Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs

Hamid Reza Zohouri*, Naoya Maruyama†*, Aaron Smith‡,
Motohiko Matsuda† and Satoshi Matsuoka*
*Tokyo Institute of Technology, †RIKEN Advanced Institute for Computational Science, ‡Microsoft Research
Email: *{zohouri.h.aa@m, matsu@is}.titech.ac.jp, †{nmaruyama, m-matsuda}@riken.jp, ‡aaron.smith@microsoft.com

***Abstract*—We evaluate the power and performance of the Rodinia benchmark suite using the Altera SDK for OpenCL targeting a Stratix V FPGA against a modern CPU and GPU. We study multiple OpenCL kernels per benchmark, ranging from direct ports of the original GPU implementations to loop-pipelined kernels specifically optimized for FPGAs. Based on our results, we find that even though OpenCL is functionally portable across devices, direct ports of GPU-optimized code do not perform well compared to kernels optimized with FPGA-specific techniques such as sliding windows. However, by exploiting FPGA-specific optimizations, it is possible to achieve up to 3.4x better power efficiency using an Altera Stratix V FPGA in comparison to an NVIDIA K20c GPU, and better run time and power efficiency in comparison to CPU. We also present preliminary results for Arria 10, which, due to hardened FPUs, exhibits noticeably better performance compared to Stratix V in floating-point-intensive benchmarks.**

***Index Terms*—FPGA, Performance evaluation, OpenCL, Heterogeneous computing**

I. Introduction

FPGAs are a middle-ground between general-purpose processors and specialized ASICs. They offer better performance per watt in comparison to general-purpose processors for a wide range of applications but fall short of the efficiencies of specialized hardware. FPGAs' reconfigurable nature however provides the flexibility to accelerate a wider range of applications than specialized hardware. Harnessing this reconfigurability though is challenging even for specialists due to low level Hardware Description Languages (HDL) such as Verilog and VHDL which lack many of the high-level constructs of conventional software programming languages and employ a parallel/data flow model rather than a sequential one.

Attempts have been made throughout the years to make FPGAs more attractive to software programmers, such as C-to-Gates or C-to-hardware converters that translate programs written in a software programming language to HDL for use on FPGAs. This process is usually called High-level Synthesis (HLS). Examples of C-to-Gates converters are AutoESL Autopilot [1] and Synopsys Synphony C Compiler [2], which can convert C and C++ to synthesizable HDL.

To make FPGAs more attractive to software programmers, Altera [3] and Xilinx [4] have recently developed HLS toolchains based on OpenCL, a royalty-free, open source and portable programming language [5]. This approach to programming FPGAs enables the possibility of porting existing OpenCL kernels for CPUs and GPUs to FPGAs. However, there are few studies on benchmarking FPGA performance with OpenCL and on the performance portability of OpenCL kernels among such devices.

To determine the effectiveness of utilizing FPGAs in future HPC systems, we evaluate the performance of the Rodinia suite [6] compiled with the Altera SDK for OpenCL targeting a Terasic DE5-Net board equipped with an Altera Stratix V 5SGXA7 FPGA. We additionally evaluate the OpenCL kernels on a modern CPU and GPU and compare run time and performance per watt among the three devices.

In addition to evaluating the original Rodinia benchmarks, we also explore the effectiveness of FPGA-specific parallelization and optimizations. The original Rodinia implementations are meant to be used for GPU-like, highly multi-threaded processors, and even though Altera FPGAs support executing such programs by pipelining the execution of multiple threads, it is likely that such programs will perform sub-optimally due to barriers. An example of a more FPGA-friendly programming model, as recommended in the Altera programming and optimization guides [7, 8], is to pipeline loop iterations, where data dependencies across iterations can be resolved with sliding windows. The effectiveness of this second programming model depends on the target algorithm, though; algorithms that are inherently parallel and use little to no barriers may perform better with the first model initially designed for GPU-like devices.

We evaluate the possibility of using FPGAs as an HPC accelerator using six representative HPC benchmarks. We find that in the majority of cases the original multi-threaded kernels do not perform as efficiently as those optimized with FPGA-specific techniques, indicating the importance of such optimizations. Fortunately with these optimizations the FPGA achieves highly promising performance. Overall, our results using an Altera Stratix V 5SGXA7 FPGA indicate that with FPGA-specific optimizations it is possible to achieve up to 3.4x better power efficiency in comparison to an NVIDIA K20c GPU, and faster run time and much better power efficiency in comparison to CPU. More specifically we show that:

- Optimization strategies like sliding windows are very effective on FPGAs. However, such optimizations are

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00 ©2016 IEEE

implemented in a completely different manner compared to common OpenCL parallelization strategies on GPUs.
- Transforming GPU-optimized code to FPGA-optimized code yields significant speedup over running the GPU-optimized code on the FPGA. Often the FPGA-optimized versions are one to two orders of magnitude faster with much better power efficiency.
- Although for the overall execution time GPUs are still faster in most cases, FPGAs excel in energy efficiency by several factors over GPUs. This itself is already a favorable result despite that the Stratix V FPGA we used is not optimized for HPC workloads.
- We found that for some benchmarks significant manual parameter tuning is required to obtain the best performance. Although auto-tuning in such cases would be desirable, the long compilation time of FPGAs (multiple hours), coupled with high parameter sensitivity, could pose significant problems in the future. This suggests the importance of future research investigating techniques to prune the search space.
- We also present preliminary results for one compute-intensive benchmark using the new Arria 10 FPGA, which has much better support for floating-point operations. We show that Arria 10 can achieve 2x better performance compared to Stratix V and compete with modern CPUs and GPUs.

II. Background

FPGAs are reconfigurable silicon devices that consist of combinatorial logic elements interconnected by programmable networks. Modern FPGAs are equipped with dedicated hardened structures such as memory blocks, digital signal processors, DRAM, PCI Express and other I/O controllers.

Since HDLs are notoriously difficult to use for software programmers due to the lack of high-level constructs and the fact that they expose much lower details of hardware circuits, multiple different HLS tools have been created to make FPGA programming more attractive to such developers, including the Altera SDK for OpenCL. Altera's SDK [9, 10] consists of a high-level synthesis tool for compiling OpenCL kernels to Verilog HDL. It also includes a host runtime library to use FPGAs as a valid OpenCL accelerator. We describe a brief overview of the OpenCL programming abstractions and how they are implemented using reconfigurable and hardened resources of an FPGA.

A. Overview of the OpenCL Programming Model

OpenCL is an open standard that defines a programming model for heterogeneous systems using accelerators [5]. Similar to NVIDIA's CUDA API, OpenCL consists of a set of common APIs for managing accelerators at run time, and a C-based language to write programs, i.e., kernels, that are offloaded to accelerators. Unlike CUDA, the OpenCL specification does not favor any vendor or hardware type, allowing applications developed with OpenCL to also run on a multitude of devices such as FPGAs.

An OpenCL kernel represents a sequence of instructions executed by a single thread or a work-item in the OpenCL terminology. Most commonly, a large number of work-items are executed with a single kernel, by decomposing a problem into equally loaded chunks of work. Work-items are hierarchically organized with a three-level multidimensional index space called NDRange, where a set of work-items are organized into a work-group, and multiple work-groups are spawned when a kernel is launched. Note that while the multi-threading model is the most common pattern of using accelerators with OpenCL, it is not mandatory, and in fact single work-item kernels are employed as a preferred execution model in Altera's OpenCL implementation as explained below.

Among several types of memories available in OpenCL, the local and global memories are most commonly used. The former is generally smaller but faster than the latter, thus using it instead of the latter as much as possible is one of the most important optimization techniques in OpenCL kernels.

B. Altera's OpenCL Implementation

Figure 1 shows Altera's OpenCL compilation flow. In this flow, the host code and the kernel code are compiled separately since Just In Time (JIT) kernel compilation is not supported due to RTL synthesis taking multiple hours. User optimizations are directly applied to the OpenCL kernel code, then Altera's OpenCL compiler converts the kernel to LLVM IR, applies necessary optimizations, converts to Verilog and then Synthesis, Placement and Routing are performed to generate the final FPGA bitstream.

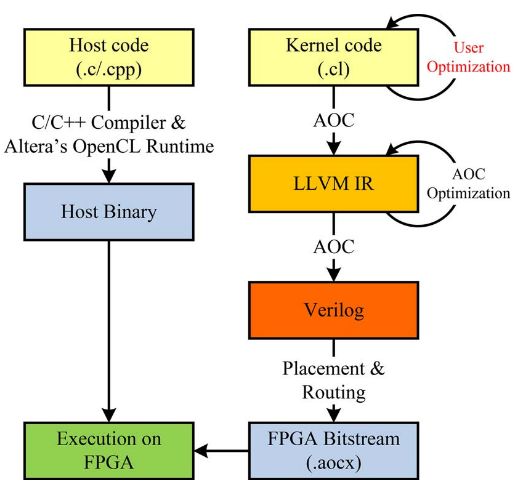

Fig. 1. Altera's OpenCL Flow, "AOC" refers to "Altera OpenCL Compiler"

Two types of parallel execution models are supported in Altera's OpenCL implementation: *explicit multi-threading* and *implicit loop pipelining*. The former employs the NDRange-based programming mechanism for explicit multi-threading, and iteratively issues each thread into the kernel pipeline, achieving pipeline parallelism across threads.

In the loop-pipelining model, the parallelization is implicit, i.e., the programmer writes sequential code in kernel functions. If a kernel function does not access any of the indices in the NDRange index space, the compiler automatically assumes that the kernel is intended to be used in a single-threaded manner, and attempts to pipeline the loops inside the kernel. Unlike explicit multi-threading, the compiler may fail to parallelize loops even when no true dependency exists, since it has to make a conservative decision with, e.g., pointer aliasing, which can be avoided with the `restrict` keyword.

The OpenCL global memory is implemented in DRAM external to the FPGA chip. Similar to discrete GPUs, typical PCI Express FPGA boards have multiple gigabytes of DRAM. OpenCL local memory, on the other hand, depending on size and access pattern of the variable, is either implemented using registers or Block RAMs available within the FPGA fabric, which provide significantly higher throughput and lower latency than the external DRAM. Local memory consistency can be ensured with the `barrier` primitive; however, it results in a highly costly pipeline flush, potentially negating the benefit of using on-chip memory blocks. Depending on the algorithm, the on-chip memory might be used more efficiently with the loop-pipelined model as explained in Section V.

C. Limitations

While FPGAs have architectural advantages compared to conventional processors, there are also some limitations that can potentially hinder adoption in realistic complex HPC applications, as discussed below.

One such constraint is that every computation physically needs its own implementation using available logic elements, thus each kernel has to be small enough to fit on a given FPGA; otherwise, it needs to be decomposed into multiple kernels that are small enough to fit. Furthermore, multiple kernels in a single application should all fit together; otherwise, the FPGA will have be dynamically reconfigured during run time which incurs additional delay in the process. While this problem is not specific to OpenCL, the adverse effect of this issue might become more noticeable when combined with OpenCL, since it results in an additional area overhead for supporting the OpenCL runtime.

Another drawback compared to other accelerators is that kernel compilation takes an extremely long time. Even a small kernel with only tens of lines of code may take hours to compile. While the functional correctness can be verified with an FPGA emulator provided with the Altera SDK for OpenCL, experimental performance tuning can be overly time-consuming unless carefully guided.

III. Methodology

To understand the performance characteristics of FPGAs as an accelerator for a wide range of HPC applications, we use the Rodinia benchmark suite as a representative set of such computations [6], and compare its performance on FPGAs against both CPUs and GPUs. Rodinia consists of twenty-one benchmarks, each of which represents one of the computational patterns compiled by [11], with implementations of two kinds of parallelism: fork-join coarse-grained parallelism based on OpenMP, and fine-grained highly-multi-threaded parallelism based on CUDA and OpenCL. The former is intended to be used on multi-core CPUs, whereas the latter is intended for GPUs. Considering the fact that the Rodinia suite is frequently used in literature and multiple optimization techniques from different publications [12–16] have been incorporated into the suite, we believe that the original benchmarks are already well optimized for CPUs and GPUs, and that comparison between our FPGA-optimized versions and the original CPU and GPU versions is a fair comparison between well-optimized benchmarks and not biased towards FPGAs.

We use the multi-threaded OpenCL version as the baseline implementation for each benchmark. We slightly modify them to use the offline compilation model since JIT compilation is not supported in Altera OpenCL, and verify that they produce correct results when running the kernels on an FPGA. Note that the kernel code is not modified in the baseline version, allowing us to understand the FPGA performance when they are merely used as a replacement of GPUs.

Since the baseline version is written for GPUs, it may not be well optimized for FPGAs, and in fact there are several optimization techniques that are unique to FPGAs as suggested in the programming and optimization guides by Altera [7, 8]. To study the effects of such optimizations, we incrementally apply them to the baseline version using a performance model as a guiding principle. Details of the performance model and optimizations are explained in Sections IV and V.

This work uses six representative benchmarks from the Rodinia suite: NW (dynamic programming), Pathfinder (dynamic programming), SRAD (structured grid), Hotspot (structured grid), LUD (dense linear algebra), and CFD (unstructured grid).[1] A brief description of each benchmark is presented in Section VI-A.

IV. Performance Model

To guide performance optimization, we build an analytical performance model of OpenCL kernels on FPGAs using the information supplied by the compiler. Our analysis models the performance of a single loop of statements that are implemented as a pipeline on an FPGA. Note that it can be considered a simple proxy even for multi-threaded kernels, since loop iterations effectively correspond to multiple threads. We first build a model with a single pipeline, and then extend it for data-parallel pipelines.

A. Single-pipeline Model

Let N_s be the number of pipeline stages encompassing a loop, L the loop trip count, and R_1 the pipeline throughput defined as the number of finished iterations per cycle. The

[1]The source code of the modified benchmarks is available at https://github.com/fpga-opencl-benchmarks.

maximum throughput is achieved when each iteration is issued to the pipeline at every clock, therefore $0 < R_1 \leq 1$. The estimated number of cycles for the loop is $N_s + L/R_1$. Thus the execution time T_1 of the loop with a single pipeline can be expressed as:

$$T_1 = (N_s + L/R_1)/F_p \tag{1}$$

where F_p is the pipeline frequency, which is typically around 200 MHz with the current generation of Altera FPGAs. Given a sequence of statements, the values of N_s and F_p are determined by the OpenCL compiler with no explicit programmer control; although, as a general rule of thumb, simpler code can lead to a smaller number of stages and higher frequency. Therefore, performance optimization should be mainly focused on improving R_1.

The pipeline throughput mainly depends on two factors: dependencies across iterations and memory access latencies. We discuss iteration dependencies in both the explicit multi-threading and implicit loop-pipelining models. In the former, where each thread corresponds to a loop iteration in the above performance model, the maximum throughput can be achieved if no synchronization primitive is used, as no inter-thread dependency exists by definition of the OpenCL memory consistency model. With a barrier synchronization, the throughput is effectively halved since the pipeline needs to be flushed once at the synchronization point. More generally, $R_1 < 1/(N_b + 1)$, where N_b represents the number of barriers.

In the implicit loop-pipelining model, since the loop is expressed as set of sequential computations, there can be arbitrary dependencies across iterations. The compiler performs standard loop dependence analysis and determines the number of stall cycles that need to be inserted between iterations. With the Altera SDK for OpenCL, this information can be quickly obtained by running the first phase of kernel compilation without synthesizing RTL. Let N_d be the number of stall cycles, $R_1 < 1/(N_d + 1)$.

We model the throughput reduction by memory access latencies as follows. Let N_m be the number of total memory access transactions at each cycle in the pipeline and W be the number of maximum transactions per cycle. We estimate N_m from the kernel source code and the underlying memory technology. We assume that the compiler does not automatically create a cache memory for DRAM accesses, but attempts to coalesce multiple memory requests if possible, as documented in the programming guides [7, 8]. Thus, every load and store instruction results in an off-chip memory transaction, potentially coalesced with other such instructions. In our model we estimate N_m as the minimum number of coalesced transactions necessary to fulfill the loads and stores. The memory transaction throughput W can be calculated from the memory performance specification as well as the pipeline frequency. The number of cycles for the memory accesses is then estimated as $\lceil N_m/W \rceil$, therefore $R_1 < 1/\lceil N_m/W \rceil$.

Putting it all together, the pipeline throughput is bounded as follows:

$$R_1 < \min\left(1/\lceil N_m/W \rceil, \left\{ \begin{array}{l} 1/(N_b+1) \\ 1/(N_d+1) \end{array} \right. \right) \tag{2}$$

Intuitively, the throughput can be improved by reducing barriers, iteration dependencies, and memory accesses. The impact of each optimization depends on the specific kernel code, the underlying FPGA, and the memory performance.

B. Extension for Data Parallelism

When the loop has no dependencies and the FPGA area has sufficient resources, it may be possible to improve the throughput by exploiting data parallelism across loop iterations. In fact, the Altera OpenCL compiler can replicate pipelines or generate SIMD-parallel pipelines upon user annotations, as explained in Section V. Let N_p be the degree of parallelism. The execution time, T_m, is modeled as:

$$T_m = (N_s + L/(R_m))/F_p \tag{3}$$

where R_m represents the total pipeline throughput, which is N_p if no overhead exists. Since the memory hardware is not replicated, unlike computation pipelines, the memory pressure is effectively increased by a factor of N_p, and the number of cycles for the memory accesses is $\lceil (N_m N_p)/W \rceil$. Thus, R_m is bounded as follows:

$$R\ _m < N_p \min\left(1/\lceil N_m N_P/W \rceil, \left\{ \begin{array}{l} 1/(N_b+1) \\ 1/(N_d+1) \end{array} \right. \right) \tag{4}$$

V. Optimizations

As indicated in the performance model, the major strategies to optimize OpenCL kernels for FPGAs are two-fold: *exploiting data parallelism* and *improving pipeline throughput*. This section presents several such techniques that can be applied using Altera's OpenCL implementation.

A. Exploiting Data Parallelism Using Replication and SIMD

Many of existing OpenCL kernels are designed to exploit data parallelism as it is efficiently supported in many-core accelerators such as GPUs. In Altera's OpenCL implementation, a multi-threaded kernel can be annotated in two ways to guide the compiler to generate data-parallel RTL.

First, the programmer can use a kernel attribute, `num_compute_units(`N`)`, to specify the number of pipelines for a kernel. At run time, the work groups are distributed across the replicated pipelines. As indicated in Equations (3) and (4), replicating the kernel pipeline potentially increases the total processing throughput by a factor of N, if there are sufficient resources and the memory bandwidth is not saturated.

Similarly, an attribute, `num_simd_work_items(`N`)`, can be used to annotate a multi-threaded kernel to generate a pipeline that processes work items in N-way SIMD parallelism. While this can fail when the kernel contains SIMD-unfriendly code such as thread-dependent branches, the pipeline throughput can be improved by a factor of N if successful. It is recommended to use the SIMD attribute if possible, rather than kernel pipeline replications, as the area overhead is generally smaller.

B. Improving Throughput by Loop Unrolling

In Equation (1), L is usually expected to be much larger than N_s and hence, having a much bigger effect than N_s in run time. Yet, since L depends on the algorithm, it is not normally possible to reduce L and gain higher throughput. When a loop is unrolled, the pipeline will become longer by the unroll factor, and N_s will be multiplied by that factor. L, on the other hand, will effectively decrease by the unroll factor since the loop step size increases and iteration count decreases. If the unroll factor is X, Equation (1) will change to Equation (5) under the presence of loop unrolling:

$$T_2 = (N_s X + L/(X R_1))/F_p \tag{5}$$

If L is large enough compared to N_s, T_2 will be smaller than T_1 and performance will increase.

C. Improving Pipeline Performance by Using Shift Register

Since most floating-point operations cannot be done in one clock on FPGAs, reduction operations in which the same variable appears on both sides of the assignment will result in an inefficient pipeline due to data dependency on that variable, and input has to be issued once every multiple clocks for correct operation (e.g. 8 clocks for floating-point addition). This will significantly reduce R from Equation (1) and result in low throughput. In such cases, a shift register with a depth equal to the input initiation interval can be inferred so that the reduction input is read from the beginning of the shift register, and output written to the end of it [7]. This effectively resolves the data dependency and results in an efficient pipeline with $R = 1$. An extra reduction on the final content of the shift register will be necessary here to obtain the final output.

D. Improving Pipeline Throughput Using Sliding Windows

As indicated in Equations (2) and (4), the pipeline throughput can be improved by reducing memory accesses, barriers in multi-threaded kernels, and iteration dependencies in loop-pipelined kernels. Common optimization techniques such as data blocking in the local memory are also effective with FPGAs; however, the throughput improvement may be limited by the barrier synchronizations necessary for the local memory consistency.

Number of memory accesses can also be reduced using sliding windows which can be used in single-threaded loop-pipelined kernels [7]. In such case, as the kernel is single-threaded, barrier synchronization is not necessary; i.e., $N_b = 0$. When a statically-sized array is used with static memory access patterns, the compiler creates a shift register using FPGA registers. The shift register can be used as a sliding window to efficiently forward data from iteration i to iteration j, $i < j$ allowing data reuse across iterations. As accesses to registers can be served without a pipeline delay, the loop can be perfectly pipelined without stalling.

To illustrate the technique, we use Rodinia NW benchmark as an example. NW computes a 2-D matrix with a neighbor data dependency on the above, left, and top left elements. The original version uses wavefront parallelization, where the inter-wave data dependency is resolved by using the local memory. This method is also applicable to FPGAs; however, it is very inefficient due to the throughput reduction caused by local memory barriers. To resolve these dependencies more efficiently, we use a sliding window and two extra registers that are implemented using FPGA registers.

For this benchmark, we split the matrix vertically, with the width of each chunk being the same as the length of the sliding window. The size of the sliding window is limited by FPGA resources. At the beginning, we load the first row of the first chunk from global memory to the sliding window, and compute the new value of the top-left index of the matrix by accessing the sliding window for the top dependency, using zero for the top-left dependency, and accessing global memory for the left dependency. Fig. 2(a) depicts this step. In the next step, we first move the current value of the first index in the sliding window to an extra *diag* register (Fig. 2(b)), which will be used to resolve the top-left dependency of the next index in the row, and then use the newly-calculated value to update both this index in the sliding window (top dependency for next row) and an extra *left* register (left dependency for the next index in the current row) (Fig. 2(c)). From here until the end of the row in this chunk, all dependencies can be resolved using the sliding window and the extra registers (Fig. 2(d)). For the left-most index of next row in the chunk, we will again need to access global memory to resolve the left and top-left dependencies, but for the rest of that row, all dependencies are resolved by accessing and updating the sliding window and extra registers. After finishing all of the chunk, the next chunk will be processed in the same manner.

VI. Experimental Settings

In this section we describe the benchmarks, hardware and software platforms, and power measurement methodology that were used in our experimental evaluation.

A. Benchmarks

a) NW: Needleman-Wunsch (NW) is a dynamic programming benchmark based on a sequence alignment algorithm. A pair of strings is organized as the top-most row and the left-most column of a 2-D matrix. The algorithm computes a score for each matrix element from the top-left position to the bottom-right position. Each score is computed based on its neighbor scores at the top, left, and top-left positions, resulting in diagonal data dependency. Wavefront parallelization is implemented in both the original OpenMP and CUDA/OpenCL versions. No floating-point arithmetic is used in this benchmark.

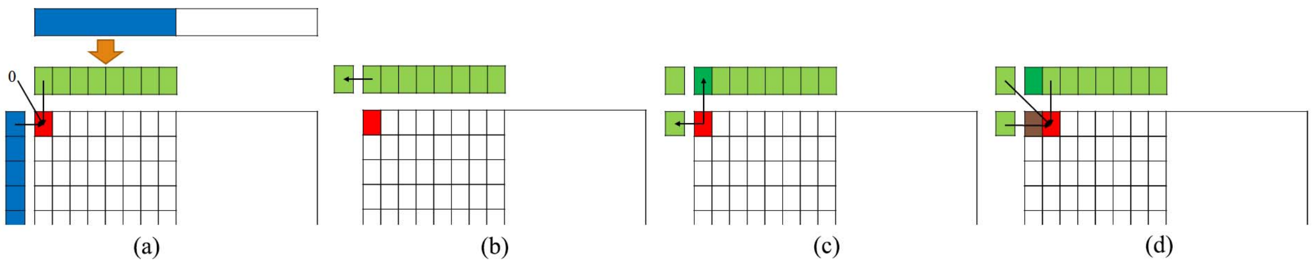

Fig. 2. Sliding window optimization technique; blue shows global memory, green shows local memory, and red shows current index in the matrix.

b) Pathfinder: Pathfinder is a dynamic programming benchmark that attempts to find a path with smallest accumulated weight from the bottom of a 2-D grid to its top. Movement direction is either straight ahead or diagonally ahead and calculation is done row by row. This benchmark has one kernel and is integer.

c) Hotspot: Hotspot is a structured grid benchmark. It simulates microprocessor temperature based on a stencil computation on 2-D structured grids and uses single-precision floating-point values. In the original CUDA version the 2-D grid is decomposed into sub-grids, each of which is computed by a thread block. The Rodinia implementation incorporates an optimization technique that saves global memory accesses by redundantly computing wider halo regions.

d) SRAD: SRAD is a structured grid benchmark that processes 2-D medical images with PDE-based diffusion kernels. Similar to Hotspot, its computation involves stencil computations with single-precision floating-point values. Unlike Hotspot, it also includes reduction of grids and dependency between iterations.

e) LUD: LU Decomposition (LUD) is a dense linear algebra benchmark that decomposes an arbitrary-sized square matrix to the product of a lower-triangular and an upper-triangular matrix. In the version we studied, the matrix indexes are single-precision floating-point numbers. This benchmark is compute-intensive with multiple instances of floating-point multiplication, addition and reduction.

f) CFD: CFD Solver (CFD) is an unstructured grid benchmark that solves 3-D Euler equations for compressible flow using the standard CSR format. The main bottleneck in unstructured grid algorithms is the indirect memory accesses, which is also the case with this benchmark. In addition, the kernel has highly intensive single-precision floating-point computations; thus, with the Stratix V FPGA which lacks hardened FPUs, performance is not expected to be as high as GPUs.

B. Software and Hardware Platform

We use a Terasic DE5-Net board that contains an Altera Stratix V 5SGXA7 FPGA and 4GB of 1600 MHz DDR3 memory in a 2-bank configuration. The FPGA has 234,720 Adaptive Logic Modules (ALMs), 938,880 registers, 2,560 M20K blocks, and 256 DSP blocks. ALM refers to Altera's unit of soft-logic elements which, in the case of Stratix V, consists of an 8-input Adaptive LUT, two embedded adders and four registers. M20K refers to 20 Kbit-sized on-chip memory blocks, and DSP refers to variable-precision arithmetic blocks that can perform arithmetic operations on up to two 27-bit inputs. We use Altera Quartus v15.1.2 and v16.0 and Terasic's Board Support Package 14.1 for this board.

For CPU performance evaluation, we use an 8-core Xeon E5-2670 with 32 GB of DDR3 memory and the OpenMP version of the benchmarks in the Rodinia suite, compiled with both GCC v5.4.0 and ICC 2016. We use 16 threads for each benchmark and report the best time between ICC and GCC. For GPU performance evaluation, we use the CUDA version of the benchmarks with CUDA v7.5 and a Tesla K20c. This specific CPU and GPU were chosen to keep comparison fair, since they are of similar age compared to the Stratix V FPGA family.

C. Power Measurement

We extract power results for FPGA by running quartus_pow on fully place-and-routed OpenCL projects. These values are an estimation of the power usage of the FPGA itself and do not reflect the power consumption of all of the FPGA board. For CPU, we used the MSR [17] driver available in most Linux distributions to extract the CPU's energy usage (in joules). For GPU, we used NVIDIA's NVML [18] library to read the GPU's power consumption (in watts) with a sampling rate of 10 ms during kernel execution. Directly comparing the GPU measurement with FPGA is, however, not completely fair, as the GPU power also includes the power usage of the on-board memory. Thus, we adjust the power consumption of the FPGA by adding an estimated power usage of its memory as well. Since we are unable to measure the actual power usage of the FPGA memory, we use its maximum power consumption as a conservative estimate. In this evaluation, we use 1.17 watts from the datasheet of a similar memory module as the one used on our FPGA board [19], and add 2.34 watts to the FPGA power consumption as it has two memory modules. Finally, energy usage (energy-to-solution) is calculated as run time (s) $\times$ power usage (watt), and is used for comparing power efficiency between the different platforms, with lower energy usage being better.

In two benchmarks, NW and Pathfinder, even with the largest possible input settings, execution finished too quickly and prevented a correct GPU power reading. In our

TABLE I
NW FPGA RESULTS

Type	Optimization Level	F_{max} (MHz)	Run Time (ms)	Power Usage (Watt)	Energy Usage (J)
Thread	None	277.23	16574.74	12.01	199.13
Loop	None	243.48	117523.09	10.60	1245.27
Thread	Basic	194.70	2445.92	16.95	41.45
Loop	Basic	249.19	116457.60	9.93	1156.77
Loop	Advanced	148.06	251.29	15.44	3.88

TABLE II
HOTSPOT FPGA RESULTS

Type	Optimization Level	F_{max} (MHz)	Run Time (s)	Power Usage (Watt)	Energy Usage (J)
Thread	None	302.48	85.22	10.99	936.52
Thread	Basic	269.69	45.14	14.44	651.97
Loop	Basic	196.19	48.02	13.70	657.67
Loop	Advanced	227.84	8.14	12.54	102.12

experience, even with very high sampling rates, measuring the power consumption of kernels that last less than 1 second usually resulted in inconsistent numbers. To overcome this issue, we wrapped the kernel of these two benchmarks in a for loop to force the benchmark to repeat the same computation multiple times, and successfully extracted average power consumption.

VII. RESULTS AND COMPARISON

In this work we focus on kernel run time and disregard host to device memory transfer. Although this can put the CPU at a disadvantage, it simplifies exploring potential performance gains using FPGAs. Run times are the average of five runs and have been extracted using the default execution settings from the Rodinia suite, unless stated otherwise. The latest version of Rodinia suite (v3.1) was used to ensure that latest CPU and GPU-based optimizations are already applied.

For the sake of brevity in Tables I to VI multi-threaded kernels have been marked as "Thread" and loop-pipelined kernels have been marked as "Loop". In all benchmarks, the multi-threaded version with no optimization is the original kernel from the Rodinia suite, and the rest of the versions are distinguished by their execution type and optimization level. Optimization level "None" shows the baseline. "Basic" shows performance that can be achieved with basic optimization techniques and a small amount of effort. "Advanced" shows performance with non-trivial FPGA-specific optimizations. Also the power usage reported in these tables does not include the FPGA DRAM power.

A. NW

Table I shows performance and power results for the NW benchmark. The length of strings is 16384 in our evaluation. The basic optimization of the multi-threaded version adds 4-way SIMD and `restrict` annotations compared to the baseline version, which we experimentally found to be most efficient among other possible optimizations for the multi-threaded version. The baseline loop-pipelined version is a straightforward sequential implementation of the algorithm. Its basic optimization adds the `restrict` keyword. The advanced optimization uses a sliding window as explained in Section V-D.

As shown in the table, the sliding window-based version performs most efficiently among all five versions. The baseline multi-threaded kernel (no optimization) performs approximately 66x slower than the fastest version (loop-pipelined with advanced optimization). This difference clearly shows the inefficiency of running GPU code on FPGAs, and the importance of using FPGA-specific optimization techniques.

B. Hotspot

Table II shows performance and power results for the Hotspot benchmark. For our evaluation, the size of computed grids is 1024x1024 and the number of iterations is 10000. For this benchmark, the multi-threaded version with basic optimization uses 4-way SIMD and the `restrict` keyword for the input parameters, which was experimentally found to be the most optimal setting, as was done for the NW case. Compared to the baseline loop-pipelined version, the version with basic optimization uses the `restrict` keyword and unrolls the inner loop, and the version with advanced optimization uses the sliding window optimization. Unlike NW, the sliding window in this benchmark is used to cache neighboring indexes loaded in one iteration, to be reused in other iterations and reduce off-chip memory access, rather than saving and forwarding partial results to subsequent iterations. Results of the baseline loop-pipelined version are not reported here due to very high (hours) run time.

Although the multi-threaded version with basic optimization performs better than the loop-pipelined version with basic optimization, the effect of the sliding window optimization, which achieves 10.5x better performance compared to baseline, is significant as shown in the final version. This shows the effectiveness of the sliding window technique for programs with structured grid computation pattern.

C. Pathfinder

Table III shows performance and power results for the Pathfinder benchmark. In this benchmark, the lengths of each row and column are 100,000 and 100, respectively. For the baseline loop-pipelined version, the `restrict` keyword is added, and the for-loop on the matrix rows is moved from host code to device code, which is not allowed in the multi-threading model due to the lack of global memory synchronization. The multi-threaded version with basic optimization, on the other hand, retains the structure of the baseline multi-threaded version, but uses the `restrict`

TABLE III
PATHFINDER FPGA RESULTS

Type	Optimization Level	F_{max} (MHz)	Run Time (ms)	Power Usage (Watt)	Energy Usage (J)
Thread	None	302.48	151.23	10.56	1.6
Loop	None	285.87	41.93	10.72	0.45
Thread	Basic	164.74	115.34	19.61	2.26
Loop	Basic	194.32	18.48	12.79	0.24
Loop	Advanced	142.49	4.57	13.72	0.06

TABLE IV
SRAD FPGA RESULTS

Type	Optimization Level	F_{max} (MHz)	Run Time (s)	Power Usage (Watt)	Energy Usage (J)
Thread	None	253.67	81.62	17.35	1416.38
Loop	None	278.7	73.99	14.79	1094.19
Thread	Basic	233.69	80.54	21.20	1707.30
Loop	Basic	207.77	12.93	19.74	255.19
Loop	Advanced	186.11	4.17	16.08	67.07

keyword, 16-way SIMD, and two replicated pipelines; further replications in this version resulted in lower performance since multiple local memory accesses were forced to share the same Block RAM port due to lack of sufficient on-chip memory. The loop-pipelined version with basic optimization uses loop unrolling both on the outer loop and the two inner loops, and the version with advanced optimization uses the sliding window technique.

As shown in Table III, the results again demonstrate the importance of the FPGA-specific optimization. Compared to the original baseline multi-threaded version, the fastest version achieves more than 33x performance improvement.

D. SRAD

Table IV shows performance and power results for the SRAD benchmark with a 4000x4000 input size and 100 iterations. In this benchmark, the multi-threaded version with basic optimization uses 2-way SIMD and `restrict`. Among the loop-pipelined kernels, the baseline implementation uses `restrict` and retains the multi-kernel structure of the original multi-threaded implementation, and the version with basic optimization uses shift register for efficient reduction and adds some unrolling.

The version with advanced optimization is created by merging all the original kernels into a single one. Specifically, the original *prepare* and *reduction* kernels are merged into one pipelined *for* loop, and the *srad* and *srad2* kernels are merged into another. With local data sharing in this new kernel, many global memory buffers are removed and global memory access is significantly reduced (nearly 10 fold). Furthermore, computation is partitioned and one sliding window is used in form of a static cache to locally address repeated reads from the same memory locations in the first pass of the computation (original *srad* kernel), similar to Hotspot. In the second pass (original *srad2* kernel), the computation direction is changed from top right, downwards, to bottom left, upwards, and another sliding window is used to efficiently resolve data dependency between iterations, similar to NW. Both windows are forced to be implemented using registers, instead of Block RAMs, using Altera's memory attributes. A global memory buffer is also used to resolve data dependency on partition boundaries.

Here, the final version achieves 19.6x speedup compared to the baseline multi-threaded implementation.

E. LUD

Table V shows performance and power results for the LUD benchmark with an input size of 8192. In this benchmark, the baseline loop-pipelined version uses the implementation called "base" provided by the benchmark author, and the version with basic optimization uses shift register for efficient reduction.

The version with advanced optimization was created by computing in a block-based manner, and temporal blocking was employed, similar to the multi-threaded kernels. Inner loop headers that depend on the iteration of the outer loop, were converted to fixed headers with if-else statements inside the loop for correct pipelining. The two innermost loops in the *internal* kernel were also fully unrolled. Still, even the last version performs only slightly faster than the baseline multi-threaded version. The loop-pipelined versions with no and basic optimization were not included in Table V due to slowness. We believe these versions cannot achieve good performance due to complex memory access pattern and unbalanced nested loop that result in inefficient pipelining.

Compared to the baseline, the multi-threaded version with basic optimization uses fixed work group sizes as outlined in Altera's documentation, and fully unrolls the loop in the internal kernel. Since this benchmark was local memory-intensive on Stratix V, in the version with advanced optimization, multiple techniques were employed to minimize the number of accesses to local buffers and the number of times these buffers are replicated to enable stall-free access. Specifically, a temporary register variable is used for reduction in the *diameter* and *perimeter* kernels instead of using the Block RAM-based local variables directly. Also to enable coalesced accesses to the buffers when unrolling is used, the local buffer in the *diameter* kernel and one of the buffers in the *perimeter* kernel were replicated manually, with one being loaded column-wise and the other row-wise. The loading direction of another buffer in the *perimeter* kernel was also changed to column-wise for the same reason. Finally, partial unrolling was used in the *diameter* and *perimeter* kernels, and multiple compute units were used for the *perimeter* and *internal* kernels to maximize resource usage. This version is DSP-bound on our Stratix V board.

Here, the fastest version (multi-threaded with advanced optimization) is 133.6x faster than the baseline.

TABLE V
LUD FPGA RESULTS

Type	Optimization Level	F_{max} (MHz)	Run Time (s)	Power Usage (Watt)	Energy Usage (J)
Thread	None	251.31	731.11	17.60	12865.38
Thread	Basic	256.20	46.37	16.61	770.07
Thread	Advanced	240.50	5.47	26.68	145.95
Loop	Advanced	216.40	517.17	20.91	10811.33

TABLE VI
CFD FPGA RESULTS

Type	Optimization Level	F_{max} (MHz)	Run Time (s)	Power Usage (Watt)	Energy Usage (J)
Thread	None	250.94	15.61	13.71	214.02
Loop	None	271.58	52.72	16.72	881.55
Thread	Basic	244.25	15.61	17.84	278.49
Loop	Basic	213.62	12.15	17.75	215.71

F. CFD

Table VI shows the performance and power results for the CFD benchmark with a default input data file with 97047 mesh points. For the multi-threaded version, only minor modifications such as annotating the input pointers with `restrict` are possible, as the remaining area resources, especially the logic elements and DSP blocks, are not enough to do any further optimizations such as pipeline replication. Due to the nature of algorithm, blocking optimization is not used in this benchmark and the sliding window optimization does not apply to it. Furthermore, since no barrier synchronization is used in the multi-threaded versions, it is expected that both the multi-threading and loop-pipelining models will perform similarly, which is in fact demonstrated as shown in the results. The baseline loop-pipelined version is created by wrapping the original multi-threaded kernel with an outer loop for iterating the index space. In the basic optimization, we solve the reduction data dependency in the inner-most loop using a shift register, as outlined in Section V, which greatly improved the performance compared to the initial loop-pipelined version. Overall, as expected, the results show that our optimization has little impact on this benchmark due to the FPGA resource limitation.

The problem of limited resources is particularly profound in this benchmark due to floating-point-heavy computations. Since the Stratix V FPGA does not have hardened FPUs, such kernels require a large amount of logic elements as well as DSP blocks. This will be alleviated with the next generation of FPGAs with hardened FPUs, thus we expect the performance to greatly improve to the point where the memory bandwidth is saturated. It should be noted that since the number of points is uniformly fixed in this implementation, typical optimizations for sparse matrices such as reordering are not necessary for this benchmark.

G. Comparison with CPU and GPU

Figure 3 shows speed-up and power efficiency of the Stratix V FPGA and GPU against CPU for all of our benchmarks.

For CFD, the FPGA version has only gone through basic optimization and is slower than CPU but more power efficient. Compared to GPU, the FPGA is quite a bit slower and also less power efficient.

Among the rest of the benchmarks for which the FPGA version has gone through advanced optimization, CPU is faster than FPGA only in Hotspot and otherwise slower. FPGA is also more power efficient than CPU in all of these benchmarks, including Hotspot, up to an order of magnitude. Comparison between GPU and FPGA among these benchmarks shows that even though FPGA is slower than GPU in every case, it is still more power efficient in all cases. The best case here is NW in which FPGA is only 48% slower but 3.4 times more power efficient. The worst case is Pathfinder in which FPGA is 5.3 times slower but is still 20% more power efficient than GPU.

These results show that despite the current Stratix V FPGAs not being optimized for HPC workloads, with proper optimization, they can achieve multiple times better power efficiency compared to state-of-the-art GPUs. With the latest Arria 10 and upcoming Stratix 10 FPGAs being more optimized towards HPC workloads, we expect such FPGAs to be much more competitive against modern GPUs.

H. Preliminary Arria 10 Results

To test the new Altera Arria 10 FPGA family that utilizes new floating-point DSPs, we used a Bittware A10PL4 board with an engineering sample of an Arria 10 GX 10AX115 FPGA. This FPGA has 427,200 ALMs, 1,708,800 registers, 54,260 M20K memory blocks, and 1,518 variable-precision DSP blocks, where each DSP block contains two 18x19 bits IEEE 754-compliant multipliers and one adder that run at 450 MHz at maximum and have a theoretical single-precision computational performance of 1.3 TFLOPS. This board also has 8 GB of 2133 MHz DDR4 memory in a 2-bank configuration.

To keep the comparison fair, we tested this FPGA against a more recent CPU and GPU, namely a 10-core Intel Xeon E5-2650 v3 (with 20 OpenMP threads) and an NVIDIA GTX 980 Ti. Our software settings and test methodology is the same as before, with the exception that since the A10PL4 board has a built-in power sensor, instead of estimating its power consumption, we periodically read the power sensor during kernel execution and calculate the average power consumption of the board, similar to our methodology for GPUs.

At the time of writing the paper, we encountered some issues with Altera's compiler for Arria 10, namely compilation failures (e.g. NW) and lower operating frequency compared to Stratix V with the exact same kernel settings. Apart from this, since Bittware's BSP for this board only supports one of the two memory banks on the board at this time, our memory-intensive benchmarks (e.g. SRAD and

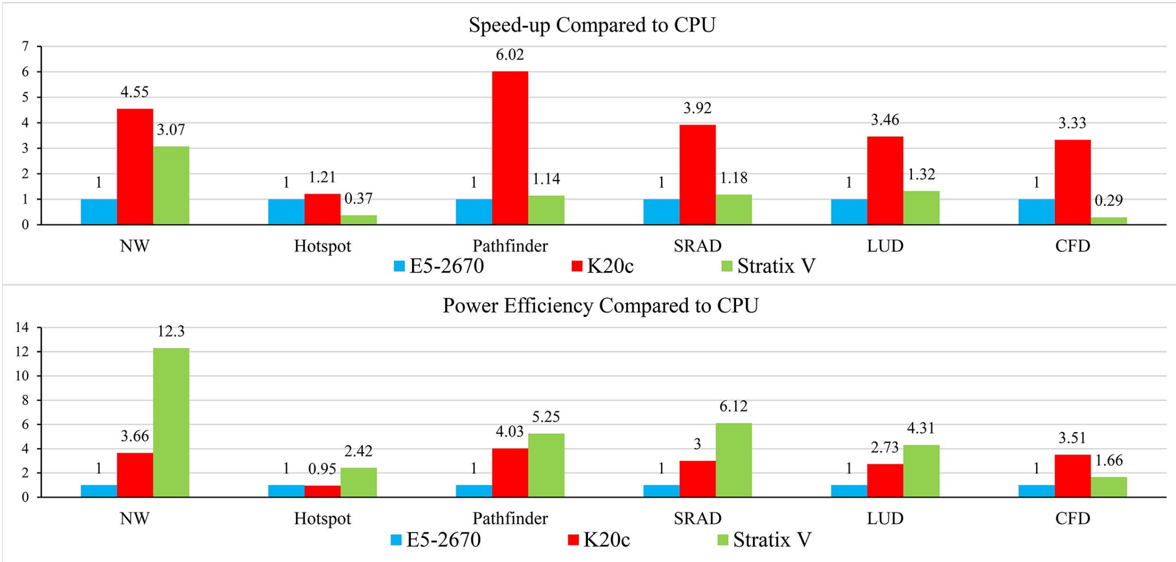

Fig. 3. Speed-up and Power Efficiency of FPGA and GPU compared to CPU, CPU bars show the normalized value of 1

TABLE VII
LUD RESULTS FOR ARRIA 10 AND NEW CPU AND GPU

Device	Run Time (ms)	Speed-up	Power Usage (Watt)	Power Efficiency
Stratix V	5470	0.97	29.02	2.64
Arria 10	2745	1.94	29.48	5.18
E5-2650 v3	5329	1	78.64	1
980 Ti	496	10.74	184.41	4.58

Hotspot) exhibit worse performance compared to the Stratix V board, due to available memory bandwidth being much lower (16 GBps vs. 23.5 GBps). The only one of our benchmarks that exhibited meaningful performance scaling on Arria 10 is the compute-intensive LUD benchmark. Table VII shows the performance and power efficiency results of both FPGA boards and new CPU and GPU for this benchmark. Speed-up and power efficiency numbers are against the E5-2650 v3 CPU.

The LUD implementation on Arria 10 uses the same code as the Stratix V implementation, but with bigger block size, higher unrolling factor in the *diameter* and *perimeter* kernels, and SIMD for the *internal* kernel. The *internal* kernel is completely memory-bound here due to very low memory bandwidth, and the other kernels cannot be unrolled any further due to lack of enough Block RAMs.

Unlike the old CPU, the new CPU can beat the older Stratix V FPGA in run time, but still not in power efficiency. It loses to the Arria 10 FPGA in both cases, though. The 980 Ti GPU is much faster than all platforms, and manages to beat the power efficiency of the older Stratix V FPGA, but it still loses to Arria 10 in this metric, despite the ultra low memory bandwidth on our Arria 10 board, which is very promising. The 2x speed-up achieved with Arria 10 compared to Stratix V clearly shows the advantage of the new DSPs. We hope that once the compiler and BSP issues are resolved, we will be able to get even better performance in LUD, and similar speed-up in other benchmarks.

VIII. RELATED WORK

One of the earliest attempts in utilizing OpenCL for FPGA-based programming was presented in [20], where the authors describe a source-to-source converter called SOpenCL (Silicon-OpenCL) that is capable of producing synthesizable HDL code from OpenCL kernels. Similarly, in [21], the authors present another source-to-source converter capable of converting CUDA programs to synthesizable code for FPGAs. Both of these papers aim at creating a platform for automatic conversion of GPU code to synthesizable code for FPGAs.

In [22], Krommydas et al. present performance evaluation using SOpenCL. Similar to our results, they used benchmarks derived from the Rodinia suite, including GEM, NW, SRAD and BFS, and evaluated their performance on a wide range of hardware including a Xilinx Virtex-6 LX760 FPGA. Unlike our work, that paper mainly used the kernels written for GPUs, and does not discuss FPGA-specific optimizations in detail, which is crucial for obtaining optimal performance on FPGAs, as shown in our work.

Since the Altera SDK for OpenCL is relatively new, there have been very few studies on its performance. Pu et al. present an implementation of the k-NN algorithm on the Terasic DE4 board with an Altera Stratix IV 4SGX530 FPGA, using the Altera SDK for OpenCL [23]. Speed and performance per watt comparison is also provided for CPU and GPU. Settle presents an implementation method of the Smith-Waterman algorithm using the Altera SDK for

OpenCL, which is similar to Needleman-Wunsch studied in our work [24]. Our optimization using sliding windows is based on the method described in this work. In [25], three image processing kernels (Canny, Sobel, and SURF) were implemented on three OpenCL-capable FPGA boards using the Altera SDK for OpenCL, and area usage, operating frequency and productivity results have been compared with HDL implementations of the same kernels on one of the boards. While the preceding results complement the results reported in this paper, our work further extends the depth and breadth of the benchmarking.

Che et al. presented a comparison of GPU, FPGA, and CPU for three benchmarks from the Rodinia benchmark suite [26]. This paper compared the number of cycles on each processor for a given algorithm when implemented with CUDA, VHDL, and OpenMP for GPU, FPGA, and CPU. Our work is complementary to theirs since we evaluated the same algorithms using OpenCL instead of HDL. However it is not straightforward to compare our results with theirs since the hardware and software platforms are significantly different.

Recently different frameworks for generating synthesizable high-level language code for FPGAs are emerging. In [27], Lee et al. present an OpenACC-based framework that can convert C code using OpenACC directives to OpenCL code compatible with Altera's compiler. Del Sozzo et al. present a preliminary framework for automating implementation of CNNs on FPGAs targeting Xilinx Vivado HLS [28]. Also in [29], Wang et al. introduce an OpenCL-based MapReduce framework for FPGAs which allows users to write familiar MapReduce interfaces in C which are then converted to OpenCL code targeting Altera's compiler.

IX. Conclusion and Future Work

In this work we presented the results of porting and optimizing a subset of the Rodinia benchmark suite to the FPGA platform, using the Altera SDK for OpenCL, and compared run time and power efficiency with an NVIDIA Tesla K20c GPU and an Intel E5-2670 CPU. We also presented a simple model that can be used to guide optimizations that are possible on this platform.

Based on our findings, even though we could not match the speed of the K20c GPU in our benchmarks, we could achieve up to 3.4x better power efficiency in comparison to this GPU. Compared to the Xeon CPU, we could beat its performance in most, and power efficiency in all benchmarks that used advanced optimization for FPGA. It is highly promising that such performance and power efficiency can be achieved on FPGAs, even with a "high-level" language such as OpenCL.

We also presented our preliminary results on the new Arria 10 FPGA from Altera that utilizes the new floating-point DSPs, for the compute-intensive LUD benchmark, and showed that despite much lower available memory bandwidth due to BSP issues, we can still achieve 2x speed-up compared to Stratix V, better performance compared to the modern Xeon E5-2560 v3 CPU, and better power efficiency compared to both this CPU and the modern NVIDIA 980 Ti GPU.

Our work is still ongoing and we will continue porting the rest of the benchmarks from the Rodinia suite for FPGAs and apply more aggressive optimizations on the benchmarks that only have gone through basic optimization so far. Automatic or guided parameter space exploration for multi-threaded kernels, which can significantly reduce optimization time for such kernels compared to exhaustive search, is a subject that needs further study. For loop-pipelined kernels, automatic implementation of optimization techniques such as sliding windows, based on the characteristics of the algorithm, can be an interesting path to extend this work. We believe that such research will be an important foundation for performance portability across different types of the current and future architectures.

Acknowledgments

We thank Altera, Altima, Elsena and Bittware for providing us generous support on Altera FPGAs as well as its software. This project was partially supported by JST, CREST through its research program: "Highly Productive, High Performance Application Frameworks for Post Petascale Computing" as well as the Microsoft Research Asia CORE program.

References

[1] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, *High-Level Synthesis: From Algorithm to Digital Circuit*. Dordrecht: Springer Netherlands, 2008, ch. AutoPilot: A Platform-Based ESL Synthesis System, pp. 99–112.

[2] Synopsys, Inc. (2014) Synopsis Synphony C Compiler Datasheet. [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Documents/synphonyc-compiler-ds.pdf

[3] Altera Corporation. Altera SDK for OpenCL. [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html

[4] Xilinx, Inc. Xilinx SDAccel. [Online]. Available: http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[5] Khronos OpenCL Working Group, "The OpenCL Specification: Version 1.0," October 2011. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[7] Altera Corporation, "Altera SDK for OpenCL: Best Practices Guide," May 2015. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/

en_US/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf

[8] ——, "Altera SDK for OpenCL Programming Guide," May 2015. [Online]. Available: https://www.altera.com/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[9] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 531–534.

[10] Altera Corporation, "Implementing FPGA Design with the OpenCL Standard," November 2013. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf

[11] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009.

[12] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron, "Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–12.

[13] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. New York, NY, USA: ACM, 2008, pp. 73–82.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. New York, NY, USA: ACM, 2008, pp. 260–269.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1370–1380, Oct. 2008.

[16] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," in *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. New York, NY, USA: ACM, 2009, pp. 256–265.

[17] (2009, March) Linux Programmer's Manual: MSR. [Online]. Available: http://man7.org/linux/man-pages/man4/msr.4.html

[18] Nvidia Corp. (2015, May) NVML API Reference Guide. [Online]. Available: http://docs.nvidia.com/deploy/pdf/NVML_API_Reference_Guide.pdf

[19] Kingston Technology. (2013, December) Kingstone KVR16S11S6/2 Memory Module Specification. [Online]. Available: http://www.kingston.com/dataSheets/KVR16S11S6_2.pdf

[20] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos, "Synthesis of platform architectures from opencl programs," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, May 2011, pp. 186–193.

[21] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "Efficient compilation of cuda kernels for high-performance computing on fpgas," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 25:1–25:26, Sep. 2013.

[22] K. Krommydas, W. chun Feng, M. Owaida, C. Antonopoulos, and N. Bellas, "On the characterization of opencl dwarfs on fixed and reconfigurable platforms," in *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, June 2014, pp. 153–160.

[23] Y. Pu, J. Peng, L. Huang, and J. Chen, "An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, May 2015, pp. 167–170.

[24] S. Settle, "High-performance dynamic programming on fpgas with opencl," in *High Performance Extreme Computing (HPEC), 2013 IEEE 17th Annual Conference on*, September 2013. [Online]. Available: http://ieee-hpec.org/2013/index_htm_files/29-High-performance-Settle-2876089.pdf

[25] K. Hill, S. Craciun, A. George, and H. Lam, "Comparative analysis of opencl vs. hdl with image-processing kernels on stratix-v fpga," in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, July 2015, pp. 189–193.

[26] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *Application Specific Processors, 2008. SASP 2008. Symposium on*, June 2008, pp. 101–107.

[27] S. Lee, J. Kim, and J. S. Vetter, "Openacc to fpga: A framework for directive-based high-performance reconfigurable computing," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 544–554.

[28] E. D. Sozzo, A. Solazzo, A. Miele, and M. D. Santambrogio, "On the automation of high level synthesis of convolutional neural networks," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 217–224.

[29] Z. Wang, S. Zhang, B. He, and W. Zhang, "Melia: A mapreduce framework on opencl-based fpgas," *IEEE Transactions on Parallel and Distributed Systems*, no. 99, pp. 1–1, 2016.