Directive-Based, High-Level Programming and Optimizations for High-Performance Computing with FPGAs

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

Seyong Lee
Oak Ridge National Laboratory
lees2@ornl.gov

Jungwon Kim
Oak Ridge National Laboratory
kimj@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory
vetter@computer.org

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

ABSTRACT

Reconfigurable architectures like Field Programmable Gate Arrays (FPGAs) have been used for accelerating computations from several domains because of their unique combination of flexibility, performance, and power efficiency. However, FPGAs have not been widely used for high-performance computing, primarily because of their programming complexity and difficulties in optimizing performance. In this paper, we present a directive-based, high-level optimization framework for high-performance computing with FPGAs, built on top of an OpenACC-to-FPGA translation framework called OpenARC. We propose directive extensions and corresponding compile-time optimization techniques to enable the compiler to generate more efficient FPGA hardware configuration files. Empirical evaluation of the proposed framework on an Intel Stratix V with five OpenACC benchmarks from various application domains shows that FPGA-specific optimizations can lead to significant increases in performance across all tested applications. We also demonstrate that applying these high-level directive-based optimizations can allow OpenACC applications to perform similarly to lower-level OpenCL applications with hand-written FPGA-specific optimizations, and offer runtime and power performance benefits compared to CPUs and GPUs.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; *Emerging languages and compilers*; • **Software and its engineering → Source code generation**; • **Computing methodologies →** *Concurrent programming languages*; • **Computer systems organization →** *Pipeline computing*;

KEYWORDS

OpenACC, FPGA, OpenCL, directive-based programming, reconfigurable computing, OpenARC, sliding window

ACM Reference format:
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 *Proceedings of 2018 International Conference on Supercomputing, Beijing, China, June 12–15, 2018 (ICS '18),* 12 pages.
https://doi.org/10.1145/3205289.3205324

1 INTRODUCTION

Heterogeneous computing using manycore processors (e.g., GPUs or Xeon Phis) is a popular solution to hardware performance constraints in existing and upcoming architectures. Reconfigurable architectures like Field Programmable Gate Arrays (FPGAs) have received renewed interest because of their unique combination of flexibility, performance, and energy efficiency. Their reconfigurable nature allows the FPGA hardware to be customized for applications so they can achieve much higher energy efficiencies compared to conventional CPUs and GPUs. Consequently, FPGAs have been increasingly adopted in various industry and research platforms, such as data centers [18], storage systems [12], cloud systems [5], and research prototypes [23].

Despite this interest, FPGAs have not been widely adopted for high-performance computing (HPC), mainly because of their programming complexity and difficulties in optimizing performance. Traditionally, programming FPGAs required substantial knowledge about the underlying hardware design and low-level hardware description languages (HDLs), such as Verilog and VHDL. To reduce FPGA programming complexities, several high-level synthesis (HLS) programming models have been proposed [2–4, 6, 17, 21]. OpenCL is the first standard programming model that is adapted by all major FPGA vendors (i.e., Intel [1] and Xilinx [3]) and that is functionally portable across diverse heterogeneous architectures.

Despite offering the potential for portability to GPUs and Xeon Phis, programming FPGAs with OpenCL has two contradicting problems: (1) the semantic gap between the OpenCL abstraction and the low-level hardware design is too wide, and (2) the programming abstraction offered by OpenCL is still considered too low for typical HPC programmers. Because of the first issue, the efficiency of the OpenCL compiler is critical to achieve performance because it synthesizes all hardware logic for the input program. The practical limits are mainly caused by the semantic gap, so existing OpenCL compilers targeting FPGAs tend to be very sensitive to specific code patterns. For example, a study on manual porting of OpenCL kernels to FPGAs showed that a minor change in the OpenCL code may have huge effects on how the underlying OpenCL compiler interprets the input code and how resources are used to implement the input algorithms, compile times, and performance [25]. Although lowering the programming abstraction level offered by OpenCL could reduce the compiler's sensitivity caused by the semantic gap, this would negatively affect programmability.

To address these issues, we propose a directive-based, high-level programming and optimization framework for efficient FPGA computing, built on top of the existing OpenACC-to-FPGA translation framework called *OpenARC* [16]. This directive-based, high-level programming model offers better programmability, and the semantic information offered by user directives can better inform the OpenARC compiler for OpenACC-to-OpenCL translation. Our goal is to generate specific OpenCL code patterns such that the underlying OpenCL compiler can infer the intended semantics.

This paper makes the following contributions:

- We design, implement, and evaluate directive-based, high-level FPGA-specific optimizations, exploiting known FPGA programming paradigms, such as shift registers and sliding windows, to reduce overall FPGA runtimes and resource usage.
- We port two existing Intel OpenCL benchmarks to OpenACC, apply proposed optimizations to the ported OpenACC benchmarks and other existing OpenACC benchmarks, and evaluate their performance on the Intel Stratix V FPGA.
- We compare the performance of OpenACC applications with FPGA-specific optimizations to manual OpenCL applications with hand-tuned FPGA-specific optimizations.
- We compare the runtime and power usage of OpenACC FPGA executions against OpenACC GPU executions and OpenMP CPU executions.

2 BACKGROUND

2.1 OpenACC

OpenCL is the first standard, portable programming model adapted by major FPGA vendors to provide ease of programming over complex FPGA architectures. However, OpenCL has some critical issues to be settled; rewriting large legacy HPC applications written for traditional CPU-only machines using OpenCL can be difficult and impractical. Developing OpenCL applications for heterogeneous hardware accelerator-based systems requires significant rewriting and restructuring of hundreds of thousands of code lines, and reorganization of major data structures. Furthermore, OpenCL does not provide performance portability across different hardware accelerators, because it still exposes low-level hardware architecture abstractions to the programmers [20]. The OpenACC programming model addresses these challenges by providing a higher-level approach compared to OpenCL. OpenACC has the potential to provide better code- and performance-portability across a wide range of hardware accelerators [19]. The OpenACC API contains a set of directives that enable offloading of kernel code to accelerators. The directives extend ANSI C, C++, and Fortran-based languages in a way that allows programmers to migrate applications incrementally to accelerators and requires few modifications of the legacy HPC application code.

2.2 OpenARC as an OpenACC-to-FPGA Translator

Our work uses OpenARC [16] as the baseline translation framework of OpenACC to FPGA. OpenARC is an open-source compiler framework for directive-based heterogeneous programming research, and it is the first OpenACC compiler supporting FPGAs [14]. It performs source-to-source translation and optimization of the input OpenACC program into OpenCL code, which is further compiled into a hardware image by a backend FPGA OpenCL compiler such as the Intel Offline Compiler. OpenARC already supported several FPGA-specific compiler optimizations prior to this work, which assisted the underlying FPGA compiler in generating more efficient FPGA hardware configuration files [14]. Even though some optimizations (e.g., dynamic memory-transfer alignment) are generally applicable, most optimizations proposed in the previous work focus on massively parallel compute regions executed by many threads, which is a common HPC pattern preferred by both CPUs and GPUs. The deeply pipelined nature of the FPGA architecture also offers an alternate, single-threaded, or single work-item, approach. The single work-item approach sequentially executes the parallel region by pipelining the entire region, which grants the compiler more opportunities to maximize the parallel pipeline stages and may also enable true data pipelining across loop iterations using built-in *shift registers* when applications share or reuse data across work-items. To better exploit the fully customizable pipelining, which is an essential component of the FPGA-based accelerator computing, we propose several directive-based, high-level FPGA-specific optimizations. These are explained in the following section.

3 DIRECTIVE-BASED, HIGH-LEVEL FPGA-SPECIFIC OPTIMIZATIONS

This section presents a directive-based, high-level FPGA-specific optimization framework, which consists of directive extensions and corresponding compiler optimizations to generate more efficient FPGA hardware configuration files. The proposed directives are designed for programmers either to provide key information necessary for the compiler to automatically generate output OpenCL code that enables FPGA-specific optimizations, or to control important tuning parameters of those optimizations at a high level.

We want to clarify how we use the term *optimization*. There is a distinction between programmer-written OpenCL optimizations (like the shift-register reduction pattern and the sliding window pattern) and compiler optimizations implemented in OpenARC (like the reduction transformation and window transformation). In the Intel FPGA SDK for OpenCL [1], programmers can use FPGA-specific features like shift registers and sliding windows by programming in OpenCL using very specific patterns. These programming patterns are nonintuitive for most OpenCL programmers and can be error-prone. Currently, the OpenCL compiler does not offer a directive- or compiler-based approach to generating these programming patterns. A primary goal of this study is to create transformations in OpenARC that automatically generate these non-intuitive programming patterns from OpenACC directives. Doing so greatly simplifies the implementation of FPGA-specific features, and allows programmers without knowledge of shift registers and sliding windows to create more efficient FPGA programs.

Listing 1: OpenACC nested loops with collapse clause

```
#pragma acc parallel loop num_gangs(1) num_workers(1) vector_length(1) collapse(2)
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++) {  ...  }
```

The following optimizations were inspired by the Intel OpenCL SDK documentation [1]. We primarily chose to implement in OpenARC optimizations that potentially apply to a wide range of applications; for example, loop collapsing, scalar reduction, and branch-variant code motion optimizations are generally beneficial when they are applicable, whereas the sliding window optimization can benefit applications with stencil patterns.

3.1 Single Work-Item Optimization

As explained in the previous section, a common approach in general CPU- and GPU-based computing is to develop massively parallel applications that can be partitioned across multiple computation units. Although this approach can be effective when targeting FPGAs, FPGAs alternatively offer a single-threaded approach, which is generally preferred for efficient FPGA computing. In OpenCL terminology, the massively parallel or multiple work-item approach is known as an NDRange kernel, and the single-threaded approach is referred to as a single work-item kernel [1]. To execute an OpenACC compute region in a single work-item fashion, the numbers of *gangs*, *workers*, and *vectors* should be explicitly set to *1*, respectively.

Some applications, like embarrassingly parallel algorithms, are well-suited to NDRange execution. For other algorithms with data dependency or data reuse across work-items, the simple single work-item optimization alone may increase performance when executing on an FPGA. In addition to the stand-alone benefits, this optimization is notable because it is a prerequisite for the following optimizations: collapse optimization (Section 3.2), reduction optimization (Section 3.3), and sliding window optimization (Section 3.4).

3.2 Collapse Optimization

In a massively parallel computing approach, a loop collapse optimization is commonly used either to increase the amount of computations to be parallelized or to change the mapping of iterations to processing units. Loop collapsing is already a part of the OpenACC standard, and OpenARC supports the collapse clause. In Listing 1, we see a pair of perfectly nested loops with a collapse clause and single work-item directives. In the current OpenARC implementation (V0.11), collapsing of perfectly nested loops is achieved by creating a new loop expression with a newly defined iteration variable. OpenARC recalculates the values of the original iteration variables at each iteration using division and modulus operators. However, both division and modulus operations are relatively expensive on the FPGA in terms of both resource usage and execution time [1]. In our FPGA-specific collapse optimization, we can replace these expensive operations with row and column counters. These counters are updated at each iteration with cheaper addition operations, which allows us to more efficiently collapse nested loops in an FPGA single work-item execution. We can see the resulting OpenACC code after applying OpenARC's collapse transformation in Listing 2.

Listing 2: OpenACC loop after collapse transformation

```
// Traditional transformation
#pragma acc parallel loop num_gangs(1) num_workers(1) vector_length(1)
for (iter = 0; iter < M*N; iter++)
{  i = iter / N; j = iter % N;
   ...
}

// FPGA−specific transformation
i = 0; j = 0;
#pragma acc parallel loop num_gangs(1) num_workers(1) vector_length(1) firstprivate(i,j)
for (iter = 0; iter < M*N; iter++)
{  ...
   j++; if (j == N) { j = 0; i++; }
}
```

The FPGA-specific collapse optimization can be automatically applied any time loop collapsing occurs within a single work-item execution context. Because the row and column counters create dependencies within the loop, in multi-threaded contexts we revert to the traditional collapse transformation. We support application of the collapse optimization in conjunction with our reduction (Section 3.3) and sliding window (Section 3.4) optimizations. Integrating these optimizations allows application of the reduction (Section 3.3) and sliding window (Section 3.4) optimizations to a wider variety of benchmarks containing nested loops, without the performance penalty from OpenARC's traditional collapse transformation.

3.3 Reduction Optimization

Scalar reductions are common patterns used in many algorithms, such as the Rodinia Benchmark SRAD [8], to compute averages, find maximum values, and so on. Most high-performance approaches to scalar reduction involve a multi-threaded, tree-based parallelization scheme [15]. However, because a pipeline-parallel approach is often more efficient than a massively parallel approach when executing on an FPGA, an alternate FPGA-specific strategy to the scalar reduction is required.

Our reduction optimization compiler technique allows users to utilize single work-item kernels and shift registers in OpenACC using only previously existing directives. When using OpenACC to target an FPGA device, the user must first indicate a single work-item execution (Section 3.1). Within a single work-item compute region, the user can then annotate any loop with the OpenACC reduction directive and a supported reduction operation. Finally, to increase the performance the user can also append an optional unroll annotation, at the cost of additional FPGA resources. Under these circumstances, we can safely and efficiently apply our reduction optimization to implement the FPGA-specific shift-register based reduction. We can see an application of the OpenACC FPGA-specific sum reduction in Listing 3, with *N* referring to the desired level of replication. OpenARC currently supports addition, multiplication, and maximum and minimum value operations for FPGA-specific reductions.

For FPGA execution, scalar reductions are an example of programming patterns where the single work-item optimization (Section 3.1) alone does not increase performance relative to the traditional NDRange implementation. Because most floating point operations on an FPGA require multiple clock cycles, traditionally programmed scalar reductions perform poorly in the pipeline parallel model or single work-item approach (Section 3.1). This results

Listing 3: OpenACC sum reduction

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

from the pipeline stalling each iteration until the dependency on the reduction variable is resolved.

These pipeline stalls during loop execution are formalized in the Intel FPGA SDK documentation by the term *initiation interval*, or *II* [1]. The initiation interval specifically refers to the number of FPGA clock cycles that a pipeline is stalled to launch each successive iteration of a loop execution. A loop with several loop-carried dependencies, like scalar reduction, may have a high *II*, while a loop without dependencies may have a lower *II*. When executing in a loop-pipeplined single work-item approach, an *II* of 1 leads to optimal performance, indicating that successive iterations are launched every clock cycle.

The stand-alone single work-item approach does not outperform the multi-threaded tree-based method for scalar reductions on an FPGA. However, a sufficiently sized shift register in addition to this approach can significantly improve performance. In the shift-register approach to scalar reductions, we use the shift register to accumulate partial results as we iterate over the input array. This is followed by a standard reduction over the much smaller shift-register array. This approach increases the reduction variable dependence distance, relaxing the loop-carried dependency on the reduction variable. As a result, the reduction loop attains the desired *II* of 1. The exact shift-register size or depth required depends on the data type, reduction operation, and unrolling or replication factor.

Fortunately, the underlying Intel OpenCL compiler provides information about loop initiation intervals at compile time that can be used to determine an appropriate shift-register depth. With this information, we performed a number of tests with different reduction configurations, and made some general observations about the relationships between the data type, reduction operation, unrolling factor, and their effects on the shift register depth required to attain the desired *II* of 1. For example, on the Stratix V FPGA, we observe that without shift registers or loop unrolling, scalar reduction using single precision floating point addition leads to an *II* of 8 cycles, while using double-precision floating point multiplication leads to an *II* of 16 cycles. We also observe that loop unrolling acts as a multiplier to the initiation interval. For example, an unroll factor of 4 in the previous example leads to an *II* of 32 and 64 cycles, respectively. From these observations, we expect the following to be valid:

$$register\ depth \approx (operator\ latency) * (unroll\ factor) \tag{1}$$

In the equation above, *register depth* refers to the expected size of the shift registers required to attain an *II* of 1, and *operator latency* refers to the device-specific cost of the data type and operation used. This equation along with pre-calculated operator costs are used in the reduction optimization to calculate efficient shift register depths. However, after compiling reduction codes with different configurations, we find that the following unexpected equation holds true:

$$register\ depth \approx \frac{(operator\ latency) * (unroll\ factor)}{2} \tag{2}$$

Listing 4: OpenCL generated from OpenARC's FPGA-specific reduction transformation

```
1  #define REGISTER_DEPTH (8 * N) // OpenARC calculated shift−register depth
2
3  float shift_reg[REGISTER_DEPTH + 1] = {0}; //Create and initialize shift registers.
4
5  #pragma unroll N
6  for (int i = 0; i < SIZE; ++i) {
7    shift_reg[REGISTER_DEPTH] = shift_reg[0] + input[i]; //Perform partial reduction.
8
9    for (int j = 0; j < REGISTER_DEPTH; ++j)
10   {   shift_reg[j] = shift_reg[j + 1];   } //Shift values in shift registers.
11 }
12
13 #pragma unroll
14 for (int i = 0; i < REGISTER_DEPTH; ++i)
15 {   sum += shift_reg[i];   } //Perform final reduction on shift registers.
```

That is, by halving the expected minimum register depth required for an *II* of 1, we still attain an *II* of 1.

Because of the significant performance advantages of launching successive iterations every cycle and attaining an *II* of 1, under certain situations the underlying compiler can force an *II* of 1 by intentionally throttling or reducing the maximum FPGA circuit frequency for the entire offloaded kernel [1]. That is, to reduce the number of cycles stalled each iteration, the compiler can increase the amount of time per cycle. Although the ability to successfully launch iterations every cycle may benefit a specific loop, reducing the maximum circuit frequency can negatively affect performance in other regions of the offloaded kernel. Therefore, by default in the *Reduction Optimization*, we use the original equation without halving (Equation 1) to calculate the register depth. We currently hard-code operator latencies specific to the Stratix V, but these can easily be reconfigured for other devices.

In Listing 4, we see the OpenCL code generated by applying the reduction optimization to the OpenACC scalar reduction code from Listing 3, targeting a Stratix V FPGA. We see the OpenARC-calculated shift register depth is appropriately set to $8*N$ for floating point addition and an unroll factor of N (line 1). We next declare and initialize the shift registers, used for storing the accumulated partial sums (line 3). In the main loop, we now add each successive value to the oldest partial sum present in the shift registers (line 7), followed by a shift of the entire shift register array (lines 9–10). In this execution pattern, an assigned partial result is not accessed until it has been shifted through the entire register array, which relaxes the loop-carried dependency. After accumulating partial results over the entire array, we perform a final sequential reduction over the partial results in the shift registers (lines 13–15).

The OpenCL programming patterns generated by OpenARC (Listing 4) direct the underlying Intel OpenCL compiler to implement scalar reduction using single work-item execution and shift registers. With the FPGA-specific reduction optimization compiler transformation, we allow users to use existing OpenACC directives to generate these non-intuitive code patterns without specialized knowledge of shift registers, initiation intervals, and operator latencies.

3.4 Sliding Window Optimization

3.4.1 Basic Sliding Window Optimization. Structured-grid applications that compute grid-cell values using a surrounding neighborhood of cells, like stencil applications, can underperform in an FPGA single work-item approach because of redundant and expensive memory accesses. To avoid redundant memory accesses across iterations, a sliding window approach can be used. In the sliding window approach, we maintain the required neighborhood of relevant data in shift registers, shifting a new value in and an old value out each time an iteration begins. This approach allows us to efficiently forward data across iterations, allowing for data reuse. This also significantly reduces the number of memory operations required each iteration because we are able to access the neighboring values stored in the sliding window without pipeline delays.

We propose an OpenARC directive extension implementing the sliding window approach to address this performance issue. The *window* directive can be applied to loops within an OpenACC compute region, specifically where the loop reads from an input array, performs computations, and writes to an output array. However, only certain types of loops can benefit from application of the window directive, such as loops where each iteration contains several noncontiguous input array accesses, and loops where the same memory locations are redundantly accessed across different loop iterations. These programming patterns are common in stencil-based scientific codes.

The window directive imposes several restrictions for safe and efficient application. The optimization requires the neighborhood of cells accessed each iteration to be a fixed size. This fixed size is used to determine the size of the sliding window. The optimization also requires that the neighbor cells (array elements) accessed each iteration have constant offsets relative to the current iteration. For example, a loop that accesses a random assortment of neighbors each iteration would not be appropriate. Finally, in the current version of the sliding window optimization, the loop iteration variable must increase monotonically and have a step size of 1. These requirements ensure that the underlying OpenCL compiler can successfully and effectively infer and implement a sliding window approach using shift registers. OpenARC enforces these requirements by analyzing the loop control statement and requiring the index expressions of the input array to be affine, where the coefficient of the index variable is either *1* or *-1*. Violations of these requirements cause OpenARC to issue errors or warnings, depending on the offense.

In Listing 5, we show an example of a simple OpenACC stencil code with the window directive applied, where each iteration in a loop contains multiple noncontiguous input array accesses. Also, each element in the input array is accessed several times over multiple iterations. Because this example code meets the requirements mentioned above, it is safe to apply the window directive.

Using only the code provided in Listing 5, OpenARC can analyze the input array index expressions to calculate the following values needed to implement the sliding window transformation: neighborhood size (*NBD_SIZE*), window offset (*SW_OFFSET*), and reading offset (*READ_OFFSET*). The neighborhood size refers to the smallest number of contiguous array elements needed to encapsulate the neighbors required to compute one iteration. The window offset refers to the difference between the current value of the iteration variable and the minimum index value of neighbor cells for a given iteration. This offset is used when replacing input array accesses with accesses to the sliding window. Finally, the reading offset refers to difference between the maximum index of the current neighbors and the current index. This offset determines the index used to read from the input array each iteration and to calculate the number of initialization iterations required. These offsets are calculated internally using the following equations, where *index* refers to the index of a given iteration, and *max_index* and *min_index* refer to the largest and smallest values used to access the input array for that same iteration.

Listing 5: OpenACC with window directive

```c
#define ROWS ...
#define COLS ...

#pragma acc parallel loop num_gangs(1) num_workers(1) vector_length(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;
}
```

$$NBD\_SIZE = max\_index - min\_index + 1 \tag{3}$$

$$SW\_OFFSET = index - min\_index \tag{4}$$

$$READ\_OFFSET = max\_index - index \tag{5}$$

In the proposed sliding window optimization, calculating the above three equations is key; for this, we exploit the built-in symbolic analysis tools in OpenARC [9]. If the target loop body does not contain inner loops, the compiler symbolically calculates the differences between any two index expressions used for the input array accesses and derives the *min_index* and *max_index* expressions by symbolically comparing those differences. If the target loop body contains inner loops, the compiler applies a symbolic range analysis [9], which computes integer variables' value ranges at each program point to find the symbolic ranges of index variables of the inner loops. The calculated symbolic ranges are used to calculate the symbolic differences between two index expressions for the input array accesses.

Once the above three values (neighborhood size, window offset, and reading offset) are calculated and determined to be constant, the remaining step is to transform the target loop into a specific programming pattern so that the underlying OpenCL compiler is able to generate the hardware logic required for efficient sliding window execution.

In Listing 6, we show the resulting OpenCL code after the proposed sliding window optimization has been applied. We first see the results of OpenARC's calculations using the above equations (lines 5–7), followed by a declaration for the sliding window array (line 9). The initial value of the loop iteration variable is offset by the read offset (line 11). This allows for additional iterations to properly initialize the sliding window array, ensuring that the necessary neighborhood of values is present in the sliding window for the first non-initialization iteration. Within the loop, we first shift the sliding window each iteration (lines 12–13). Although this programming pattern is inefficient on non-FPGA platforms, it is

Listing 6: Transformed OpenCL sliding window code

```
1  #define ROWS ...
2  #define COLS ...
3
4  // OpenARC calculated values
5  #define NBD_SIZE    (2*COLS + 1)  // Neighborhood size
6  #define SW_OFFSET   (COLS)        // Window offset
7  #define READ_OFFSET (COLS)        // Read offset
8
9  float sw[NBD_SIZE]; //Create a sliding window array.
10
11 for (int index = −(READ_OFFSET); index < ROWS*COLS; ++index) {
12   for (int i = 0; i < NBD_SIZE − 1; ++i)
13   {  sw[i] = sw[i + 1]; } //Shift values in the sliding window array.
14
15   //Load an input array element into the sliding window array.
16   if (index + READ_OFFSET < ROWS*COLS)
17   {  sw[NBD_SIZE − 1] = input[index + READ_OFFSET]; }
18
19   if (index >= 0) { //Main computation body which uses sliding window
20     float N = sw[SW_OFFSET − COLS];
21     float S = sw[SW_OFFSET + COLS];
22     float E = sw[SW_OFFSET + 1];
23     float W = sw[SW_OFFSET − 1];
24     output[index] = sw[SW_OFFSET] + N + S + E + W;
25   }
26 }
```

required by the underlying OpenCL compiler to infer a shift register implementation of the intended sliding window array. We next read one value from the designated input array into the sliding window array, using the pre-calculated read offset (lines 16–17). Finally for every non-initialization iteration, we perform the calculations from the original loop (lines 20–24). We see that each read from the original input array has been replaced with one read from the sliding window array, and in the sliding window array index expressions, the iteration variable has been replaced with the window offset.

Although Listing 5 provides an ideal case for the window directive, the sliding window compiler transformation is robust enough to handle more complex indexing expressions, including expressions within nested loops containing multiple iteration variables. Also, algorithms without a separate output array that write computation results back to the original input array, like the Rodinia Benchmark NW [8], are handled by the compiler transformation using special-case code. The OpenARC window directive exemplifies the need for high-level programming constructs to enable widespread adoption of FPGA programming for HPC. This OpenACC directive extension enables programmers to use the performance-critical sliding window pattern on an FPGA without specific knowledge of shift registers, neighborhood sizes, and nonintuitive OpenCL programming patterns.

3.4.2 Sliding Window Optimization with Loop Unrolling.

3.4.2 Sliding Window Optimization with Loop Unrolling. Like the reduction optimization (Section 3.3), we can increase the performance of the shift-register–based sliding window optimization by applying loop unrolling. This unrolling can effectively increase the pipeline depth, allowing for a higher degree of pipeline parallelism and reducing the number of iterations required. This can decrease overall runtime but at the cost of increased FPGA resource usage. For applications with a low base resource usage, loop unrolling can be used to utilize unused resources while improving performance.

To enable loop unrolling in conjunction with the sliding window approach, users can add an additional *#pragma unroll UNROLL_FACTOR* annotation to any loop annotated with a window directive. Here *UNROLL_FACTOR* refers to the degree of unrolling and the number of times the sliding window logic should be replicated. We have integrated the sliding window approach with loop unrolling by creating an extension to the sliding window compiler transformation. Although we could simply lower the unroll directive to the underlying OpenCL compiler, we can further optimize this approach by separating the shift register and memory operations from the primary computation operations. This separation allows us to reduce the number of sliding window shifts and perform coalesced memory reads and writes, while only replicating code used in the primary computation. This models the approach used in the Intel OpenCL SKD FD3D design example [1].

We see the resulting OpenCL code generated from applying an optional loop unroll pragma along with the window directive in Listing 7. In this transformation, the size of the sliding window is dictated by a new compile-time constant *SW_SIZE* (line 8). The increased size of the sliding window is needed to accommodate the additional operations from loop unrolling. Because we now process multiple values each iteration, the loop step size is increased to *UNROLL_FACTOR* (line 12). Instead of shifting the sliding window one position each iteration, we now shift *UNROLL_FACTOR* positions (lines 13–14), thus reducing the overall number of shifts required. We then perform a coalesced read of *UNROLL_FACTOR* values from the input array (lines 16–19). We declare a statically sized array to temporarily store output values (line 21). The primary computation is then replicated by the enclosing fully unrolled loop (lines 23–33), with each access to the sliding window offset by the unrolled loop iteration index. Finally, we perform a coalesced write from the temporary array to the output array (lines 35–38).

The loop unrolling pragma can be applied to any loop optimized with the window directive as long as the unroll factor evenly divides the iteration space of the original main loop. For example, in Listing 7, the user-provided unroll factor must divide $ROWS*COLS$. Violation of this restriction results in an OpenARC compiler error.

3.5 Branch-Variant Code Motion Optimization

Among devices used as hardware accelerators, the concept of directly managing hardware logic generation at the programming level is unique to FPGAs. Because programming logic is mapped directly to FPGA hardware, programming patterns and coding styles that may only affect source code length on devices like GPUs or CPUs can make concrete differences in FPGA resource usage.

Loop-invariant code motion is a common computation-reduction optimization applied across all hardware devices. In the same fashion, we can apply branch-invariant code motion. This optimization normally would not lead to a performance benefit for more traditional devices like CPUs and GPUs because the number of operations executed remains unchanged. However, when compiling for FPGAs, logic from both branches is required to be implemented in hardware, leading to increased resource usage from the redundant code. Therefore, factoring out branch-invariant code can reduce the overall resources required to implement the hardware logic, and thus the Intel OpenCL compiler supports the branch-invariant code motion optimization.

To reduce the resource usage further, we propose a branch-variant code motion optimization, which transforms branch-variant codes and factors out codes with the same computation patterns.

Listing 7: Transformed OpenCL sliding window code with loop unrolling

```
#define ROWS ...
#define COLS ...

// OpenARC calculated values
#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.
 //Main body replicated by UNROLL_FACTOR
 #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 8, Listing 9, and Listing 10 illustrate how the proposed optimization works: Listing 8 shows an input code that contains branch-variant codes, so the traditional branch-invariant code motion optimization cannot be applied. However, if we transform the code into a form in Listing 9, codes with common computation patterns can be hoisted out of the conditional, as shown in Listing 10. The key part of this optimization is identifying a common computation pattern, which is an expression that exists in all branch bodies and performs the same sequence of computations with branch-variant operands. For example, in assignment expressions, a common computation pattern could be statements whose *lvalues* (i.e., an object that appears on the left side of the expression) is branch-invariant, whereas the right side of the expression is branch-variant. To identify these patterns, the compiler transforms the input conditional code into a form where non-constant operands in expressions within branch bodies, except for the left sides of the assignment expressions, which are replaced with temporary variables, even though variable assignments should be done in a specific order (Listing 9). Then, common computation patterns existing in all branch bodies are factored out of the conditional (Listing 10). If the left-hand side of an assignment statement is used as an input to a subsequent statement within the branch bodies, the assignment statement and subsequent statement can be factored out only if both statements are common computation patterns. Otherwise, the conditional should split into multiple conditionals. If the conditional itself is dependant on the common statement, the code motion optimization does not apply.

Listing 8: Input conditional

```
if (condition) {
 output += A[i] * B[i];
} else {
 output += A[i−1] * B[i−1];
}
```

Listing 9: Modified conditional

```
if (condition) {
 t1 = A[i]; t2 = B[i];
 output += t1 * t2;
} else {
 t1 = A[i−1]; t2 = B[i−1];
 output += t1 * t2;
}
```

Listing 10: After code motion

```
if (condition) {
 t1 = A[i]; t2 = B[i];
} else {
 t1 = A[i−1]; t2 = B[i−1];
}
output += t1 * t2;
```

FPGA resource usage can indirectly impact runtime performance in several ways. High resource usage can cause the hardware design to suffer from routing congestion, negatively affecting performance. Also, applications with higher base resource usage benefit less from loop unrolling techniques because they quickly exhaust FPGA resources even with small unroll factors. Section 5.5 presents an example of this behavior.

4 EXPERIMENTAL SETTING

4.1 Benchmarks

We use multiple benchmarks to test the viability, correctness, and performance of our FPGA-specific optimizations. Table 1 provides a summary of the benchmarks and their properties. The Sobel and FD3D benchmarks are taken from the Intel *High-Performance Computing Platform Examples* [1], and the HotSpot, SRAD, and NW benchmarks originate from the Rodinia Benchmark Suite 3.1 [8]. NW can be classified as a dynamic programming algorithm, but the rest can be classified as structured grid algorithms. We use the same input sizes and input parameters as the original Intel or Rodinia source codes, with the exception of FD3D. The original FD3D OpenCL code from Intel supports an input size of $504 \times 504 \times 504$ points by dividing the input into $64 \times 64 \times 504$ blocks. This blocking is necessary to meet FPGA resource usage requirements. However, because OpenARC does not currently support this type of custom blocking with OpenACC directives, we use an input size of $64 \times 64 \times 64$ single-precision floating-point values.

Base OpenACC versions of the Intel OpenCL SDK design examples were created directly from the OpenCL code by replacing the low-level OpenCL constructs with their high-level OpenACC counterparts and removing any FPGA-specific optimizations. A primary goal of this study is to reintroduce these optimizations using directives. Base OpenACC versions of the Rodinia benchmarks were sourced from the OpenARC repository. These benchmarks were adapted from the Rodinia 1.0 OpenMP benchmarks [16], although in this study we update them with any changes in Rodinia 3.1. The OpenCL benchmarks evaluated in Section 5.6 sourced directly from [1] and [25] without modification. The OpenMP benchmarks evaluated in Section 5.7 come from the Rodinia repository [8].

4.2 Hardware and Software Platform

On all platforms, OpenACC code is compiled using OpenARC V0.11 as the front end. We calculate energy (J) as runtime (s) × power (watts). Rutimes reported are the average of five executions.

4.2.1 FPGA Platform. The FPGA platform consists of a Nallatech 385 board containing a Stratix V 5SGSD5 FPGA. This FPGA has 172K ALMs, 690K registers, 2,014 M20K blocks, 1,590 DSP blocks, and 8 GB DDR3 device memory. Our host code executes on an Intel(R) Xeon(R) E5520 CPU, with a clock frequency of 2.27 GHz.

Table 1: Applications and optimizations (A: Collapse, B: Reduction, C: Sliding window, D: Sliding window with unrolling, and E: Code motion).

Application	Source	Description	Input Size	Data Type	Iterations	A	B	C	D	E
Sobel	Intel	Image edge detection algorithm	1,920 × 1,080	integer	1			X	X	
FD3D	Intel	3D finite difference computation	64 × 64 × 64	floating-point	100	X		X	X	
HotSpot	Rodinia	Compact thermal modeling	1,024 × 1,024	floating-point	10,000	X		X	X	X
SRAD	Rodinia	Speckle reducing anisotropic diffusion	4,096 × 4,096	floating-point	100	X	X	X	X	
NW	Rodinia	Needleman–Wunsch algorithm	4,096 × 4,096	integer	1	X		X		

We use the Intel FPGA SDK for OpenCL Offline Compiler V16.1.0 as the primary compiler and the back end runtime for OpenCL code. We obtain FPGA power usage estimations using the Quartus Power Analyzer [1] on fully compiled and routed applications. For a fair comparison with GPU and CPU power calculations, we add 2.34 W to the power estimations to account for the FPGA memory modules, as in [25]. Resource usage percentages are provided by the backend OpenCL compiler. The runtime variance was below 1.5% of the mean runtime for all applications, with most variances falling below 0.1%.

4.2.2 GPU Platform. For the GPU comparisons in Section 5.7, we use an NVIDIA Tesla K40c GPU. The OpenACC code relies on the NVIDIA CUDA compiler V8.0 as the back end. We calculate energy consumption using NVIDIA NVML to sample power usage every 10 ms.

4.2.3 CPU Platform. For the CPU comparisons in Section 5.7, we use a 16-core Intel(R) Xeon(R) E5-2683 v4 CPU with 2-way hardware multi-threading. We compile the OpenMP benchmarks using GCC 4.8.5 with the *-O2* flag, and execute them using 32 OpenMP threads. We collect CPU energy usage information using the Intel Running Average Power Limit (RAPL) interface.

5 EVALUATION RESULTS

5.1 Single Work-Item Optimization

By using directives to dictate a single work-item execution context, we can transform a traditional multi-threaded approach into an FPGA-specific pipeline-parallel single-work item approach. We evaluate the effectiveness of the single-work item approach by comparing it to the multi-threaded approach. Both approaches were programmed using OpenACC and executed on an FPGA. Figure 1 shows the FPGA performance of the two approaches across each benchmark. In this figure, the multi-threaded approach (*NDRange*) is used as a baseline, and the single work-item approach is compared in terms of speedup. We can see that for two applications (Sobel and HotSpot), applying the single work-item alone improves runtime performance. For the other applications (FD3D, SRAD, and NW) this optimization can actually degrade performance. However, in both cases the single work-item optimization enables us to apply the more advanced collapse, reduction, and sliding window optimizations, ultimately leading to higher performance than the multi-threaded approach for all benchmarks.

5.2 Collapse Optimization

The FD3D, HotSpot, SRAD, and NW benchmarks all contain nested loops within their main computation kernels. As a result of restrictions from the underlying OpenCL compiler, to apply the sliding

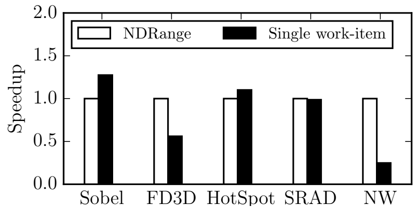

Figure 1: Multi-threaded and pipeline-parallel approaches on an FPGA.

window and unrolling optimizations, we first need to apply loop collapsing to remove the nested loops. Traditional loop collapsing techniques can be used to remove the nested loops; however, because the sliding window and other optimizations require a single work-item context, we can apply the single work-item FPGA-specific loop collapse optimization, replacing the division and modulus operations with more efficient addition operations along with row and column counters. Table 2 demonstrates the modest performance and resource usage improvements realized when applying the FPGA-specific collapse optimization in single work-item executions.

Table 2: FPGA-specific collapse clause performance comparison.

Application	Collapse Type	Runtime	Resource Usage (%)
FD3D	Standard	190.935 (ms)	39
FD3D	FPGA-specific	180.149 (ms)	36
HotSpot	Standard	47.882 (s)	32
HotSpot	FPGA-specific	47.371 (s)	30

5.3 Reduction Optimization

We use the SRAD benchmark to experimentally verify the observations in Section 3.3. First, we evaluate the relationships between different programmable parameters in the FPGA-specific single-work item scalar reduction. We isolate the reduction in SRAD, removing other computations in the benchmark. This results in a single-precision floating-point sum reduction over an input array of size 4096 × 4096. Removing the non-reduction code allows us to better observe the relationships between shift register depth, initiation interval, resource usage, and runtime. In the initial experiment, we use a constant unroll factor of 8 and manually vary the shift register depth.

In Figure 2, we see that increasing the shift register depth reduces the initiation interval, at the cost of increased resource usage. This reinforces observations about relationship between shift register depth and initiation interval introduced by Equation 1. As we increase the shift register depth, for certain depth values we observe an unexpected decrease in circuit frequency and a corresponding unexpected decrease in the initiation interval. These specific values indicate instances where the compiler has intentionally sacrificed or throttled the circuit frequency to attain a lower initiation interval. For example, in Figure 2, at register depths 16 and 32 we notice a decrease in II and a corresponding significant drop in circuit frequency. As the shift register depth continues to increase, the circuit frequency restabilizes, steadily increasing while the initiation interval remains unchanged.

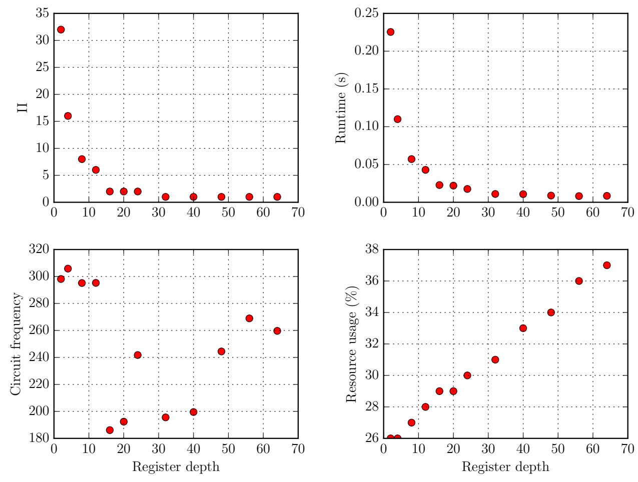

Figure 2: Initialization interval (II), circuit frequency, runtime, resource usage, and shift-register depth relationships.

In the second experiment, we evaluate the performance improvements by applying the single work-item FPGA-specific reduction optimization, compared to traditional approaches to scalar reduction. For this experiment we use the entire SRAD benchmark, as changes in the reduction implementation can also affect execution in other code regions. We compare three different approaches to scalar reduction: (1) a tree-based reduction, (2) a basic single work-item reduction, (3) and the FPGA-specific shift register reduction.

In Table 3, we see the basic single-threaded approach performs poorly compared to the hardware-agnostic multi-threaded tree-based reduction. Consequently, scalar reduction represents a code pattern where the single work-item optimization alone does not lead to improvements in performance. However, by combining the single work-item approach with the FPGA-specific shift-register based optimization, we can significantly outperform the other approaches to scalar reduction, but this performance comes at the cost of increased resource usage.

Table 3: SRAD FPGA reduction performance comparison.

Reduction Type	Runtime (s)	Resource Usage (%)
Multi-threaded Tree-based	31.053	45
Single Work-item	78.307	38
Single Work-item Shift Register	23.239	50

5.4 Sliding Window Optimization

5.4.1 Basic Sliding Window Optimization. The sliding window optimization (Section 3.4) can safely be applied to non-nested loops in a single work-item execution context. Therefore, by first applying the single work-item optimization (Section 3.1) and, when appropriate, the collapse optimization (Section 3.2), we can then apply the sliding window optimization to all five benchmarks.

We evaluate the effectiveness of the sliding window optimization for each benchmark by comparing a massively parallel multi-threaded approach, a basic pipeline-parallel single work-item approach, and a pipeline-parallel single work-item approach using a sliding window. We see significant performance improvements across all benchmarks when applying the sliding window optimization. The results of the sliding window evaluation are presented in Figure 3. The runtime for OpenACC implementation with only the single work-item optimization applied is used as a baseline, and the performance of the same OpenACC implementation with both the single work-item and sliding window optimizations applied is compared in terms of speedup.

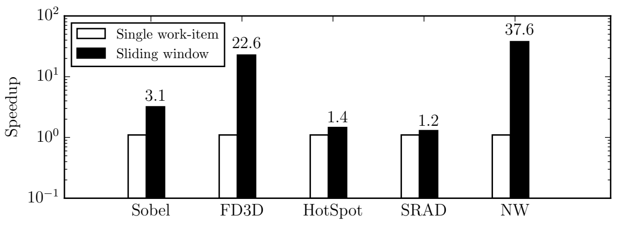

Figure 3: Comparison of a single work-item and a single work-item with shift-register sliding window approach on an FPGA.

We see that the performance of the NW benchmark improves exceptionally after applying the sliding window optimization. Unlike the other applications, NW reads from and writes to the same array, instead of writing to a separate output array. When executing in a single work-item context, this creates a memory dependency on the load and store operations to and from this array. This memory dependency causes successive iterations to be launched only once every 328 cycles, severely degrading performance, as we see in NW's basic single work-item approach. Applying the sliding window optimization to the single work-item implementation of NW shifts the memory dependency to a local data dependency. The sliding window allows successive iterations to be launched every cycle, significantly improving performance. Additionally, the expensive load operations for neighboring array elements are replaced with sliding window, or shift register, accesses.

We can also conjecture that the degree of speedup when applying the sliding window optimization is proportional to the size of the stencil computation. For example, the Sobel (9-point stencil) and FD3D (19-point stencil) realize a greater speedup than HotSpot and SRAD (4-point stencils).

5.4.2 Sliding Window Optimization with Loop Unrolling. We evaluate the effectiveness of using loop unrolling in conjunction with the sliding window optimization (Section 3.4.2) in each benchmark by comparing the performance of the single work-item sliding window approach with various degrees of loop unrolling applied.

The results of this evaluation are presented Figure 4. For each benchmark, the runtime of the application with the sliding window optimization without unrolling (Section 3.4.1) is used as a baseline. These times are annotated with a 1 above the bar. We compare each baseline to the same benchmark with different unrolling factors applied, visible over each bar. In general, we see that we can utilize previously unused FPGA resources to increase runtime performance. We can also see that performance improvements diminish with high unroll factors, as resources become scarce.

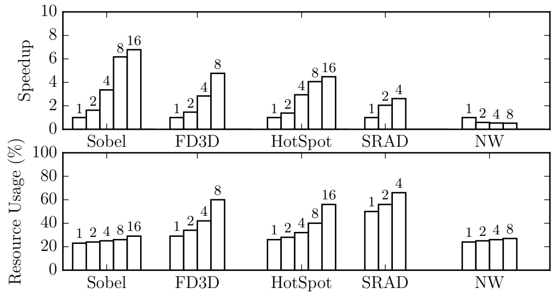

Figure 4: Sliding window optimization with different unroll factors applied.

We see in Figure 4 that the Sobel benchmark is an ideal candidate for loop unrolling. Because of the benchmark's low base resource usage, we can apply a high unrolling factor without exhausting FPGA resources. In contrast, applying loop unrolling to the NW benchmark actually degrades performance. As previously mentioned, the NW benchmark is unique in that the same array is used for both input and output values. This creates a dependency between loop iterations. We see performance benefits by using the sliding window optimization because of the replacement of expensive memory operations with shift register operations. However, we cannot increase the level of pipeline parallelism by unrolling the inner loop because the operations are serialized because of the loop dependency.

5.5 Branch-Variant Code Motion Optimization

We use the HotSpot benchmark to measure the performance and resource usage effects of the branch-variant code motion optimization. In this benchmark, a nine-way conditional is used to determine if the current index is an edge, corner, or neither. Several common operations occur within each branch of this conditional, including several multiplication and addition operations, and an expensive load operation. These common operations result in a relatively high base resource usage for the application. By applying branch-variant code motion, we can factor or hoist the common computation code from each branch, significantly reducing the number of multiplications, additions, and loads required to be mapped to the hardware logic. This results in a lower base resource usage.

Table 4 shows the results of executing HotSpot with the sliding window applied and different loop unroll factors with and without branch-variant code motion. We see that applying the resource usage reduction optimization does not directly or significantly affect runtime. However, as we unroll the inner loop, the version with the common operations in each branch of the conditional quickly encounters performance degradation due to resource exhaustion, but the optimized version with the hoisted code continues to see performance improvements.

Table 4: HotSpot code motion experiments.

	Base		Hoisted	
Unroll Factor	Resource Usage (%)	Runtime (s)	Resource Usage (%)	Runtime (s)
1	28	36.842	26	35.622
2	31	24.656	28	25.796
4	39	16.625	32	12.106
8	54	29.442	40	8.770
16	84	50.702	56	7.953

5.6 OpenACC and OpenCL Performance Comparison

To explore the viability of using a high-level language like OpenACC for FPGA programming, we compare the performance of all five benchmarks to the performance of those same benchmarks implemented directly in OpenCL. The OpenCL versions manually implement several of the same optimizations generated by the OpenARC compiler, but they also contain other FPGA-specific optimizations not currently supported by OpenARC, such as blocking and halo regions with sliding window arrays.

We can see the comparison between the best-performing OpenACC implementation and the manual OpenCL implementations in Figure 5. In this figure, the OpenACC runtimes are used as baselines, and the OpenCL runtimes are compared in terms of speedup. We can see that the OpenACC applications FD3D, HotSpot, and SRAD perform comparably to the manual OpenCL versions, with performances varying by less than a factor of 2.

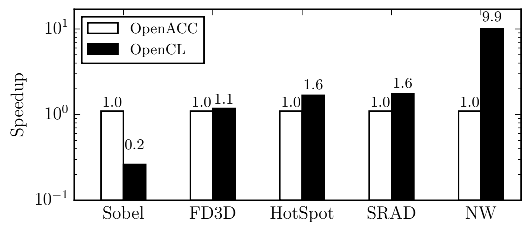

Figure 5: OpenACC and OpenCL with FPGA-specific optimizations.

The OpenACC version of the NW benchmark is around 10 times slower than the OpenCL version. This is because Rodinia's OpenCL version of NW, on which the FPGA-specific OpenCL version in is based, employs a significantly different programming pattern than the straightforward serial version of NW used to develop our OpenACC version. These patterns are currently unreproducible using our OpenACC directives for FPGA-specific optimizations, so NW represents a class of applications where our current FPGA-specific optimizations fail to realize the performance of manually tuned OpenCL. In contrast, the OpenACC version of the Sobel Filter actually outperforms the OpenCL version from the Intel SDK design examples. Although the manual code also uses a sliding window approach, it does not perform loop unrolling, resulting in the performance differences observed.

5.7 Performance and Power Comparisons of FPGAs, GPUs, and CPUs

To evaluate the viability of OpenACC FPGA programming, we compare OpenMP programs executed on a CPU (Section 4.2.3) and OpenACC programs executed on a GPU (Section 4.2.2) against OpenACC programs executed on an FPGA (Section 4.2.1). The results of this evaluation are shown in Figure 6. In this figure we compare runtimes, measured in terms of speedup from the CPU baseline, and energy consumption, measured in Joules and normalized to a CPU baseline of 1.

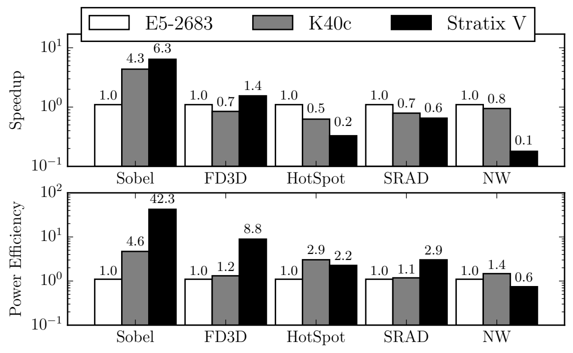

Figure 6: Comparison of OpenMP CPU executions, OpenACC GPU executions, and OpenACC FPGA executions.

The NW benchmark performs relatively poorly both in terms of runtime and power usage on the FPGA. This stems from the same algorithmic differences mentioned in Section 5.6. However for every other benchmark, the FPGA outperforms at least one of the other newer devices in either runtime or power usage when programmed using high-level frameworks.

6 RELATED WORK

6.1 High-level Programming Models for FPGAs

Several high-level programming models have been proposed to lower the barrier preventing wide-spread adaption of FPGAs further. The SRC-6 reconfigurable computer consists of multiple general-purpose CPUs and user-programmable Xilinx FPGAs [21]. The SRC compiler translates the users' functions written in high-level programming languages, such as C and FORTRAN, directly into VHDL. Xilinx SDSoC [3] supports compilation from a behavioral description written in C/C++/OpenCL to VHDL/Verilog RTL. Xilinx SDSoC provides a set of high-level directives including loop flattening, unrolling, and pipelining to improve performance by exploiting the parallelism between loop iterations. However, the programmers have to write additional code manually to exploit the shift registers in their reduction or sliding window codes, while our compiler automatically generates the shift register–friendly codes.

6.2 Loop Pipelining

Loop pipelining is an important compiler optimization in the HLS context used to fully exploit the deeply pipelined nature of the FPGA architecture [1, 3, 24]. Typically, the HLS tools [3, 7, 11] employ software pipelining techniques for hardware pipelining synthesis. If there are no inter-iteration dependencies and resource conflicts, the tools schedule the iterations of a loop to be continuously initiated at constant intervals [13]. Sliding window applications are a sub-domain of digital signal processing and highly amenable to loop pipelining on FPGAs [22]. Fowers et al. [10] and Zohouri et al. [25] perform extensive analyses of sliding window applications for different use cases by considering performance and energy consumption.

7 CONCLUSION

This paper presents a directive-based, high-level FPGA-specific optimization framework, consisting of a set of user directives and corresponding compiler optimizations, for more efficient FPGA computing. The proposed framework enables directive-based interactive programming by allowing users to provide important information to the compiler using directives. These directives instruct the compiler to automate FPGA-specific optimizations and allow control of important tuning options at a high level. We have developed several FPGA-specific optimizations in the OpenARC compiler framework, such as a reduction optimization to exploit shift registers, sliding window optimizations to enable more efficient pipelining, and branch-variant code motion optimization to reduce overall resource usage. We evaluate the proposed framework by porting five OpenACC benchmarks and comparing them against manually optimized OpenCL versions. The preliminary results show that the directive-based, semi-automatic optimizations can successfully realize performance comparable to the hand-written, low-level codes in many cases, and that OpenACC FPGA programs can have performance benefits over OpenACC GPU programs and OpenMP CPU programs in terms of runtime and power usage.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under contract number DE-AC05-00OR22725.

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.

This research was supported in part by an appointment to the Oak Ridge National Laboratory ASTRO Program, sponsored by the U.S. Department of Energy and administered by the Oak Ridge Institute for Science and Education.

REFERENCES

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

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

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

[4] S. R. Alam, P. K. Agarwal, M. C. Smith, J. S. Vetter, and D. Caliga. 2007. Using FPGA Devices to Accelerate Biomolecular Simulations. *IEEE Computer* 40, 2 (2007), 66–73.

[5] Amazon. [n. d.]. Amazon EC2 F1 Instances. ([n. d.]). https://aws.amazon.com/ec2/instance-types/f1/

[6] Matthew Aubury, Ian Page, Geoff Randall, Jonathan Saul, and Robin Watts. 1996. Handel-C Language Reference Guide. Oxford University Computing Laboratory.

[7] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*. 33–36.

[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 *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*.

[9] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. 2009. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *IEEE Computer* 42, 12 (2009), 36–42. http://www.ecn.purdue.edu/ParaMount/publications/ieeecomputer-Cetus-09.pdf

[10] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. 2012. A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-window Applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '12)*. 47–56.

[11] L. Gao, D. Zaretsky, G. Mittal, D. Schonfeld, and P. Banerjee. 2009. A software pipelining algorithm in high-level synthesis for FPGA architectures. In *2009 10th International Symposium on Quality Electronic Design*. 297–302.

[12] Intel. [n. d.]. Altera FPGA-Based Storage Reference Design. ([n. d.]). https://newsroom.intel.com/news-releases/altera-fpga-based-storage-reference-design-doubles-life-nand-flash/

[13] M. Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. 318–328.

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

[15] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 101–110.

[16] Seyong Lee and Jeffrey Vetter. 2014. 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*.

[17] Preeti Ranjan Panda. 2001. SystemC: A Modeling Platform Supporting Multiple Design Abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS '01)*. 75–80.

[18] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 13–24. http://dl.acm.org/citation.cfm?id=2665671.2665678

[19] Amit Sabne, Putt Sakdhnagool, Seyong Lee, and Jeffrey S. Vetter. 2015. Evaluating Performance Portability of OpenACC. In *Languages and Compilers for Parallel Computing*. 51–66.

[20] S. Seo, G. Jo, and J. Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*. 137–148.

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

[22] Greg Stitt, Eric Schwartz, and Patrick Cooke. 2016. A Parallel Sliding-Window Generator for High-Performance Digital-Signal Processing on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 9, 3, Article 23 (2016), 23:1–23:22 pages.

[23] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. 2017. ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 13–26. https://doi.org/10.1145/3140659.3080244

[24] Ritchie Zhao, Mingxing Tan, Steve Dai, and Zhiru Zhang. 2015. Area-efficient Pipelining for FPGA-targeted High-level Synthesis. In *Proceedings of the 52Nd Annual Design Automation Conference (DAC '15)*. Article 157, 157:1–157:6 pages.

[25] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. 35:1–35:12.