FPGA HPC using OpenCL: Case Study in 3D FFT

Ahmed Sanaullah Martin C. Herbordt
Department of Electrical and Computer Engineering; Boston University, Boston, MA

Abstract*—FPGAs have typically achieved high speedups for 3D Fast Fourier Transforms (FFTs) due to the presence of hard floating point units, low latency specialized pipelines, and support for complex connectivity among processing elements. Previous implementations have relied on FFT IP cores for performing the computation due to the complexity of manually developing and maintaining/upgrading efficient pipelines in HDL. These IP cores, however, are bulky and cannot be fully tuned for specific FFT sizes due to use of generic architectures. HLS tools, such as OpenCL, offer a more customizable alternative but have suffered from worse performance than HDL in previous work. In this paper we show that, using a set of code structure optimizations, OpenCL designs can be compiled to Radix-2 FFT pipelines which outperform IP core based designs for the same throughput. We further show that the HDL generated by the OpenCL compiler can be isolated and seamlessly integrated into existing 3D FFT shells to reduce implementation effort. Our single device design, tested on the Altera Arria10X115 FPGA, achieves an average speedup of 29x vs CPU-MKL, 4.1x vs GPU cuFFT and 1.1x vs IP Core FFT implementations for $16^3, 32^3$ and 64^3 FFTs. Moreover, OpenCL generated compute pipelines for $8^3, 16^3, 32^3$ and 64^3 FFTs use an average of 7.5x fewer ALMs and 1.6x fewer DSPs than corresponding IP core versions.

I. Introduction

FPGAs enable developers to design custom systems that leverage application-specific optimizations [1]–[3] including many that use the 3D FFT [4]–[6]. Unlike GPUs, FPGAs are not constrained to data parallelism, a co-processor configuration, standard data types, or other limitations of fixed architectures. They can support virtually any computation, can be tailored based on their 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 have helped reduce the effort, but offer limited customizations and are therefore unable to fully tailor the design for a given application. For example, IP cores have been used in numerous previous 3D FFT implementations for both single [7]–[10] and multi FPGA systems [11]–[13]. These cores only allow users to modify trivial parameters such as FFT size, data types, data order, and the resource type used to implement them. The architecture remains generic, i.e., each core operates as an independent entity and processes one element of a vector at a time using a streaming interface. While this is ideal for larger FFT sizes, since the problem can be folded to accommodate limited resources, it is suboptimal for smaller ones where the board is large enough for an entire vector to be streamed every cycle. Having a broadside approach reduces resource usage by simplifying the algorithm, data path, and processing elements. It is able to perform global optimizations since the scope of doing so is not limited to datapaths within an individual core. Consequently, there is a need for higher levels of abstraction for creating and maintaining 3D FFT designs which gives greater flexibility than IP cores, but without a degradation in performance.

HLS tools, such as Altera OpenCL [14], 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.

Previous work on FPGA OpenCL has typically shown worse performance than both GPU and HDL based implementations. Authors in [15] explore the performance of OpenCL by porting the Rodinia benchmark. They have performed standard optimizations for a number of kernels but nearly all have shown worse performance than the GPU design. In [16], the authors have developed multi-producer single-consumer architectures using OpenCL for processing particle interactions. The design suffers from a significant reduction in performance as compared to Verilog designs. Authors in [17] have implemented a particle in cell simulation on an Arria 10 board using OpenCL, but only managed a 2.5x speedup after optimizations over a single core CPU. These algorithms have traditionally shown orders of magnitude better performance for FPGAs over CPUs due to significant opportunities for parallelism. Similarly, for Smith-Waterman which has typically achieved speedups over GPUs using HDL implementations, [18] has shown that GPUs outperform FPGAs when OpenCL is used. In [19], the authors have implemented an anisotropic Huber-L optical flow estimation algorithm on high-end FPGA and GPU boards. Their results show that the GPU implementation has an average speedup of 20x over FPGAs.

In our work, we implement a broadside Radix-2 FFT module using OpenCL that outperforms CPUs, GPUs, and even HDL for FFT sizes that can fit on a single FPGA without folding. By using a set of code structure optimizations, we ensure that the compiler is able to leverage the design flexibility of FPGAs to identify and exploit all forms of possible parallelism effectively. Once the compiler translates code

into HDL, we isolate compute pipelines from the remaining OpenCL system to address language limitations in expressing complex data structures efficiently. This generated logic is seamlessly integrated into existing 3D FFT shells that were initially constructed for IP core based designs.

Our contributions in this paper are as follows:

- We present a broadside Radix-2 FFT kernel implemented using Altera OpenCL SDK. By using a set of code structure optimizations, we demonstrate that the compiler is able to implement efficient, deep, pipelines with large degrees of exploited parallelism.
- Through generated HDL isolation, we demonstrate that the compute pipelines can be easily integrated into legacy HDL codes.
- Our OpenCL design is shown to be more resource efficient that IP cores for the considered FFT sizes.
- We demonstrate that OpenCL for FPGAs can outperform CPU, GPU, and HDL implementations. Previous work has typically shown performance far worse than HDL.

The rest of this paper is organized as follows. Section 2 discusses the traditional Altera OpenCL toolflow and highlights the drawbacks of a full compilation. Section 3 presents our set of optimizations–in order to improve compiler effectiveness–on a baseline 1-D FFT kernel code and describes the resulting generated hardware. Section 4 shows how this hardware can then be integrated into existing IP core based 3D FFT shells. Section 5 presents the results of our implementations.

II. Altera OpenCL

In this section, we discuss the manner in which applications are typically implemented using OpenCL and the potential drawbacks of this approach. We then introduce OpenCL-HDL as an approach which allows users to take advantage of the compiler's efficient C-HDL translation, without being tied down to the discussed drawbacks of a full compilation.

A. Toolflow

Unlike traditional OpenCL toolflows, the Altera OpenCL SDK does not support run-time compilation of kernel files since there is no pre-existing architecture being targeted. Instead, it relies on performing offline compilations to infer the desired architecture, followed by a fitting operation to determine a valid board logic configuration for implementing this architecture. An AOCX file is generated which can be programmed onto the board using host functions. Partial reconfiguration is preferred over a full one since the former is faster and thus better suited for run-time programming. Board vendors provide a set of pre-existing modules, referred to as the Board Support Package (BSP), which define a partial reconfiguration space for compute pipelines. The BSP also provides interfaces for these pipelines with external hardware such as host (via PCIe), off-chip memory, and network ports. Therefore the toolflow wraps user defined hardware with vendor supplied BSP logic (and a freeze wrapper), and then compiles the entire system rather than isolated kernel logic.

B. Drawbacks

In our work we avoid using the full OpenCL generated system due to the following reasons. These result from both the semantics of the language itself, as well as the associated generic toolflow.

1) Language Capability: OpenCL was primarily designed for GPUs and thus the syntax is not broad enough to effectively leverage a number of essential FPGA features. These include complex memory structures, such as interleaved memory and ring buffers, as well as arbitrary-sized data types. Moreover, the OpenCL programming model only transfers data between host and device off-chip memory. While this is required for GPUs, since global memory is the only memory accessible to all blocks, FPGAs are not bound by such architectural constraints. Performance of data sets of up to a few MBs would instead benefit more from being copied directly into FPGA on-chip BRAMs from host memory.

2) Board Support Package: Since it is designed to support a number of interfaces, the BSP is too large. Irrespective of how many of the available interfaces are used by a user application, the BSP will always have a fixed resource overhead since they are all implemented regardless. While some vendors offer different BSP types for particular sets of use cases, they cannot be configured to each individual application and thus a significant portion of the chip (as high as 30%) is wasted. Moreover, since the interfaces will lie in the critical path of communication and memory intensive applications, the performance of user defined logic depends heavily on how efficient the BSP implementation is.

3) Porting Legacy Codes: There exists a large family of legacy codes that would require substantial effort to port to OpenCL as custom RTL due to required support for stall/valid interfaces, as well as defining path latencies. Moreover, the possibility of OpenCL syntax being unable to express certain features can result in users being unable to emulate the design and verify functional correctness before a full compilation.

4) Fitting effort: Partial reconfiguration limits the board area where logic can be implemented. This restriction makes it difficult to place and route larger designs. Fitting effort for a given resource utilization that typically took hours can now potentially take days, if it succeeds at all.

C. OpenCL-HDL

The OpenCL toolflow is mainly composed of two compilation stages. In the first stage, kernel code is translated into equivalent HDL while the second stage, performed by the Quartus compiler, synthesizes and fits this design. We use a compiler augmentation to break the toolflow after the translation stage and isolate the HDL corresponding to our compute pipelines from the remaining system (including BSP). This logic is referred to as OpenCL-HDL. OpenCL-HDL can be re-interfaced with custom input and output buses and then connected to existing user defined HDL logic. Thus, we can use OpenCL generated logic to optimize individual components in larger systems instead of redoing the entire architecture. In this work, we isolate our 1D FFT pipelines

and integrate them into an HDL shell containing buffers and crossbars in order to perform a 3D FFT.

III. OpenCL 1-D FFT Kernel

In this section, we present our OpenCL FFT code. Starting from a baseline version, we outline the optimizations done in restructuring the code so that the compiler can better infer and implement efficient pipelines. We also give a description of the generated OpenCL-HDL architecture.

A. Baseline Radix-2 Kernel Code

Algorithm 1 gives an overview of our baseline 1-D Radix-2 FFT code based on OpenMM [20]. Each N-element input vector is processed for log(N) stages. For a given stage and iteration, vector elements are processed in pairs; each pair writes two values to the input vector for the subsequent stage. The variables L and m are used to determine these input and output pairs, as well as corresponding twiddle factors. The entire computation is done using two float2 arrays in a ping-pong manner, with the buffers swapped after every stage.

Algorithm 1 Radix-2 1D FFT Baseline Kernel

```
1:  N ← FFT Size
2:  for Array of 1D Vectors →j = 1 : M do
3:      float2 data_0[N], data_1[N]
4:      Initialize data_0
5:      int L = N/2
6:      int m = 1
7:      for Stage →s = 1 : log(N) do
8:          data_1 ← COMPUTE(data_0,L,m)
9:          Swap (data_0 , data_1)
10:         L = L >> 1
11:         m = m << 1
12:     end
13: end
```

B. Optimizations

1) Standard Optimizations: Altera recommends a number of design best practices, such as Single Work Item kernels, loop unrolling, and inferring registers using small arrays. However, a significant number of existing FPGA OpenCL programs, e.g., OpenDwarfs [21] benchmarks, use alternatives such as Multiple-Work Item kernels. This may be due to the reduced effort in porting GPU OpenCL codes. Our optimizations both build on recommended best practices (3,5,6 below), as well as introduce novel approaches (2,4 below). We present observations as to why these optimizations are found to be better suited for our implementations. In the case of recommended best practices, we provide further insight that is beyond the information given in Altera's documentation.

2) Single Kernel Implementation: The OpenCL compiler automatically detects the worst case latency between all input/output port pairs and then adds delay modules in parallel data-paths (where needed) to ensure pipelines are synchronized. It can also shuffle pipeline stages, while maintaining data dependencies, in order to overlap delay modules and effectively remove them. Having multiple kernels with channel based connectivity isolates different pipeline stages, restricting the compiler to only consider local data paths when performing the above optimizations. This reduces compiler efficiency and suboptimal pipelines are potentially generated. We therefore implement our FFT code within the same kernel and rely on compiler optimizations to minimize the overall pipeline depth.

Algorithm 2 Compute Function Code

```
1:  for base = 0 : (N/2) − 1 do
2:      int j = base/m; int k = j*32/L; int p = (j+1)*m;
3:      float c0_r = stage0_r[base];
4:      float c0_i = stage0_i[base];
5:      float c1_r = stage0_r[base+32];
6:      float c1_i = stage0_i[base+32];
7:      stage1_r[base+j*m] = c0_r+c1_r;
8:      stage1_i[base+j*m] = c0_i+c1_i;
9:      float c2_r = c0_r-c1_r; float c2_i = c0_i-c1_i;
10:     stage1_r[base+p] =w_r[k]*c2_r - w_i[k]*c2_i;
11:     stage1_i[base+p] =w_r[k]*c2_i + w_i[k]*c2_r;
12: end
```

3) Single Work Item (SWI) Kernel with Unrolled Compute Loops: Use of Multiple Work Item (MWI) kernels can result in substantial inefficiency and overhead. Some of these include a) inability of compute units to directly communicate with each other (and instead having to rely on global memory access), b) resource usage by the scheduler, implemented with sufficient complexity to ensure stall free pipelines, c) static SIMD lane sizes being applied to irregular shaped pipelines, d) wrapper logic for individual compute units to interface with external memory, e) complex memory management unit needed to process multiple, potentially un-coalesced requests from different compute units, and f) overhead of local and global synchronization needed through barriers, including memory needed for preserving variable states.

Algorithm 3 shows unrolled compute loops within a SWI kernel. This gives the compiler flexibility in detecting and implementing all possible forms of parallelism such as data, task, and instruction. The compiler performs several levels of optimizations, tuning the architecture at the granularity of a pipeline stage in order to maximize implementation efficiency. No scheduler is required since only one work item exists and external interfaces are greatly simplified. By binding all computations to a single outer loop variable, inherent synchronization is achieved. Moreover, this approach also results in a tight coupling of all compute pipelines, making them easier to isolate from the overall OpenCL generated system.

4) Explicit Computations: We perform computations explicitly, using a large number of intermediate variables as shown in Algorithm 2. This helps the compiler infer extra pipeline stages which it can use to break data dependencies, reduce the critical path, improve fanout and increase operating frequency. If the pipeline is already optimal, the compiler synthesizes away these extra variables and does not unnecessarily increase pipeline latency. We also declare separate variables

for each computation stage, rather than using only two sets of variable arrays (in a ping-pong manner). This further helps the compiler in determining the correct order of pipeline stages, and associated dependencies, which is then used to exploit instruction parallelism.

Algorithm 3 Optimized 1D FFT Kernel

```
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[N]
5:      int L = N/2 , int m = 1
6:      #pragma unroll
7:      for Loop 1 →k = 0 : (N/2) − 1 do
8:          stage_1 ← COMPUTE(stage_0,L,m)
9:      L = L >> 1 , m = m << 1
10:             ⋮
11:     #pragma unroll
12:     for Loop logN →k = 0 : (N/2) − 1 do
13:         stage_logN ← COMPUTE(stage_logN-1,L,m)
14: end
```

5) Inferring Pipeline Registers as Registers: OpenCL limits the size of a register array in SWI kernels and converts them to BRAM based storage if that limit is exceeded. Since BRAMs are unable to source and sink data as easily as registers, the compiler is forced to employ inefficient hardware to meet the throughput constraints. This typically involves use of barrel shifters and memory replication, both of which can result in a manifold increase in memory usage and exceed device limits (even for simple designs). Moreover, the compiled pipelines may also not be able to launch iterations every cycle due to memory dependencies. Our solution here is to break down large arrays into smaller ones. Specifically, we use separate arrays for real and imaginary parts of all stage variables. This ensures that our variables are synthesized as registers. For large FFT sizes, these arrays would need to be further broken down to ensure the limit for register arrays is met.

6) Twiddle Factors: Since the size of interest for the FFT implementation is known beforehand, we implement twiddle factors as constant arrays instead of inputs to the kernel or real-time computations. The compiler identifies the floating point multiplication/dot-product units where these values are used and applies them at the corresponding inputs. This not only reduces pipelines latency, but also allows the compiler to eliminate floating point units in cases where the multiplication is with -1, 0 or 1.

C. Generated Hardware

Figure 1 shows the processing element generated for each iteration of each compute loop in our kernel. FIFOs implemented using registers are used to source and sink data to the pipelines, along with providing delays to synchronize pipelines and reducing the critical path. The compiler also infers the two different forms of dot products and generates appropriate modules (Dot2 and MDot2). As discussed before, for twiddle factor values (w_r and w_i) equal to -1, 0, or 1, dot product units are omitted and extra depth is added to the first input FIFO of the data-path. Processing elements are placed in a data parallel manner for a given stage and cascaded to implement inter-stage stall-free connectivity. Therefore, the FFT unit can sink and source an entire N-element 1-D vector every cycle.

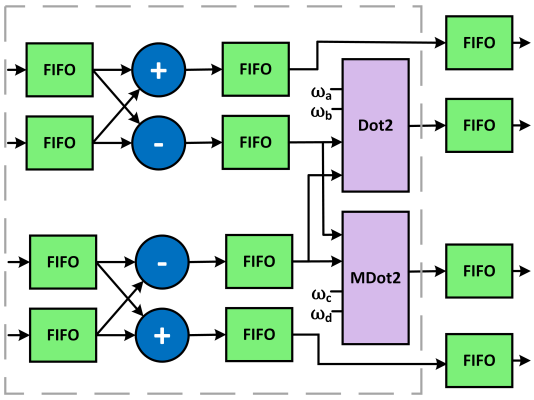

Fig. 1. OpenCL-HDL processing element architecture generated by the OpenCL compiler. Dot product units are inferred and implemented instead of individual adders/multipliers. Twiddle factors are declared as constants and hard coded to corresponding inputs of the dot product units.

IV. 3D FFT Shell

In this section, we demonstrate that both OpenCL-HDL and IP core based designs can utilize the same shell for performing a 3D FFT. Consequently, the former can be seamlessly integrated into existing logic initially designed for the latter. Since FFT is a linear operation, a high dimensional FFT can be broken down into a series of 1D directional transforms in any order. In order to avoid constructing complex and expensive memory structures that can stream data every cycle in all dimensions, a transpose is performed on the overall directional FFT result to reorder data within buffers. This reordering rotates the grid and allows a different directional 1D FFTs to be performed for the same data access pattern.

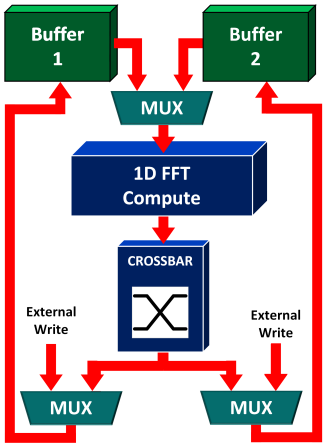

Fig. 2. Architecture for a 3D FFT using 1D FFT compute pipelines.

Figure 2 shows the typical architecture for performing a 3D FFT using 1D FFT modules. A set of ping-pong buffers are used to source and sink vectors. The initial grid is loaded into

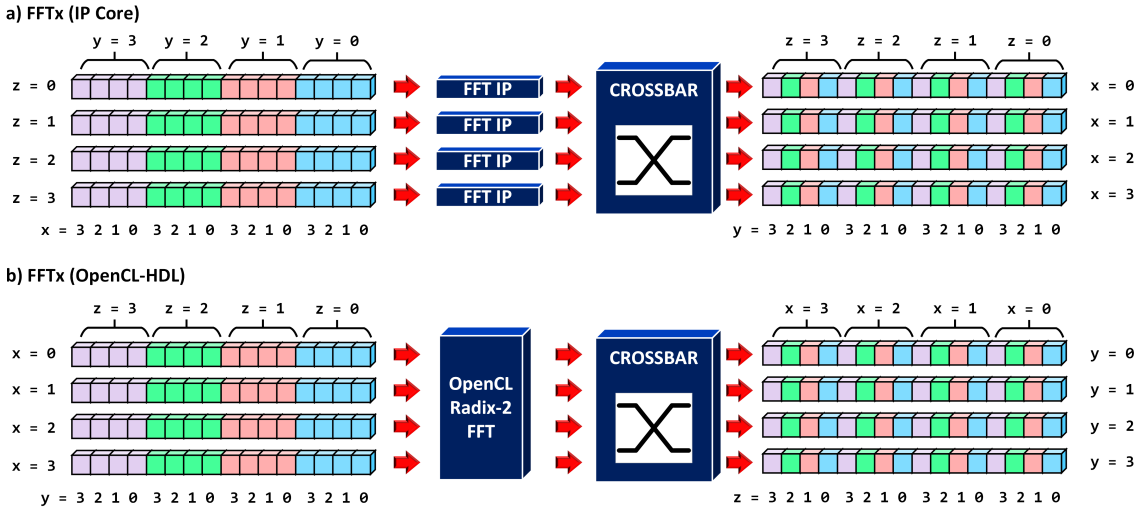

Fig. 3. Data source and sink patters for (a) IP Cores and (b) OpenCL-HDL pipelines. The same 3D FFT shell can be used for both since the access pattern is persistent. Only the dimensional order of FFT changes, but since FFT is a linear operation, the final result will remain the same.

Buffer 1 while the final result can be read from Buffer 2. By providing a variable data path for writes to both buffers using MUXs, the 3D FFT shell can be used as an intermediate stage for a number of different applications .

Figure 3 shows the detailed organization of Buffer 1 and 2 for 4 element, 1-D FFTs in the x-dimension. The throughput of both IP core and OpenCL-HDL Radix-2 implementations is constrained to be that of the Radix-2 design. Therefore, for an N^3 FFT, buffers are constructed using N banks of size N^2 each in order to stream the required N elements per cycle. The crossbar performs a transpose using a parallel to serial transformation since all N results per cycle are written to the same bank. It can operate stall free since the same output bank is accessed once every N cycles. From the figure, we observe that FFTx for OpenCL-HDL Radix-2 is the same as FFTy for IP cores. By extension, $FFTy(OpenCL-HDL) = FFTz(IPCore)$ and $FFTz(OpenCL-HDL) = FFTx(IPCore)$. Since FFT is a linear operation and the order is independent, $FFTxyz(IPCore) = FFTzxy(OpenCL-HDL)$. This means that we can replace FFT IP cores with OpenCL-HDL designs without having to modify almost any data or control paths in existing 3D FFT implementations. One potential exception (to the control path) is adjusting for differences in loading and unloading latencies.

V. Results

In this section, we presents two sets of results. The first is a comparison of implementation efficiency in terms of resource usage and latency, between FFT IP Cores and OpenCL-HDL for the same throughput. The second compares execution times for the OpenCL-HDL implementation with those of CPU, GPU and IP Cores.

A. Hardware Specifications

Our FPGA designs are implemented on the Altera Aria10X115 FPGA with 427,200 ALMs, 53Mb of BRAM, and 1518 DSP blocks. OpenCL code is compiled using Altera OpencL SDK v16.0.2. The IP Core used is [22]. Our CPU code is implemented on a fourteen-core 2.4 GHz Intel Xeon E5-2680v4 with ICC compiler and MKL DFTI [23]. The GPU used is NVIDIA TESLA P100 PCIe 12GB. It has 3584 Cuda cores and peak bandwidth of 549 GB/s. FFT code is written using cuFFT library [24] and compiled with CUDA 8.0.

B. Implementation Efficiency

Table I and II show the latency and resource usage for OpenCL-HDL and IP Core designs for $8^3, 16^3, 32^3$ and 64^3 FFTs. Figure 4 further illustrates the IP core resource usage as ratio of OpenCL-HDL usage for these FFT sizes. From the results we can see that OpenCL-HDL is significantly more resource efficient, with an average of 7.5x fewer ALMs used and 1.6x fewer DSP blocks. For N=64, we could not fit N parallel IP cores when using hard DSP blocks. Therefore, only 50 were implemented on DSP blocks while the remaining used ALMs. With regards to latency, the IP Core takes $2N$ cycles since input and output ordering is set to *Natural*. On the other hand, OpenCL-HDL latency grows by an average of only 1.4x for every 2x increase in FFT size.

TABLE I
Latency and Resource usage for OpenCL-HDL 1D FFT

FFT Size	Latency (cycles)	ALM	DSP
8	20	1,849(<1%)	56(4%)
16	37	4,387(1%)	168(11%)
32	41	7,237(2%)	456(30%)
64	53	18,705(4%)	1160(76%)

TABLE II
Latency and Resource usage for IP-Core 1D FFT

FFT Size	Latency (cycles)	ALM	DSP
8	16	26,759(6%)	96(6%)
16	32	11,132(3%)	256(17%)
32	64	63,322(15%)	832(55%)
64	128	176,285(41%)	1412(93%)

C. Performance

We compare the total execution time for $16^3, 32^3$ and 64^3 OpenCL-HDL against CPU, GPU and IP core implementations. Results (Table III) show that the average speedup achieved is 29x vs CPU-MKL, 4.1x vs GPU cuFFT and 1.1x vs IP Core FFT.

TABLE III
EXECUTION TIME (US) FOR 3D FFT IMPLEMENTATIONS

Design	16^3	32^3	64^3
CPU	24.0	196.9	1628.9
GPU	20.7	23.6	43.1
IP Core	1.8	6.8	31.1
OpenCL-HDL	1.8	6.6	25.8

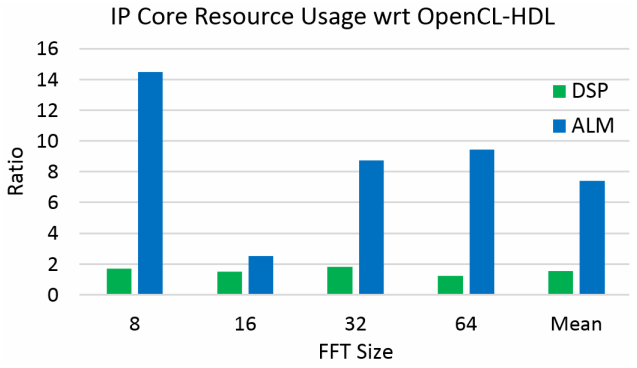

Fig. 4. IP core resource usage with respect to OpenCL-HDL. OpenCL-HDL designs consume both fewer ALMs and DSPs.

VI. CONCLUSION

In our work, we have demonstrated the effectiveness of OpenCL as an alternative to HDL in implementing complex FFT pipelines efficiently. Using a set of optimizations, the compiler can translate high level language code into an architecture that leverages all available forms and degrees of application specific parallelism on the FPGA. We highlighted the drawbacks of both the OpenCL toolflow, as well as the language itself, and presented an approach to isolating and using only the compute pipelines specified by users. The generated hardware, OpenCL-HDL, was integrated into an existing 3D FFT shell without having to modify nearly all interfaces. Our results show that OpenCL-HDL uses fewer ALMs and DSPs than IP core designs for the same throughput. Moreover, the OpenCL-HDL code was able to perform 3D FFTs of various sizes with lower execution times than corresponding CPU, GPU and IP core implementations.

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] M. Chiu and M. C. Herbordt, "Molecular Dynamics Simulations on High-Performance Reconfigurable Computing Systems," *ACM Trans on Reconfigurable Technology and Systems*, vol. 3, no. 4, p. 23, 2010.

[2] A. Sanaullah, C. Yang, Y. Alexeev, K. Yoshii, and M. C. Herbordt, "Real-Time Data Analysis for Medical Diagnosis Using FPGA-Accelerated Neural Networks," *BMC Bioinformatics*, 2018.

[3] A. Sanaullah, A. Khoshparvar, and M. C. Herbordt, "FPGA-Accelerated Particle-Grid Mapping," in *Field-Programmable Custom Computing Machines*, 2016, pp. 192–195.

[4] T. VanCourt, Y. Gu, and M. Herbordt, "FPGA acceleration of rigid molecule interactions," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2004.

[5] B. Sukhwani and M. Herbordt, "Acceleration of a Production Rigid Molecule Docking Code," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2008, pp. 341–346.

[6] B. Sukhwani and M.C. Herbordt, "FPGA Acceleration of Rigid Molecule Docking Codes," *IET Computers and Digital Techniques*, vol. 4, no. 3, pp. 184–195, 2010.

[7] S. Lee, *An FPGA Implementation of the Smooth Particle Mesh Ewald Reciprocal Sum Compute Engine (RSCE).* University of Toronto, 2005.

[8] C.-L. Yu, K. Irick, C. Chakrabarti, and V. Narayanan, "Multidimensional DFT IP Generator for FPGA Platforms," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 4, pp. 755–764, 2011.

[9] J. Andrade, V. M. M. da Silva, and G. F. P. Fernandes, "From OpenCL to Gates: The FFT." in *GlobalSIP*, 2013, pp. 1238–1241.

[10] B. Humphries, H. Zhang, J. Sheng, R. Landaverde, and M. C. Herbordt, "3D FFTs on a Single FPGA," in *Proc. Int. Symp. Field-Programmable Custom Computing Machines*. IEEE, 2014, pp. 68–71.

[11] J. Sheng, B. Humphries, H. Zhang, and M. C. Herbordt, "Design of 3D FFTs with FPGA clusters," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 2014, pp. 1–6.

[12] J. Sheng, C. Yang, and M. Herbordt, "Towards Low-Latency Communication on FPGA Clusters with 3D FFT Case Study," *Proc. Highly Efficient and Reconfigurable Technologies*, 2015.

[13] J. Sheng, C. Yang, A. Sanaullah, M. Papamichael, A. Caulfield, and M. C. Herbordt, "HPC on FPGA Clouds: 3D FFTs and Implications for Molecular Dynamics," in *Proc Field Programmable Logic and Applications*, 2017, pp. 1–4.

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

[15] 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.

[16] 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.

[17] 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. Int. Symp. Field-Programmable Custom Computing Machines*. IEEE, 2017, pp. 76–79.

[18] 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.

[19] U. Ulutas, M. Tosun, V. E. Levent, D. Büyükaydın, T. Akgün, and H. F. Ugurdag, "FPGA Implementation of a Dense Optical Flow Algorithm using Altera OpenCL SDK," in *International Conference on ICT Innovations*. Springer, 2017, pp. 89–101.

[20] P. Eastman and V. Pande, "OpenMM: a Hardware-Independent Framework for Molecular Simulations."

[21] 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.

[22] Altera, "FFT IP Core User Guide," www.altera.com/documentation/hco1419012539637.html, accessed: 2010-08-29.

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

[24] NVIDIA, "CUFFT library," http://docs.nvidia.com/cuda/cufft, 2018.