Similarly, outputs are grouped together such that a single 256-bit write stores several *call* and *put* results.

To facilitate fast access to memory, we created bit-width adapting buffers for the kernel. The input buffer takes in 256-bit wide data and stores it in an array of eight 96-bit by n buffers, as shown in Figure 4.

3.2 Bloom Filter

In networking applications, Bloom filters [7] are used to scan the payload of a packet to help determine if a packet should be dropped or kept. For example, a packet router may wish to determine if a message it is about to forward contains unwanted data. If so, a packet can be dropped, protecting the destination computer from a potential hazard.

Bloom filters scan the payload of a packet by looking for signatures of interest. While a usual search could be slow, Bloom filters employ hash tables, where each signature is hashed using several hash functions. This allows the filter to compute a hash value for the data as it arrives, and perform a quick hash lookup to determine if a signature is of interest.

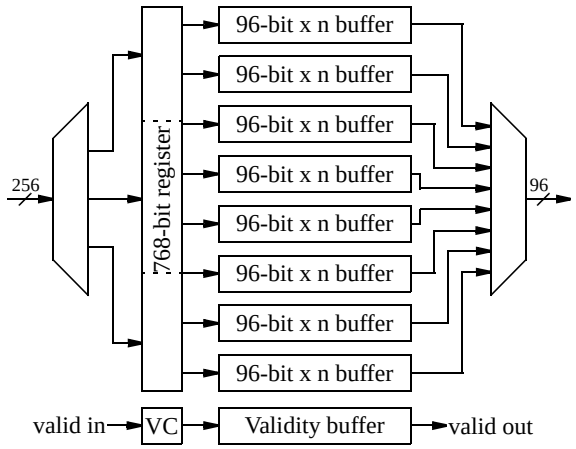

Figure 4: Bit-width adapting input Buffer

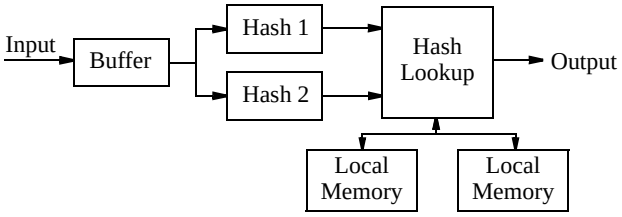

Figure 5: High-Level Diagram of a Bloom Filter

A high-level diagram of a Bloom filter is shown in Figure 5. In this design, a packet is first loaded into a buffer. The packet data is then sent through two parallel hash computation functions. Once each hash value is computed, a hash lookup is performed. If both values at addresses specified by the hash values are 1, then a Bloom filter reports it has found a signature of interest. To achieve a high throughput for this design, it may be necessary to replicate local memory that stores the hash values, as indicated in the figure.

This type of a search can be efficiently implemented on an FPGA. A recent paper [8] showed an implementation of such a filter that can sustain a throughput of approximately 3.2 Gbps. Implementation by Suresh et al. [9] provided a throughput of 18.6 Gbps on an FPGA.

Our filter is designed to scan the payload of a packet, expected to contain simple text. The text is parsed and words are identified and compared against a dictionary of approximately 150000 words. The result of the Bloom filter, is a pair of values that indicate how many words in every packet are spelled correctly and how many are not.

3.2.1 Bloom Filter Core

Pseudo-code for the Bloom filter is given in Figure 6. In this kernel, we process each packet one character at a time. For every character in a packet, we detect if it is a separator. If it is not, then we update the *hash_1* and *hash_2* values as per the else clause, updating the length of the word currently being processed. Once a separator is detected, we perform a lookup into a hash table, updating the *hit* and *miss* registers. This approach matches the high-level design presented in Figure 5.

In this kernel, the hash lookup is performed at the end of every word and hence can be executed in parallel with the rest of the kernel. However, it has to be able to stall the pipeline when the *perform_hash_lookup* function is executing

```
__kernel void bloom_filter(read_only int *d_lookup,
        unsigned int *d_result, unsigned char *d_packet_mem,
        int *d_packet_size, unsigned int packet_count) {
    for(packet_id = 0; packet_id < packet_count; packet_id++) {
        int packet_length = d_packet_size[packet_id];
        unsigned int offset = 1024*packet_id;
        unsigned int hash_1 = 0, hash_2 = 0, miss = 0, hit = 0;
        short int length = 0;
        for (index = 0; index < packet_length; index++) {
            unsigned char packet_ch = d_packet_mem[offset+index];
            char is_end_word= is_separator(packet_ch);
            int temp_hash_1 = update_hash_1(hash_1, packet_ch);
            int temp_hash_2 = update_hash_2(hash_2, packet_ch);
            if ((is_end_word) && (this_word_length > 0))
                perform_hash_lookup(d_lookup,
                        hash_1 % 2500000, hash_2 % 2500000,
                        &hit, &miss);
            hash_1 = (is_end_word) ? 0 : temp_hash_1;
            hash_2 = (is_end_word) ? 0 : temp_hash_2;
            length = (is_end_word) ? 0 : length+1;
        }
        d_result[packet_id] = (miss << 16) | hit;
    }
}
```

Figure 6: Bloom Filter kernel

and another lookup is required. Such an implementation allows a lookup to be concurrent with the processing of the next word, increasing the throughput of the circuit.

3.2.2 Shared Hash Table(s)

A key aspect of performance in this design is the hash table lookup. The lookup takes several cycles, which makes it possible for another word to be processed before the lookup is completed. In such a case, it is important to design the lookup circuitry with minimal latency. Also, to enable higher throughput, we need to be able to share the hash table between a set of kernels.

We implemented a 2.5Mbit multi-ported shared hash table using dual-ported on-chip memory. To create more than two ports for the hash table, we divided it into five segments, each implemented as a dual-ported memory with 16384 32-bit words. Each segment can simultaneously read data from two distinct memory locations, making a total of 10 ports on the hash table available for simultaneous access.

4. STUDY ANALYSIS

In this section, we discuss the performance results we obtained in our case study of two example benchmarks, in an effort to determine the key aspects that we will need to address when creating an OpenCL-to-FPGA compiler.

4.1 Area and Performance

We implemented each of the case studies on an Altera DE4 board, comprising an Altera Stratix IV 230GX device and two DDR2 memory interfaces. In each case, we began with an OpenCL description of an application, manually created a DFG for each kernel, and coded each kernel hardware module in Verilog HDL. We created a system comprising many instances of hardware kernels on which threads could be executed, as well as a Nios II processor to run the host program. Table 1 presents the clock frequency, logic utilization and throughput results for each circuit in our case study.

The Black-Scholes option pricing circuit comprised four accelerators, where floating-point cores were set for minimum latency. To obtain the throughput results, we ran the system at a clock rate of 150 MHz, allowing each instance of the accelerator to produce two results (*call* and *put*) per cycle. To exchange data with the external DDR2 memory, we connected each pair of accelerators to a single DDR2 memory controller. Because of the limited memory bandwidth in comparison to the required number of load and store operation, four accelerators fully utilized the memory bandwidth when producing 885.6 million results per second.

Table 1: Performance and Area Results

Name	Freq. (MHz)	Utilization (%)	Throughput
Black-Scholes	150	76	885.6 M results/s
Bloom Filter	150	46	25.559 Gbps

To implement the bloom filter, we instantiated 48 kernels to process packets in parallel. Each kernel was connected to DDR2-800 memory to load data packets, and used local on-chip memory for intermediate data storage. The system was clocked at a rate of 150MHz and occupied 46% of the available device resources. The resulting throughput of the circuit was 25.6 Gbps.

4.2 Observations from Case Studies

The initial case studies yielded both compelling results as well as a number of interesting challenges that need to be overcome when developing an OpenCL-to-FPGA compiler. We summarize the key results in three categories: memory access, pipelining, and hardware replication.

4.2.1 Memory Access

The first challenge we encountered was one of effectively utilizing the available memory bandwidth. This was particularly an issue with the Black-Scholes case study, as the kernel itself was able to process a lot of data very quickly. To address this challenge we introduced buffers to store input and output data. While these buffers were not explicitly defined in OpenCL, they could be inferred. With the help of input and output buffers we were able to group memory transfers into sizable batches, such that the kernel could be kept occupied 100% of the time.

One of the less obvious tasks from the point of view of the programmer was to ensure that the memory interface, in our case DDR2-800 memory, and the kernel hardware needed to have matching bus widths. This is something a usual high-level language programmer does not need to think about, as the underlying architecture does not change. In our case, the process of synthesizing a circuit from scratch requires such details to be handled. To overcome such obstacles, our compiler will have to take care of low-level implementation details, while being mindful of the external memory it is interfacing with.

The key lesson we learnt here was that our compiler will have to be able to generate efficient memory interface architecture to external RAM on a per application basis. This will be a key consideration for many applications.

4.2.2 Pipelining

The second challenge dealt with using pipelining to implement high-performance hardware for kernels. While in the case of the Black-Scholes, this can be done easily (by hand or in an automated tool), implementing a Bloom filter as a pipelined circuits was more difficult.

To implement a Bloom filter as a pipelined circuit, we had to break the kernel up into two parts: the hash computation, and the hash lookup, each of which could run independently. The hash computation was responsible for processing packet data, one character at a time, and computing a hash value for every word in the packet. When a word was found, the hash lookup was executed. At the same time another set of characters were being processed and a hash value was computed for them. If the computation of a hash value for the next word completed before the hash lookup, it was necessary for the hash lookup to stall hash computation, until the lookup was completed.

The key lesson we learnt was that a pipelined architecture for many kernels is possible, especially in an OpenCL context where explicit parallelism is defined. The pipeline would have to allow sections of the circuit to stall when a long memory access is required and thus a flexible design will be required. This is particularly important when dealing with loop constructs where it is possible for the loop to be only able to handle a limited number of threads simulteneously.

4.2.3 *Hardware Replication*

The third challenge in the implementation of both designs was the level of hardware replication needed to achieve high throughput. While in the case of Black-Scholes an entire kernel was replicated, in the case of the Bloom filter it turned out that a major bottleneck was the hash table.

We noted earlier that the hash table had to be replicated to achieve high throughput, allowing only half of the kernels (24) to use each hash table. One way to define such replication in OpenCL is to provide a kernel with several hash tables as inputs. A better approach would be to be able to model a kernel to be able to detect such problems, and replicate resources shared by kernel instances.

The lesson we learnt here was that some level of control over the architecture of the system needs to be provided, to allow the designer to analyze the performance results and then affect the architecture of the kernel hardware to improve throughput. While the ultimate goal is to be able to automatically determine the correct level of replication required for an application, a reasonable first approach is to allow user some control over the compiler output.

5. OPENCL-TO-FPGA COMPILER

In this section we present the compiler we devised based on the lessons we learnt from initial case studies.

Figure 7 presents the flow of our compilation framework, based on an LLVM compiler infrastructure [10]. 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 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 using Altera Complete Design Suite (ACDS) [11].

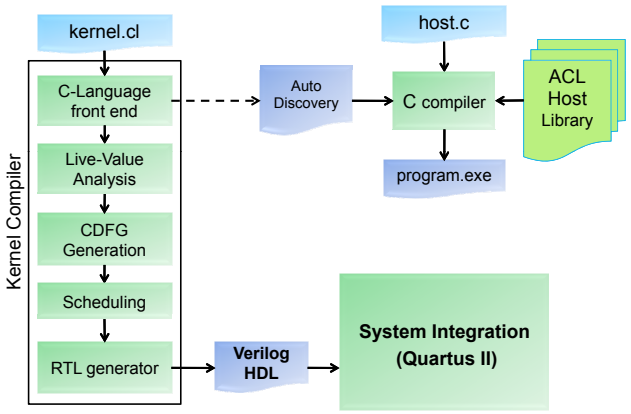

Figure 7: OpenCL-to-FPGA framework.

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 (ACL) Host Library, which implements OpenCL function calls that allow the host 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.

6. IMPLEMENTATION DETAILS

Our framework comprises a Kernel Compiler, the ACL Host Library, and System integration. The Kernel Compiler implements OpenCL kernel functionality as a circuit described in Verilog HDL and produces a description of the generated circuit for the host program in a form of an Auto-Discovery module. The C compiler then takes that description as well as the ACL Host Library and compiles the host program. The host executable as well as the Verilog HDL are then put together using a System Integration tool (Qsys) and compiled using ACDS [11].

6.1 ACL Host Library

The Altera OpenCL (ACL) host library implements most of the Platform and Runtime APIs of the OpenCL 1.0 specification [5]. The Platform APIs allow the host program to discover the FPGA accelerator device and manage execution contexts. The Runtime APIs are used to manage command queues, memory and program objects, and to discover precompiled kernels and invoke them.

The ACL Library comprises two layers: a platform independent layer that performs device independent processing, and a hardware abstraction layer that adapts to platform specifics. The platform independent layer provides the user-visible OpenCL API functions and performs generic bookkeeping and scheduling. The hardware abstraction layer pro-

```
__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;
}
```

Figure 8: Example OpenCL Kernel Program

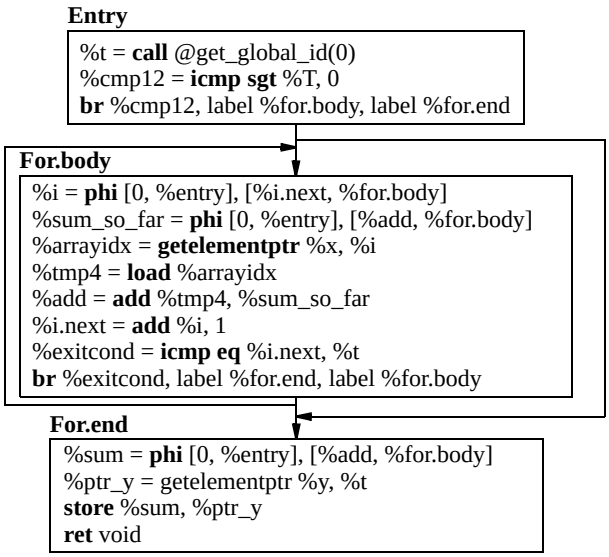

Figure 9: Intermediate Representation Example

vides low-level services to the platform independent layer. These services include: device and kernel discovery, raw memory allocation, memory transfers, kernel launch and completion. In particular, all communication between the host and the kernels goes through this layer.

6.2 Kernel Compiler

To compile OpenCL kernels into a hardware circuit, we extended the LLVM Open-Source compiler [10] to target an FPGA platform as shown in Figure 7. 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 above abstraction allows us to implement a kernel from basic block modules. Each basic block module comprises an input and an output interface with which it talks to other basic blocks. In the cases of a first and a last basic block, their interfaces have to be exposed at the top level, forming primary inputs and outputs of a kernel module.

6.2.1 C-Language Front-End

The first step in the conversion of a high-level description to a hardware circuit is to produce an intermediate representation (IR). To illustrate the IR, consider a program in Figure 8. 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*. Finally, the result is stored in array *y*.

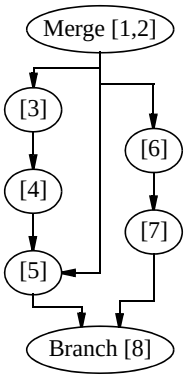

Figure 10: Basic Block Module

C-Language front-end parses a kernel description and creates an LLVM Intermediate Representation (IR), which is based on static single assignment [12]. It comprises basic blocks connected by control-flow edges as shown in Figure 9. The first basic block, **Entry**, performs initialization for the kernel and ends with a branch instruction that decides if a thread should bypass the loop. The second basic block represents the loop body and the last basic block stores the result to memory. To determine the data each basic block consumes and produces, we perform Live Variable Analysis.

6.2.2 Live Variable Analysis

Live Variable Analysis identifies variables consumed and produced by each basic block. In our example, the **Entry** basic block contains only kernel arguments as input variables (x, y). At the output of the basic block, variables *sum*, t and i are also created. This tells us that each thread produces these values when it completes execution in this basic block. The **For.body** basic block includes all kernel arguments as well as the three arguments produced by the first basic block. 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. Finally, the last basic block has input live variables y, t and *add*, while no variables are live after the return instruction.

6.2.3 CDFG Generation

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, shown in Figure 10, 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 a basic block needs 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 [13, 14], however our implementation is much smaller and simpler because each operation has fixed latency. In particular, in [14] a convergence of data flow from two operations feeding a single one required 2 FFs and 15 gates, where communication between elastic nodes requires four signals. In [13], the approach is simpler with two signals in each direction and a total of 4 gates in what they refer to as a join operation. In our implementation we use two signals and two gates. The valid signal into a node is an AND gate of its data sources (called *ready*). The stall to each predecessor node is computed as *!ready+stall_out*. The fanout splitting, or fork, logic in our implementation is also distinct from [13, 14]. In our case, each output of a node has an associated register called *consumed* that indicates if a specific successor already consumed the data being produced. If so, the register is set to 1. When all consumed register are, or are about to be set to 1, the functional unit producing a value is unstalled (its stall_in is cleared).

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.

6.2.4 Loop Handling

Loops are handled at a basic block level. A simple example of a loop is a basic block whose output is also an input to it, such as shown in Figure 9. The loop itself presents a problem in that it is entirely possible that a loop can stall. To remedy the problem, we insert an additional stage of registers into the merge node, that allows the pipeline to have an additional spot into which data can be placed.

When loops comprise many basic blocks, it is possible that stalling can occur when loop-back paths are unbalanced. In such cases, we instantiate a loop limiter that allows only specific number of threads to enter the loop. The number of threads is equal to the length of the shortest path in a loop.

6.2.5 Scheduling

Once each basic block is represented as a CDFG, scheduling is used to determine the clock cycles in which each operation is performed; however, since no sharing of resources occurs, the key contribution of scheduling is to sequence the operations into a pipeline where independent operations occur in parallel. This is important because not all instructions require the same number of clock cycles to complete. For example, an AND operation may be purely combinational, but a floating point addition may take eight cycles. Thus if possible, we would like to schedule operations that add up to 8 cycles while the adder performs computation maximizing the throughput of the hardware circuit, while reducing the area at the same time. In some cases, it may be necessary to insert pipeline balancing registers into the circuit because one execution path is longer than another.

To solve the scheduling problem we apply SDC scheduling algorithm [15]. The SDC scheduler uses a system of linear equations to schedule operations, while minimizing a cost function. In the context of scheduling, each equation represents a clock cycle relationship between connected operations. For example, in implementing an equation $f = a * b + c * d$, the scheduler has ensure that both multiplications occur before addition. 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.

6.2.6 Hardware Generation

To generate a hardware circuit for a kernel we build it out of basic block modules. To achieve high performance, we implement each module as a pipelined circuit, rather than a finite state machine with datapath (FSMD). This is because a potentially large number of threads need to execute using a kernel hardware, and their computation is largely independent. Hence, the kernel hardware should be able to execute many threads at once, rather than one at a time.

In a pipelined circuit, a new thread begins execution at each clock cycle. Thus, a basic block with pipeline depth of 100 executes 100 threads simultaneously. This is similar to replicating an FSMD circuit 100 times, except that subsequent threads execute different operations. In our design, the *valid_in-stall_out* pairs are used as a handshaking mechanism to synchronize subsequent operations.

Once each basic block is implemented, 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. In our case, we implement all load and store instructions as Avalon Memory-Mapped Master interfaces [16] that can access data from global or local memory. In addition, the wrapper keeps track of kernel execution, issuing workitems into the pipelined circuit and signals when the kernel has completed execution.

7. SYSTEM INTEGRATION

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, as shown in Figure 11. 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. This section can include many kernels, possibly replicated several times, where each kernel has a dedicated local memory segment associated with it.

8. MEMORY ORGANIZATION

OpenCL defines three types of memory spaces: global, local and private. The global memory space is designated for access by all threads. Read and write operations to this memory can be performed by any thread; however, OpenCL does not guarantee memory consistency between an arbitrary pair of threads, thus one thread may execute fully before the other one even begins running. Thus, this type of memory is usually used to store data threads will require for computation, as well as any results the threads produce.

In our implementation, the global memory space resides in

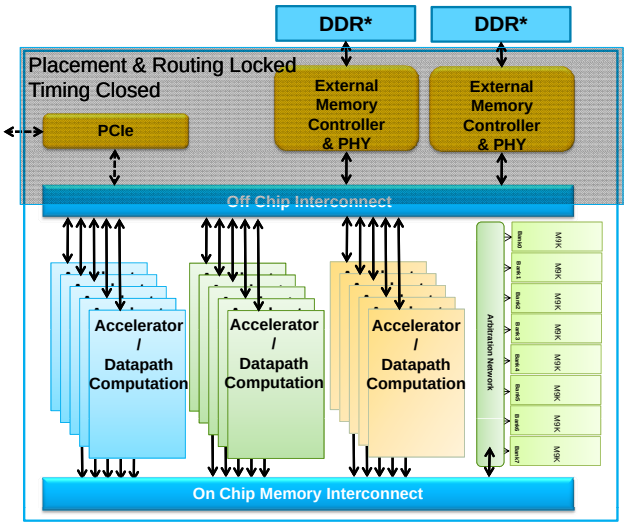

Figure 11: System Integration Details

off-chip DDR2-800 memory. It has large capacity allowing us to store data, but long access latency. Accesses to this memory are coalesced when possible to increase memory throughput. In the compiler we can detect when access to particular memory arrays are random or sequential and take advantage of this. When an array is determined to be accessed sequentially by consequtive threads, we create a special read or write module that includes a large buffer for burst transactions. This is very important, because if we know that the next set of threads will request consecutive data, we can request a large burst transaction the moment the first tread requests data. As a result we incur a small delay penalty loading/storing the data for the first thread, and no latency on the subsequent threads.

When memory accesses are random we create a coalescing unit that collects requests to see if the seemingly random memory accesses address the same 256-bit word in memory. If so, the requests are be coalesced into a single data transfer, improving memory throughput.

Local memory is used by work groups to enable synchronized exchange of data. To synchronize threads within a workgroup, barriers/memory fences are used to force threads to wait on one another before proceeding further. This allows complex algorithms that require collaboration of threads to be implemented (ex. bitonic sort).

Local memory is implemented using on-chip memory. It has short latency and multiple ports, allowing the kernel to access it efficiently. To do this we create a shared memory space for each load and store unit to any local memory. For example, if a kernel uses an array which it reads and writes a total of 4 times, then the compiler creates four memory ports for local memory. The four ports are logical (not physical), and are then mapped into a set of dual-ported on-chip memory modules. In a case where it is necessary to read more data in one cycle than the dual-port memory can provide we split memory into banks to enable faster data access, allowing for higher on-chip memory bandwidth.

Private memory is implemented with registers that store the data on a per-thread basis, and are pipelined through the kernel to ensure that each thread keeps the data it requires as it proceeds with the execution through the kernel.

9. OPENCL-SPECIFIC FEATURES

OpenCL defines features to allow synchronization of threads, such as barriers. In our framework, a barrier is a reordering First-In First-Out (FIFO) buffer. The buffer contains dedicated logic to force an entire workgroup to enter the FIFO before the first element out of a workgroup is allowed to exit. This ensures that all threads have stopped execution at a predefined location and performed all data accesses as required. This is essential, because with the exception of barriers and memory fences the OpenCL paradigm does not guarantee any ordering on thread execution. Thus, any data dependencies between threads must be guarded by a barrier.

The presence of barriers can help in many applications, especially when reordering is made possible. This is because a barrier that is followed by a global memory transation has the property that the threads exiting the barrier are in order. Thus, we can predict how they will access global memory and instantiate an appropriate memory access module as described in the previous section.

Because we use on-chip memory for barriers and local memory, it places a constraint on how many workgroups can simultaneously occupy the same kernel hardware. To ensure that this limitation is not violated, a workgroup limiter is included inside of a kernel. It permits only a specific number of workgroups to enter a kernel at any given time.

In addition to barriers, function calls to obtain global and local IDs, and other work-item intrinsics are provided. In our framework, these function calls are replaced by kernel arguments, whose value is filled by the kernel wrapper as each thread is issued.

10. EXPERIMENTAL RESULTS

To evaluate the quality of results produced by our OpenCL compiler, we implemented several OpenCL applications on a Terasic DE4 board, with the host program running on a Windows XP64-based machine. Each application was compiled to generate both the host program and the kernels. The kernels were synthesized, placed and routed using ACDS 12.0 and downloaded onto the DE4 board. We then ran the host programs to obtain performance results, and use ACDS 12.0 report files to obtain fmax and area metrics.

10.1 Benchmark Applications

The applications we implemented are: Monte Carlo Black-Scholes, matrix multiplication, finite differences, and particle simulation. The Monte Carlo Black-Scholes (MCBS) simulation is an option pricing approximation. It computes the same data as our initial case study, but it does so using a Monte-carlo approach. Each simulation in this benchmark requires a randomly distributed number generator, a Mersenne Twister, as well as floating-point computations, including exponent, logarithm and square root functions. Matrix multiplication is an application that exhibits easy-to-visualise parallelism. Finite differences (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. Particle simulation is a demo distributed with NVIDIA's OpenCL package, that simulates collisions of particles in a cube. It is a broad test of language coverage.

The area and fmax measurements for each circuit are listed

in Table 2. The first column shows the circuit name, the second column shows clock frequency of the kernel, and the remaining columns show the area of each application including the memory and host interfaces. The circuit area is specified in terms of ALMs, FFs, number of DSP blocks and memory bits, as well as an overall logic utilization metric (*%Util*) on an Altera Stratix IV-530 device. The PCIe and DDR2 interfaces are included in the resource utilization, and comprise approximately 11% of resources. The throughput of each application is summarized in the last column of Table 2.

10.2 Discussion

Each application had its unique challanges that we needed to address to obtain high-quality results.

Monte Carlo Black Scholes simulation requires a Mersenne Twister to implement the random number generator. Describing it 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, it is possible to break that dependency by using barriers.

To do this, we synchronize the data accesses such that each workgroup accesses a subset of random numbers, while generating values for the next simulation. When a simulation completes, all other threads enter a barrier and wait. Once all threads enter a barrier, they are allowed to proceed further and compute the next result. This prevents any race conditions from occuring. To speed the process up, local memory is used and as such each workgroup uses a dedicated random number generator, which is initialized when the workgroup enters a kernel.

When we compare the circuit generated by our compiler to a hand-coded implementation [17], the compiler performed very well. We achieved a throughput of 2.2 billion simulations per second (Gsims/sec) in comparison to a hand-crafted design that achieved a 1.8 Gsims/sec [17] using two FPGAs. The difference in the number of FPGAs used can be omitted here, because Stratix IV device is larger than the Stratix III FPGAs used in [17]. It is expected that the circuit designed in [17] would fit successfully providing similar performance to our compiler-generated design if the same Stratix IV device was used.

The matrix multiplication application (1024x1024 floating point) uses on-chip local memory to store a part of a row and column in local memory. 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 account for practical aspects of this application, we chose a sufficiently large matrix size such that external memory storage was required and the effects of communication between the FPGA and DDR2 memory were account for. A choice of 1024x1024 matrix size was sufficient as storing three 1024x1024 matrices requires 12 MBs of memory, which exceeds the on-chip memory capacity of the Stratix-IV device.

In this application, each thread is requires to access and process the same information other threads use. For example, when computing a column for a resulting matrix, each thread needs to access the same row of the first input matrix causing contention for memory access. To alleviate the problem, we vectorize the kernel to allow each thread to simultaneously perform a computation for four matrix entries. This increased our throughput by a factor of 4.

To utilize the high memory bandwidth, the inner loop is unrolled to perform 64 floating-point multiplications and addition simultaneously. This implementation permits a maximum throughput of 89.6 GFLOPS and we achieve 88.4 GFLOPS with some losses due to communication with the host. In comparison to [18], our compiler produces a faster circuit (16ms [18] to compute 64x64 matrix multiplication). A recent work using double-precision floating point on a Virtex5 device showed a performance of 29.8 GFLOPs [19]. A throughput of approximately 15.6, 15 and 8 GFLOPS was also reported in [20], [21] and [22] respectively. Integer matrix multiplication was implemented in [23], where 128 cores filled a Xilinx Virtex5 device. Our implementation comprises a pipeline of 500 clock cycles, still leaving a significant amount of the FPGA unused. This demonstrates that while an FSM-based approach can produce a small circuit for a single thread, over many threads a pipeline-based approach is superior.

The Finite Differences application is similar in nature to matrix multiplication in that the key operations happen in a tight loop comprising 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 the loop, more preprocessing is required than in the case of matrix multiplication. While an FSM-based HLS compiler would attempt to apply loop pipelining in this case, it is not a necessary step in our flow. The key reason for it is that the design is already pipelined, so while each thread is takes several cycles to process the loop, several threads are being processed through the same loop simultaneously. Because we have many threads reading data from memory, we can detect when a series of threads accesses sequential memory locations and optimize memory accesses. In a case of loop pipelining, we would be accessing data in strides unless it was reordered in memory.

The final application is a particle simulation. The most time consuming part of this application is collision detection. It comprises several steps: partitioning, sorting and collision computation. Partitioning divides the cube in which particles collide into smaller subcubes. Each particle in a subcube is known to only be able to collide with particles within the same subcube, or one of the 8 adjacent subcubes. The particles are then sorted based on the subcube index they belong to. Once sorted, the collision detection is engaged independently for each subcube.

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 on an FPGA takes only 9ms, while the remaining time is spent copying data from the FPGA to the GPU for rendering.

11. CONCLUSION

In this paper we presented the prototyping and development of an OpenCL-to-FPGA compiler. We showed through two case studies that the OpenCL-to-FPGA flow is not only feasible, but also effective in designing high-performance circuits. When discussing the initial case studies, we covered

Table 2: Circuit Summary and Application Throughput Results

Circuit Name	Fmax (MHz)	ALUTs	FFs	DSPs	Mem. Mbits	Util (%)	Num. copies	Throughput
MCBS	192.2	175687	261716	988	6.56	71%	12	2181 MSims/sec
MatMult	175	204894	254880	1024	5.12	80%	1	88.4 GFLOPS
FD	163.5	125662	180321	108	5.05	55%	4	647.6 Mpoints/sec
Particles	179.2	168527	206137	444	7.39	69%	1	62 FPS

major lessons we learnt from this process that helped us shape the architecture of the automated compiler. These lessons were incoporated in the framework, which we implemented and evaluated on a set of applications.

Our work shows OpenCL is well-suited to automatic generation of high-performance circuits for FPGAs. Our framework generated circuits for the benchmark suite that provide high throughput 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. Finally, we have shown the ability to automatically implement complex OpenCL applications that comprise not only an FPGA-based computation engine, but also host processing that interfaces with peripherals such as a GPU.

The circuits generated by 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. Thus in a true sense, this kernel architecture is a multiple-instruction multiple-data style design. This accounts for its size, and also high performance.

12. 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] "Medical imaging implementation using FPGAs," *Altera Corporation - White Paper*, 2010.

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

[4] K. Wakabayashi, "C-based behavioral synthesis and verification analysis on industrial design examples," *ASPDAC*, pp. 344–348, 2004.

[5] 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

[6] *Nios II Processor Reference Handbook*, Altera, Corp., San Jose, CA, 2011.

[7] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *ACM*, vol. 13, pp. 422–426, May 1970.

[8] Y.-H. Yang, H. Le, and V. Prasanna, "High performance dictionary-based string matching for deep packet inspection," in *INFOCOM*, 2010, pp. 1–5.

[9] D. Suresh, Z. Guo, B. Buyukkurt, and W. Najjar, "Automatic compilation framework for bloom filter based intrusion detection," in *Reconfigurable Computing: Architectures and Applications*, ser. Lec. Notes in CS. Springer Berlin / Heidelberg, 2006, vol. 3985, pp. 413–418.

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

[11] A. Corp., "Altera complete design suite 12.0," *http://www.altera.com*, 2012.

[12] 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.

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

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

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

[16] *Avalon Interface Specifications*, Altera, Corp., San Jose, CA, 2011.

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

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

[19] 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.

[20] Y. Dou, S. Vassiliadis, G. Kuzmanov, and G. Gaydadjiev, "64-bit floating point fpga matrix multiplication," in *In Proc. of the 13th Int. Symp. on FPGAs*, Feb 2005, pp. 86–95.

[21] L. Zhuo, S. Choi, and V. Prasanna, "Analysis of high-performance floating-point arithmetic on FPGAs," in *Parallel and Distributed Processing Symposium*, Apr 2004, p. 149.

[22] M. Smith, J. Vetter, and S. Alam, "Scientific computing beyond CPUs: FPGA implementations of common scientific kernels," in *MAPLD*, 2005.

[23] 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.