Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolflow

Ahmed Sanaullah Martin C Herbordt
Deptartment of Electrical and Computer Engineering, Boston University

***Abstract*—Improved support for OpenCL has been an important step towards the mainstream adoption of FPGAs as compute resources. Current research has shown, however, that programmability derived from use of OpenCL typically comes at a significant expense of performance, with the latter falling below that of hand-coded HDL, GPU, and even CPU designs. This can primarily be attributed to 1) constrained deployment opportunities, 2) high testing time-frames, and 3) limitations of the Board Support Package (BSP). We address these challenges by penetrating the toolflow and utilizing OpenCL-generated HDL (OpenCL-HDL), which is created as an initial step during the full compilation. OpenCL-HDL can be used as an intermediate stage in the design process to get better resource/latency estimates and perform RTL simulations. It can also be carved out and used as a building block for an existing HDL system. In this work, we present the process of generating, isolating, and re-interfacing OpenCL-HDL. We first propose a kernel template which reliably exploits parallelism opportunities and ensures all compute pipelines are implemented as a single HDL module. We then outline the process of identifying this module from the thousands of lines of compiler generated code. Finally, we categorize the different types of interfaces and present methods for connecting/bypassing them in order to support integration into an existing HDL shell. We evaluate our approach using a number of benchmarks from the Rodinia suite and Molecular Dynamics simulations. Our OpenCL-HDL implementations of all benchmarks show an average of 37x, 4.8x, and 3.5x speedup over existing FPGA/OpenCL, GPU, and FPGA/Verilog designs, respectively. We demonstrate that OpenCL-HDL is able to deliver hand-coded HDL-like performance with significantly less development effort and with competitive resource overhead.**

***Index Terms*—FPGA; OpenCL; Rodinia**

I. Introduction

FPGAs enable developers to design custom systems that leverage application-specific optimizations. Unlike GPUs, FPGAs are not constrained to data parallelism, a coprocessor configuration, standard data types, or other limitations of fixed architectures. They can support virtually any computation, can be tailored based on the application's nature and context, have very high resource utilization, and consume much less energy.

Despite these advantages, FPGAs have faced difficulty in mainstream adoption due to complexity of HDL programming. Creating efficient configurations requires (at least) good knowledge of the underlying hardware and how to get the tools to map the application to it. IP cores and hierarchal design approaches have helped reduce the effort, but as newer technologies become available, these designs still need to be updated regularly to make efficient use of available resources. Moreover, if support for a particular IP does not exist for particular target, the effort required increases substantially and could even result in a complete overhaul being needed. Consequently, there is a need for higher levels of abstraction for creating and maintaining designs.

HLS tools, such as Intel/Altera OpenCL [1], have provided a popular alternative to HDLs. They allow developers to express designs in high level languages such as C99, and automate the process of converting code into board configuration files. Use of OpenCL automatically addresses design complexities that extend beyond HDL coding to even more fundamental tasks. These include dealing with fan-outs, latency matching for variable sized data paths, stall hardware, interfaces, and the entire control plane. OpenCL can also scale these appropriately for a given problem size. For features that cannot be described using the OpenCL specification, pragmas, attributes, and custom RTL libraries can be included in the design.

Reports of experiences using OpenCL, however, have shown that this improved programmability and portability comes at the expense of performance, even for applications that have traditionally given FPGAs speedups over CPUs and GPUs when using hand coded HDL. This can largely be attributed to the shortcomings of the traditional OpenCL toolflow. That toolflow, from C to bitstream, constrains the manner in which the compiler can be used and, due to its limited scope, diminishes several key potential benefits. We broadly partition the negative impact into the following three categories.

- Deployment: Lack of support for deploying OpenCL designs on host-independent applications, including real-time and streaming, and for integrating OpenCL generated architectures into existing HDL systems.
- Testability: Lack of reliable estimates for resource usage (post synthesis), latency, and operating frequency without running the complete fitting operation; this results in potentially hours spent per iteration to find better optimizations.
- Board Support Package (BSP): No way of avoiding using the BSP to reduce resource and latency overhead of potentially inefficient designs.

We address the above challenges for Altera OpenCL by incorporating an extra stage into the compilation flow: using the OpenCL-generated HDL (OpenCL-HDL) which is generated early during compilation (typically after $<1\%$ of the compilation time). The OpenCL-HDL generated at this stage represents the transition phase between code compilation and hardware generation/fitting.

We find that using OpenCL-HDL provides a number of benefits. It enables developers to get very quick reliable

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

estimates of latency and resource usage. More significantly, it allows developers to quickly gauge the effectiveness of the C-to-HDL translation, e.g., to find whether the compiler has unrolled a summation loop to make an adder tree, or serial pipelines stages with a single addition per cycle. With this additional step in the toolflow, only after the design has been verified do developers then proceed to full compilation; this significantly reduces the number of fitting iterations and thus the development time. Moreover, for applications that do not require host-machines, or to eliminate BSP overhead, OpenCL-HDL can be carved out of a full system and integrated seamlessly into existing HDL designs. This is also useful with large applications where a parameter change then only requires a specific OpenCL-HDL building block to be recompiled and re-interfaced, rather than the entire architecture. Since standard translation rules enable the entire process to be done using scripts, OpenCL-HDL can be used by developers across the HDL expertise spectrum.

The contributions of this paper are as follows:

- We use OpenCL-HDL to help bridge the Performance-Programmability gap for FPGAs. OpenCL-HDL is generated as an intermediate stage in the standard OpenCL compilation and addresses deployment, testability, and BSP bottlenecks of traditional FPGA OpenCL designs.
- We present our process of generating, isolating, and re-interfacing OpenCL-HDL modules. This includes (i) a kernel template which reliably exploits parallelism opportunities and ensures all compute pipelines are implemented as a single HDL module, (ii) identification of this module from the thousands of lines of compiler generated code, and (iii) categorization of the different types of interfaces and methods for connecting/bypassing them.
- We demonstrate the effectiveness of our approach by comparing the performance of our OpenCL-HDL modules against existing designs for a number of benchmarks from the Rodinia suite [2] and from Molecular Dynamics simulations.

The rest of this paper is organized as follows. Section 2 gives an overview of previous work in use of OpenCL for FPGAs. Section 3 provides details of our proposed steps for penetrating the standard OpenCL toolflow and carving out OpenCL-HDL compute modules. Section 4 describes the benchmarks used to test our approach and their kernel code structure. Section 5 provides the resource and performance results with respect to existing FPGA and GPU designs.

II. Previous Work

In this section, we discuss previous work on OpenCL for FPGAs in the context of compiler constraints and kernel structure. To the best of our knowledge, there is no previous work that deals with isolating, instrumenting, or embedding OpenCL-HDL.

We find that implementations of various applications have performed optimizations on baseline codes and achieved substantial speedups over that original code. Authors in [3] have implemented a particle-in-cell simulation on an Arria 10 board using OpenCL wit a 2.5x speedup after optimizations over a single core CPU. Work in [4] presents OpenCL designs for doing convolution on FPGAs and describes several optimizations which can improve performance up to 20x over baseline. Authors in [5] achieve a 23x improvement over their baseline GEM code using optimizations. A PDE solver for Poisson's equation using the Conjugated Gradient method is implemented in [6], which demonstrates average speedups of 86x, 192x and 1517x over the baseline OpenCL code for their Scaleadd, Dotc and LaplaceApply kernels, respectively. Authors in [7] have designed FPGA OpenCL kernels for k-nearest neighbor, Monte-Carlo, and bitonic sorting, with some results showing speedups over GPUs. For Smith-Waterman (SW), [8] has shown 1.53x improvement over an Altera staff implementation [9]. Authors in [10] explore the performance of OpenCL by porting some parts of the Rodinia benchmark. They have performed the standard optimizations for a number of kernels, but nearly all applications have shown worse performance than the corresponding GPU design. In [11], the authors have developed multi-producer single-consumer architectures using OpenCL for processing particle interactions with asynchronous producer-consumer pipelines. The design suffers from a significant reduction in performance as compared with the reference Verilog design.

Summarizing these reports, despite the speedups over baseline, most of these implementations have performed worse than reference GPU designs even for application characteristics that traditionally favor FPGAs configured using HDLs.

III. OpenCL-HDL

We outline the steps needed to obtain isolated OpenCL-HDL designs as part of the standard OpenCL compilation process. We first present a kernel template that not only exploits many forms of parallelism, but also reduces the OpenCL-HDL isolation effort by implementing all compute pipelines in a single HDL module. We then identify the point where compilation must be stopped and give the location of the source files. These include the kernel implementation and the pre-existing modules used to build it. Within the kernel implementation, we identify the lowest level module in the design hierarchy that contains the entire ICL. Finally, we describe the module interface and how the different ports can be connected.

A. Kernel Template

Algorithm 1 illustrates our proposed single-kernel single work-item kernel template. We implement all computations within a single Outer Compute Loop (OCL). Each iteration of the OCL represents an algorithm progression, with the actual amount depending on whether the problem was folded or not. Folding can be required since there may not be sufficient board resources to implement a full algorithm iteration for each iteration of the OCL. Outer Persistent Variables (OPVs) exist throughout the duration of a kernel execution. Hence their state is explicitly preserved by the compiler using feedback logic. Typically, OPVs are used to hold intermediate values

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

being passed to the next algorithm iteration. *Inner Persistent Variables (IPVs)* are used to source and sink data for the compute pipelines. Based on the size of an IPV array, it can implemented using registers (small) or SRAM (large). The *Inner Compute Loop (ICL)* contains a complete description of all compute pipelines in the design. All loops within the ICL, and including it, are fully unrolled. The ICL uses access to data sources and sinks, along with the order in which instructions are specified, to enable the compiler to determine the structure of compute pipelines.

Algorithm 1 Kernel Template

```
__kernel void kernel_name (.....
Declare and Initialize Outer-Persistent Variables
for Memory Loop = 1 : N do
    Initialize Data for Outer-Persistent Variables
for Outer Compute Loop = 1 : K do
    Declare and Initialize Inner-Persistent Variables
    for Memory Loop = 1 : J do
        Initialize Data for Inner Persistent Variables
    #pragma unroll
    for Inner Compute Loop = 1 : J do
        COMPUTE
    for Memory Loop = 1 : J do
        Store Results to Outer-Persistent Variables
        Output Inner-Persistent Variable Data
for Memory Loop = 1 : M do
    Output Outer-Persistent Variable Data
```

A fully unrolled ICL enables all compute infrastructure to be grouped into the same OpenCL-HDL module, which reduces isolation effort. From a performance perspective, by expressing all computations in the same loop, the compiler can establish data dependencies and identify opportunities for data, task and instruction parallelism.

Data Parallelism: Unrolled loops with no data dependencies between iterations are implemented in a SIMD manner. This enables each loop to have an arbitrary number of SIMD lanes, as opposed to a fixed global value defined in multiple work-item kernels.

Task Parallelism: Code fragments with no data dependencies are implemented as separate pipelines that operate concurrently, irrespective of their ordering in the ICL. Use of a shared variable can allow communication between these tasks, similar to blocking OpenCL channels. However, task parallel pipelines in ICL do not stall in order to wait for a new shared variable value. Instead, delay modules are added to ensure that the new value is available on the exact cycle that it is needed. It is possible to synchronize pipelines in this manner using our template because all pipelines operate based on the same OCL and ICL iterators.

Instruction Parallelism: For all computations performed, the compiler attempts to maximize the amount of work done per pipeline stage (minimize latency), while ensuring pipelines are deep enough for the design to operate at high frequencies.

B. Optimizations

Optimizations are a critical part of any OpenCL coding effort. Within the kernel template, we perform the following.

Minimizing OPVs: We avoid using OPVs whenever possible due to data dependencies across OCL iterations. Every iteration of the OCL can potentially be stalled for a significant number of cycles to ensure updated values of OPVs are available. This also prevents the compiler from inferring deeper pipelines.

Constants: Use of constants not only reduces the interfacing effort, but also saves resources by removing the associated logic for fetching data and applying values directly at the inputs of compute units.

Register Arrays: We ensure all IPVs are inferred as registers by breaking large variable arrays, which are inferred as SRAMs, into smaller ones. Use of registers allows a large number of concurrent reads and writes to different elements in a variable array. On the other hand, parallel accesses of an SRAM based array results in memory replication, which, in turn, can cause device resources to be exceeded.

Detailed Computations: We perform all computations in as much detail as possible to minimize the dependence on the compiler to correctly infer design patterns or potential pipeline stages.

Selective Operations: We avoid using conditional statements in the ICL because hardware is persistent and cannot be reliably turned off. Instead, we prevent results of invalid operations from modifying the system state by assigning safe values to their outputs, e.g., zeroing out the result if it is connected to an adder.

C. Compilation Breakpoint

OpenCL-HDL is available early in the compilation process, immediately after the first stage completes, but its extraction currently requires manual termination. One method is to compile with the verbose (*-v*) flag and stop compilation once the message regarding successful generation of source files has been posted.

D. Source Files

The compiler places source files in a directory relative to the kernel file. Specifically this is *[Path to Kernel File]/ <kernel_filename>/ kernel_subsystem/ <kernel_filename>_system_140/ synth /.* The file <kernel_filename>.v contains the implementations for all kernels in the compiled OpenCL source. Other files include <kernel_filename>_system.v, which is a wrapper, and the Altera modules that form the lowest level in the design hierarchy. These building blocks can range from commonly used components, such as FIFOs and Load-Store Units (LSUs), to kernel-specific ones, such as floating point arithmetic units. While Altera maintains a large number of modules, only those identified by the compiler as required are copied here. Before using these files, however, it is important to change their file extension to $.sv$ from $.v$ due to certain syntax that would otherwise prevent compilation as

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

Verilog. In the remaining sections of this paper, we refer to <kernel_filename>.v as the OpenCL-HDL source.

E. Basic Blocks

The OpenCL toolflow compiles kernels through multiple modules called basic_blocks, and uses a function module to describes their connectivity. The observed rules for basic_block module generation are as follows. Typically, each loop generates a basic_block module. Nested normal loops will generate their independent modules and are connected by their parent loop module. Unrolled loops will also generate a separate module. However, consecutive unrolled loops, or any unrolled nested loop within an unrolled loop, will not generate a new module. Based on these rules, the entire ICL is created in a unique basic_block which significantly reduces the effort of isolating it.

F. Interfaces

We broadly categorize common interfaces to be either LSU, Direct, Control, Feedback, or Don't Care. A basic_block can have variations in the number and types of these ports based on individual applications and problem sizes. Modifying interfaces eliminates hardware that was required for interfacing the pipelines with the BSP OpenCL wrappers (and hence does not impact the functionality of our compute pipelines). This reduces resource usage and simplifies data paths resulting in lower latency. Once the interfaces have been mapped, the module can be used for RTL simulation or integrated into a different HDL shell for compilation.

Load Store Unit (LSU) ports consist of Avalon interfaces to LSU modules. LSU modules source and sink data to compute pipelines and are typically created, one per variable per operation (read/write), when a memory access depends on the outer loop iterator. Both the module, as well as the Avalon protocol, have associated overheads which can be removed by interfacing pipelines directly with variable data. We bypass these LSU modules by creating explicit variable input/output ports and connecting them to the corresponding LSU module's source/sink interfaces (*o_readdata*/*i_writedata*). The Quartus compiler then removes the LSU modules during fitting optimizations.

Direct ports are automatically generated by the compiler to supply data directly to compute pipelines without requiring LSU modules or the Avalon protocol. They can be further categorized into Constants, Variables, and Initialization. Constant direct ports correspond to kernel inputs that are individual elements (instead of pointers). Variable direct ports correspond to cases where the LSU unit is moved outside the basic block. This can occur if the outer loop variable is not used to index/address memory accesses, or if the problem size is too small. Initialization direct ports are used to load initial values for (outer/inner) persistent variables.

Control ports primarily consist of clock, reset, *Stall*, and *Valid* ports. *Stall* ports are used by a basic_block to stall upstream modules, while *Valid* ports are used to stall downstream basic_blocks. Since we isolate the compute pipelines from the overall system, we hardwire both *Stall* and *Valid* ports. As a result, the Quartus compiler minimizes or removes the associated stall hardware during fitting optimizations.

Feedback ports are typically generated to maintain the state of Outer Persistent Variables (OPVs) across algorithm iterations. They are created in input-output pairs, with the output feedback ports being looped back and connected to input ports of the same basic_block. However, if the algorithm does not use OPVs, we do not connect Feedback ports. This allows the Quartus compiler to remove the unnecessary associated logic. If the OpenCL compiler generates individual feedback inputs (and not pairs), then we hardwire them since these correspond to selecting between initial and steady-state values of OPVs.

Don't Care ports can correspond to a variety of logic such as parallel control and data paths that do not interact with compute pipelines, or logic that interfaces LSU modules, e.g., address computation. Leaving Don't Care ports unconnected in our HDL shell optimizes away these unnecessary resources.

IV. Evaluation and Benchmarks

We evaluate our kernel optimizations by porting a number of benchmarks from Rodinia and Molecular Dynamics to SWI OpenCL and comparing the performance of OpenCL-HDL against existing implementations. The design pattern is given in parentheses. Benchmarks from Rodinia are Needlman-Wunsch (Dynamic Programming) and Pathfinder (Dynamic Programming). For Molecular Dynamics, the benchmarks are FFT (Spectral), Range-Limited Electrostatics Force Calculation with Filtering (N-Body), and Charge Mapping (Structured Grids).

A. Hardware Specifications

We implemented our designs on an Altera Arria 10AX115H3F34E2SGE3 FPGA using Altera OpenCL SDK 16.0. The FPGA has 427,200 ALMs, 1506K Logic Elements, 1518 DSP blocks, and 53Mb of on-chip storage. For reference, for the Rodinia benchmarks, we use the Tesla P100 PCIe 12GB GPU. It has 3584 Cuda cores and peak bandwidth of 549 GB/s.

B. Needleman-Wunsch

Needleman-Wunsch is a pairwise sequence alignment algorithm that is a fundamental application in bioinformatics. It evaluates all possible alignments by using two strings to fill a table and then finding a minimum through the table path using Dynamic Programming (DP). The two strings form the first row and column of this table, respectively, with index (0,0) being the top-left corner of this table. Each table entry is scored based on the values of the input sequences. The best path score at each entry is found by looking at the already computed path scores of its immediate top, left, and top-left indices. The computation on FPGAs is typically performed using a systolic array that processes entire rows or columns in sequence (as opposed to evaluating diagonals/wavefronts). Algorithm 2 presents the structure of our ICL. A single variable, value[i], is used to represent the state of each systolic array processing element [12], [13].

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

In our implementation, we used a 1024 element systolic array to process string sizes of 16,384 (similar to those in [10]). The data type used here is *char*. The table is divided into blocks and the algorithm was executed 16 times in total. For each execution, the running time was 16,384 cycles, plus a load latency of 1 cycle and an unload latency of 512 cycles (2 systolic modules processed per pipeline stage). The GPU implementation from the Rodinia suite is run as is, i.e., using the integer data type. We also perform a direct comparison with [10], who used a Stratix V 5SGXA7 FPGA, since the design does not use DSP blocks. From the results (Table I), we can see that OpenCL-HDL outperforms both the high end GPU and existing OpenCL implementation by at least 40x. Moreover, since only 13% ALMs were used, greater parallelism can be exploited by having more processing elements in the systolic array or by replicating the pipelines.

Algorithm 2 Needleman Wunsch ICL structure

```
1:  char value[N]
2:  Initialize value array
3:  for k = 1 : M do
4:      #pragma unroll
5:      for i = 1 : N do
6:          if i < N then
7:              b[i+1]←value[i] + ref[k*N+i+1]
8:          if i == 1 then
9:              value[i] = north[k]
10:         else
11:             a[i]←value[i]-penalty
12:             c[i]←value[i-1]-penalty
13:             d[i]←MAX(a[i],b[i])
14:             value[i]←MAX(d[i],c[i])
```

TABLE I
RESOURCE USAGE AND PERFORMANCE FOR NEEDLEMAN-WUNSCH

Design	ALM	Freq.(MHz)	Time(ms)
OpenCL [10]	-	148	251.3
OpenCL-HDL	56274(13%)	307	0.9
GPU	-	1328	36.7

C. Path Finder

Pathfinder is a grid traversal algorithm that determines the minimal cost of moving from each point on the first row of a grid to any point in the last row. Movement is constrained to be either vertical or diagonal. We follow the algorithm provided in the Rodinia suite as shown in Algorithm 3 and implement it using systolic arrays.

Similar to [10], our evaluation is done using row sizes of 100,000. However, we use a column size of 1024 instead of 100 in order to perform a more reliable comparison with the GPU (implemented in CUDA 8.0). Load and unload latencies for the OpenCL-HDL design are negligible. Similar to Needleman-Wunsch, we compare with [10] directly since no DSP blocks are used. From Table II, we see that OpenCL-HDL achieves ≈10x speedup over the FPGA OpenCL design despite having 10x more columns. With respect to GPU, the

Algorithm 3 Pathfinder ICL structure

```
1:  int value[N]
2:  Initialize value array
3:  for k = 1 : M do
4:      #pragma unroll
5:      for i = 1 : N do
6:          West ← value[(i==1) ? 1 : i-1]
7:          East ← value[(i==N) ? N : i+1]
8:          Center ← value[i]
9:          Up ← ref[i]
10:         Smallest = MIN (West, East, Center)
11:         value[i]← Smallest + Up
```

P100 was <2x faster than OpenCL-HDL. In comparison, in [10] the Tesla K20c (2496 cores, 208GB/s) GPU was ≈5x faster than the OpenCL FPGA version.

TABLE II
RESOURCE USAGE AND PERFORMANCE FOR PATHFINDER

Design	ALM	Freq.(MHz)	Time(ms)
OpenCL [10] (col:100)	-	143	4.57
OpenCL-HDL	129,508 (30%)	228	0.44
GPU-P100 (CUDA)	-	1328	0.25

D. 1D FFT

FFT is a critical step in a number of modern scientific applications where the k-space transformation allows time domain computations, such as convolutions, to be performed with *O(NlogN)* complexity. We discuss FFT in detail in a separate publication [14] and hence will provide only a brief outline here. As shown in Algorithm 4 we have implemented a Radix-2 1D FFT compute kernel that produces a complete transformed vector every cycle. We compare the performance of our design, for a 64 points FFTs, against the same algorithm implemented in HDL, OpenCL and using IP core [15] based designs [16]–[18]. The throughput of all designs is constrained to be 64. For IP core instantiations that do not have available DSPs, we use ALMs to implement them. Table III lists the results of our implementations. OpenCL-HDL has similar resource overhead and frequency with respect to the pure HDL design. While manual datapath tuning in pure HDL resulted in better latency, that impact is diminished as a larger number of vectors is processed. IP cores, on the other hand, were significantly worse than both HDL and OpenCL-HDL, consuming 10x more ALMs and ≈300 more DSP blocks to get the same performance.

TABLE III
RESOURCE USAGE AND PERFORMANCE FOR FFT

Design	ALM	DSP	Freq.(MHz)	Latency
Verilog	15,558(4%)	1160 (76%)	488	37
IP Core	176,285(41%)	1412 (93%)	408	64
OpenCL	65,257(15%)	1280 (93%)	240	-
OpenCL-HDL	18,705(4%)	1160(76%)	483	53

E. Range-Limited Electrostatics with Filtering

Filtering uses a small amount of hardware to determine whether a much larger computation needs to be executed (see,

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

Algorithm 4 1D FFT ICL structure

```
1: N ← FFT Size
2: for Array of 1D Vectors →j = 1 : M do
3:     float2 stage_0[N] .... stage_logN[N]
4:     Initialize stage_0
5:     int L = N/2; int m = 1;
6:     #pragma unroll
7:     for Loop 1 →k = 1 : N/2 do
8:         stage_1 ← COMPUTE(stage_0)
9:     L = L >> 1; m = m << 1;
10:            ⋮
11:    #pragma unroll
12:    for Loop logN →k = 1 : N/2 do
13:        stage_logN ← COMPUTE(stage_logN-1)
```

e.g., [19]–[21]). Since the result of the filter is not known *a priori,* these designs are asynchronous producer-consumer. In Molecular Dynamics simulations, a set of filters is used to pre-process particle pairs and only those within cut-off radius are passed to the compute pipelines. Algorithm 5 gives the ICL for our Filtered Pipeline implementation. The overall computation is implemented as three task parallel stages, i.e., arbiter (Line 6-12), force pipeline (Line 13), and filters (Line 14-20). Following [11], we use 8 filters per pipeline.

Algorithm 5 Filtered Pipeline ICL structure

```
1: N ← Filters per Pipeline , P ← Particles per Filer
2: T ← Threshold
3: for Algorithm Iterations →j = 1 : M do
4:     #pragma unroll
5:     for Filters →i = 1 : N do
6:         largest_L ← Elements[i] > Elements[1→i-1]
7:         largest_R ← Elements[i] ≥ Elements[i+1→N]
8:         largest_C ← Elements[i] > 0
9:         cmp[i] ← largest_L & largest_R & largest_C
10:    Pipeline_In = ∑_{i=1}^{N} cmp[i] * Buffer[i][Elements[i]]
11:    Pipeline_Valid = ∑_{i=1}^{N} cmp[i]
12:    Pipeline_Out = f(Pipeline_In , Pipeline_Valid)
13:    #pragma unroll
14:    for Filters →i = 1 : N do
15:        Particle ← j ≤ P ? (x,y,z) : (T,T,T)
16:        FilterPass[i] ← x*x + y*y + z*z ≤ T ? 1 : 0
17:        Buffer[i][Elements[i]-cmp[i]] ← Particle
18:        Elements[i] ← Elements[i] + FilterPass - cmp[i]
19:        Elements[i] ← Elements[i] < 0 ? 0 : Elements[i]
```

Table IV shows the results of our OpenCL-HDL design as compared to the implementations given in [11]. The Verilog design refers to the state-of-the art, while OpenCL refers to their best case implementation (which uses distributed control). We use the data set values given in the reference work, i.e., 216 slices, 27 slices per filter, and 15 particles per slice. The load latency for OpenCL-HDL is at least 67 cycles, with the value increasing based on the complexity of the compute pipeline. In the results we observe that worst-case OpenCL-HDL execution time outperforms both the state-of-the-art and the existing best-case OpenCL design by at least 50x. While the results are similar to the Verilog, we can also fit up to 20 copies of the filtered pipeline system using OpenCL-HDL.

TABLE IV
RESOURCE USAGE AND PERFORMANCE FOR FILTER PIPELINE

Design	ALM	DSP	Freq.(MHz)	Time(ms)
Verilog	654(1%)	53(2%)	195	0.5
OpenCL	39709(5%)	56(4%)	257	3.1
OpenCL-HDL	6206(1%)	67(4%)	324	0.01

F. Charge Mapping

Charge Mapping is an interpolation operation that maps point charges, distributed randomly in space, to a regular grid [**?**], [22], [23]. It is a critical step in enabling FFT computation, which in turn reduces the complexity of Coulombic force evaluation to $O(NlogN)$. While Charge Mapping itself is an $O(N)$ problem, the high arithmetic workload (almost 500 FLOPs per particle) highlights the importance of accelerating this computation. The interpolation operation is performed by evaluating four polynomials per dimension (12 total). Each unique combination of one polynomial per dimension corresponds to a specific grid point (64 total per particle). Algorithm 6 illustrates our Charge mapping kernel. Table V compares our implementation with the Arria-10 Verilog design presented in [24]. The load latency of our OpencL-HDL design is 35 cycles. We achieved similar performance to the reference design, but have significantly less DSP resource overhead.

Algorithm 6 Charge Mapping ICL structure

```
1: N ← PME_Order (4), M ← Num_Atoms
2: const factors3 [4] ← {-0.5, 1.5, -1.5, 0.5}
3: const factors2 [4] ← {1.0, -2.5, 2.0, -0.5}
4: const factors1 [4] ← {-0.5, 0.0, 0.5, 0.0}
5: const factors0 [4] ← {0.0, 1.0, 0.0, 0.0}
6: for Atoms →j = 1 : M do
7:     x = px[j]; y = py[j]; z = pz[j]; q = pq[j];
8:     #pragma unroll
9:     for xx = 1 : 4 do
10:        #pragma unroll
11:        for yy = 1 : 4 do
12:            #pragma unroll
13:            for zz = 1 : 4 do
14:                index ← (xx << 4) + (yy << 2) + zz
15:                a ← f(x, xx, factors 0-3)
16:                b ← f(y, yy, factors 0-3)
17:                c ← f(z, zz, factors 0-3)
18:                output[index] ← a*b*c*q
```

TABLE V
RESOURCE USAGE AND PERFORMANCE FOR CHARGE MAPPING

Design	ALM	DSP	Freq.(MHz)
Verilog	3752 (1%)	319 (20%)	571
OpenCL-HDL	1900 (<1%)	117(8%)	488

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

V. Overall Performance

Figure 1 summarizes the performance of OpenCL-HDL versus FPGA OpenCL, GPU, Verilog, and IP-core-based designs. The reference execution time for Needleman-Wunsch (NW) and Path Finder (PF) is from the GPU implementation. The reference time for FFT, Electrostatics with Filtered Pipelines (FP), and Charge Mapping (CM) is Verilog. We demonstrate that in all cases, the optimized OpenCL-HDL performance is competitive and can be up to 40x and 50x higher than GPUs and Verilog, respectively.

Figure 2 presents the geometric mean for all comparisons. It shows that, on average, OpenCL-HDL outperforms previous FPGA OpenCL, GPU, and Verilog codes by 37x, 4.8x and 3.5x, respectively. This demonstrates that, with the correct code structure and bypassing the BSP, FPGA OpenCL can achieve both good programmability, and high performance. The result is particularly significant with respect to Verilog. While it is expected that hand-coded HDL will outperform OpenCL-HDL, writing optimal HDL to do so requires significant expertise and effort. On the other hand, by using OpenCL-HDL, we have the advantage of not only reducing programming effort, but also of having the code compile to an efficient architecture that provides HDL-like performance.

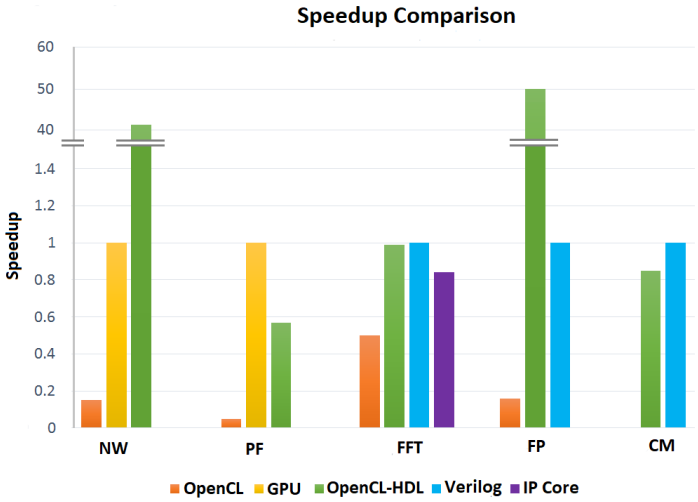

Fig. 1. Performance comparison for all five benchmarks

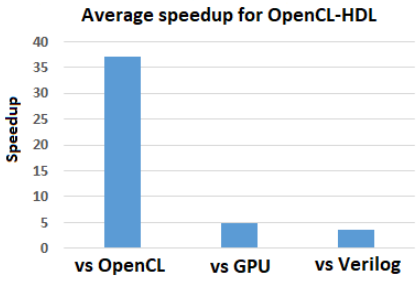

Fig. 2. Geometric mean of OpenCL-HDL speedup across all applications

VI. Conclusion

In this paper, we presented our approach for addressing the Performance-Programmability challenges of FPGAs using OpenCL-HDL. By penetrating the traditional OpenCL toolflow, we have developed an approach for carving out compute pipelines from OpenCL wrappers and integrating them into custom HDL shells for RTL simulation, more accurate resource estimation and deployment as host-independent applications. We also propose a kernel template, which not only efficiently leverages application-specific opportunities for parallelism, but also reduces the effort of isolating OpenCL-HDL from thousands of lines of compiler generated code. We demonstrate that OpenCL-HDL can be used for a variety of design patterns by testing our approach using benchmarks from Rodinia and Molecular Dynamics. The results are promising as the OpenCL-HDL designs outperform existing FPGA OpenCL implementations, GPU codes, and Verilog (for Filtered Pipelines), with the average speedup being 37x, 4.8x and 3.5x, respectively. OpenCL-HDL also used fewer resources on average to implement designs, as compared with existing FPGA OpenCL codes.

Acknowledgement

This work was supported in part by the National Science Foundation through Awards #CNS-1405695 and #CCF-1618303/ 7960; by grants from Microsoft and Red Hat; and by Altera through donated FPGAs, tools, and IP.

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. 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 International Symposium on Workload Characterization*, 2009, pp. 44–54.

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

[4] 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.

[5] 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.

[6] 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.

[7] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis," *IEEE Access*, vol. 5, pp. 2747–2762, 2017.

[8] E. Rucci, C. Garcia, G. Botella, A. De Giusti, M. Naiouf, and M. Prieto-Matias, "Accelerating Smith-Waterman Alignment of Long DNA Sequences with OpenCL on FPGA," in *Int. Conf. on Bioinformatics and Biomedical Engineering*, 2017, pp. 500–511.

[9] S. O. Settle, "High-performance dynamic programming on FPGAs with OpenCL," in *Proc. IEEE High Performance Extreme Computing Conference*, 2013, pp. 1–6.

[10] 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.

[11] C. Yang, J. Sheng, R. Patel, A. Sanaullah, V. Sachdeva, and M. C. Herbordt, "OpenCL for HPC with FPGAs: Case study in molecular electrostatics," in *IEEE High Performance Extreme Computing Conference*, 2017, pp. 1–8.

[12] T. VanCourt and M. Herbordt, "Families of FPGA-based algorithms for approximate string matching," in *Proc. Int. Conf. on Application Specific Systems, Architectures, and Processors*, 2004, pp. 354–364.
[13] A. Mahram and M. Herbordt, "NCBI BLASTP on High Performance Reconfigurable Computing Systems," *ACM Trans. Reconfigurable Tech. and Sys.*, vol. 15, no. 4, pp. 6.1–6.20, 2016.
[14] A. Sanaullah and M. Herbordt, "FPGA HPC using OpenCL: Case Study in 3D FFT," *Proc. Highly Efficient and Reconfigurable Technologies*, 2018.
[15] "FFT IP Core User Guide," www.altera.com/ documentation/ hco1419012539637. html, accessed: 2010-08-29.
[16] B. Humphries, H. Zhang, J. Sheng, R. Landaverde, and M. Herbordt, "3D FFT on a Single FPGA," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2014.
[17] J. Sheng, B. Humphries, H. Zhang, and M. Herbordt, "Design of 3D FFTs with FPGA Clusters," in *IEEE High Perf. Extreme Computing Conf.*, 2014.
[18] J. Sheng, C. Yang, A. Caulfield, M. Papamichael, and M. Herbordt, "HPC on FPGA Clouds: 3D FFTs and Implications for Molecular Dynamics," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2017.
[19] M. Chiu and M. Herbordt, "Efficient filtering for molecular dynamics simulations," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2009.
[20] M. Chiu and M. Herbordt, "Molecular dynamics simulations on high performance reconfigurable computing systems," *ACM Trans. Reconfigurable Tech. and Sys.*, vol. 3, no. 4, pp. 1–37, 2010.
[21] M. Chiu, M. Khan, and M. Herbordt, "Efficient calculation of pairwise nonbonded forces," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2011.
[22] Y. Gu and M. Herbordt, "FPGA-based multigrid computations for molecular dynamics simulations," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2007, pp. 117–126.
[23] A. Sanaullah, K. Lewis, and M. Herbordt, "GPU Accelerated Particle-Grid Mapping," in *IEEE High Perf. Extreme Computing Conf.*, 2016.
[24] A. Sanaullah, A. Khoshparvar, and M. C. Herbordt, "FPGA-Accelerated Particle-Grid Mapping," in *Field-Programmable Custom Computing Machines*, 2016, pp. 192–195.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE