An Empirically Guided Optimization Framework for FPGA OpenCL*

Ahmed Sanaullah Rushi Patel Martin C. Herbordt
ECE Department; Boston University

Abstract—**FPGAs have been demonstrated to be capable of very high performance, especially power-performance, but generally at the cost of hand-tuned HDL code by FPGA experts. OpenCL is the leading industry effort in improving performance-programmability. But while it is recognized that optimizing OpenCL code using published best practices is critical to achieving good performance, even optimized code has so far rarely matched that of HDL code, or that available with competing technologies such as GPUs. In this paper we propose a series of systematic and empirically guided code optimizations that augment current best practices and substantially improve achieved performance. Our work characterizes and measures the impact of all of these optimizations. This enables programmers to not only follow a script when optimizing their own kernels, but also opens the way for the development of autotuners to perform optimizations automatically. We also demonstrate that, by applying these proposed code design practices to a number of parallel computing dwarfs, our optimized kernels outperform CPU and previous FPGA OpenCL implementations by $1.2\times$ and $5\times$ respectively. Moreover, our optimizations enable OpenCL FPGA codes to consistently achieve performance within striking distance of ($\approx 2\times$) best current equivalent code for GPUs and HDL. To the best of our knowledge, this is at least $2\times$ better than previous characterizations of OpenCL FPGA optimizations.**

I. Introduction

Programmability has been an ever-present challenge for FPGAs that has hindered adoption in production HPC. High Level Synthesis (HLS) tools provide an alternative to HDL programming by enabling designs to be expressed with higher levels of abstraction. Intel FPGA OpenCL [1] has received much attention and is our focus here. The methods used, however, are general and can easily be applied to other tools. Despite the advantages of OpenCL, it has still not managed to effectively bridge the performance-programmability gap for FPGAs. OpenCL kernels often end up having orders of magnitude worse performance than functionally equivalent HDL designs. Significant expertise in how the C-to-Hardware translation works is typically required for good performance.

Automatic compilation to a complex architecture is well-known to be an extremely difficult problem; even after decades of research it has been only partially solved. In HPC, programmers achieve high performance by augmenting the coding process, first, by integrating optimized libraries, and then, when these are not sufficient, by optimizing the code themselves [2]. Since this process is challenging even for experienced programmers, a vast area of research has grown up around automating it through autotuning [3].

We propose that an analogous approach be applied to coding OpenCL for FPGAs. We broadly categorize types of code optimizations in this domain into three sets: (i) Intel Best Practices (IBPs), (ii) Universal Code Optimizations (UCOs), and (iii) FPGA-Specific Optimizations (FSOs). **IBPs** refer to design strategies given in the Intel Best Practices guide [4]. These provide insight into how to effectively express hardware using OpenCL semantics. We separate these from UCOs and FSOs because IBPs are well-known to the FPGA OpenCL community and there have been several studies characterizing their behavior. **UCOs** consist of general approaches to optimizing programs that, to a large degree, are independent of the compute platform. Examples of UCOs include use of 1D arrays, records of arrays, predication, loop merging, scalar replacement, and precomputing constants. While described, e.g., in [2], they are largely missing from IBP documentation. **FSOs** consist of a number of FPGA-specific optimizations that typically augment IBPs. They are based on (a) obtaining a particular FPGA-specific mapping not found as an IBP, (b) facts stated in IBPs, but which must be leveraged and converted into optimizations, and (c) anti-IBPs, i.e., practices that are part of IBPs, but which (we have found) should actually be avoided.

In this paper we propose a systematic and empirically guided series of code optimizations for creating High Performance FPGA OpenCL kernels, using a combination of IBPs, UCOs, and FSOs. *These are aimed at giving the OpenCL compiler sufficient freedom to infer and exploit all possible forms and degrees of parallelism, along with more aggressive steps such as restructuring pipeline stages to minimize latency and resource overhead.* It is important to note here that, while some FSOs may be commonly known, since they can be based on UCOs and IBPs, our work is novel in their application to FPGAs. We also characterize and measure the impact of all optimizations. These results not only enable programmers to follow a script when optimizing their own kernels, but also open the way for the development of autotuners to perform optimizations automatically. We also demonstrate that, by applying these proposed code modifications to a number of parallel computing dwarfs, OpenCL FPGA code can consistently achieve performance within striking distance of (≈2x) best current equivalent code for GPUs and HDL. To the best of our knowledge, this is at least 2x better (for GPUs) than previous optimization studies of OpenCL for FPGAs.

*This work was supported in part by the NSF through Awards #CNS-1405695 and #CCF-1618303/7960; by the NIH through Award #1R41GM128533; by grants from Microsoft and Red Hat; and by Altera through donated FPGAs, tools, and IP.

The contributions in this paper are as follows:

- We propose systematic code optimizations for Intel FPGA OpenCL that can bridge the Performance-Programmability gap for FPGAs. Programmers can easily apply these optimizations to their kernels by following a script. Autotuners can also be developed to automate the process of high-performance hardware generation.
- To the best of our knowledge, our work is the first effort towards (a) characterizing optimizations beyond IBPs (UCOs and FSOs), (b) finding Anti-IBPs, and (c) the application of UCOs and FSOs to FPGAs.
- We show the incremental impact of each optimization using benchmarks for seven distinct parallel computing dwarfs: MMM, SpMV, FFT, CRC, Needleman-Wunsch (NW), Range-Limited MD, and Particle Mesh Ewald.
- We show that these optimized kernels outperform existing CPU and FPGA OpenCL codes by approximately $1.2\times$ and $5\times$ on average respectively. For certain benchmarks, we also outperform the GPU and Verilog codes; these include SpMV ($2.6\times$) and PME ($2.8\times$) for GPU, and SpMV ($2.1\times$) and Range Limited ($25\times$) for Verilog.

II. Previous Work

Most previous work related to FPGA OpenCL focuses on performing optimizations to achieve speedups for a number of applications. These speedups, however, are mostly with respect to either a CPU baseline code, or an OpenCL baseline code with no optimizations (e.g., [5]–[7]); this significantly reduces the utility of reported results. The space covered in previous studies characterizing FPGA OpenCL optimizations [6], [8]–[10] has mostly been limited. Often only a common set of simple optimizations is applied, with the initial code being written for a GPU with Multiple Work Item Kernels. Characterizations may be limited to varying the number of SIMD lanes or compute units.

One of the most important studies in this area, by Zohouri, *et al.* [11], implements both GPU based and FPGA-specific codes. The latter, referred to as loop-pipelined kernels, employ Single Work Item kernels and IBPs such as sliding windows and shift registers. Our work advances that of [11] in a number of ways. We have implemented many more optimizations; authors in [11] only use IBPs, while we characterize UCOs and FSOs as well. Performance values for best case implementations in [11] show an average of $35\times$ improvement over un-optimized baselines while our work shows an average speedup of $288\times$ for the final optimized version (with respect to the baseline). Moreover, the average speedup reported by Zohouri, *et al.* over GPUs is approximately $0.25\times$, while we achieve approximately $0.5\times$. Finally, unlike our reference GPU implementations, the GPU used in [11] does not have High Bandwidth Memory.

III. Optimizations

In this section, we present our design pattern agnostic OpenCL code optimizations that help the compiler infer and generate optimal architectures. There are seven code versions,

TABLE I
Summary of code versions and optimizations applied therein

Ver.	Optimizations	Type
0	(**GPU code** for porting to FPGA OpenCL)	—
1	**Single thread code with cache optimization**	IBP,FSO
2	Implement **task parallel computations in separate kernels** and connect them using channels	IBP
	Fully unroll all loops w/ *#pragma unroll*	IBP,UCO
	Minimize variable declaration outside compute loops – use temps where possible	IBP,UCO
	Use **constants** for problem sizes and data values – instead of relying on off-chip memory access	IBP,FSO,UCO
	Coalesce memory operations	IBP,UCO
3	Implement the entire computation within a **single kernel** and avoid using channels	FSO
4	**Reduce array sizes** to infer pipeline registers as registers, instead of BRAMs	FSO
5	Perform **computations in detail**, using temporary variables to store intermediate results	FSO,UCO
6	Use **predication** instead of conditional branch statements when defining forks in the data path	FSO,UCO

which are incrementally developed. Each version contains one or more applied optimizations. Version 1 is a cache optimized CPU code for the application of interest. Version 2 is obtained by applying the IBPs to the baseline code. Versions 3-6 involve applying a number of additional optimizations that, not only maximize opportunities for parallelism, but also reduce the complexity (and hence resource usage and latency) of the generated control and data planes. Table I summarizes the optimizations and their type (IBP, FSO, and/or UCO). We illustrate each set of optimizations through a running example, the Needleman-Wunsch (NW) benchmark.

Algorithm 1 Needleman Wunsch-V0

```
 1: int tx = get_local_id(0)
 2: __local int* temp
 3: __local int* ref
 4: Initialize temp from global memory
 5: barrier(CLK_LOCAL_MEM_FENCE);
 6: Initialize ref from global memory
 7: barrier(CLK_LOCAL_MEM_FENCE);
 8: for i = 1 : SIZE do
 9:     if tx≤i then
10:         compute t_idx_x and t_idx_y based on tx and i
11:         temp[t_idx_y][t_idx_x] =
12:                 max( temp[t_idx_y-1][t_idx_x-1] +
13:                 ref[t_idx_y-1][t_idx_x-1],
14:                 temp[t_idx_y][t_idx_x-1] - penalty,
15:                 temp[t_idx_y-1][t_idx_x] - penalty);
16:     barrier(CLK_LOCAL_MEM_FENCE);
17: barrier(CLK_LOCAL_MEM_FENCE);
18: for i = SIZE − 2 : 0 do
19:     Perform computations similar to above
20: barrier(CLK_LOCAL_MEM_FENCE);
21: Store temp to global memory
```

A. *Version 0: Sub-Optimal Baseline Code*

A popular starting point (e.g., in [9]) is kernels based on Multiple Work Items (MWI) such as is appropriate for GPUs. Advantages of starting here include ease of exploiting data parallelism through SIMD, and Compute Unit Replication (CUR), which is exclusive to MWI structures.

Algorithm 1 shows a V0-type kernel (based on [12]). The core operation is to populate a matrix using known values

of the first row and the first column. Each unknown entry is computed based on the values of its left, up, and up-left locations. This is achieved using loops which iterate in-order over all matrix entries. The max function is implemented using 'if-else' statements. In Algorithm 1, SIZE represents the dimension of blocks of matrix entries being processed.

B. *Version 1: Preferred Baseline Code (used for reference)*

Algorithm 2 Needleman Wunsch-V1

```
1: for i = 1 : Vector_B_Size do
2:     for j = 1 : Vector_A_Size do
3:         Out[i,j] = max( Out[i-1,j] - penalty,
4:                   Out[i-1,j-1] + ref[i,j] , Out[i,j-1] - penalty)
```

A less intuitive, but preferred, alternative is to use (as a baseline) single threaded CPU code. In particular, initial designs should be implemented as Single Work Item (SWI) kernels. While use of MWI kernels best matches the original purpose of OpenCL, enabling various expressions of parallelism, there are number of disadvantages for FPGAs.

Scheduling work-groups/items: Similar to GPUs, a scheduler is required to balance workloads and ensure all pipelines are kept filled. Using CUR increases scheduling effort since work-groups must be scheduled across the different compute units. Use of a simple scheduler can result in under-utilized pipelines. But more complex schedulers can require more resources and increase latency.

Under-utilization: CUR helps fill the chip in order to maximize use of available resource. However, because entire workgroups are assigned to each compute unit, the latter must divide the former perfectly; often, areas of the chip are mostly idle.

Static SIMD size: MWI kernels require a global SIMD size to be defined. This is sub-optimal for asymmetric pipelines where opportunities for data parallelism can vary frequently.

Global synchronization overhead: Compute units cannot communicate with each other directly. As with GPUs, data transfers between workgroups across compute units must go through off-chip memory. In addition to large synchronization overhead, the advantage of connectivity within FPGAs is lost.

Wrapper Overhead: Extra OpenCL wrapper logic is required to individually interface compute units with the host and external memory. Arbitration is often required, which incurs overheads similar to that of the scheduler.

Symmetry constraints: All work-items and work-groups are assigned equal amounts of private and local memory, respectively. This symmetry is not favorable when the workload varies as a factor of the work-item/work-group number.

In contrast, SWI kernels do not require a scheduler; pipelines are customized for a given application; parallelism is inferred and exploited in SWI kernels by analyzing the computational flow and dependencies; the compiler can employ an arbitrary number of registers and dimensions of BRAMs; communication is global because any set of pipeline registers can transfer data to each other; and wrapper overhead is substantially reduced.

The CPU-like baseline code should also be optimized for cache performance; this helps the compiler infer connectivity between parallel pipelines (i.e., whether data can potentially be directly transferred between pipelines instead of being stored in memory), improves bandwidth for on-chip data access, and efficiently uses the internal cache of Load Store Units which are responsible for off-chip memory transactions.

Algorithm 2 shows the preferred baseline kernel. The first row and column of the matrix are Vector_A and Vector_B respectively.

C. *Version 2: IBPs*

Algorithm 3 Needleman Wunsch-V2

```
1: N ← Size of systolic array
2: PE_k → Kernel Begin
3: int up, left, up_left, cached_up, cached_up_left
4: for i = k : N : Vector_A_Size do
5:     Initialize cached_up & cached_up_left
6:     for j = 1 : 1 : Vector_B_Size do
7:         left ← read_channel (PE_{k-1})
8:         up = cached_up
9:         up_left = cached_up_left
10:        cached_up = max(up - penalty,
11:                  left - penalty , up_left + ref [j,i])
12:        cached_up_left = left
13:        Out[j,i] = cached_up
14:        write_channel (PE_{k-1})← cached_up
```

Given the preferred baseline code, we then apply the following commonly used IBPs.

1) Multiple Task Parallel Kernels: Task parallelism is conventionally leveraged by implementing independent tasks as individual kernels. FPGA OpenCL implements direct connectivity between these kernels using channels (FIFOs) of variable data widths and depths, with support for both blocking and non-blocking operations. Channels are critical to performance in this approach, since all kernels operate concurrently and potentially large amounts of data are transferred between them. Availability of data transfers directly between pipelines located in separate kernels avoids off-chip memory accesses.

2) Fully unroll all loops: All loops must be fully unrolled whenever possible. Partial unrolls should be *avoided* if resources are limited since that can add significant complexity and overhead to pipelines. Rather, the problem can be folded by increasing the outer loop limit and doing less work per iteration. This allows the compiler to exploit all forms and degrees of parallelism at very fine granularity.

3) Minimizing State Register Usage: State registers are a special type of pipeline register whose value persists across algorithm iterations. State registers are inferred using variables declared outside the loops where they are used. The compiler generates feedback hardware for them to explicitly pass values to and from compute pipelines every iteration. Moreover, the state register hardware can also interface off-chip memory for loading initialization values. The initialization loop should be unrolled so that the state registers can be loaded in parallel.

We observe that the compiler is bound to ensure state register availability across subsequent iterations. For computations involving complex updates to the variable (as opposed to simple operations like increment/decrement), the compiler can

generate enough pipeline stages to prevent data from being forwarded stall-free to the next iteration. As a result, the pipeline is either stalled, or the operating frequency is lowered to accommodate a larger combinational path. There is also a resource overhead of implementing the feedback logic.

As a result, we attempt to minimize stage register use by moving variable declarations to within the outer loop whenever possible. Since the compiler is then no longer bound to ensure availability of these variables across iterations, it can perform more aggressive optimizations such as pipeline re-ordering.

4) Constant Arrays: Determining whether variables of known values should be initialized as constant arrays is a simple, yet important design decision since it impacts the implementation beyond reducing memory accesses. The compiler analyzes how the values are used in the context of corresponding computations and generates hardware accordingly. If the constant array has static accesses, i.e. persistently accessing the same value at a particular pipeline stage, the compiler can attempt to minimize the resources needed for that computation by pre-computing results if possible. For example, if a number is persistently multiplied with 1, the compiler will replace the DSP block with a delay module that simply passes the input to the output whilst ensuring data-paths continue to be synchronized. On the other hand, random accesses of the constant array by multiple pipelines can result in memory replication with a unique copy of the constant array generated for each pipeline. If the array was large enough, this could saturate board resources and the design will not compile. Therefore, while constant arrays should be used whenever possible to improve performance, their size should be minimized to ensure the design can successfully compile.

5) Coalescing: As with GPUs, coalescing is critical to effectively utilizing memory bandwidth. The impact is more so for FPGAs since the available DDR3 bandwidth is already small. Even a single un-coalesced read per iteration can result in stalls which leads to poor performance. It is preferable to fetch larger blocks of data in one access and initialize state registers, which in turn supply data to the pipelines.

Algorithm 3 shows the Needleman Wunsch kernel structure after we apply IBPs. Parallelism is exploited using a systolic array, with each Processing Element (PE) implemented in a separate kernel. Channels are used to connect PEs in a specified sequence. For each inner loop iteration, PEs compute consecutive columns within the same row. This ensures spatial locality for memory transactions. The drawback is data dependencies between kernels, which cannot be reliably broken down by the compiler since it optimizes each kernel as an individual entity. Thus, the overhead of synchronizing data paths can result in performance degradation.

D. Version 3: Single Kernel Design

In Version 3 we merge the IBP optimized task parallel kernels and declare all compute loops within the same kernel. This is because the compiler is still able to automatically able to infer task parallel pipelines, and having single kernel carries a number of advantages over multi kernel approaches.

Algorithm 4 Needleman Wunsch-V3

```
1:  N ← Size of systolic array
2:  int value[N+1], left[Vector_B_Size]
3:  left ← Vector_B
4:  for i = 1 : 1 : Vector_A_Size/N do
5:      base = f(i)
6:      value ← Vector_A[base:base+N+1]
7:      for j = 1 : 1 : Vector_B_Size do
8:          int up_left[N+1]
9:          for k = 2 : 1 : N + 1 do
10:             up_left[k] = value[k-1]
11:         value[1] = left[j]
12:         #pragma unroll
13:         for k = 2 : 1 : N + 1 do
14:             value[k] = max(value[k-1] - penalty,
15:               up_left[k] + ref[j, base+k] , value[k] - penalty)
16:         left[j] = value[N+1]
17:         Out ← value[2:N+1]
```

There is inherent global synchronization since all computations are tied to the same outer loop variable. Moreover, there is direct connectivity between pipelines which lowers communication overhead and enables pipelines stages, previously isolated due to channels, to be merged. Within a kernel, these loops should be placed in the same outer loop, which represents the algorithmic flow. Each loop iteration can correspond to a complete application stage, or have multiple variables derived from the loop iterator to emulate nested loops. Outer loops are used by the compiler to determine data-path latencies and synchronize them. Delay modules using FIFOs are added in case of a latency mismatch. Since the compiler only needs the correct variable values at the end of an iteration, it is not bound to follow explicit C code steps. Rather, pipelines can be reordered which can result in merged computation with reduced resource usage, as well as overlap of delay modules across pipelines to reduce/eliminate them. This approach also reduces the control logic for tracking application progress, which could be as simple as a counter-comparator circuit. Nested loops are typically avoided since there are a small number of stall cycles after each outer loop iteration.

Algorithm 4 shows the kernel structure for implementing the systolic array as a single kernel. The compiler can now optimize the entire computation, as opposed to individual PEs. Synchronization overhead is also reduced since almost all computation is tied to a single loop variable (j). Nested loops are used since, in this particular case, the cost of initiation intervals is outweighed by the reduction in resource usage. This is because the compiler was unable to infer data access patterns when loops were coalesced.

E. Version 4: Reduced Array Sizes

OpenCL limits the size of a register array in SWI kernels. If this limit is exceeded, the arrays are converted to BRAM based storage. While this is acceptable for data memory/cache, inferring pipeline registers as BRAMs can have significant drawbacks on the design. Since BRAMs cannot source and sink data with the same throughput as registers, barrel shifters

and memory replication is required which drastically increases resource usage. Moreover, the compiler is also unable to launch stall-free iterations of compute loops due to memory dependencies. Our solution is to break large arrays corresponding to intermediate variables into smaller ones. Ideally, arrays should be avoided altogether where ever possible. Instead, scripts can be used to create and reference individual variables.

Algorithm 5 Needleman Wunsch-V4

```
 1: N ← Size of systolic array
 2: int value_1, value_2 ... value_N_plus_1
 3: int left [Vector_B_Size]
 4: left ← Vector_B
 5: for i = 1 : 1 : Vector_A_Size/N do
 6:     base = f(i)
 7:     value_1 ← Vector_A[base]
 8:             ↓
 9:     value_N_plus_1 ← Vector_A[base+N+1]
10:     for j = 1 : 1 : Vector_B_Size do
11:         int up_left_2 ... up_left_N_plus_1
12:         up_left_2 = value_1
13:                 ↓
14:         up_left_N_plus_1 = value_N
15:         value_1 = left [j]
16:         value_2 = max(value_1 - penalty,
17:             up_left_2 + ref[j,base+2], value_2 - penalty)
18:                 ↓
19:         value_N_plus_1 = max(value_N - penalty,
20:                 up_left_N_plus_1 + ref[j,base+N+1],
21:                 value_N_plus_1 - penalty)
22:         left[j] = value_N_plus_1
23:         Out ← value_2 ... value_N_plus_1
```

Algorithm 5 shows the kernel structure for inferring pipeline registers as registers. All arrays are expressed as individual variables, with the exception of local storage of Vector_B in 'left,' which has low throughput requirements.

F. Version 5: Detailed Computations

The OpenCL compiler does not reliably break down large computations being assigned to a single variable into intermediate stages. As a result, dependency across iterations can be considered as the worst case, i.e., the overall result of the computation is required for the next iteration's first evaluation in the computation chain. The compiler thus stalls the pipeline for the required number of cycles to address this. Our solution is to do computations in as much detail as possible by storing results in intermediate variables. This helps the compiler infer potential pipeline stages with forwarding hardware. Memory dependencies are removed and the critical path is decreased. If the pipeline is already optimal, these variables will be synthesized away and resource is not wasted.

Algorithm 6 shows the kernel structure after performing computations in detail with a number of intermediate variables added. The 'max' function is also explicitly implemented.

Algorithm 6 Needleman Wunsch-V5

```
 1: N ← Size of systolic array
 2: int value_1, value_2 ... value_N_plus_1
 3: int left [Vector_B_Size]
 4: left ← Vector_B
 5: for i = 1 : 1 : Vector_A_Size/N do
 6:     base = f(i)
 7:     value_1 ← Vector_A[base]
 8:             ↓
 9:     value_N_plus_1 ← Vector_A[base+N+1]
10:     for j = 1 : 1 : Vector_B_Size do
11:         int a_2 = value_1 + ref[j,base+2];
12:         value_1 = left[j]
13:
14:         int b_2 = value_1 - penalty
15:         int a_3 = value_2 + ref[j,base+3];
16:         int c_2 = value_2 - penalty
17:
18:         if ((a_2 ≥ b_2) && (a_2 ≥ c_2))
19:                 value_2 = a_2
20:         else if ((b_2 > a_2) && (b_2 ≥ c_2))
21:                 value_2 = b_2
22:         else
23:                 value_2 = c_2
24:
25:         int b_3 = value_2 - penalty
26:         int a_4 = value_3 + ref[j,base+4];
27:         int c_3 = value_3 - penalty
28:                 ⋮
29:         left[j] = value_N_plus_1
30:         Out ← value_2 ... value_N_plus_1
```

G. Version 6: Predication

We optimize conditional operations by explicitly specifying architecture states which ensure the validity of the computation. Since hardware is persistent and will always exist once synthesized, we avoid using conditional branch statements. Instead, variable values are conditionally assigned such that the output of invalid operations is not committed and hence does not impact the overall result. Examples of this include zeroing out variables and pointer arithmetic. Algorithm 7 shows the 'if-else' operations replaced with conditional assignments.

Algorithm 7 Needleman Wunsch-V6

```
 1:         ⋮
 2: int a_2 = value_1 + ref[j,base+2];
 3: value_1 = left[j]
 4:
 5: int b_2 = value_1 - penalty
 6: int a_3 = value_2 + ref[j,base+3];
 7: int c_2 = value_2 - penalty
 8:
 9: int d_2 = (a_2 > b_2) ? a_2 : b_2
10: value_2 = (c_2 > d_2) ? c_2 : d_2
11:         ⋮
```

IV. RESULTS

In this section, we present the results of applying our optimizations to the benchmarks listed in the previous section. We not only evaluate the impact of individual optimizations to each benchmarks, but also demonstrate the importance of selecting the correct baseline code structures. We also compare the performance of generated pipelines against other

TABLE II
BENCHMARK SUMMARY – NOT ALL OPTIMIZATIONS ARE APPLICABLE TO ALL CODES

Benchmarks	Dwarf	Problem Size	V-1	V-2	V-3	V-4	V-5	V-6
NW	Dynamic Programming	16K x16K Integer Table	✓	✓	✓	✓	✓	✓
FFT	Spectral Methods	64 point Radix-2 1D FFT, 8192 Vectors	✓	✓	✓	✓	✓	
Range Limited	N-Body	180 particles per cell, 15% pass rate	✓	✓	✓	✓	✓	✓
PME	Structured Grids	1,000,000 Particles, 32^3 grid, 3D Tri-Cubic Interpolation	✓	✓	✓			✓
MMM	Dense Linear Algebra	1K x 1K Matrix, Single Precision	✓	✓	✓		✓	
SpMV	Sparse Linear Algebra	1K x 1K Matrix, Single Precision, 5%-Sparsity, NZ=51122	✓	✓	✓		✓	✓
CRC	Combinational Logic	100MB CRC32	✓	✓	✓		✓	✓

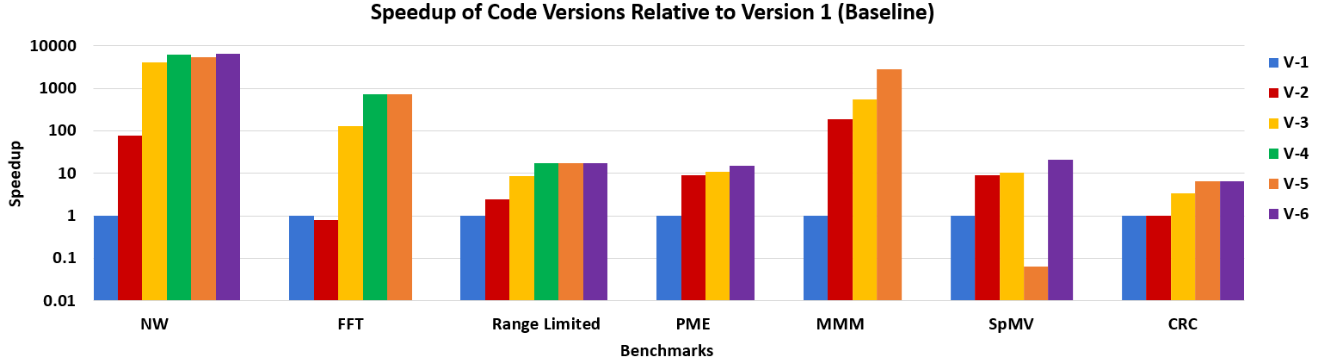

Fig. 1. Impact of systematic application of proposed optimizations to a cache-optimized CPU baseline code. In almost all cases, every subsequent code version shows increasing performance, with up to orders of magnitude better performance possible for full optimized kernels over ones with only IBPs (V-2).

platforms/approaches such as CPU, GPU, Verilog, and exiting FPGA OpenCL implementations. To ensure fairness, values used for comparisons are all either obtained from literature, or from implementations of available source codes/libraries.

A. Benchmarks

We tested our approach using the Needleman Wunsch (NW) [13], [14], Fast Fourier Transform (FFT) [15], [16], Range Limited Molecular Dynamics (Range Limited) [17], [18], Particle Mesh Ewald (PME), Dense Matrix Matrix Multiplication (MMM), Sparse Matrix Dense Vector Multiplication (SpMV), and Cyclic Redundancy Check (CRC) benchmarks. Table II provides a summary of these benchmarks, their associated dwarfs, tested problem sizes, and applicable code versions. Blank table entries indicate that the version was not created since the corresponding optimizations were not possible in the context of the application. For example, *selective operations* does not apply to FFT since there is a single, fixed data path.

B. Hardware Specifications

We implement the designs using an Altera Arria 10AX115H3F34I2SG FPGA and Altera OpenCL SDK 16.0. The FPGA has 427,200 ALMs, 1506K Logic Elements, 1518 DSP blocks, and 53Mb of on-chip storage. For GPU implementations, we use the Tesla P100 PCIe 12GB GPU with CUDA 8.0. It has 3584 Cuda cores and peak bandwidth of 549 GB/s. CPU codes are implemented on a 14 core 2.4 GHz Intel Xeon E5-2680v4 with Intel C++ Compiler v16.0.1.

C. Impact of Optimizations

Figure 1 shows the results of individual optimizations. In almost all cases, the same trend can be observed where IBPs (V-2) only result in a fraction of the speedup possible. The shortcomings of IBPs are especially highlighted in CRC

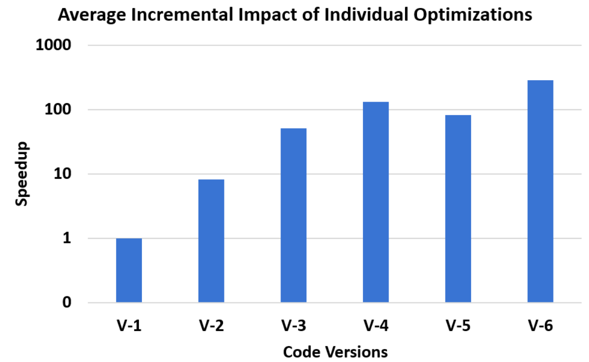

Fig. 2. Performance for different code versions, obtained by averaging the speedup of all applicable benchmarks.

and FFT. For the former, the V-2 code for CRC has the same performance as the baseline. Results for the latter, FFT, are worse still since implementing multi-kernel designs with channels results in a lower performance than even the baseline. On the other hand, by applying the additional optimizations on top of V-2, the achieved performance is improved by orders of magnitude. The average impact of individual optimizations is shown in Figure 2. Generally, each successive set of optimizations applied results in increasing performance. The exception is V-5. This is due to higher execution times of V-5 for NW and SpMV. In both cases, performing computations in as much detail as possible results in the use of conditional statements that outweigh benefits of the optimization. Once these statements are removed in V-6, the speedup increases.

D. Impact of Initial Code Structure

We highlight the importance of selecting the correct initial code structure for kernel development by implementing MMM kernels with three different approaches (Figure 3). MMM-JIK

is the naive approach: the outer loops, i and j, select the row and column of two matrices A and B, respectively. The innermost loop, k, iterates over all elements in the selected row and column. j is selected as the outermost loop so that the column vector, which has a poor access pattern, is only read once from global memory, stored in local variables and reused.

MMM-KIJ swaps the loops, moving the k loop to the outermost location in the hierarchy. It is unable to outperform MMM-JIK, despite a better access pattern, because of the writes in the inner loop. Finally, MMM-Block is a blocked version that targets high data reuse and minimal memory access. It is thus able to achieve the lowest execution time. In the case of MMM-Block, we also demonstrate that despite having the worst performance in V-1, the final version V-6 has the lowest execution time since it benefits more from the applied optimizations. On the other hand, while improvements are seen for the other two versions as well, the benefits are relatively small.

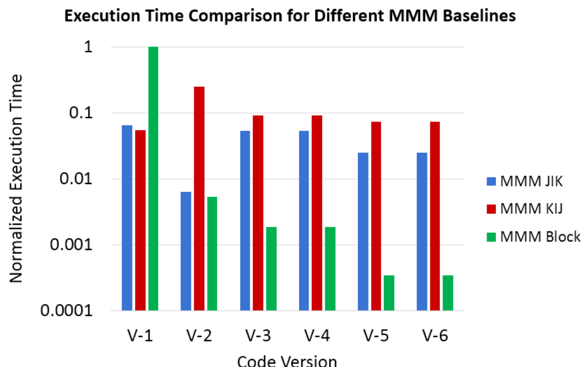

Fig. 3. Optimizations performed for MMM versions with different initial code structures. The observed trend is that better memory access patterns of a given baseline results in a larger impact of each optimizations and lower overall execution time.

TABLE III
REFERENCES FOR EXISTING IMPLEMENTATIONS

Benchmarks	CPU	GPU	Verilog	OpenCL
NW	Rodinia* [12]	Rodinia* [12]	Benkrid [14]	Zohouri [11]
FFT	MKL* [19]	cuFFT* [20]	Sanaullah [16]	Altera* [15]
Range Limited	-	-	Yang [18]	Yang [18]
PME	Ferit [21]	Ferit [21]	Sanaullah [22]	-
MMM	MKL* [19]	cuBLAS* [23]	Shen [24]	Spector* [25]
SpMV	MKL* [19]	cuSparse* [26]	Zhou [27]	OpenDwarfs* [9]
CRC	Brumme* [28]	-	Anand [29]	OpenDwarfs* [9]

E. Overall Performance

To demonstrate the overall effectiveness of the approach, we compare the performance of the optimized kernels against existing CPU, GPU, Verilog, and FPGA-OpenCL implementations. Table III lists the references for these designs; they are either obtained from the literature or implemented using available source code/libraries. The latter is illustrated using an asterisk (*). Verilog FFT measurement from [16] has been extended to include off-chip access overhead. Figure 4 shows the average speedup achieved over the CPU code while Figure 5 shows the normalized execution times for all implementations.

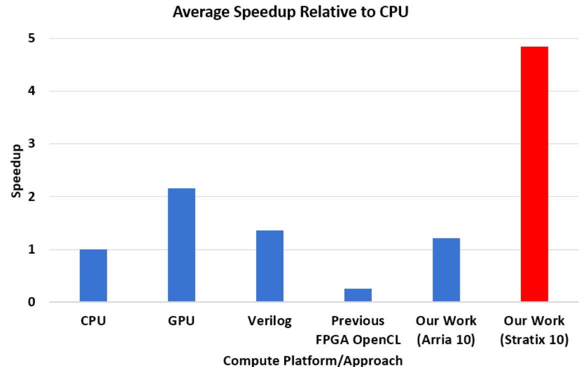

Fig. 4. Average speedup wrt CPU across all applicable benchmarks.

From the results, we observe that our work outperforms multi-core CPU implementations by approximately $1.2\times$ due to the performance of codes written using Intel MKL. We have also achieved an average of approximately $5\times$ lower execution time than existing FPGA OpenCL work. The exception is FFT, where we have a $3.7\times$ higher execution time than the reference FPGA OpenCL implementation. Similar to MKL, this reference design was developed by Intel engineers, who are familiar with the low level details of the C-to-HDL translation, and the optimizations performed cannot be applied in a general way to all applications.

The GPU speedup of $2.4\times$ relative to our work is due to the use of a high-end GPU, Tesla P100, and a medium-end FPGA, Arria-10. We therefore also provide an estimate of a high end FPGA performance, Stratix-10, using a conservative factor of $4\times$ that accounts for an increase in resource only. Results show that the optimized kernels on Stratix-10 are expected to outperform GPU designs by 65%, on average. Comparison with existing Verilog implementations show that the kernels are, on average, within 12% of hand-tuned HDL. This demonstrates that the optimizations are successfully able to bridge the performance-programmability gap for FPGAs and deliver HDL-like performance using OpenCL.

V. CONCLUSION

In this paper, it is shown that the Performance-Programmability gap of FPGA OpenCL can be reduced and that performance comparable to GPUs, and even Verilog-based implementations, can be achieved. By using CPU code as a starting point and performing a series of simple optimizations that augment common best practices, highly efficient FPGA hardware can be generated. The performance impact of all of the optimizations is characterized using a number of parallel computing dwarfs. The overall impact of this characterization is that programmers can now follow a script for optimizing their FPGA OpenCL kernels and achieve HPC performance. Moreover, auto-tuners can be developed to automate the generation of efficient hardware. The optimized kernels have

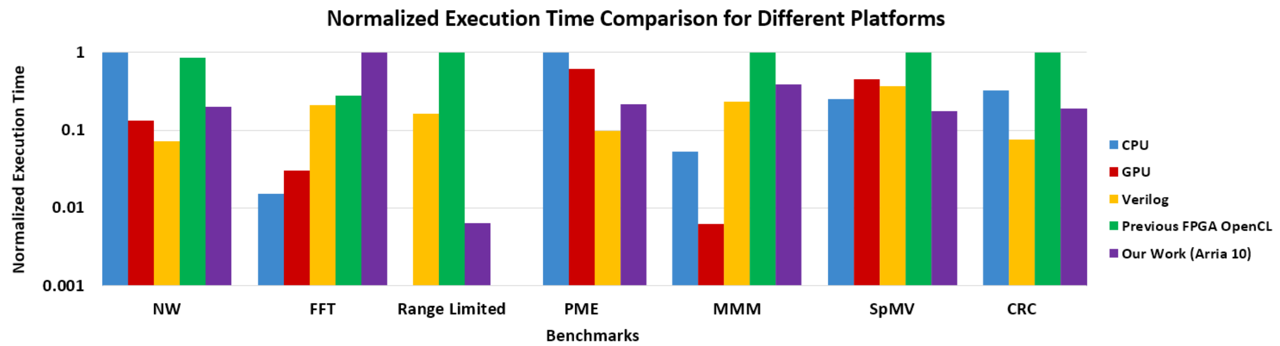

Fig. 5. Performance of Our Work as compared with existing CPU, GPU, Verilog and FPGA OpenCL implementations. Our work outperforms CPU and OpenCL for most of the benchmarks. Moreover, we also achieve speedups over GPU (SpMV, PME) and Verilog (SpMV, Range Limited).

been shown to outperform CPU and previous FPGA OpenCL designs. Using Stratix-10, they can also outperform a high-end GPU. Most importantly, it is demonstrated that the optimizations can, on average, achieve performance values within 12% of hand-tuned HDL code.

In related work [30], [31] we have described methods for isolating and simulating HDL compute pipelines generated by the OpenCL compiler, in order to rapidly optimize the kernel code. Moreover, if required, this HDL can be re-interfaced and integrated into existing HDL systems that serve as an alternative to the standard OpenCL Board Support Package. An in-depth case study is presented in [16].

References

[1] "Intel FPGA SDK for OpenCL," https://www.altera.com/ products/ design-software/ embedded-software-developers/ opencl/ developer-zone.html, accessed: 2017-01-16.

[2] S. Chellappa, F. Franchetti, and M. Pueschel, "How To Write Fast Numerical Code: A Small Introduction," in *Generative and Transformational Techniques in Software Engineering II, Lecture Notes in Computer Science v5235*, 2008, pp. 196–259.

[3] J. Moura, M. Pueshel, D. Padua, and J. Dongarra, Eds., *Proceedings of the IEEE Special Issue: Program Generation, Optimization, and Platform Adaptation*. IEEE, 2005.

[4] "Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide," https://www.intel.com/ content/ www/us/en/ programmable/ documentation/ mwh1391807516407.html, accessed: 2018-07-30.

[5] A. Abedalmuhdi, B. E. Wells, and K.-I. Nishikawa, "Efficient Particle-Grid Space Interpolation of an FPGA-Accelerated Particle-in-Cell Plasma Simulation," in *Proc Field-Programmable Custom Computing Machines*, 2017, pp. 76–79.

[6] C. Rodriguez-Donate, G. Botella, C. Garcia, E. Cabal-Yepez, and M. Prieto-Matías, "Early Experiences with OpenCL on FPGAs: Convolution Case Study," in *Field-Programmable Custom Computing Machines*, 2015, pp. 235–235.

[7] D. Weller, F. Oboril, D. Lukarski, J. Becker, and M. Tahoori, "Energy Efficient Scientific Computing on FPGAs using OpenCL," in *International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 247–256.

[8] Z. Jin, K. Yoshii, H. Finkel, and F. Cappello, "Evaluation of the Single-precision Floating Point Vector Add Kernel Using the Intel FPGA SDK for OpenCL," Argonne National Laboratory, Tech. Rep., 2017.

[9] K. Krommydas, A. E. Helal, A. Verma, and W.-C. Feng, "Bridging the Performance-Programmability Gap for FPGAs via OpenCL: A Case Study with Opendwarfs," Department of Computer Science, Virginia Polytechnic Institute & State University, Tech. Rep., 2016.

[10] Z. Jin, K. Yoshii, H. Finkel, and F. Cappello, "Evaluation of the OpenCL AES Kernel using the Intel FPGA SDK for OpenCL," Argonne National Laboratory, Tech. Rep., 2017.

[11] H. R. Zohouri, N. Maruyamay, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs," in *Int. Conf. for High Performance Computing, Networking, Storage and Analysis, SC16*, 2016, pp. 409–420.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE Int Symp on Workload Characterization*, 2009.

[13] T. VanCourt and M. Herbordt, "Families of FPGA-based accelerators for approximate string matching," *Microprocessors and Microsystems*, vol. 31, no. 2, pp. 135–145, 2007.

[14] K. Benkrid, Y. Liu, and A. Benkrid, "A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 4, pp. 561–570, 2009.

[15] "FFT (1D) Design Example," https://www.altera.com/support/support-resources/design-examples/design-software/opencl/fft-1d.html, accessed: 2018-01-16.

[16] A. Sanaullah and M. Herbordt, "FPGA HPC using OpenCL: Case Study in 3D FFT," in *Proc. Int. Symp. on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2018.

[17] M. Chiu, M. Khan, and M. Herbordt, "Efficient calculation of pairwise nonbonded forces," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2011.

[18] C. Yang, J. Sheng, R. Patel, A. Sanaullah, V. Sachdeva, and M. Herbordt, "OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics," in *IEEE High Perf. Extreme Computing Conf.*, 2017.

[19] Intel, "Computing an FFT," https://software.intel.com/en-us/mkl-developer-reference-c-computing-an-fft, 2018.

[20] C. Nvidia, "CuFFT Library," 2010.

[21] F. Büyükkeçeci, O. Awile, and I. F. Sbalzarini, "A Portable OpenCL Implementation of Generic Particle–Mesh and Mesh–Particle Interpolation in 2D and 3D," *Parallel Computing*, vol. 39, no. 2, pp. 94–111, 2013.

[22] A. Sanaullah, A. Khoshparvar, and M. Herbordt, "FPGA-Accelerated Particle-Grid Mapping," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2016.

[23] Nvidia, "CUBLAS Library," *NVIDIA Corp, Santa Clara, CA*, 2008.

[24] J. Shen, Y. Qiao, Y. Huang, M. Wen, and C. Zhang, "Towards a Multi-Array Architecture for Accelerating Large-Scale Matrix Multiplication on FPGAs," in *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 1–5.

[25] Q. Gautier, A. Althoff, P. Meng, and R. Kastner, "Spector: An OpenCL FPGA Benchmark Suite," in *Int Conf Field-Programmable Tech*, 2016.

[26] Nvidia, "CuSparse Library," *NVIDIA Corp, Santa Clara, CA*, 2014.

[27] L. Zhuo and V. K. Prasanna, "Sparse Matrix-Vector Multiplication on FPGAs," in *Proc. Int. Symp. on Field-Programmable Gate Arrays*, 2005.

[28] S. Brumme, "Fast CRC32," http://create.stephan-brumme.com/crc32/, 2018.

[29] P. A. Anand *et al.*, "Design of High Speed CRC Algorithm for Ethernet on FPGA using Reduced Lookup Table Algorithm," in *India Conference (INDICON), 2016 IEEE Annual*. IEEE, 2016, pp. 1–6.

[30] A. Sanaullah and M. Herbordt, "Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolow," in *IEEE High Perf. Extreme Computing Conf.*, 2018.

[31] A. Sanaullah, C. Yang, D. Crawley, and M. Herbordt, "SimBSP: Enabling RTL Simulation for Intel FPGA OpenCL Kernels," in *Proc. Heterogeneous High Performance Reconfigurable Computing*, 2018.