In-Depth Optimization with the OpenACC-to-FPGA Framework on an Arria 10 FPGA

Jacob Lambert
University of Oregon
jlambert@cs.uoregon.edu

Seyong Lee
Oak Ridge National Laboratory
lees2@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory
vetter@ornl.gov

Allen Malony
University of Oregon
malony@cs.uoregon.edu

***Abstract*—The reconfigurable computing paradigm that uses field programmable gate arrays (FPGAs) has received renewed interest in the high-performance computing field due to FPGAs' unique combination of performance and energy efficiency. However, difficulties in programming and optimizing FPGAs have prevented them from being widely accepted as general-purpose computing devices. In accelerator-based heterogeneous computing, portability across diverse heterogeneous devices is also an important issue, but the unique architectural features in FPGAs make this difficult to achieve. To address these issues, a directive-based, high-level FPGA programming and optimization framework was previously developed. In this work, developed optimizations were combined holistically using the directive-based approach to show that each individual benchmark requires a unique set of optimizations to maximize performance. The relationships between FPGA resource usages and runtime performance were also explored.**

***Index Terms*—FPGA, OpenACC, OpenARC, directive-based programming, compiler optimization**

I. Introduction

Accelerator-based heterogeneous computing, which typically employs devices such as GPUs and many-core processors, has become a mainstream approach in high-performance computing (HPC) to solve performance, power efficiency, reliability, and cost issues caused by increasing power densities in conventional von-Neumann architectures. Similarly, reconfigurable computing that uses FPGAs and coarse-grained reconfigurable devices has received renewed interest due to the unique combination of performance and energy efficiency through flexible hardware customizations. FPGAs' reconfigurable nature allows these architectures to be customized to match the needs of a given application and achieve much higher energy efficiency and/or performance gains compared with conventional CPUs and GPUs. As a result, FPGAs have been deployed in various application domains, such as finance [1], database systems [2], machine learning [3], image processing [4], graph analysis algorithms [5], and others. Moreover, recent trends in FPGA technologies—such as supporting hardened floating-point data signal processing blocks and integrating CPUs, GPUs, and FPGAs as a new system-on-chip devices—make FPGA-based high-performance reconfigurable computing more attractive for serious exploration in scientific simulation and data analytics.

However, FPGAs have not been widely adopted as general-purpose computing devices primarily due to programming and optimization difficulties. In accelerator-based heterogeneous computing, functional and performance portability across diverse architectures are important and desirable. In contrast, traditional FPGA programming that uses low-level hardware description languages (HDLs) exposes too many details involving complex hardware and software designs to the programmers, inhibiting programmability and portability. To alleviate the programmability concern in FPGA computing, several high-level synthesis (HLS) programming models have been proposed [6]–[11]. OpenCL (Open Computing Language) is the first standard programming model that is functionally portable across diverse heterogeneous architectures and has been adopted by major FPGA vendors (e.g., Intel and Xilinx) [12]. Despite its potential to offer better programmability and portability than HLS approaches, programming and optimizing FPGAs with OpenCL is still considered to be very complex and difficult due to the semantic gap between the OpenCL abstraction and the low-level hardware design. For example, the current OpenCL abstraction does not provide a straightforward method for programmers to express specific hardware features, such as shift registers, hardware channels, and pipeline delays. Instead, the underlying OpenCL compiler implicitly derives and synthesizes all hardware logic from an input program, and there are many practical limits in compilation for finding optimal hardware designs. Consequently, existing FPGA OpenCL compilers can be very sensitive to specific code patterns. One way to solve this problem is to lower the programming abstraction level offered by OpenCL to expose the low-level hardware design to the programmers. However, lowering the programming abstraction would sacrifice the portability benefits of OpenCL and negatively affect its programmability.

To address the problems caused by the aforementioned semantic gap, a directive-based, high-level programming and optimization framework for efficient FPGA computing was proposed in previous works [13], [14]. This framework takes a standard, portable OpenACC program as input and generates an output OpenCL code, which the underlying OpenCL compiler further compiles into an FPGA hardware configuration file. The proposed framework solves the semantic-gap issue using directive-based, high-level FPGA-specific optimizations in which programmers provide important characteristics of the input program via a set of directives. The framework then generates specific OpenCL code patterns so the underlying back-end OpenCL compiler can infer known FPGA

programming paradigms, including shift registers, hardware pipelines, and sliding windows. The proposed OpenACC-to-FPGA translation framework offers enough abstraction over low-level hardware designs and complex OpenCL programming syntax and also provides high-level control over various FPGA-specific optimizations. As a result, the programmer can specify FPGA optimizations with user-friendly, high-level OpenACC directives and keywords and will leave the lower level error-prone OpenCL FPGA-specific syntax generation to the compiler.

Previous works [13], [14] focused primarily on developing and evaluating each individual optimization in isolation to maximize performance gains when applying a specific optimization. This work focuses on evaluating the optimization framework holistically to maximize performance for each application using the array of available optimizations.

Although there is a higher level of abstraction in the OpenACC-to-FPGA framework, the reconfigurable nature of the underlying FPGA architecture exposes multiple design choices for users to decide. The different threading-model approaches of multi-threaded parallelism and pipeline parallelism present design choices with significantly different performance for different applications. Furthermore, within each threading approach are many other optimizations that may or may not apply and may or may not improve performance for that specific application. In OpenCL, restructuring and reformatting source code to implement the multiple different combinations of optimizations are very time consuming and error prone. A thorough design exploration and holistic application of optimization arrays are greatly simplified by using a directive-based approach, which typically only requires changing a few lines of code for each individual design.

The contributions in this work are as follows.

- A categorical organization and summary of optimizations presented in [13], [14] are provided.
- Developed optimizations on an array of benchmarks are holistically evaluated using an Arria 10 FPGA.
- The effects of FPGA resource usages and kernel frequencies on runtime performance are explored.
- The necessity of high-level frameworks for efficient FPGA optimization and design exploration and the need to transition to a more automated process are demonstrated.

II. Background

A. FPGAs

FPGAs are composed of digital signal processing (DSP) blocks, registers, adaptive look-up tables (ALUTs), and other typical hardware components. At runtime, the FPGA is configured to use a subset of these hardware components using programmable interconnects. This runtime-configuration property provides several advantages for FPGAs compared with other accelerators. First, specific resources can be allocated to meet the needs of specific applications, leading to performance improvements. Additionally, because only crucial components are configured, FPGAs can maintain a low-power state. However, configuring the FPGA for specific applications has traditionally required programming in HDLs at the register-transfer level (RTL).

RTL FPGA programming in VHDL or Verilog is inaccessible to most application programmers because it requires in-depth knowledge of the FPGA, such as cycle-by-cycle descriptions of hardware, and hardware-clock timing considerations. To address these programmability issues, two leading FPGA manufacturers, Intel/Altera and Xilinx, have provided an OpenCL SDK for their FPGA devices.

B. Intel FPGA SDK for OpenCL

The OpenACC-to-FPGA project relies heavily on the Intel (formerly Altera) FPGA SDK for OpenCL [15]. Although this SDK is the first major vendor-supported project to bridge the gap between HPC and FPGAs, the abstraction level is inadequate for most HPC programmers. Although the SDK technically allows shift registers, sliding windows, and other hardware-level FPGA constructs to be implemented, expressing these constructs in OpenCL clashes against traditional HPC programming intuition because they require programming patterns and loop constructs antithetical to conventional programming principles. This is a primary motivation for the OpenACC-to-FPGA project.

Compilation times using the Intel SDK are significantly longer than traditional CPU or GPU compilation times, often taking several hours. This is generally true of all HLS tools. However, the Intel SDK does provide a significant amount of information about the application and how it will be mapped to hardware before attempting a full compilation. The estimated resource usages and design layout, which are neatly presented in HTML format, were very useful for guiding optimizations, even when working at the OpenACC level.

C. OpenACC

The source-level portability of OpenCL allows programmers to write applications once and run them on any OpenCL-compliant hardware accelerators, such as CPUs, GPUs, Xeon Phis, DSPs, and FPGAs. However, OpenCL does not provide performance portability across different types of accelerators because the abstraction level typically requires device-specific code to optimize performance. Additionally, OpenCL is still considered a low-level programming model for scientific application developers due to the required knowledge of different hardware architectures and machine models.

OpenACC addresses these challenges. OpenACC is a directive-based, performance-portable parallel programming model for a wide variety of hardware accelerators. The model outsources device-specific implementation details to the compiler to reduce the required programming effort and increase performance portability. The OpenACC API—which consists of compiler directives, library routines, and environment variables—allows programmers to augment applications with information, exposing available parallelism within an application. A core OpenACC facility is to offload the burden

of mapping parallelism directly to devices from the user to the underlying compiler. Because of its simplistic API, maintainability, usability, and portability, OpenACC is increasingly being considered as an alternative to lower level accelerator programming models.

D. OpenARC

The OpenACC-to-FPGA translation framework discussed in this work is built inside the OpenARC compiler framework [16]. OpenARC is a research-oriented OpenACC compiler that specializes in rapidly prototyping new optimizations, API features, and device-support for emerging technologies. This makes OpenARC an ideal platform for the initial implementation of OpenACC-to-FPGA translation, which was first introduced by Lee et al. [13].

OpenARC takes an input C program that is annotated with OpenACC directives, performs several optimization and translation passes, and generates an optimized output host and kernel code in CUDA or OpenCL. The CUDA or OpenCL output is then further compiled using a low-level device compiler, such as NVCC or Intel's OpenCL compiler. In this work, OpenARC was used to generate OpenCL specific to Intel FPGAs and to apply FPGA-specific optimizations that were developed previously [13], [14].

III. Optimization Background

This section provides a high-level overview and categorical classification of the different optimizations developed in previous works [13], [14]. The optimizations are segregated into three primary categories: (1) automatically applied optimizations requiring no user intervention, (2) repurposed directives in which existing OpenACC directives are implemented in an FPGA-specific way, and (3) directive extensions in which FPGA-specific directives are developed outside the established OpenACC standard. This classification is presented as a summary of previous work compulsory to the presented evaluations and as a structured template for adding new optimizations as the OpenACC-to-FPGA framework continues to improve. The cited works provide more detailed descriptions on individual optimizations, their implementations inside the OpenARC framework, and evaluations of their efficacy.

A. Automatically Applied

The first class of optimizations in the OpenACC-to-FPGA framework represents optimizations for which no user intervention is required. These optimizations can be safely applied any time the compiler encounters specific constructs and are applied independently from any user-supplied directives.

1) Dynamic Memory Transfer Alignment: In a typical FPGA-based heterogeneous system, an FPGA is attached to the host CPU via PCIe bus as a discrete device with a separate memory. Therefore, for a device kernel to access the host data and vice versa, data should be explicitly transferred between the host and device memory. Existing FPGA OpenCL runtimes, such as Intel OpenCL runtime, use direct memory access (DMA) for higher throughput and lower latency. To exploit DMA, the host-side buffer and device-side buffer should be aligned. Although device buffers are automatically allocated in an aligned way, host buffers should be allocated with special memory allocators (e.g., `posix_memalign()` in Linux). Even if both host and device buffers are allocated in an aligned way, the transfer of partial arrays might not exploit DMA if at least one of the start addresses is not aligned. The OpenACC-to-FPGA framework runtime dynamically analyzes memory alignment and employs temporary buffers to satisfy alignments without user interference, as described in Lee et al. [13].

2) Code Motion: Loop-invariant code motion is a common optimization generally applicable to all hardware devices, which can reduce redundant computations in a loop. Branch-invariant code motion is less commonly applied because it does not reduce the computation amount. However, in FPGA-based reconfigurable computing, the branch-invariant code motion can reduce the overall hardware resource requirements by removing redundant hardware logic generation for the invariant codes in both branches. Therefore, the underlying Intel OpenCL compiler supports the branch-invariant code motion optimization.

To further reduce the resource usage, the OpenACC-to-FPGA framework also includes branch-variant code motion optimization, which factors out branch-variant codes if both branches have the same computation patterns. This optimization does not reduce the amount of computations; instead, it reduces the resource usage by sharing hardware logic for the common computation patterns in both branches, which can improve performance by increasing the kernel operating frequency. It also allows for more aggressive unrolling and increased parallelization.

3) Boundary Check Elimination: When an OpenACC compute region is translated into a device kernel, each iteration in a work-sharing loop will be mapped to each device thread (work-item in OpenCL) according to the OpenACC execution model. If the total number of device threads is not the same as the number of corresponding loop iterations, then the device kernel should be executed so that only device threads with valid mapping execute the loop body, which is usually implemented using control statements. Generally, control flow divergence by control statements is less of an issue in FPGA computing than in GPU computing because the reconfigurability in FPGA can completely eliminate the diverging control paths of thread executions by using hardware predicates if conditional structure is simple enough. However, if the device kernel has complex control structures such as thread-dependent backward branching, then the underlying OpenCL compiler cannot flatten the control structures, which could cause significant performance penalty by disallowing various advanced compiler optimizations, such as kernel vectorization. To alleviate the burden for the underlying OpenCL compiler to flatten the control structure, a compiler pass was developed that uses built-in symbolic analysis tools to check and eliminate unnecessary loop-boundary check code at compile time.

B. Repurposed OpenACC Directives

The second class of optimizations in the OpenACC-to-FPGA framework represents optimizations that users can optionally apply using existing OpenACC directives and clauses. Many clauses are typically implemented by compilers in a specific way to optimize GPU performance. In the OpenACC-to-FPGA framework, these clauses were re-implemented to optimize FPGA performance without changing their syntax or context from a programming perspective.

1) Single Work-Item Clauses: Because FPGAs can leverage deeply pipelined execution, single-threaded pipeline-parallel implementations can outperform their multi-threaded counterparts in many situations. This contrasts with GPU execution, which explicitly relies on multi-threaded execution. Because OpenACC was primarily developed with a focus on GPU execution, the default execution model assumes multi-threaded parallelism using multiple gangs, workers, and/or vectors.

In the proposed OpenACC-to-FPGA framework, existing OpenACC directives were repurposed to allow programmers to indicate that a kernel should be executed as a single work item. By setting the `num_gangs`, `num_workers`, and `vector_length` clauses of an OpenACC parallel directive to 1, OpenARC can generate the appropriate OpenCL code for the underlying back-end compiler to correctly infer a pipeline parallel execution.

2) Collapse Clause: Loop collapsing is a common optimization used across several directive-based languages, including OpenMP and OpenACC. This optimization requires a compiler to combine two tightly nested loops into a single loop, which typically requires the original iteration variables to be recalculated at each iteration. In a multi-threaded context, this recalculation can be done using division and modulus operations, deriving the old iteration index values from the collapsed iteration index value.

In an FPGA context, these division and modulus operations are relatively expensive in terms of execution time and resource usage. However, in a single work-item context, recalculating at each iteration is unnecessary. Instead, row and column counters can be used, which require only addition and comparison operations. If the given kernel is executed in the single work-item context, OpenARC generates a row and column counter approach when encountering collapse clauses in the OpenACC-to-FPGA framework instead of using the costly division and modulus approach.

3) Reduction Clause: Scalar reductions represent an operation commonly optimized by compilers. For implementations that target multi-threaded CPUs or GPUs, this optimization is typically a tree-based approach. The leaves represent the array of values, and the roots represent the combination of those values by some scalar operation.

This tree-based implementation can also be used in an FPGA context and could outperform a straightforward serialized approach. However, a more efficient approach can be realized using FPGA-specific shift registers [15]. In this approach, partial sums are accumulated in a shift register, and then the final value is computed by doing a traditional reduction over the partial sums.

In the OpenACC-to-FPGA framework, OpenARC implements this shift–register-based approach when encountering reduction clauses. This FPGA-specific alternative to the GPU-specific approach can lead to significant performance improvements at the cost of a small increase in resource usage [14].

C. Directive Extensions

For some FPGA-specific optimizations, automatic application by the compiler is difficult. Also, there might not be a straightforward mapping to existing directives that programmers could use to optionally apply the optimizations since these optimizations might not be relevant in GPU or multi-threaded CPU contexts.

In these cases, novel directive extensions are developed that can be recognized by the OpenARC compiler framework. The goal of these extensions is to allow programmers with limited FPGA knowledge to leverage FPGA-specific optimizations that could largely affect performance.

1) Kernel Vectorization Directive: In the Intel FPGA OpenCL programming, kernel vectorization allows multiple work items (device threads) in an OpenCL work group to execute in a single instruction multiple data (SIMD) fashion, which is implemented by replicating the kernel data paths while sharing control logic across each SIMD vector lane. Kernel vectorization is usually beneficial, but its additional resource requirement could contend with other optimizations. Although the OpenACC `vector` clause has similar effects, the vectorization behavior in the OpenACC execution model is not the same as that of the Intel OpenCL kernel vectorization. In OpenACC, vector lanes execute only in a SIMD manner if a kernel is in vector-partitioned mode and might not execute in a lockstep manner. In contrast, OpenCL kernel vectorization exercises a strict lockstep vectorization.

2) Compute Unit Replication Directive: The reconfigurable nature of FPGAs allows multiple compute units to be generated for each kernel so that the hardware controller in FPGA can distribute work groups to available compute units in addition to running multiple work groups in a pipeline of a compute unit. Increasing the number of compute units can achieve higher throughput, but it also increases bandwidth pressure to the global memory and requires more hardware resources, whose optimal number should be carefully tuned.

3) Channels Directive: In the current OpenACC execution model, there is no mechanism to allow fine-grained synchronization between actively running device kernels, and the device kernels can communicate with each other only through the device global memory. Therefore, both kernels require reading and writing to and from global memory to communicate, and the communication is serialized due to kernel communication. Moreover, the limited bandwidth and long latency of the global memory could become another performance-limiting factor. To address these issues, the underlying Intel OpenCL provides a hardware mechanism called *channel*, which two concurrently running kernels can use to communicate with each other in

a fine-grained manner without using the expensive global memory. If two or more OpenACC kernels execute in a sequential order and communicate with each other using temporary device buffers, then these kernels might be able use the channel mechanism when running on an FPGA. However, for the kernels to use this mechanism without breaking the original execution semantics, these kernels should communicate in specific patterns, which are not easy for the compiler to detect automatically. To enable the channel mechanism in OpenACC, a set of new backward-compatible OpenACC data clauses were proposed with the existing OpenACC data clauses that will preserve functional portability across FPGAs and non-FPGA devices.

4) Window Directive: Applications relying on stencil computations are common in scientific computing. Many algorithms operating on a grid or matrix apply a stencil pattern at each input location, relying on neighboring locations. These patterns and operations can result in redundant, expensive memory operations on devices such as FPGAs.

However, with a pipeline-parallel single work-item kernel, sliding window optimizations can be used to greatly improve the performance of these stencil computations [14] [15]. The sliding window is constructed using a shift–register-based buffer and used to store input data values relevant to the stencil computation at every iteration. The benefit of the sliding window approach is that in each iteration, only one value is loaded from memory. The other values needed for the stencil pattern are loaded from the shift-register buffer, which is a relatively inexpensive operation on FPGAs.

The *window* directive extension (Listing 2) allows OpenACC programmers to use this shift–register-based approach without actually implementing the shift-register logic, which can be difficult for programmers without experience with the Intel OpenCL SDK.

D. Code Example

Listing 1 provides an example OpenCL kernel implementing an FPGA-specific sum reduction and an FPGA-specific stencil operation using a sliding window. For brevity, variable initialization was omitted, as well as the host code involved with setting up the OpenCL command queue, data buffering and transfers, and other standard OpenCL API calls. In the OpenCL implementation, programmers manually manage shift register initialization, population, and shifting via loops. Mismanaging the shift register logic could cause the underlying compiler to not infer a shift register in hardware and result in catastrophic performance results.

Listing 2 provides an example of the same algorithm using OpenACC instead of OpenCL. In the OpenACC implementation, shift-register management and other low-level, OpenCL-specific error-prone features are abstracted away. Also, although not provided, the OpenACC host code is much simpler than the verbose OpenCL host code and requires only simple data movement directives.

Listing 1. OpenCL version of FPGA reduction and sliding window.

```
// OpenCL FPGA Reduction
#define REGISTER_DEPTH (8 * N) // unroll factor N
float shift_reg[REGISTER_DEPTH + 1] = {0}; // create and initialize shift registers.
#pragma unroll N
for (int i = 0; i < SIZE; ++i) {
  shift_reg[REGISTER_DEPTH] = shift_reg[0] + input[i]; // perform partial reduction.

  for (int j = 0; j < REGISTER_DEPTH; ++j)
  {   shift_reg[j] = shift_reg[j + 1];   } // shift values in shift registers.
}

#pragma unroll
for (int i = 0; i < REGISTER_DEPTH; ++i)
{   sum += shift_reg[i];   } // perform final reduction on shift registers.

// OpenCL FPGA Sliding Window
#define ROWS ...
#define COLS ...

#define NBD_SIZE    (2*COLS + 1)  // neighborhood size
#define SW_OFFSET   (COLS)        // window offset
#define READ_OFFSET (COLS)        // read offset
#define SW_SIZE     (NBD_SIZE + UNROLL_FACTOR - 1)

float sw[SW_SIZE]; // create a sliding window array.

for (int index = -(READ_OFFSET); index < ROWS*COLS; index += UNROLL_FACTOR) {
  for (int i = 0; i < NBD_SIZE - 1; ++i)
  {  sw[i] = sw[i + UNROLL_FACTOR];  } //Shift UNROLL_FACTOR positions.
  // load UNROLL_FACTOR values to the sliding window.
  for (int ss = 0; ss < UNROLL_FACTOR; ++ss) {
    if (index + READ_OFFSET + ss < ROWS*COLS)
    {  sw[NBD_SIZE - 1 + ss] = input[index + READ_OFFSET + ss];  }
  }

  float value[UNROLL_FACTOR]; // temporary array storing outputs.
  #pragma unroll
  for (int ss = 0; ss < UNROLL_FACTOR; ++ss) {
    if (index + ss >= 0) {
      float N = sw[SW_OFFSET+ss - COLS];
      float S = sw[SW_OFFSET+ss + COLS];
      float E = sw[SW_OFFSET+ss + 1];
      float W = sw[SW_OFFSET+ss - 1];
      output[index] = sw[SW_OFFSET+ss] + N + S + E + W;
      value[ss] = sw[SW_OFFSET+ss] + N + S + E + W;
    }
  }
  // store temporary outputs to the output array.
  for (int ss = 0; ss < UNROLL_FACTOR; ++ss) {
    if (index + ss >= 0)
    { output[index + ss] = value[ss]; }
  }
}
```

Listing 2. OpenACC version of FPGA reduction and sliding window.

```
// OpenACC FPGA Reduction
#pragma acc parallel loop num_gangs(1) num_workers(1) reduction(+:sum)
#pragma unroll N
for (int i = 0; i < SIZE; ++i)
{   sum += input[i];   }

// OpenACC FPGA Sliding Window
#define ROWS ...
#define COLS ...

#pragma acc parallel loop num_gangs(1) num_workers(1)
#pragma openarc transform window (input, output)
for (int index = 0; index < ROWS*COLS; ++index) {
   float N = input[index - COLS];
   float S = input[index + COLS];
   float E = input[index + 1];
   float W = input[index - 1];
   output[index] = input[index] + N + S + E + W;
}
```

IV. Experimental Setup

A. Software

The front-end OpenACC code was compiled using OpenARC v0.11 (`openarc`). OpenARC is hosted on the internal

Oak Ridge National Laboratory (ORNL) repository site, and access can be granted upon author request. The back-end OpenCL code is compiled using the Intel FPGA SDK for OpenCL v17.1.0 (`aocl`). The software stack is built on CentOS Linux 7 (Core), and runtime measurements are recorded using C API calls, specifically `clock_gettime()`.

Several Python scripts were also built to automate batch build, compilation, and execution processes for the FPGA. These scripts also extract resource usage and other compilation information reported by the `aocl` compiler and notify users via text message/email upon compilation completion.

B. Hardware

For the following evaluations, an Intel Arria 10 FPGA, model p510t_sch_ax115 (427K ALMS, 2713 RAM Blocks, 1518 DSPs) was used. For the host CPU, a multicore Intel Xeon E5-2683 v4 was used, although all the executed host codes are single threaded.

C. Benchmarks

Several representative benchmarks were employed to holistically evaluate the OpenACC-to-FPGA optimizations. Two benchmarks—SRAD (speckle reducing anisotropic diffusion) and HotSpot—originate from the Rodinia benchmark suite. SRAD is used in medical imaging, and HotSpot is used to model heat diffusion. Although these benchmarks were originally programmed using OpenMP, OpenACC adaptations available in the OpenARC repository were used. These benchmarks were specifically chosen because they posses characteristics amenable to FPGA optimization, and other works considered them to be interesting applications in FPGA contexts [17].

Evaluations were also performed using the Sobel benchmark, which is an edge-detect4 ion image processing algorithm from the Intel FPGA OpenCL SDK. Sobel required a relatively straightforward adaptation from OpenCL to OpenACC, as performed in previous works [14].

Finally, for the sake of generality, evaluations were performed using two core algorithms: (1) Jacobi, which is commonly used for solving systems of linear equations in many scientific domains, and (2) MatMul (matrix multiplication), which is used in applications in nearly every domain.

V. Evaluation

The holistic evaluation of the numerous optimizations in the OpenACC-to-FPGA framework required many executions with different combinations of threading models, optimizations, kernel vectorization and compute unit replication factors, unrolling factors, and more. This process was manually guided, but it was also restricted by the applicability of optimizations to each algorithm and device resource limitations. The optimization process for each benchmark was greatly simplified by the directive-based approach because code changes between versions were very minimal. However, the large optimization search space also exposed the dire need for a more automated optimization process.

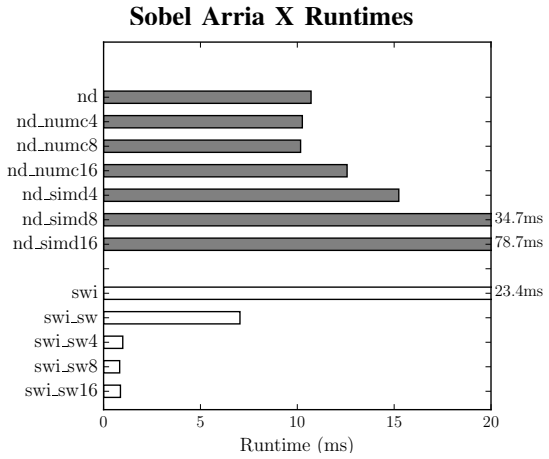

Fig. 1. Runtime performance (in seconds) of Sobel with different FPGA-specific optimizations applied. Gray bars indicate the nd-range approach, and white bars indicate the single work-item approach (smaller is better).

This section presents individual evaluations of each benchmark with accompanying discussions on the effects of various optimizations. An evaluation of resource usages for the Jacobi and SRAD benchmarks is also presented.

A. Individual Benchmark Optimizations

To evaluate each individual benchmark, first only the multi-threaded and single work-item implementations of each kernel were evaluated. Then, optimization directives and clauses were incrementally applied to each version where possible. Finally, different replication factors were tested when possible by varying the number of compute units and SIMD parallelism in the multi-threaded kernels and by varying the reduction and sliding window unrolling factors in the single work-item kernels. In Figs. 1–5, the version name is a concatenation of the applied optimizations. The following key explains the abbreviations:

- `nd`: multi-threaded kernel
- `numcX`: number of compute units (X: replication factor)
- `simdX`: vectorization (X: replication factor)
- `elim`: kernel boundary elimination optimization
- `coll` collapse optimization
- `swi`: single work-item kernel
- `redX`: reduction optimization (X: unroll factor)
- `swX`: sliding window optimization (X: unroll factor)
- `hoist`: code motion optimization

1) Sobel: The Sobel benchmark iterates over a 1D image array and performs a stencil operation. Sobel is unique among the evaluated benchmarks because it relies on integer operations, which can be implemented very efficiently in FPGA logic. In Fig. 1, the baseline `nd` implementation outperforms the baseline `swi` implementation. However, if the replication factors are scaled for `simd` and `numc` in the multi-threaded version, then the performance degrades significantly. This is mostly likely due to the high cost of the memory operations for the 9-point stencil, which is relative to the cheap cost of integer and bit arithmetic.

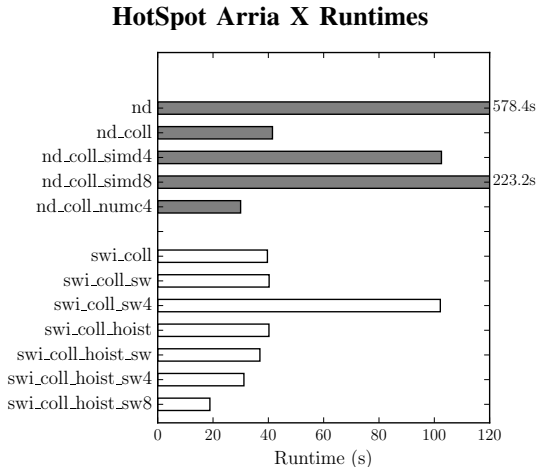

Fig. 2. Runtime performance (in seconds) of HotSpot with different FPGA-specific optimizations applied. Gray bars indicate the nd-range approach, and white bars indicate the single work-item approach (smaller is better).

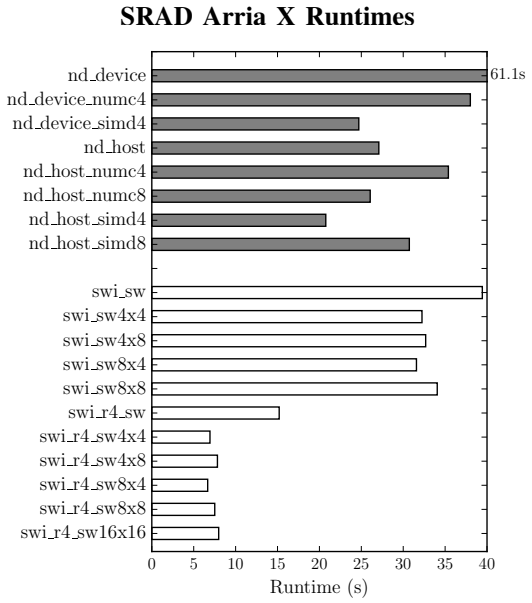

Fig. 3. Runtime performance (in seconds) of SRAD with different FPGA-specific optimizations applied. Gray bars indicate the nd-range approach, and white bars indicate the single work-item approach (smaller is better).

Conversely, applying additional optimizations and replication to the single work item significantly improves performance. Applying the sliding window pattern effectively reduces the ratio of memory operations to computation, and applying loop unrolling significantly increases the parallelism.

Sobel represents a unique but critical example in which the multi-threaded kernel initially outperforms the single work-item kernel until applying sufficient optimization results in a performance reversal. This example exposes a pitfall of manual intuition-guided optimization and further motivates an automated analytical optimization solution.

2) HotSpot: Like Sobel, HotSpot also consists of a stencil operation, although HotSpot relies on a 5-point stencil instead of a 9-point stencil and uses floating-point values instead of integer values.

As shown in Fig. 2, HotSpot experiences a significant performance difference if the collapse optimization is applied. This result contradicts previous results on the Stratix V [14] in which the collapse optimization achieved only modest performance. This could be attributed to the difference in FPGAs, but it is more likely an artifact of the different compiler version and how v17.1 of the SDK interprets the nested loops.

In the multi-threaded kernels, HotSpot does not respond well to kernel vectorization, likely due to the high degree of branching and numerous conditionals. However, HotSpot does experience modest performance improvement from compute unit replication (41.49 s vs. 29.97 s).

If only the collapse optimization is applied, the single work-item kernels perform very similarly to the multi-threaded kernels. Only with significantly more optimization (collapse, code motion, sliding window, and unrolling) does the single work-item approach achieve a lower runtime (18.87 s vs. 29.97 s). As with the Stratix V device evaluations in Lambert et al. [14], no performance improvements were seen from the window optimization with replication unless the common operation hoisting optimization (code motion) was also applied.

3) SRAD: The SRAD algorithm consists of three separate kernels, one large reduction, and two 5-point stencil loops. In Fig. 3, the `device` and `host` keywords for the multi-threaded kernels indicate whether the reduction is performed on the FPGA device or on the host (and are updated via an `acc update` directive). For the single work-item kernels, the reduction is always performed on the device. The distinct sliding window unrolling factors for the first and second stencil loops are separated by the `x`. For example, in the `nd_host_simd8` version, the reduction is performed on the host CPU and the results that are copied to the device, and the two stencil loops are vectorized with an SIMD factor of 8. In the `swi_r4_sw_8x4`, the first kernel is optimized using the reduction optimization with an unroll factor of 4, and the second and third kernels are optimized using the sliding window approach with unroll factors of 8 and 4, respectively.

Unlike HotSpot and Sobel, SRAD multi-threaded kernels respond well to compute unit replication and kernel vectorization; the 4-way kernel vectorization performs the best overall.

The vectorized multi-threaded kernels outperform many single-threaded kernels, even after applying the sliding window and unrolling optimizations to the second two loops. However, after applying a combination of sliding window and reduction optimizations, the single-threaded kernel significantly surpassed the multi-threaded counterpart. Again, the trend is seen only where sufficiently optimized single-threaded kernels outperform nd-range kernels.

4) MatMul: Compared with the other evaluated benchmarks, MatMul is the most simplistic and has no conditionals, limited arithmetic, and few memory operations.

As shown in Fig. 4, there is little opportunity for optimiza-

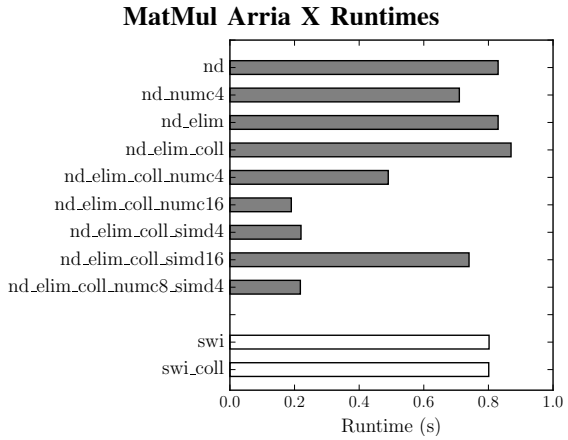

Fig. 4. Runtime performance (in seconds) of MatMul with different FPGA-specific optimizations applied. Gray bars indicate the nd-range approach, and white bars indicate the single work-item approach (smaller is better).

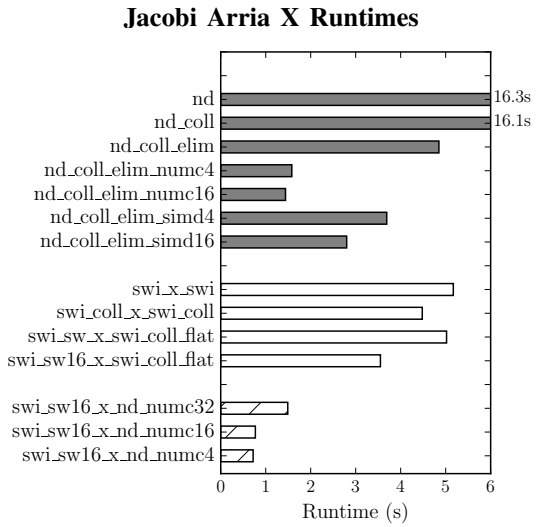

Fig. 5. Runtime performance (in seconds) of Jacobi with different FPGA-specific optimizations applied. Gray bars indicate the nd-range approach, white bars indicate the single work-item approach, and hatched bars indicate a hybrid nd-range+single work-item approach (smaller is better).

tion in the single-threaded approach because only the collapse optimization can be applied with no significant difference in performance.

However, as a result of the simplicity, MatMul responds very well to kernel vectorization and compute unit replication. MatMul is the only example in which the nd-range kernels significantly outperform the single work-item counterparts.

However, this could be an example in which the OpenACC-to-FPGA contains insufficient optimizations for this application. Including more advanced optimizations that are not yet supported by the framework (e.g., blocking, tiling, local memory buffering) could improve performance.

5) Jacobi: Jacobi, which was the final application evaluated, was the most interesting. Jacobi consists of two kernels, one 5-point stencil operation with very little arithmetic (floating point), and an array-copy kernel.

In Fig. 5, there are three different groupings of performance measurements. The top grouping represents versions in which a multi-threaded approach was used for both kernels by applying the same optimizations to each kernel.

The second grouping conversely represents versions in which the single-threaded or single work-item approach was used for both kernels. The `x` separates the optimizations applied to the first and second kernels. This distinction was made because the window optimization applies only to the first stencil-based loop. The `flat` keyword represents versions in which the original 2D array is replaced with a 1D array, and the indices are adjusted accordingly. Currently, the window optimization supports only 1D arrays, so this manual code modification was needed even though manually modifying code is generally avoided and only directives are applied.

The third grouping represents a novel hybrid approach in which a single work-item approach is used for the first kernel and a multi-threaded approach is used for the second kernel. Although the `flat` keyword is omitted, these versions also revert to a 1D array to apply the window optimization.

As shown in Fig. 5, the best-performing hybrid approach (0.72 s) outperforms the best-performing nd-range kernel approach (2.8 s) and the best-performing single work-item kernel approach (3.55 s). These results further complicate the manual optimization process because different threading models can lead to optimal performance, even within a single application. Fortunately, using the high-level directives in the OpenACC-to-FPGA framework to switch between threading models requires modifying only two clauses in the enclosing OpenACC directive compared with OpenCL, which requires modifying the host and device code.

B. Performance Discussion

Unsurprisingly, there is not one set of optimizations that can be universally applied to guarantee the best performance. Threading models, replication approaches, and even optimal replication and unrolling factors vary between benchmarks. This variety in best-performing optimizations demonstrates the crucial need for a simple, straightforward, and error-free method of applying, removing, and modifying optimizations, which this project's directive-based OpenACC-to-FPGA framework provides.

However, despite the variability in optimizations, several general trends apply across most benchmarks. The collapse optimization is universally beneficial in the context of multi-threaded and single work-item kernels if it can be applied. Similarly, when applicable, the sliding window optimization typically results in significant improvement, and single work-item kernels with these optimizations normally outperform their multi-threaded counterparts.

Furthermore, the straightforward multi-threaded version (`nd`) and straightforward single work-item version (`swi`) typically perform very similarly. The deciding factor between threading models is decided by the performance gains when applying various optimizations and opportunities for replication. Replication in multi-threaded kernels is always possible

by increasing the number of compute units, and it is usually possible using SIMD parallelization unless the compiler cannot vectorize the kernel. Conversely, in the single work-item context, replication in the form of loop unrolling is possible only if the reduction or sliding window optimizations can be applied. As a result of this distinction, applications in which the reduction and sliding window optimizations can be applied often perform best with the single-threaded model (Sobel, SRAD, HotSpot, Jacobi kernel 1), whereas other applications perform best with the multi-threaded model (MatMul, Jacobi kernel 2).

C. Resource Usage Evaluation

This section evaluates the relationship between the reported resource usages and kernel frequency (fmax) and runtime performance. It also evaluates the two benchmarks with the highest variety of code versions, SRAD and Jacobi.

1) SRAD Resources: In the SRAD multi-threaded kernels, increasing the replication factors scaled the resources used. Conventional logic suggests that using more of the FPGA resources would result in higher performance, and this is often the case. However, scaling the replication factors—and thus the degree of parallelism—also lowers the operating fmax, which typically degrades performance. Therefore, there is often a trade-off between resource use and operating frequencies, and the goal is often to achieve a balance between increasing parallelism and maintaining a high fmax. The effects of this relationship are shown in Table I. The lowest runtime multi-threaded version employed a sufficient degree of 4-way SIMD parallelism while still maintaining a high fmax relative to the other versions at 226.8 MHz.

In the single work-item kernels, a similar behavior was observed. Although replication generally increases performance, higher degrees of unrolling the trade-offs between parallelism and fmax are relevant. The best-performing kernel achieved a balance between a high degree of parallelism (replication factors of 4, 8, and 4) and a high fmax (286.69 MHz).

2) Jacobi Resources: By using the multi-threaded kernel approach in the Jacobi kernel (Table II), applying the collapse and kernel boundary elimination optimizations significantly improved performance without significantly changing resource use. However, because the replication factors are increased using the compute unit and kernel vectorization optimizations, significant increases in resources usages occurred. This parallelism increase scarified the fmax, which increased the overall runtime. Although the `numc16` replication was still well under the resource usage restrictions, the OpenCL implementation limited the quantity of compute units.

In the single work-item approach, window optimization significantly decreased the fmax, especially when unrolling was applied aggressively. However, as with the multi-threaded case, the large increase in parallelism more than offsets the decrease in fmax, which resulted in a lower overall runtime. Also, applying the window and unrolling optimizations significantly increased the DSP usage, which is a powerful resource in the Arria 10 FPGA that generally improves performance when used fully [18].

In the hybrid-threading approach, the window optimization still uses a significant portion of DSPs. However, the multi-threaded nature of the array copy kernel also consumes a significant portion of RAM blocks. Using these resources resulted in the lowest overall runtime, even with a lower fmax.

VI. Related Work

Watanabe et al. [19] presented preliminary results on a very closely related project and targeted OpenACC using the OmpSs compiler, which, like OpenARC, can generate output OpenCL. However, instead of developing optimizations inside the OpenACC to OpenCL translation, they generate SPD code for SPGen (Stream Processing Generator) alongside OpenCL. The separation program designator (SPD) code bypasses the OpenCL abstraction layer, translating directly into HDL. This seems to be a promising project to complement the OpenACC-to-FPGA framework.

Sommer et al. [20] presented a fully functional implementation of the OpenMP device offloading for the Xilinx FPGAs. The work integrated a custom compiler toolflow into the LLVM/Clang OpenMP offloading infrastructure. The input program contains one or more OpenMP target directives. The compiler generates a complete FPGA design, including a ThreadPoolComposer device software executable, Vivado HLS input file, and kernel description. In the prototype, the FPGA offloaded versions show slower performance than one 4-core CPU.

Zohouri et al. [17] ported and optimized a subset of the GPU-friendly Rodinia OpenCL benchmark suite [21] to the Altera FPGA OpenCL platform. The work optimized the original OpenCL kernels with FPGA-specific techniques, such as sliding windows. The results show that FPGAs can achieve better power efficiency compared with GPUs, even though they could not show competitive performance with GPUs.

Brahm et al. [18] compared a highly optimized GPU version and a highly optimized FPGA version of an astronomy code using the Arria 10 FPGA and Intel OpenCL interface. The work found that the FPGA can achieve competitive performance with the GPU by maximizing DSP usage and liberally using the channels extension, which warrants further investigation.

The r DataCentric (DaCe) project [22] recently integrated FPGA support. DaCe relied on Python and a graphical user interface-based dataflow diagram to map computations to hardware. DaCe might be an interesting option for new FPGA-centric applications, but it would require significant code restructuring and algorithm modifications to existing applications.

VII. Conclusion

This work examined the directive-based high-level FPGA programming approach implemented in the OpenARC compiler. The presented project takes an OpenACC program with optional user directives as input and generates an output

TABLE I
SRAD RESOURCE USAGE.

Thread model	Version	ALUTs	DSP blocks	RAM blocks	fmax	Runtime (s)
multi	nd_reduce	82303	88	594	247.155	61.13
multi	nd_reduce_simd4	152466	271	1025	228.51	24.7
multi	nd_update	62013	58	537	257.99	27.09
multi	nd_update_simd4	102068	193	898	226.8	20.76
multi	nd_update_simd8	148427	373	1441	214.4	30.73
swi	swi_sw	65028	59	545	256.6	39.44
swi	swi_r4_sw	68033	129	516	252.14	15.2
swi	swi_r4_sw4x4	82398	264	601	284.57	6.94
swi	swi_r4_sw4x8	89032	284	643	257.46	7.83
swi	swi_r4_sw8x4	96493	424	695	286.69	6.668
swi	swi_r4_sw8x8	102911	444	737	270.48	7.51
swi	swi_r4_sw16x16	133672	804	1064	245.33	7.98

TABLE II
JACOBI RESOURCE USAGE.

Thread model	Version	ALUTs	DSP blocks	RAM blocks	fmax	Runtime (s)
multi	nd	42884	3	416	290.95	16.32
multi	nd_collapse	42682	3	409	279.64	16.07
multi	nd_collapse_elim	42403	3	403	285.06	4.85
multi	nd_collapse_elim_numc4	64210	12	634	262.74	1.58
multi	nd_collapse_elim_numc16	152581	48	1558	210.03	1.44
swi	swi_x_swi	44646	3	544	267.8	5.17
swi	swi_collapse_x_swi_coll	42903	3	407	311.52	4.48
swi	swi_window_x_swi_flat_coll	41709	3	421	287.93	5.02
swi	swi_window16_x_swi_flat_coll	124581	768	852	208.63	3.55
hybrid	window16_x_nd_permute_numc4	133044	768	914	217.39	1.49
hybrid	window16_x_nd_permute_numc16	170596	768	1154	212.94	0.77
hybrid	window16_x_nd_permute_numc32	220912	768	1474	196.11	0.72
hybrid	window16_x_nd_permute_simd16	123687	768	854	220.26	4.07

OpenCL program, which is further synthesized into hardware logic to be executed on FPGAs. The main goal of this directive-based high-level FPGA programming is to strike a balance among programmability, performance, and portability. The directive-based approach allows programmers to provide important characteristics of an input program and control the overall translation and optimization while achieving enough abstraction over FPGA-specific hardware features and complex low-level programming models.

The optimizations developed in previous works were classified into three categories: automatically applied optimizations independent of user input, existing OpenACC directives repurposed for FPGA-specific implementation, and directive extensions supporting other FPGA-specific features. Then, these optimizations were holistically evaluated against a set of representative benchmarks using an Arria 10 FPGA.

The experimental results show that multi-threaded and single-threaded kernels can perform well on FPGAs, depending on which optimizations can be applied to a specific application. For example, most applications that allow for advanced single-threaded optimizations outperform their multi-threaded counterparts. In contrast, applications in which these single-threaded optimizations do not apply might perform best using multi-threaded compute unit or SIMD replication.

The relationship between resource usage and runtime performance was also explored. In general, higher resource usage indicates better utilization that typically results from replication, which leads to better performance. However, there are also several exceptions to this trend. In some cases, if two benchmark versions employ the same degree of parallelism, then higher resource usage can indicate less-efficient routing and could hurt performance. In other cases, if additional logic is implemented that results in higher resource usage and sacrifices the kernel fmax, then performance can suffer. Finally, even if more logic were implemented without sacrificing the fmax, lower performance is still observed if the initiation interval is increased [14].

The team plans to aggressively extend this project in the future. An immediate target is to integrate the Aspen performance modeling tool [23], [24] into the OpenACC-to-FPGA translation to automate the optimization process, including threading-model selection and lower level tuning of replication and unrolling factors.

The team also aims to support Xilinx devices in the near future. Developing *hlslib* [25] and other cross-platform tools at the OpenCL level can greatly simplify multidevice support in the OpenACC-to-FPGA framework.

Although the presented directive-based framework has exclusively relied on OpenACC as the front-end programming model, the team envisions supporting OpenACC and OpenMP within this project due to the introduction and increased popularity of the OpenMP offloading model. By employing tools

such as CCAMP [26] for OpenMP to OpenACC translation and developing an analogous FPGA-specific API for OpenMP, the project can be extrended to support OpenMP offloading models.

Finally, because the OneAPI approach was introduced, Intel's FPGA support is projected to shift from OpenCL to OneAPI's SYCL/DPC++ implementation. Likewise, the team's long-term goal is to migrate the OpenACC-to-FPGA framework to use these newer intermediate representations.

Acknowledgment

The authors would like to acknowledge the ORNL Experimental Computing Laboratory team for its support with the compute resources and the software stack.

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the US Department of Energy (DOE) Office of Science (OS) and the National Nuclear Security Administration.

This material is based upon work supported by the DOE OS Advanced Scientific Computing Research under contract number DE-AC05-00OR22725. The US government retains—and the publisher, by accepting the article for publication, acknowledges that the US government retains—a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

References

[1] Q. Tang, L. Jiang, M. Su, and Q. Dai, "A pipelined market data processing architecture to overcome financial data dependency," in *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, Dec. 2016, pp. 1–8.

[2] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, "Database Analytics Acceleration Using FPGAs," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 411–420. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370874

[3] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang, "FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-Off," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2017, pp. 160–167.

[4] Y. Kim, S. Jadhav, and C. S. Gloster, "Dataflow to Hardware Synthesis Framework on FPGAs," in *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, Oct. 2016, pp. 91–96.

[5] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj, "A framework for FPGA acceleration of large graph problems: Graphlet counting case study," in *2011 International Conference on Field-Programmable Technology*, Dec. 2011, pp. 1–8.

[6] *Mentor - DK Design Suite*, https://www.mentor.com/products/fpga/handel-c/dk-design-suite/.

[7] *Xilinx - SDSoC Development Environment*, https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html.

[8] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts, "Handel-C Language Reference Guide." Oxford University Computing Laboratory, August 1996.

[9] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11, 2011, pp. 33–36.

[10] P. R. Panda, "SystemC: A Modeling Platform Supporting Multiple Design Abstractions," in *Proceedings of the 14th International Symposium on Systems Synthesis*, ser. ISSS '01, 2001, pp. 75–80.

[11] M. C. Smith, J. S. Vetter, and X. Liang, "Accelerating Scientific Applications with the SRC-6 Reconfigurable Computer: Methodologies and Analysis," in *19th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '05, 2005.

[12] K. Group, "Open Compute Language (OpenCL)," [Online]. Available: http://www.khronos.org/opencl/, 2013, (Accessed Mar. 11, 2020).

[13] S. Lee, J. Kim, and J. S. Vetter, "OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing," in *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '16, 2016, pp. 544–554.

[14] J. Lambert, S. Lee, J. Kim, A. Malony, and J. Vetter, "Directive-Based, High-Level Programming and Optimizations for High-Performance Computing with FPGAs," in *Proceedings of the 2018 IEEE International Conference on Supercomputing Symposium*, ser. ICS '18, 2018, pp. 544–554.

[15] *Intel FPGA SDK for OpenCL*, https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[16] S. Lee and J. Vetter, "OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing," in *HPDC '14: Proceedings of the ACM Symposium on High-Performance Parallel and Distributed Computing, Short Paper*, June 2014.

[17] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, 2016, pp. 35:1–35:12.

[18] B. Veenboer and J. W. Romein, "Radio-astronomical imaging: Fpgas vs gpus," in *European Conference on Parallel Processing*. Springer, 2019, pp. 509–521.

[19] Y. Watanabe, J. Lee, K. Sano, T. Boku, and M. Sato, "Design and preliminary evaluation of openacc compiler for fpga with opencl and stream processing dsl," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*, 2020, pp. 10–16.

[20] L. Sommer, J. Korinth, and A. Koch, "OpenMP device offloading to FPGA accelerators," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2017, pp. 201–205.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. ha Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[22] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[23] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *SC12: International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, 2012, pp. 1–11.

[24] S. Lee, J. S. Meredith, and J. S. Vetter, "COMPASS: A framework for automated performance modeling and prediction," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. Newport Beach, California, USA: ACM, 2015, pp. 405–414.

[25] J. de Fine Licht and T. Hoefler, "hlslib: Software engineering for hardware design," *arXiv preprint arXiv:1910.04436*, 2019.

[26] J. Lambert, S. Lee, A. D. Malony, and J. S. Vetter, "CCAMP: OpenMP and OpenACC Interoperable Framework, Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms," in *Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar), in conjunction with Euro-Par19*, Gottigen, Germany, 2019.