Toward Evaluating High-Level Synthesis Portability and Performance between Intel and Xilinx FPGAs

Anthony M. Cabrera[1], Aaron R. Young[1], Jacob Lambert[2], Zhili Xiao[3], Amy An[3], Seyong Lee[1], Zheming Jin[1], Jungwon Kim[1], Jeremy Buhler[3], Roger D. Chamberlain[3], Jeffrey S. Vetter[1]

[1] Oak Ridge National Laboratory, Oak Ridge, TN, USA
[2] University of Oregon, Eugene, OR, USA
[3] Washington University in St. Louis, St. Louis, MO, USA
{cabreraam,youngar,lambertjb,lees2,jinz,kimj,vetter}@ornl.gov
{xiaozhili,a.an,jbuhler,roger}@wustl.edu

ABSTRACT

Offloading computation from a CPU to a hardware accelerator is becoming a more common solution for improving performance because traditional gains enabled by Moore's law and Dennard scaling have slowed. GPUs are often used as hardware accelerators, but field-programmable gate arrays (FPGAs) are gaining traction. FPGAs are beneficial because they allow hardware specific to a particular application to be created. However, they are notoriously difficult to program. To this end, two of the main FPGA manufacturers, Intel and Xilinx, have created tools and frameworks that enable the use of higher level languages to design FPGA hardware. Although Xilinx kernels can be designed by using C/C++, both Intel and Xilinx support the use of OpenCL C to architect FPGA hardware. However, not much is known about the portability and performance between these two device families other than the fact that it is theoretically possible to synthesize a kernel meant for Intel to Xilinx and vice versa.

In this work, we evaluate the portability and performance of Intel and Xilinx kernels. We use OpenCL C implementations of a subset of the Rodinia benchmarking suite that were designed for an Intel FPGA and make the necessary modifications to create synthesizable OpenCL C kernels for a Xilinx FPGA. We find that the difficulty of porting certain kernel optimizations varies, depending on the construct. Once the minimum amount of modifications is made to create synthesizable hardware for the Xilinx platform, more nontrivial work is needed to improve performance. However, we find that constructs that are known to be performant for an FPGA should improve performance regardless of the platform; the difficulty comes in deciding how to invoke certain kernel optimizations while also abiding by the constraints enforced by a given platform's hardware compiler.

CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable Computing**; • **Extreme Heterogeneity**;

KEYWORDS

FPGA, high level synthesis, Rodinia, Xilinx, hardware accelerator, performance, portability

ACM Reference Format:
Anthony M. Cabrera[1], Aaron R. Young[1], Jacob Lambert[2], Zhili Xiao[3], Amy An[3], Seyong Lee[1], Zheming Jin[1], Jungwon Kim[1], Jeremy Buhler[3], Roger D. Chamberlain[3], Jeffrey S. Vetter[1]. 2021. Toward Evaluating High-Level Synthesis Portability and Performance between Intel and Xilinx FPGAs. In *International Workshop on OpenCL (IWOCL'21), April 27–29, 2021, Munich, Germany*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3456669.3456699

1 INTRODUCTION

The use of hardware accelerators is increasing in prominence because Moore's law falls victim to the end of Dennard scaling [1]. GPUs are now found in numerous computing systems, from the majority of the 10 fastest supercomputers in the Top500 list [18] to data centers and edge and embedded computers. At the same time, a different type of hardware accelerator—the field-programmable gate array (FPGA)—has expanded from traditional embedded roles to more general-purpose computations [7].

FPGAs are a powerful tool, partly due to their ability to exploit fine-grained parallelism specific to a given computation in hardware. Because of their reconfigurable nature, FPGAs can exploit more types of application-specific parallelism than GPUs or CPUs with particular strength in pipelining dataflow computations. However, the benefits of FPGAs have historically been difficult to access because they are considered challenging to program. Specifically, the available languages for FPGA hardware synthesis, such as Verilog and VHDL, offer only a low level of abstraction at the level of logic gates. Therefore, application designers must consider low-level artifacts, such as timing specifications, when reasoning about the correctness and efficiency of their implementations.

High-level synthesis (HLS) frameworks and tools help lower the barrier to entry for FPGAs. HLS allows programmers to target FPGAs by using languages at a higher level of abstraction, such as C or C++. These languages allow programmers to focus less on the low-level details of using an FPGA and more on their kernel's functionality. The HLS tools then perform transformations and analyses that extract the parallelism inherent in a kernel and subsequently synthesize performant FPGA hardware.

Intel and Xilinx, two primary FPGA vendors, offer suites of tools for HLS. Both vendors' tools leverage OpenCL as a way to facilitate kernel execution and application management. Individual

IWOCL'21, April 27–29, 2021, Munich, Germany

ACM ISBN 978-1-4503-9033-0/21/04…$15.00
https://doi.org/10.1145/3456669.3456699

kernel designs are expressed in OpenCL C. Because of OpenCL's portability, an OpenCL kernel that is authored for an Intel FPGA should, in theory, be synthesizable for a Xilinx FPGA and vice versa. However, in practice, although many HLS-based application kernels exist for Xilinx and Intel hardware, little has been reported about the actual portability of HLS kernels between these two device families. Even if one kernel can be compiled to run correctly on both platforms, performance portability between platforms is far from guaranteed. Therefore, understanding the commonalities between Intel and Xilinx HLS tools and the quirks peculiar to each is a worthwhile topic for investigation. If performance portability between these FPGA families can be achieved, then it would enable application designers to confidently author their kernels once in OpenCL C and achieve high performance with each family by using its respective HLS tools.

This work presents an initial evaluation of portability and performance of OpenCL C kernels that were originally architected for an Intel FPGA reconfigured for a Xilinx FPGA. We use the Intel FPGA implementations from Zohouri et al. [20] of the Rodinia benchmark suite [8] as a baseline and investigate the process and impact of porting these implementations to a Xilinx FPGA. Our work includes the following contributions:

(1) detailing our port of a subset of FPGA kernel optimizations from an Intel OpenCL to a Xilinx OpenCL specification,
(2) contributing to the sparse literature of using OpenCL C for Xilinx platforms,
(3) presenting our experience of using Xilinx Vitis tools with OpenCL C kernels, and
(4) evaluating Xilinx OpenCL kernel portability and performance from the ported hardware kernels.

Sections 2 and 3 describe preliminaries and related work, Section 4 presents methods, Section 5 gives results, and Section 6 concludes and explores opportunities for future work.

2 PRELIMINARIES

2.1 Vendor Tooling and Flows

Although our evaluation will focus on the Xilinx platform, we begin by briefly comparing the Intel and Xilinx HLS tools. This section is especially helpful for those who want to develop kernels for one platform but have experience only with the other platform. The Intel HLS tools are provided as part of the Intel FPGA software development kit (SDK) for OpenCL, and the Xilinx tools are provided through the Xilinx Vitis platform.

Both the Intel FPGA SDK for OpenCL and Xilinx Vitis supply a *hardware compiler*, a vendor tool[1] that facilitates:

- translating the OpenCL C specification of a kernel to a register-transfer level (RTL) equivalent;
- synthesizing the RTL description into a netlist of logic primitives, such as logic gates and registers;
- placing the resulting netlist into specific resources of the target FPGA;
- routing the used resources on the FPGA; and
- generating a bitstream to program the FPGA to execute the kernel.

[1] Actually, set of tools, but we will refer to the entire set as a tool.

The hardware compiler is responsible for abstracting the low-level details of targeting an FPGA. This work refers to the Intel and Xilinx hardware compilers by the names of their principal command-line tools: `aoc` and `v++`, respectively. The extension of a compiled FPGA bitstream for an Intel design is `.aocx` and `.xclbin` for a Xilinx design.

2.2 Kernel Development Flow

When developing hardware for either platform, the vendor advises testing a kernel's functional correctness and estimating its performance based on functional emulation or hardware simulation of the RTL generated from the OpenCL C kernel. Iterative generation and testing of RTL is preferable to repeatedly generating a new FPGA bitstream because placing and routing a design just once can take many hours. In the Intel flow, `aoc` is invoked with the options `-march=emulator` or `-march=simulator` to generate a kernel binary file for software emulation or hardware simulation, respectively. For Xilinx, the `-t` or `--target` flag is specified to `v++`; `-t=sw_emu` is for software emulation and `-t=hw_emu` is for hardware simulation.

Both hardware compilers generate interactive reports that can be used to provide insight into kernel performance and opportunities for optimization. Information provided in these reports includes FPGA resource utilization and the analysis of loops within a kernel. Intel generates this report by constructing an `.html` file that can be opened in a browser, and Xilinx generates summaries that can be navigated by using the `vitis_analyzer` graphical user interface (GUI) application.

2.3 Designing Hardware Using OpenCL

OpenCL for FPGAs supports two kernel execution models: multiple work item (MWI) and single work item (SWI). OpenCL was originally intended to support the MWI execution model, which maps well to wide single instruction, multiple data (SIMD) architectures, such as GPUs. It divides the work in the kernel among multiple threads, which are scheduled across one or more compute units. FPGAs can be configured for MWI kernel execution, but they also provide high-performance support for the SWI execution model. This model uses a single thread on a single compute unit to execute all the computation within a kernel, which is typically encapsulated within one or more nested loops. Using a single thread allows the hardware compiler to construct a deep pipeline that can support nontrivial data dependencies between loop iterations while still enabling high throughput up to a level at which each successive iteration of the loop can be initiated on every clock cycle. This work focuses on kernels designed for the SWI execution model because the SWI model often allows more advanced optimizations, resulting in better overall performance, as evidenced in prior work [12–14, 20].There are counterexamples to this in which the MWI execution model outperforms the SWI execution model [6, 11, 13], although these examples are few and often represent applications without complex memory dependencies that might be more appropriate for GPU offloading.

Another consideration of FPGA design is the parameterization of certain hardware features, known as *hardware design knobs*. For some designs, it could be beneficial to enable the tuning of certain

knobs, such as how large a particular buffer should be or how many times a loop should be unrolled. Using HLS makes design knob configuration as simple as specifying a macro definition to the hardware compiler (e.g., `-DBLOCK_SIZE = 2048`). However, finding the most performant configuration of these knobs for a given design is a nontrivial task. Furthermore, related work shows that porting OpenCL designs between two FPGAs supplied by the same vendor still requires some tuning to find the locally optimal knob configuration [5]. We explore retuning the knobs between platforms to some degree in this work but leave a more rigorous design space exploration to future work.

3 RELATED WORK

The kernels that we used for our performance and portability evaluation come from the Rodinia benchmark suite created by Che et al. [8]. The original intent of Che et al. was to provide a suite of applications to evaluate heterogeneous computing systems across a wide range of parallel computing communication patterns and accelerator interfaces (e.g., OpenMP and OpenCL). Zohouri et al. later adapted the OpenCL implementations of a subset of these kernels and systematically modified them to become performant on Intel FPGAs [20]. In this work, we extended the work done by Zohouri et al. by porting these Intel OpenCL kernels to be synthesizable and performant for Xilinx FPGAs.

Work done by de Fine Licht and Hoefler incorporates software engineering design principles into HLS development [10]. These works are somewhat similar to our work in that they try to account for differences when targeting an Intel or Xilinx FPGA through a C++ library they developed called `hlslib`. In contrast, our work focused on using OpenCL C for Intel and Xilinx FPGA kernels to evaluate the portability and performance of starting from a kernel optimized for an Intel platform and then porting that specification to a Xilinx platform.

Cabrera and Chamberlain also used the work done by Zohouri et al. to evaluate performance and portability by building kernels that were originally designed for an Intel FPGA connected via a peripheral component interconnect express (PCIe) card and targeting the Intel HARPv2 platform, which combines a CPU and FPGA on the same chip package [5]. Our approach is similar, but the focus of this work was on evaluating performance and portability across different FPGA vendors.

Zhang et al. used an OpenCL C kernel specification when architecting layers for a convolutional neural network [19]. However, they did not port an existing OpenCL C specification that originally targeted an Intel FPGA.

Brown and Dolman investigated optimization strategies for FPGAs' data movement by profiling and optimizing a Xilinx HLS implementation of a kernel that contained significant direct memory access transfers. [4]. Their target application was the Met Office NERC Cloud model, an atmospheric model's advection scheme that involves 53 double-precision operations and works with a large 6.44GB dataset. However, this work differs from ours in that the language used to architect kernels is not OpenCL C but C/C++ annotated with Xilinx specific pragmas. Two other recent works by Brown evaluated the performance of Xilinx's Vitis HLS tools with the Nekbone mini-app and the Himeno benchmark [2, 3]. In porting the Nekbone AX kernel from Fortran to Xilinx FPGAs via Vitis, the author studied optimizations, including revising the algorithm from von Neumann to dataflow form, optimizing the use of memory banks, loop unrolling, and ping-pong buffering. Brown also emphasized the importance of analyzing the Vitis scheduler explorer's logs in profile-guided optimization of Himeno. We also used Xilinx Vitis in our evaluation and used the GUIs provided by Vitis to more easily visualize kernel information, but we evaluated kernels architected in OpenCL C.

On the Intel side, Sanaullah et al. explored strategies for optimizing OpenCL kernels for the Intel OpenCL FPGA compiler [17]. They developed an approach that eschewed multiple-work-item kernels in favor of pipelined computations on a SWI and divided computations into elementary steps to help Intel's compiler recognize opportunities for operation reordering, register usage, and predication. Their approach resulted in multiple order-of-magnitude speedups over unoptimized kernels and speedups of up to several times over prior OpenCL kernels targeting FPGAs. Many of the optimizations suggested by their work should apply broadly to FPGA targets. Our work explored how approaches similar to theirs interact with a different platform, namely the Xilinx HLS compiler and FPGA fabric.

Harris et al. investigated the utility of commonly used cache-focused optimizations designed for traditional cores when executing on the Intel HARPv2 platform [11]. For the matrix multiplication application that they explored, the FPGA-deployed design consistently outperformed the optimized CBLAS library by as much as 3.5×. For that application, the MWI paradigm performed better across the board than the SWI paradigm.

To provide a higher level abstraction for FPGA programming, Lee et al. proposed an OpenACC-to-FPGA framework that automatically converts an input OpenACC C code into an output OpenCL code, which can be further compiled by the back-end OpenCL hardware compilers [14]. Lambert et al. [12, 13] extended the OpenACC-to-FPGA framework by implementing more FPGA-specific optimizations—which were also inspired from the work by Zohouri et al. [20]—but their framework supports only Intel FPGAs, not Xilinx FPGAs. The work presented in this paper can serve as a preliminary reference to extend the OpenACC-to-FPGA framework for Xilinx support so that it can maximize the performance portability of directive-based, high-level FPGA programming.

4 METHODS

To evaluate portability and performance, we leveraged the Intel OpenCL FPGA implementations of the host and kernel code of the Rodinia benchmarking suite [8] from Zohouri et al. [20] as a starting point to build and run applications on the Xilinx platform. For our experiments, we used a Xilinx Alveo U250 Data Center accelerator card, which includes an XCU250 FPGA of the Xilinx UltraScale+ architecture, a Gen3 x16 PCIe interface, and 64 GB of DDR4, off-chip memory. We used the 2020.1 version of the Vitis Core Development Kit.

4.1 Host-Side Code

The host-side code for an OpenCL application is responsible for setting up the OpenCL context for one or more devices, allocating

space for device buffers, and enqueueing operations on the OpenCL device (e.g., host-to-device data movement and enqueueing kernels for execution). The structure of the host-side code for each application remains largely the same. Because both the Intel and Xilinx FPGAs are targeted by using OpenCL, the host code must set up OpenCL constructs, such as the context and its associated command queue. Selecting a Xilinx FPGA instead of an Intel FPGA requires only a straightforward alteration on the host code.

One salient modification we made to the host code in Zohouri et al. is leveraging C++-like objects and RAII (Resource Acquisition Is Initialization). Because the host code can be compiled by using C++, we designed an object-oriented approach to manage the application. The variables associated with a particular application are encapsulated into one class, whereas the OpenCL constructs to handle accelerator management and kernel execution are encapsulated into another class. For example, in the `pathfinder` application, we implemented the class `PathfinderVars` to hold the parameters for the grid size and to initialize the grid to an initial state. We designed the `PV_OpenCL` class to handle the initialization and management of the Xilinx platform's context and command queue. Taking an object-oriented approach allows for an organizational structure that makes the source code easier to read and the application testing more user-friendly.

4.2 Applications

This section presents descriptions of each of the Rodinia applications used in this work. In all cases, our evaluation uses the input problem sizes specified in the original OpenCL FPGA evaluation done by Zohouri et al. [20].

4.2.1 Pathfinder. The goal of the `pathfinder` application is to find the value of a minimum-weight path from the top row of a 2D grid to the bottom row. This computation uses a dynamic programming approach. Each element in the 2D grid is populated with a nonnegative integer weight. The path to a given element, `elt`, is determined by the taking the minimum value from the northwest, north, or northeast element relative to `elt`. The program terminates when the last row of the 2D grid has been visited.

4.2.2 CFD. The Rodinia computational fluid dynamics (CFD) application is an unstructured grid benchmark that solves 3D Euler equations for inviscid compressible flow [9]. This application comprises three kernels: compute step factor, compute flux, and time steps. The kernels are highly compute-intensive with many single-precision floating point operations, including addition, multiplication, division, and square root. The most expensive computation is in the compute flux kernel, which calculates the artificial viscosity and accumulates flux contributions across each face.

4.2.3 SRAD. The Rodinia speckle-reducing anisotropic diffusion (SRAD) benchmark models an algorithm for isotropic diffusion on ultrasound images. SRAD aims to smooth speckled imagery by using a speckle-reducing filter. Its implementation includes multiple nested loops, array reductions, and stencil code patterns. These more complex memory dependencies make SRAD an ideal candidate for FPGA execution.

4.2.4 HotSpot. The Rodinia HotSpot application is an iterative partial differential equation solver for estimating processor temperature and heat diffusion over time, based on a provided processor "floor plan" and simulations of power measurements. Given initial temperature and power data, HotSpot iterates over each cell and applies a stencil operation.

4.3 Ported Kernels

Table 1 lists the particular kernel versions of each benchmark that we used and ported in our evaluation. The version numbering follows that of Zohouri et al. in which odd-numbered kernels use the SWI execution model.

Table 1: List of ported kernels.

Application	Baseline	Best
`pathfinder`	v1	v5
`cfd`	v1	v5
`srad`	v1	v5
`hotspot`	v1	v5

For each application examined, we ported the baseline and most performant kernels (*baseline* and *best*, respectively, in Table 1). The baseline versions are SWI kernels in which there are no FPGA optimizations supplied as hints to the hardware compiler aside from use of the SWI model itself, which tells the compiler to construct a deep pipeline. The most performant kernels are the versions that were reported to give the best performance among all kernel versions for each tested application in the original work by Zohouri et al. (i.e., the best evaluated kernel when targeting Intel-based Stratix V and Arria 10 FPGAs).

We evaluated the portability and performance of these kernels by performing the minimum amount of modifications required to port the annotated hardware optimizations for each kernel from the Intel specification to Xilinx. We detail the specifics of porting from Intel FPGA optimizations to Xilinx ones in Section 4.4. For the `pathfinder` application, we further extended our initial port by evaluating the addition of other optimizations as part of a design-space search.

4.4 Porting Optimizations from Intel to Xilinx

To evaluate the performance of Intel kernels authored in OpenCL C on a Xilinx platform, we must port the kernels originally written for an Intel platform to a Xilinx platform. Xilinx supports authoring kernels by using traditional C/C++, but we instead opted to use Xilinx's OpenCL C support to maximize the reuse of the kernels from the Intel platform and to test the portability of OpenCL C kernels between the two platforms. Using C/C++ to architect kernels on the Xilinx platform affords a more fine-grained control over the resulting hardware than is possible with OpenCL C, but we leave exploration of such re-architecting to future work.

Although using OpenCL C gives us a foundation for porting kernels between the two platforms, the way in which optimizations are specified for each is different. Intel uses a combination of specific programming patterns, `#pragmas` and `__attributes__`,

to provide guidance to the hardware compiler, whereas Xilinx uses only `__attributes__`. Additionally, although there is sometimes a one-to-one mapping of kernel optimizations between platforms, this is not always the case. The following sections detail the performant FPGA optimizations we have encountered thus far, how they are expressed for an Intel platform, and the changes we made to express that same construct on a Xilinx platform.

4.4.1 Loop Unrolling. Loop unrolling is a common optimization in FPGA programming. In both the Intel and Xilinx tools, loop unrolling hints allow the hardware compiler to use additional resources to replicate the loop body. In an SWI execution context, this allows for more deeply nested pipelines, higher FPGA resource utilization, and typically better overall performance. Intel and Xilinx support unrolling loops through compiler hints. For Intel OpenCL kernels, a loop is preceded with

`#pragma unroll N`.

For Xilinx, the previous pragma is replaced with

`__attribute__((opencl_unroll_hint(N)))`.

In both cases, `N` is the loop unrolling factor. Therefore, the mapping between loop unrolling for Intel and Xilinx OpenCL is straightforward. The hardware compiler will determine whether it is possible to unroll the loop given available resources of the target FPGA. Also, the Intel and Xilinx compilers will both attempt to analyze and automatically unroll non-annotated loops, but in our experience, manually applying the directives and attributes results in more consistent compilations and performance.

4.4.2 Shift Registers. Shift registers are a performant FPGA construct that aid in efficient pipelining of loop iterations by storing data to satisfy inter-loop dependencies and avoiding redundant loads from global FPGA memory. How these shift registers are constructed depends on the vendor. Both vendors support using registers within the FPGA fabric. Depending on the size, the Intel hardware compiler might try to synthesize a shift register by using on-chip memories. Xilinx supplies a header file that allows the shift register to be synthesized by configuring lookup tables in the FPGA fabric to act as a RAM-based shift register; however, we are unable to use this feature from OpenCL C code. Unlike the case for loop unrolling, there is not a one-to-one mapping for inferring shift registers between vendors.

We show a minimum example of how to infer a shift register for Intel and Xilinx in Listings 1 and 2, respectively. In this case, the shift register is used as a delay line. For Intel, a private buffer is declared (line 1), and the size of this buffer is a compile-time constant. Shift register shifting is orchestrated in the inner loop (line 5). For the hardware compiler to infer a shift operation, the inner loop must be unrolled by prepending a `pragma`, as described in Section 4.4.1. The Xilinx setup is similar. Again, a private buffer must be declared, but an additional attribute (line 2) must be appended to this buffer. This attribute is a hint to the hardware compiler that the kernel designer wants to completely decompose the buffer into a collection of registers. The `complete` keyword indicates that the buffer must be completely decomposed into a collection of registers, and the `0` argument implies that we are performing this decomposition among all dimensions of the buffer. The inner loop that orchestrates the shifting (line 6) is then unrolled, as described in Section 4.4.1, by appending an attribute (line 5).

```
1  int shift_reg[SR_SIZE]; // where SR_SIZE is a compile time constant
2  for (int n = 0; n < N; n++) {
3      shift_reg[SR_SIZE - 1] = input_arr[n];
4      #pragma unroll SR_SIZE - 1
5      for(int i = 0; i < SR_SIZE - 1; i++)
6          shift_reg[i] = shift_reg[i + 1];
7  }
```

Listing 1: Inferring a shift register using an Intel platform.

```
1  int shift_reg[SR_SIZE]
2      __attribute__((xcl_array_partition(complete,0)));
3  for (int n = 0; n < N; n++) {
4      shift_reg[SR_SIZE - 1] = input_arr[n];
5      __attribute__((opencl_unroll_hint(SR_SIZE - 1))
6      for(int i = 0; i < SR_SIZE - 1; i++)
7          shift_reg[i] = shift_reg[i + 1];
8  }
```

Listing 2: Inferring a shift register using a Xilinx platform.

5 RESULTS

This section discusses the results of porting the Rodinia applications with Intel-specific FPGA optimizations to a Xilinx FPGA platform.

5.1 Minimum Modification Porting

Figure 1 shows the results of porting the baseline and most performant kernel versions, as detailed in Section 4. The following sections detail the process of porting each kernel to the Xilinx platform.

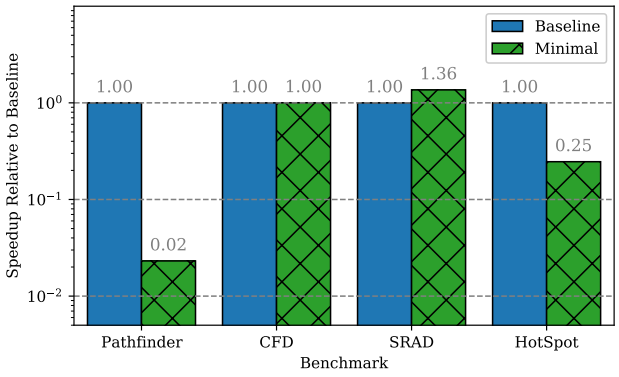

Figure 1: Each application's performance on the Xilinx platform for the port of their respective baseline kernel and the port with minimum modification of the most performant kernel when targeting the Intel platform. The performance is reported as speedup relative to the Xilinx baseline result.

5.1.1 Pathfinder. The `pathfinder` kernel version `v1` was straightforward to port because there are no compiler optimizations to port. For `pathfinder` kernel version `v5`, there are two hardware design knobs that must be set at compile time when building the kernel: `BSIZE` and `SSIZE`.

Because the pathfinder kernel performs a stencil operation, an FPGA-specific shift register or sliding window can be used to reduce redundant memory accesses, as mentioned in Section 4.4.2. The minimum size of the shift register is constrained by the smallest number of contiguous array elements required to encapsulate a single iteration of the stencil operation. In the pathfinder application, this equates to one complete row of the input array plus one additional element. For large input data sizes, even one row of the input dataset can be too large for implementation in a single shift register. The BSIZE hardware knob controls this column size and thus indirectly controls the resulting shift register size.

Although one output array element is assigned each iteration by default, the second tuning paramenter SSIZE allows multiple stencil operations per iteration. By allowing multiple operations per iteration, we can reduce the total number of iterations and increase the FPGA utilization. Increasing SSIZE can significantly improve performance if the FPGA has enough resources to support the hardware needed to perform multiple stencil operations and the hardware compiler does not have to increase the loop initiation interval or decrease the compute unit operating frequency.

In our initial port, we used the most performant parameters listed from Zohouri et al. which sets BSIZE = 32,768 and SSIZE = 32. However, we found that the kernel using this parameterization is not synthesizable immediately when building on Xilinx; the hardware compiler only allows a buffer to be partitioned 1,024 times. Therefore, it is not possible for us to infer a shift register by using the parameterization from Zohouri et al. To use the given parameters, then, we do not partition the array. The other change we made was to replace the Intel loop unrolling construct with the Xilinx one, as detailed in Section 4.4.1. At this point, the kernel was successfully built and executed on the Xilinx FPGA.

We found that the performance of the ported version of the most performant kernel is 43 times slower than the baseline version. Performance was expected to decrease without being able to infer a shift register. However, we found from the output logs generated by the hardware compiler that the main loop of computation in this kernel was not able to be pipelined; thus, each iteration of the loop must wait for the previous iteration to finish before it can start. The main benefit of the hardware compiler inferring a deep pipeline for kernel execution is that it enables the main computation loop to issue a loop iteration (ideally) at every cycle. Therefore, the performance portability of this kernel starting from the Intel specification is poor. Improving the performance of this kernel is detailed in Section 5.2.

5.1.2 CFD. The version v1 kernel of CFD is a straightforward SWI port of the original MWI kernel used in the GPU OpenCL kernel. The v1 version did not have any Intel-specific pragmas in the kernel. Therefore, no changes where made to the v1 kernel to target the Xilinx FPGA. The version v5 CFD kernel was the best performing on the Intel Stratix V FPGA, according to Zohouri et al. This kernel had various optimizations, including adding the restrict qualifier to the input arrays and using a shift register-based reduction to accumulate the flux contribution. The v5 version also had an unroll pragma, which we changed to a Xilinx unroll hint attribute.

The generated compiler information for version v1 reported that the Xilinx compiler was unable to flatten the main computation loop because the outer loop was not a perfectly nested loop. It also reported that there was a data dependency in the loop, which greatly reduced the loop iteration interval. The build reports for version v5 showed a lower loop initiation interval than the v1 kernel. However, the main iteration loop was still unable to be flattened because the the outer loop had nontrivial logic in the loop latch. Despite the reported lower initiation interval, the two kernels took essentially about the same amount of time to execute; the v5 kernel executed slightly faster with a 0.23% reduction in kernel execution time.

Overall, the performance of directly porting CFD kernels from Intel to Xilinx FPGAs was quite poor, with a 70× increase in kernel execution time compared with Zohouri et al.'s work [20]. The performance hit is not surprising when looking at the large loop latencies and initiation intervals reported in the Xilinx build reports. Further Xilinx specific optimizations and compiler hints that are beyond the scope of this paper are needed to obtain state-of-the-art performance.

Although we made minimal changes to the CFD kernels, we did encounter a few challenges when porting CFD. The first challenge was an undefined symbol when loading the .xclbin file during software emulation. This error only appeared for software emulation and did not appear during hardware emulation or when running on hardware. After exploration, we determined that the undefined symbol was from the calls to the built-in OpenCL math function sqrt. The same error also occurred for other built-in OpenCL math functions. We resolved this issue by updating the LD_LIBRARY_PATH environment variable with the path to the missing library located in the Vitis installation directory.

The second challenge we encountered was that build reports would not open correctly in the Vitis Analyzer for the CFD application when multiple kernels were compiled simultaneously. The Vitis Analyzer would report an error about an unexpected status found in the file and would open the build report with an unknown status. This issue can be avoided by compiling each kernel separately and then linking them together during the linking step.

5.1.3 SRAD. Because the SRAD application also implements an iterative stencil algorithm, it shares many of the same FPGA-specific optimizations and tuning parameters with the Pathfinder application. The SRAD v1 kernel implements a straightforward approach that extends the source Rodinia OpenCL kernel with restrict keywords on input array variables and creates SWI kernels. The highest performing kernel on the Intel platform version v5 combines the five separate kernels into a single kernel and implements a shift register-based reduction and shift register-based sliding window.

We make several changes for the minimally modified Xilinx analog kernel of the Intel-based v5. We replaced an Intel-specific attribute applied to the entire kernel,

```
__attribute__((max_global_work_dim(0)))
```

with an analogous one recognized by the Xilinx platform,

```
__attribute__((reqd_work_group_size(1, 1, 1))).
```

We also replaced instances of #pragma unroll with the previously mentioned Xilinx-specific attribute and annotated the shift register with the following Xilinx-specific attribute:

```
__attribute__((xcl_array_partition(complete, 0)))
```

For this application, we were able to completely partition the array used for the shift register operation and were not required to do a block or cyclic partition. The Xilinx platform's restrictions on the size of the shift register array are not necessarily a limitation. Although the Intel platform successfully compiles with larger shift register sizes, the larger arrays can significantly degrade performance, which is why the manual partitioning via the BSIZE variable and logic is present, even in the Intel-optimized code. Finally, we left the SSIZE replication factor at it's default value of 2.

As shown in Figure 1, the SRAD v5 kernel represents the only example in which directly porting an Intel-optimized kernel to use analogous Xilinx constructs improves performance over a more platform-agnostic baseline. This application demonstrates that directly translating constructs can improve performance over a baseline in some cases, although we do note that the absolute performance of the baseline and v5 SRAD underperform their Intel counterparts. That is, there is still a significant amount of room for Xilinx-specific improvement in these kernels.

5.1.4 HotSpot. Like Pathfinder and SRAD, the HotSpot application implements an iterative stencil. Again, the v1 version of the kernel is directly adapted from the original OpenCL, only adding restrict keywords and switching to a SWI kernel. In the v5 version, we again replaced the Intel-specific loop unrolling, kernel dimension attributes, and directives with Xilinx-specific attributes. Like Pathfinder, the default BSIZE value results in a shift register that is slightly too large for complete partitioning by the Xilinx compiler with a size of 1,032 elements against the restriction of 1,024. However, instead of defaulting to a blocking or cyclic partition scheme—which typically leads to poor performance, as shown in the following section—for the results presented in Figure 1, we instead reduced the value of BSIZE, which allowed full compilation with complete partitioning on the shift register array. Like SRAD, we again maintain the default SSIZE replication factor, which is 16 for the HotSpot application.

Figure 1 shows that, as with the Pathfinder application, directly translating the Intel-specific optimizations to their Xilinx counterparts in the v5 version degrades performance compared with a more agnostic, less-optimized baseline. HotSpot represents another example in which one-to-one kernel optimization ports do not lead to portable performance.

5.2 Extracting More Performance

Because three of the four translated (v5) kernels failed to outperform the relative baseline implementations (v1), we chose one representative application, Pathfinder, and performed a deeper exploration of potential Xilinx-specific optimizations by using the Xilinx-ported v5 as a baseline.

Although the large default setting of BSIZE prohibited the complete partitioning of an array to infer a shift register, we applied a cyclic partitioning to the array [16] in an effort to partition the array as much as possible and increase the local memory bandwidth since partitioning the array introduces more read and write ports and allows for more accesses to local memory. To do this, we append

```
__attribute__ ((xcl_array_partition(cyclic,PARTITION,1)))
```

to the buffer that is to act as a shift register. The argument cyclic specifies that we want the original array to be split into equally sized blocks, and the original elements of the array are interleaved into those blocks. The PARTITION variable specifies how many blocks to use when partitioning the array. In this work, we created the following design space for this variable:

$$\texttt{PARTITION} \in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\},$$

where the PARTITION = 1 is the same as the minimal pathfinder result from Figure 1.

The result of this design space sweep is shown by the bars starting with "partition_" in Figure 2. We observe that partitioning the array further degrades performance. The performance degradation can be broken into three regions starting at PARTITION = 2, 32, and 1,024. At PARTITION = 32, the iteration latency increases by 32,769 more cycles than when PARTITION = 16, in addition to not being pipelined. Thus, each iteration costs an additional 32K cycles. At PARTITION = 1,024, the amount of BRAM used in the design doubles from when PARTITION = 512. Because of this, the hardware compiler was unable to meet timing for the default request of 300MHz because one or more timing paths was too long. Thus, the clock frequency of the design was changed to 200.1MHz.

Another optimization employed was decreasing BSIZE so that a proper shift register could be inferred. Based on this constraint and the constraints placed by the problem, we created a design space of

$$\texttt{BSIZE} \in \{128, 256, 512\}.$$

In this case, we were able to infer a shift register, as specified in Section 4.4.2. Additionally, we limit set SSIZE = 1 because trying to compute more than one element per iteration adversely affected the initiation interval of the kernel. The results of sweeping this design space are shown by the bars starting with "sr_" in Figure 2.

We found that we achieved a minimum speedup over the baseline result from Figure 1 of 2× when BSIZE = 128 and a maximum speedup of 3× when BSIZE = 512. In addition to the shift register that is inferred, there are two salient differences between this modified version of the kernel and the version in which the partition factor is swept. The first is that the main loop in the kernel is fully pipelined. Although an iteration of the pipeline must be launched every other cycle as opposed to every clock cycle, this is a considerable improvement over previous versions that could not be pipelined. The second difference is that the iteration latency for this set of kernels is 72,000 cycles less than the minimum ported version from Section 5.1. Although the performance benefit is not as pronounced as the results found in Zohouri et al.'s work, this result reinforces that FPGA constructs that are known to aid performance (e.g., shift registers) are theoretically portable. However, porting that construct across different FPGA vendors requires nontrivial amount of work, making performance portability more difficult.

The last optimization we used was to employ predication as detailed by Sanaullah et al. [17] when optimizing for Intel FPGAs. In this optimization, conditional operations are avoided by executing both paths of a conditional branch and only committing the result that corresponds to the result of the conditional. Effectively, this means computing both paths of an if-else statement and storing the results in temporary registers. Depending on whether the condition is true or false, the selected result is stored to the target variable or memory location by using the ternary operator. To evaluate this optimization, we used the kernel with the inferred shift register, as outlined in the previous paragraph, and modified all of the if-else

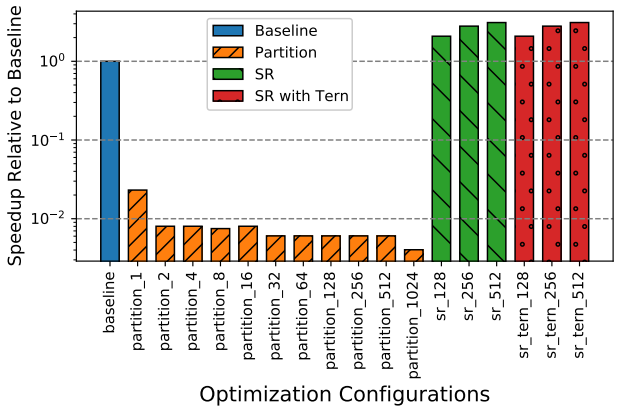

Figure 2: Applying additional optimizations to the minimally modified `pathfinder v5` kernel.

statements to use predication. The results of this evaluation are shown by the bars starting with "sr_tern_" in Figure 2. The results show no performance benefit to explicitly using predication. This is because the Xilinx hardware compiler already does this optimization as part of an optimization pass when statically analyzing the kernel source. To confirm this, we examined the compiler logs for the previous set of kernels, which show that the hardware compiler is performing if-conversion on hyperblocks that it finds in the kernel source. This is an important portability result because predication that is handled by one vendor's hardware compiler might not be handled by the other's and thus must be explicitly expressed.

6 CONCLUSION

This paper presents our efforts toward evaluating the portability and performance of kernels originally designed for an Intel FPGA platform and porting them to a Xilinx FPGA platform. To perform this evaluation, we used a subset of applications from the Rodinia benchmarking suite and their corresponding Intel OpenCL FPGA implementations. We ported baseline SWI versions of the `Pathfinder`, `CFD`, `SRAD`, and `HotSpot` computational kernels, as well as the most performant versions of those kernels based on previous executions on an Intel platform to the Xilinx platform. Porting the baseline versions is generally straightforward because there are no compiler hints supplied to the hardware compiler. Porting the versions of these kernels with more Intel-specific optimizations proved to be more difficult. The minimum amount of modification to port these kernels to the Xilinx platform varies. For some optimizations, such as loop unrolling, there is a one-to-one mapping for how to supply a hint to the hardware compiler to infer a particular optimization. For other optimizations (e.g., shift registers), the amount of modification is nontrivial, based on constraints that are embedded into the Xilinx hardware compiler. Achieving competitive performance from these ported kernels also is nontrivial. In our additional evaluation of optimizations for the `Pathfinder` kernel, we show that some optimizations might significantly degrade performance. However, we also show that FPGA constructs can perform well, regardless of the vendor specification, but might require additional modification from the kernel designer.

6.1 Future Work

There are many different avenues of future work that we plan to pursue. In the short term, there is still more work that can be done to evaluate how portable and performance-portable OpenCL kernels can be on Xilinx FPGAs. We plan to port more kernels from the Rodinia benchmark suite to expose different optimizations that we might not have encountered yet. For example, Xilinx enables the specification of which off-chip RAM banks to use for certain global buffers during the linking stage of the kernel build process. There are Rodinia kernels that take advantage of this in their corresponding Intel implementations, and we plan to evaluate that design choice, as well as others. Another short-term goal is to use C/C++ instead of OpenCL to author kernels because this is a feature supported by Xilinx. These kernels use compiler pragmas in a similar fashion to the Intel OpenCL kernels and provide a wider range of fine-grained control for how hardware gets synthesized for the FPGAs. Additionally, Xilinx provides header files and libraries that can be added to kernels designed in C/C++ that are designed to infer performant hardware constructs by the Xilinx hardware compiler. Our long-term goal is to use the lessons learned from our performance portability evaluation to automatically generate optimized Xilinx HLS kernels. As shown in our work, OpenCL is not inherently portable because it exposes the hardware details to the programmers. To achieve peak performance for different target accelerators, OpenCL code must be rewritten by using target accelerator-specific optimization techniques. This hurts the productivity and maintainability of the program. We plan to address this issue by extending a higher level compiler framework like OpenARC [15], which would allow us to generate optimized Xilinx kernels alongside Intel kernels and kernels for CPUs and GPUs.

ACKNOWLEDGMENTS

The authors would like to acknowledge the ORNL Experimental Computing Laboratory team for its support with the compute resources and the software stack. We would also like to acknowledge Elizabeth Kirby of ORNL for her helpful suggestions and edits to this work. This research was supported in part by the following sources: National Science Foundation (NSF) under grant CNS-1763503, Defense Advanced Research Projects Agency (DARPA) Microsystems Technology Office (MTO) Domain-Specific System-on-Chip Program, and the US Department of Energy (DOE) Advanced Scientific Computing Research (ASCR) program.

This manuscript has been co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan.

REFERENCES

[1] Mark Bohr. 2007. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *IEEE Solid-State Circuits Society Newsletter* 12, 1 (Winter 2007), 11–13. https://doi.org/10.1109/N-SSC.2007.4785534

[2] Nick Brown. 2020. Exploring the acceleration of Nekbone on reconfigurable architectures. In *Proc. of IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 19–28. https://doi.org/10.1109/H2RC51942.2020.00008

[3] Nick Brown. 2020. Weighing up the new kid on the block: Impressions of using Vitis for HPC software development. In *Proc. of 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 335–340. https://doi.org/10.1109/FPL50879.2020.00062

[4] Nick Brown and David Dolman. 2019. It's All About Data Movement: Optimising FPGA Data Access to Boost Performance. In *Proc. of IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 1–10. https://doi.org/10.1109/H2RC49586.2019.00006

[5] Anthony M Cabrera and Roger D Chamberlain. 2019. Exploring Portability and Performance of OpenCL FPGA Kernels on Intel HARPv2. In *Proc. of International Workshop on OpenCL*. ACM, 3:1–3:10. https://doi.org/10.1145/3318170.3318180

[6] Anthony M. Cabrera and Roger D. Chamberlain. 2020. Designing Domain Specific Computing Systems. In *Proc. of IEEE 28th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. https://doi.org/10.1109/FCCM48280.2020.00052

[7] Roger D. Chamberlain. 2020. Architecturally Truly Diverse Systems: A Review. *Future Generation Computer Systems* 110 (Sept. 2020), 33–44. https://doi.org/10.1016/j.future.2020.03.061

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[9] Andrew Corrigan, Fernando F. Camelli, Rainald Löhner, and John Wallin. 2011. Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids* 66, 2 (2011), 221–229. https://doi.org/10.1002/fld.2254

[10] Johannes de Fine Licht and Torsten Hoefler. 2019. hlslib: Software Engineering for Hardware Design. (2019). arXiv:1910.04436

[11] Steven Harris, Roger D. Chamberlain, and Christopher Gill. 2020. OpenCL Performance on the Intel Heterogeneous Architecture Research Platform. In *Proc. of IEEE High-Performance Extreme Computing Conference (HPEC)*. IEEE. https://doi.org/10.1109/HPEC43674.2020.9286213

[12] Jacob Lambert, Seyong Lee, Jungwon Kim, Jeffrey S Vetter, and Allen D Malony. 2018. Directive-based, high-level programming and optimizations for high-performance computing with FPGAs. In *Proc. of International Conference on Supercomputing (ICS)*. ACM, 160–171. https://doi.org/10.1145/3205289.3205324

[13] Jacob Lambert, Seyong Lee, Jeffrey S Vetter, and Allen Malony. 2020. In-depth Optimization with the OpenACC-to-FPGA Framework on an Arria 10 FPGA. In *Proc. of IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 460–470. https://doi.org/10.1109/IPDPSW50202.2020.00084

[14] Seyong Lee, Jungwon Kim, and Jeffrey S Vetter. 2016. OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing. In *Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 544–554. https://doi.org/10.1109/IPDPS.2016.28

[15] Seyong Lee and Jeffrey S Vetter. 2014. OpenARC: Open Accelerator Research Compiler for Directive-based, Efficient Heterogeneous Computing. In *Proc. of 23rd International Symposium on High-performance Parallel and Distributed Computing*. ACM, 115–120. https://doi.org/10.1145/2600212.2600704

[16] Xilinx Application Note. 2015. Increasing Local Bandwidth. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_increasing_local_memory_bandwidth.html.

[17] Ahmed Sanaullah, Rushi Patel, and Martin Herbordt. 2018. An empirically guided optimization framework for FPGA OpenCL. In *Proc. of International Conference on Field-Programmable Technology (FPT)*. IEEE, 46–53. https://doi.org/10.1109/FPT.2018.00018

[18] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. 2020. Top 500 List. https://www.top500.org/.

[19] Shuo Zhang, Yanxia Wu, Chaoguang Men, Hongtao He, and Kai Liang. 2019. Research on OpenCL Optimization for FPGA Deep Learning Application. *PLOS ONE* 14, 10 (2019), e0222984. https://doi.org/10.1371/journal.pone.0222984

[20] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. 2016. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 409–420. https://doi.org/10.1109/SC.2016.34