Accelerating Workloads on FPGAs via OpenCL: A Case Study with OpenDwarfs

Anshuman Verma†, Ahmed E. Helal†, Konstantinos Krommydas*, and Wu-Chun Feng*†
Department of Computer Science*, Department of Electrical and Computer Engineering†, Virginia Tech
Email: {anshuman, ammhelal, kokrommy, wfeng}@vt.edu

***Abstract*—For decades, the streaming architecture of FPGAs has delivered accelerated performance across many application domains, such as option pricing solvers in finance, computational fluid dynamics in oil and gas, and packet processing in network routers and firewalls. However, this performance comes at the expense of programmability. FPGA developers use hardware design languages (HDLs) to implement the application data and control path and to design hardware modules for computational pipelines, memory management, synchronization, and communication. This process requires extensive knowledge of logic design, design automation tools, and low-level details of FPGA architecture, this consumes significant development time and effort.**

To address this lack of programmability of FPGAs, OpenCL provides an easy-to-use and portable programming model for CPUs, GPUs, APUs, and now, FPGAs. Although this significantly improved programmability yet an optimized GPU implementation of kernel may lack performance portability for FPGA. To improve the performance of OpenCL kernels on FPGAs we identify general techniques to optimize OpenCL kernels for FPGAs under device-specific hardware constraints. We then apply these optimizations techniques to the OpenDwarfs benchmark suite, which has diverse parallelism profiles and memory access patterns, in order to evaluate the effectiveness of the optimizations in terms of performance and resource utilization. Finally, we present the performance of structured grids and N-body dwarf-based benchmarks in the context of various optimization along with their potential re-factoring. We find that careful design of kernels for FPGA can result in a highly efficient pipeline achieving 91% of theoretical throughput for the structured grids dwarf.

***Index Terms*—OpenDwarfs; FPGA; OpenCL; GPU; MIC; Accelerators; Performance Portability**

I. Introduction

For decades, the streaming architecture of FPGAs has delivered accelerated performance across many application domains, such as option pricing solvers in finance [1], computational fluid dynamics in oil and gas, and packet processing in network routers and firewalls. However, this performance comes at the expense of programmability, i.e., the performance-programmability gap. FPGA programmers use hardware design language (HDL) to implement the application data path and to design hardware modules for computational pipelines, memory management, synchronization, and communication interfaces at the Register Transfer Level (RTL), i.e., the programmers must specify the cycle-accurate behavior for the data path in every module and register in the design [2]. This process is similar to programming traditional CPUs in assembly language with the additional complexity of scheduling the instructions and data on a cycle-by-cycle basis, which requires extensive low-level knowledge of the target architecture and consumes significant development time and effort. In contrast, GPUs took the parallel computing community by storm in the late 2000s by significantly enhancing the programmability of GPUs via higher-level programming abstractions for general-purpose computing, namely CUDA and OpenCL. To address this lack of programmability of FPGAs, OpenCL provides an easy-to-use and portable programming model for CPUs, GPUs, APUs, and now, FPGAs [3] . However, this significantly improved programmability and portability can come at the expense of performance. Although FPGA compilers for OpenCL can generate functionally-correct hardware designs from architecture-agnostic OpenCL kernels, it is unlikely that these designs will utilize the FPGA resources efficiently to meet the required performance; that is, there still remains a performance-programmability gap.

In this paper, we use the OpenDwarfs benchmark suite [4], a suite of architecture-agnostic OpenCL kernels that capture common computation and communication patterns across a wide spectrum of scientific and engineering applications, to study the performance of the OpenCL programming model on FPGAs. In OpenDwarfs, none of the benchmark contains optimizations that favor a specific architecture over another.

Our contributions in this paper are following:

1) We assess the performance gap between fixed and reconfigurable architectures by characterizing the performance of a small subset of benchmarks in the OpenDwarfs benchmark suite on multi-core CPUs, GPUs, Intel MIC and FPGAs. We show that architecture-agnostic OpenCL kernels result in inefficient hardware designs on FPGAs *(Section III)*.
2) To improve the performance of OpenCL kernels on FPGAs, and thus, bridge the performance-programmability gap, we identify general techniques to optimize OpenCL kernels for FPGAs under device-specific hardware constraints. We then apply these optimization techniques to two example cases from OpenDwarfs benchmark suite. in order to evaluate the effectiveness of the optimizations in terms of performance and resource utilization and present the performance of the optimized implementations *(Section IV)*.
3) A realization of an efficient algorithm for a three-dimensional stencil using OpenCL for FPGA and an evaluation of its performance *(Section V)*.

This work was supported in part by NSF I/UCRC IIP-1266245 via the NSF Center for High-Performance Reconfigurable Computing (CHREC).

Rest of the paper is structured as follows: in Section II, we discuss the motivation behind our work, followed by performance of benchmarks across different architectures in Section III. In Section IV we discuss compiler optimizations using the Altera OpenCL compiler and provide results for these optimizations applied to the benchmarks under study followed by an accelerator design for 3D-Stencil using OpenCL in Section V. Section VI presents related work and Section VII concludes the paper.

II. Motivation and Background

OpenCL is a portable and standard programming model for heterogeneous systems that typically consists of a hierarchical array of processing elements and memory structure. At a high level, OpenCL defines a unified and abstract machine model for the different many-core architectures to provide both portability and programmability. The target architecture consists of multiple compute units (CUs) that share a single global memory and constant memory space. The global memory can be used across CUs. Each CU has its own local memory and contains multiple processing elements (PEs) that share this local memory. In addition, each PE has a low-latency private memory. Using OpenCL, the programmer can control the parallelism at different granularity levels, such as task-level parallelism and data-level parallelism, and manage data movement between memory levels.

Unlike traditional high-level synthesis (HLS) programming models [5] for FPGAs, OpenCL is explicitly parallel, which allows OpenCL-FPGA compilers to automatically generate hardware accelerator from the OpenCL kernel implementation based on the available resources on the target reconfigurable fabric. Therefore, HLS on FPGAs using OpenCL has the potential to design a custom hardware accelerator that matches the application's characteristics and improve performance and power efficiency [6].

Although OpenCL's abstract machine model allows programmers to write their applications once and run them on multiple architectures, including CPUs, GPUs, Intel MIC, and FPGAs, it is unlikely that these applications will efficiently utilize the underlying hardware architecture, i.e., OpenCL provides functional portability but not performance portability. For example, CPUs favor task-level parallelism (due to their limited vector units) and have special hardware that implicitly utilizes data locality and reduces memory access latency; in contrast, GPUs require massive data-level parallelism, and the programmer is responsible for reducing the memory access latency by explicitly utilizing the data locality. In FPGAs, the problem of efficient utilization of the target hardware is even more complicated, as the programmers have access to an array of logic elements and embedded memory blocks that can be configured to be CPU-like, a GPU-like or an application-specific architecture.

III. Performance Characterization

In this paper, our main goal is to study the performance of the OpenCL programming model on FPGAs using the OpenDwarfs benchmark suite. First, we assess the performance gap between fixed and reconfigurable architectures by characterizing the performance of the OpenDwarfs benchmark suite on multi-core CPUs, GPUs, Intel MIC and FPGA.

TABLE I: OpenDwarfs Benchmarks Used

Dwarf	Benchmark	Input data
N-body Methods	GEM	nucleosome 80 1 0
Structured Grid	3D Stencil	$8 * 256^3$

TABLE II: Specification of Test Architectures

Model	Intel i5-2400	Intel Xeon E5-2700	Intel MIC 7100	Tesla C2070	Tesla K20X
Type	CPU	CPU	Co-proc.	GPU	GPU
Freq. (GHz)	3.1	2.7	1.238	1.15	0.732
Cores	4	12	61	14	14
SIMD (SP)	8	8	16	32	192
GFLOPS (SP)	198.4	518.4	2415.6	1030	3950
On-Chip mem.	7.125	33.375	32.406	3.375	3.032
B/W (GB/s)	21	59.7	352	148.42	250
Process (nm)	32	22	22	40	28
TDP (W)	95	130	270	238	235

Table I presents the OpenDwarfs subset considered in this study, i.e., structured grid (3D stencil) and N-body (GEM), and their input data-sets and/or parameters. Table II lists the target fixed architectures. Our FPGA board is the BittWare S5-PCIe-HQ-D8 board with high-density Altera Stratix V FPGA (28nm process), supporting the Altera OpenCL SDK v14.0. Additional details about FPGA used in study are provided in Table III.

A. Benchmarks

1) Stencil: In "structured grid" algorithms, computation proceeds as a series of update steps to a regular grid data structure of two or more dimensions. The 3D stencil from OpenDwarfs is a structured grid that solves partial differential equations by applying a 7-point finite-difference algorithm on

TABLE III: FPGA Details

Device	ALM	Block Memory(bits)	DSP Blocks
5SGSMD8K2F40C2	262,400	52,572,160	1963

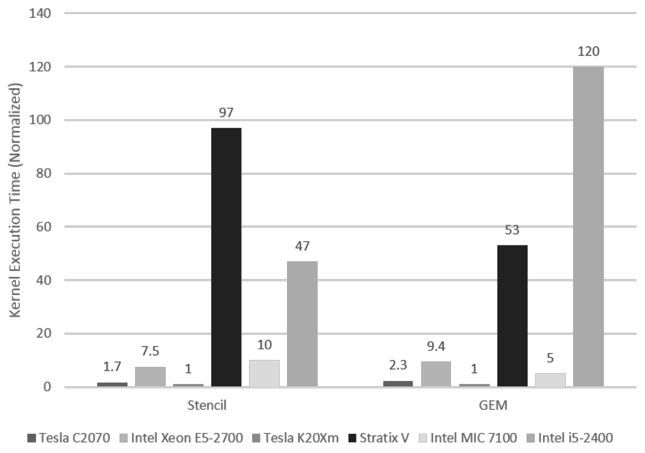

Fig. 1: Architecture-Agnostic Kernel Performance for Stencil and GEM Normalized to TeslaK20Xm (lower is better)

a three-dimensional (3D) regular grid. In the 7-point finite-difference code, each cell value is computed as a weighted average of its six immediate neighbors along the X,Y and Z dimensions, which are the east, west, north, south, top, and bottom cells. The 3D stencil, similar to all stencil computations, is a regular, memory-bound algorithm with low computational intensity, where each grid cell can be updated independently. However, synchronization is required before proceeding to the next grid update step. Therefore, it requires a large number of PEs and high memory-bandwidth, thus making suitable for GPU architectures.

2) GEM: N-body algorithms are characterized by all-to-all computations within a set of particles. In GEM, the electrostatic surface potential of a bio-molecule is calculated as the sum of charges contributed by all atoms in the bio-molecule due to their interaction with a specific surface vertex (two sets of bodies). The algorithm complexity is $O(N \times M)$, where N is the number of points along the surface, and M is the number of atoms.

B. Characterization

1) Stencil: An efficient GPU version of the 3D-Stencil algorithm divides the problem into smaller three-dimensional blocks, where each element of the smaller blocks is a computed by a separate thread. Each thread in a thread-block fetches the element to operate upon and waits for other threads to reach a barrier. Once each of the threads have reached the barrier, threads proceed to calculate the function and then wait at a barrier, once each element on the plane is computed; threads proceed to compute the next plane. Figure 1 shows the performance of the Stencil across fixed and reconfigurable architectures. Tesla K20XM and Tesla C2070 outperform other architectures for this implementation owing to efficient intra-block synchronization and utilization of (fast) shared memory, while former doing better than latter due to higher memory bandwidth. Intel MIC 7100 performs better than CPUs(Intel Xeon E5-2700 and Intel i5- 2400) because of more number of cores and higher memory bandwidth. FGPAs perform poorest due to the large synchronization overhead.

2) GEM: GEM is a regular compute-bound algorithm, given that atoms' data are reused, as each thread independently accumulates the potential at a single point due to every atom in the molecule, requiring a large number of processing elements (PEs) and being sensitive to data locality. Hence, GPUs with their massive number of PEs achieve the best performance (Figure 1).

Architecture-agnostic implementation of the OpenCL kernels results in inefficient hardware designs on FPGAs. Thus, we detail optimization strategies to improve the performance on the FPGA in Section IV.

IV. FPGA Optimizations and Insights

To improve the performance of OpenCL kernels on FPGAs, we exploit different levels of parallelism — task, data (i.e., SIMD vectorization) and pipeline parallelism — and minimize memory access latency by controlling data movement across the memory hierarchy levels and coalescing memory accesses. Because FPGAs have limited hardware resources and memory bandwidth, it is imperative that we analyze different combinations of these optimization techniques to identify the best set and generate the most efficient (in terms of performance and resource utilization) hardware design for the OpenDwarfs under consideration. We explore the FPGA-oriented optimization space and articulate insights that may be leveraged in future OpenCL compilers targeting the FPGA.

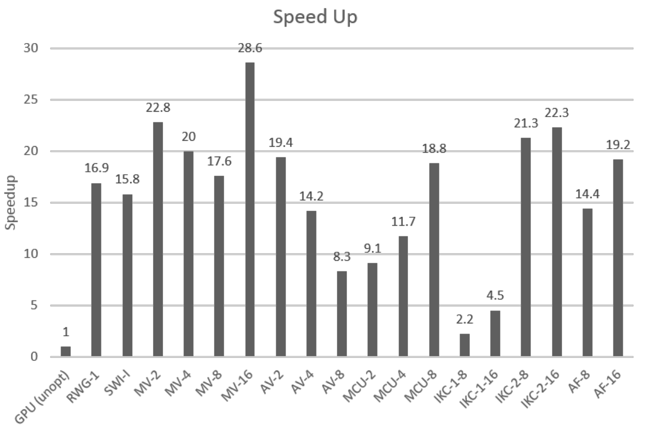

Fig. 2: Optimized Stencil Kernel implementations (Higher is better)

In the following sections we use the OpenCL-specific terminology listed below (for details refer to the OpenCL specification [7]):

- *NDRange*: The host program invokes a kernel over an index space called *NDRange*. It can be a 1-, 2-, or 3-dimensional space.
- *Work-item*: A single kernel instance at a point in the index space is called a *work-item.*
- *Work-group*: Work-items are grouped into *work-groups* that are conceptually scheduled together for execution.
- *Single work-item kernel*: A kernel specified to run as a one-dimensional NDRange, with a work-group size of 1 is called *single work-item* kernel.

A. 3D Stencil Benchmark

To improve the poor performance of NDRange kernel on FPGA, we employ different optimizations techniques. Abridged version of these optimization results are following: Rewriting the kernel as single work-item improved performance. Static memory coalescing, and algorithmic re-factoring further increased the performance of the single work-item kernel. Conversely, designs with multiple compute units, compiler vectorization, or data sharing among work-items using intra-kernel channels resulted into little or no performance improvement or even performance degradation. Insights and details of the these optimizations are discussed in the following subsections.

1) Single Work-Item Kernel: FPGAs benefit from pipeline parallelism when work-items are launched every cycle. This allows a result to be produced every cycle, once the pipeline is full. Programming a kernel as a single work-item enables the compiler to attempt creating an efficient pipeline. However, pipeline stalls reduce the throughput. Pipeline stalls can occur

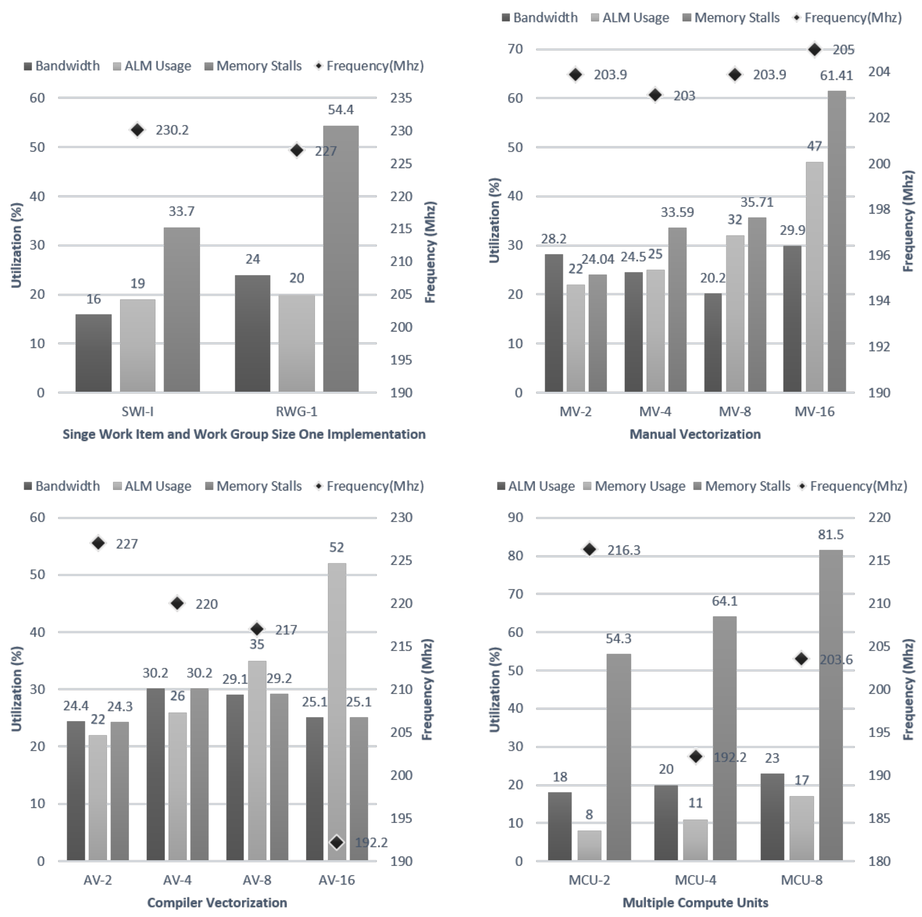

Fig. 3: Single Work-item Kernel, Manual and Compiler Vectorization, Muliple Compute Units: Profiling (Bandwidth, ALM Usage, Memory Stalls, and Clock Frequency

for many reasons. For an example, if the number of memory read accesses per cycle exceeds the number of available read ports then compiler inserts an arbiter to provide more ports. This, in turn, increases access latency and results in pipeline stalls and, overall, an inefficient implementation. Barriers can also result in pipeline stalls. The stencil NDRange kernel (as originally designed for the GPU) results in sub-par performance when compiled and synthesized for the FPGA (Stratix V in Figure 1, GPU(unopt) in Figure 2).

To benefit from pipeline parallelism on the FPGA, we implement the kernel as a single work-item kernel (SWI-1, Listing 1). An improvement of approximately 16x (SWI-1 in Fig 2) was obtained for the FPGA over the unoptimized kernel originally designed for the GPU for a grid size of $(256)^3$ double-precision floating-point elements. This kernel is launched with a work-group size of 1 and dimension of the grid provided using arguments.

We implemented a similar version (RWG-1, Listing 2) with a work-group size of 1, but without the loop and launch the kernel with a work-group size of dimension and block size 1. RWG-1 results into a larger footprint on the FPGA (has additional 5% ALM usage) and performs slightly better than SWI-1. Furthermore, it results in better memory bandwidth because of larger burst size and more efficient coalescing, as indicated by bar RWG-1 in Figure 3. Both kernels perform the exact same operations, so this difference in area could potentially be attributed to compiler optimizations done for memory access coalescing. We also noted during our experiments that SWI-1 is a better style of writing kernel for FPGA because of the benefits in using the shared memory and private registers across work items without introducing a barrier or stalls in pipeline.

```
__attribute__((reqd_work_group_size(1,1,1)))
kernel_stencil_db(dimension, in,out) {
    for(i=0; i < dimension; i++) {
        fetch_neighbors();
        do_stencil();
    }
}
```

Listing 1: Single Work-item (SWI-1)

```
__attribute__((reqd_work_group_size(1,1,1)))
kernel_stencil_db(in,out) {
    int id = get_group_id(0);
    fetch_neighbors(id);
    do_stencil();
}
```

Listing 2: Work Group Size One (RWG-1)

Insight*: Single work-group size kernels are suitable for FPGA.*

2) Manual Vectorization or Static Coalescing: FPGAs have limited memory bandwidth but high computational capability. Efficient memory accesses can potentially accelerate the application, and this is particularly true for 3D stencil. In order to improve the performance of RWG-1, we apply manual vectorization with SIMD width of 2, 4, 8 and 16. Figure 2 shows the corresponding performance in bars MV-2, MV-4, MV-8, and MV-16, respectively. MV-16 results in the highest memory bandwidth and, accordingly, in the best performance with a speed up of 29x over the unoptimized kernel. This improved performance comes at the cost of area usage (Figure 3). We did not observe an increase in memory usage by applying manual vectorization owing to the simplicity of the stencil benchmark's kernel. Increasing the SIMD length results into higher number of stalls, but performance gains are still obtained due to increased memory bandwidth.

Insight*: Manual vectorization may result into efficient memory coalescing and can potentially augment the performance.*

3) Compiler Vectorization: An OpenCL directive hints the Altera OpenCL compiler to attempt automatic code vectorization. This requires a work-group size that is a multiple of vectorization length. We choose the RWG-1 implementation to evaluate compiler-generated vectorization. Bars AV-2, AV-4, AV-8, and AV-16 in Figure 3 correspond to the performance for vectorization width of 2, 4, 8, and 16 respectively. Being a memory-bound application, stencil benchmark does not benefit from compiler vectorization. Specifically, although burst size increases because of memory access coalescing, high percentage of stalls results in poor bandwidth. Performance deteriorates for RWG-1 with increase in the vectorization width. This optimization is not as useful for memory-bound algorithms, as opposed to compute-bound. Finally, increase in vector width results in lower clock frequency and higher resource utilization (ALM usage) (Figure 3).

Insight*: Compiler vectorization may not result into improvement of performance for memory bound algorithms.*

4) Multiple Compute Units: Performance gains can be obtained by replicating the pipeline of single work-item kernels. However, since stencil implementation RWG-1 is memory-bound, we do not expect, nor obtain, any performance gain with respect to RWG-1 by this optimization, as shown by bars MCU-2, MCU-4, and MCU-8 in Figure 2, which correspond to 2, 4, and 8 compute units respectively. However, it is worth observing that the area usage, clock frequency, and memory stalls percentage (Figure 3) grow linearly as the number of compute units are increased. Clock frequency remains roughly the same for all cases of compute units.

Insight*: Multiple compute units optimization may not enhance performance for memory bound algorithms*

5) Intra-Kernel Channels: Altera introduces *channels*, a custom solution that functions as the equivalent to OpenCL-2.0 [7] pipes. Channels are communication links that can be used to transfer data among kernels. This eliminates the need for memory transfers between the host and device and the corresponding data transfer overhead. To enhance the data locality for RWG-1, we perform computation along the X

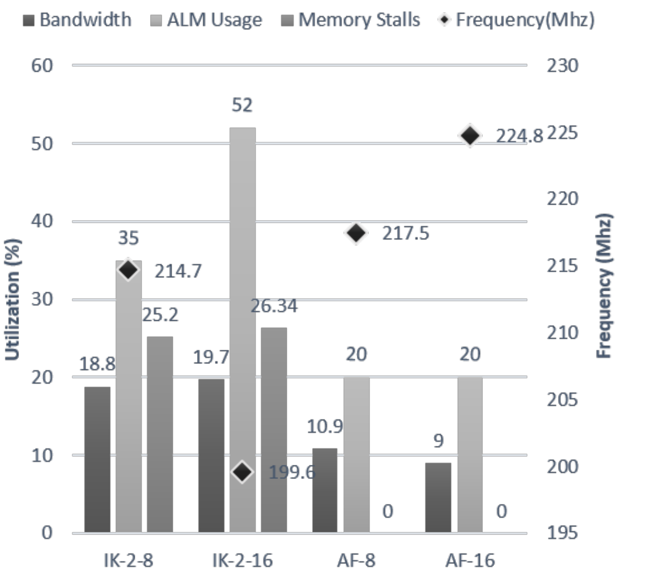

Fig. 4: Data Sharing in X direction, and Algorithmic Refactoring: Profiling

direction in blocks of 8 and 16 double words and reuse the data in the X direction. We use the channel to transfer data among the work items in the same kernels. The above optimization results into poor kernel design, as synthesized by Altera OpenCL compiler, and instead of improvement, we notice a slowdown compared to RWG-1 attributed to the channel's latency (IKC-1-8, IKC-1-16 in Figure 2), which is stalled 98.7% of the time for both implementations. The above optimization applied for SWI-1 implementation using private registers results in performance improvement (IKC-2-8 and IKC-2-16 in Figure 2) over SWI-1. This difference in performance can be likely attributed to channel-related latency considerations in the pipeline implementation.

Insight*: Intra-kernel channels may degrade the performance without proper latency considerations. However, increasing data locality for single work-items definitely improves performance.*

6) Algorithmic Refactoring: To improve data locality in Stencil, we implement an algorithm similar to the GPU implementation. The problem in 3D space is divided into smaller blocks, where each smaller block has the same height, but smaller size in the X and Y dimensions. Data is fetched into a cyclic buffer of size $(2*(x+2)*(y+2)+1)$, where x and y are the size of the smaller blocks in X and Y dimensions. An additional 2 elements (added to x) is needed to account for the boundary elements in the smaller blocks. This implementation is discussed in more detail in Section V, where we focus on designing a fixed plane size problem. Implementation for an 8x8 plane size did not result into better performance (AF-8 in Figure 2) because of poor memory coalescing. However, a 16x16 plane size (AF-16 in Figure 2) enables performance gains mainly due to higher data-reuse. Moreover, AF-8 and AF-16 result into a smaller foot-print on the FPGA (Figure 4) compared to SWI-1 and no memory stalls. The Altera OpenCL compiler could not statically resolve the complex calculations on array indices. This would have enabled memory coalescing

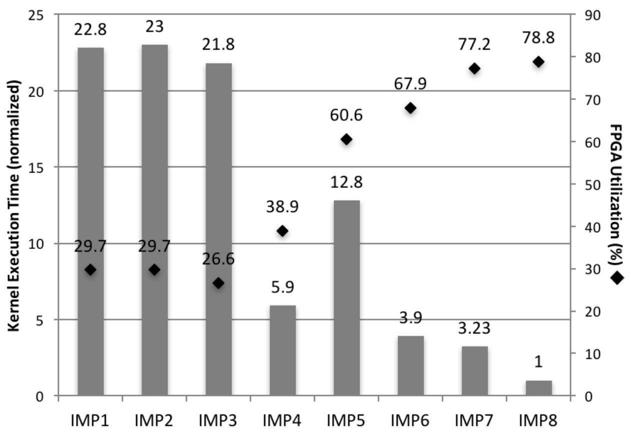

Fig. 5: Optimized GEM Kernel implementations

Implem.	Refact.	Restrict	Constant	SIMD	CU	Unroll
IMP1				1	1	1
IMP2			✓	1	1	1
IMP3	✓		✓	1	1	1
IMP4		✓	✓	1	1	1
IMP5			✓	1	1	4
IMP6			✓	8	1	1
IMP7	✓		✓	16	1	1
IMP8		✓	✓	8	1	1

TABLE IV: GEM Kernel Implementations' Features

and corresponding performance gains. However, the above optimization can be applied and its benefits are observed when the algorithm is explicitly coded with a fixed plane size of grid, as we discuss in Section V.

Insight*: Kernels designed with taking into consideration the FPGA memory hierarchy and pipeline parallelism are most likely to exploit the benefits of reconfigurable computing.*

B. GEM Benchmark

1) Use of Restrict/Const Keywords and Kernel Vectorization: An optimization strongly suggested by Altera is use of the *restrict* keyword for kernel arguments that are guaranteed to not alias (i.e., point to the same memory location). Using *restrict* allows more efficient designs in terms of performance by eliminating unnecessary assumed memory dependencies. Although a side effect of such an optimization could be lower resource utilization, we find that this is not the case in our application. Cases IMP2 and IMP4 (Figure 5) highlight the difference (1.31 times higher utilization with *restrict*) across two otherwise identical implementations. Performance-wise, IMP4 is 3.94 times faster and this stems from the vast majority of memory accesses resulting in cache hits. Conversely, IMP2 is characterized by sub-optimal memory accesses that result in cache misses and pipeline stalls (about 80% of the time). As far as *const* keyword is concerned we observe no difference neither in resource utilization, nor in execution time. Automatic kernel vectorization (SIMD), which is enabled with the appropriate OpenCL attribute, can yield easy performance gains of 5.85x (IMP4, IMP8) at the cost of increased (double) resource utilization.

Insight*: Using restrict keyword may result into higher performance but not necessarily lower footprint on FPGA*

2) Compiler Resource-Driven Optimizations: In compilation with resource-driven optimization the compiler applies a set of heuristics and estimates resource utilization and throughput given a number of kernel attributes, like loop unroll factor, kernel vectorization, number of compute units. This process should not be always expected to provide the best implementation. In our example application, we identify at least one case where manual choice of kernel vectorization width surpasses (by 3.33x) the compiler-selected attributes (*pragma unroll 4*) (IMP6, IMP5 in Figure 5). Profiling the kernel, we find that IMP6 benefits from coalesced memory accesses, while memory accesses in IMP5 result in costly pipeline stalls. Also, bandwidth efficiency is higher (more than double) in IMP6 (i.e., more of the data acquired from the global memory system is actually used by the kernel). Altera discusses the inherent limitations of static resource-driven optimizations in their optimization guide [8]. Developers should consider the aforementioned limitations when compiling using the resource-driven optimization option.

Insight*: Manual search of optimization space can out perform automatic compiler resource driven optimizations.*

3) Algorithmic Refactoring: A given algorithm implementation may solve an actual problem, but this does not mean that a set implementation is appropriate for every platform (e.g., CPU, GPU, FPGA). A different implementation for solving the same problem, i.e., produce the same output given the same input, may be necessary. While this may not be intuitive, or even applicable for all cases, certain algorithmic restructuring can prove very beneficial. To illustrate the above, we apply basic algorithmic refactoring in our example application. Specifically, we remove the complex conditional statements for different cases encapsulated in a single kernel, and tailor the kernel to the problem at hand. This provides a two-fold benefit: (a) better resource utilization (in our examples the refactored algorithm requires about 10% less FPGA resources, and (b) better performance (5% faster, IMP2, IMP3). What is more important, though, is that better resource utilization may allow wider SIMD or more compute units to fit in a given board. In our example (IMP6, IMP7 in Figure 5), the reduced resource utilization of the refactored algorithm allows SIMD length of 16, whereas the original one accommodated up to 8 (logical elements being the limiting factor). This translates to an 1.22x faster execution of the former compared to the latter.

Insight*: Our experiments with GEM refactoring reiterates our experiments with Stencil, kernel designed with FPGA architecture in consideration has the potential to exploit reconfigurable architecture*

V. Accelerator for 3D-Stencil

We design a FPGA accelerator for 3D-Stencil computation using OpenCL for a 32x32 double-word plane size and variable height. This implementation is similar to the streaming element for stencil as discussed by Sano et al. [9]. Data from memory is fetched into a cyclic buffer that has a size of (32x32x2+1) double words. Once the first two planes are fetched into local memory, computation can proceed as new data is being fetched. Full utilization of memory bandwidth

```
int sh_register[size];
#pragma unroll
for(int i = 0; i< size; i++)
    sh_register[i] = 0;

for(int k = 0; k < loop_count; k++) {
    #pragma unroll
    for(int i=0; i < (size - 1); i++)
        sh_register[i] = sh_register[i+1];

    sh_register[size-1] = input_data ;
}
```

Listing 3: Shift Register Implementation in OpenCL

stems from enhanced data reuse and locality. This implementation generates an efficient pipeline and in theory can perform one computation per data fetch, once the pipeline is full. In following subsections we discuss refactored algorithm, followed by optimizations used to design it in OpenCL, and do an evaluation of design generated by Altera compiler.

A. Refactored Algorithm

Listing 4 shows the pseudo-code for stencil implementation using a single work-item kernel. The outermost loop (variable i) iterates over the planes, the inner loop (variable j) iterates over rows, and the innermost loop (variable k) processes elements in column. Sustaining highest throughput i.e. computation of a grid-point in single cycle requires six read requests to local memory, but the available memory on board has only two ports. To avoid the need of an arbiter on the buffer, which would result in a higher latency on read request completion, the cyclic buffer is divided into multiple banks acting together as a large buffer that comprises 65 smaller buffers of size 256 bytes. Dividing the buffer into multiple smaller buffers resolves contention and reduces pipeline stalls. The benefit of splitting reflects in minimal memory stalls while unrolling the innermost loop up to 4 in Figure 6.

B. Implementing Cyclic Buffer

This implementation employs shift register inference optimization, as explained in the OpenCL optimization guide. Listing 3 is an implementation of shift registers in OpenCL using compile pragma (*#pragma unroll*). Shift registers are a powerful tool and are used extensively in hardware design. An accelerated implementation for dynamic programming [10] uses shift registers to resolve complex dependencies among elements in the Smith-Waterman algorithm for sequence matching. Small shift registers can be implemented in the register space while large shift registers can be implemented using block-RAMs (BRAM) on the FPGA.

C. Loop Unrolling

The FPGA implementation of the above algorithm without any loop unrolling suffers from low memory bandwidth utilization. Unrolling the innermost loop (variable k) results

```
#define N 32
#define SIZE (2*N*N +1 )
__attribute__((reqd_work_group_size(1,1,1)))
__kernel kernel_stencil(in_d,out_d,dim_z)
    __private double sh_mem[SIZE];
    for(int i = 0; i < dim_z; i++)
        for(int j = 0; j< N; j++) {
            #pragma unroll <UNROLL_FACTOR>
            for(int k = 0; k< N; k++) {
            __private double neighbors[6];
            __private double in;
            in = in_data[index + k];
            get_neighbors_from_buffer();
            out_d[index+k-N*N] =
                compute_stencil(neighbors,in);
            sh_mem[size -1] = in;
            shift_registers;
        }
    }
```

Listing 4: Pseudo Code For 3D-Stencil Implementation

in better utilization of memory bandwidth, as more compute units are generated. Higher number of compute units in turn increase the number of buffer accesses per cycle and results in diminishing bandwidth gains. An unroll factor of 8 results in a high number of memory stalls, however, performance still increases due to increased memory bandwidth. The gain in bandwidth does not scale with unrolling.

Insight*: Loop unroll can help improve utilization of under-utilized memory bandwidth if unrolling does not increase local memory access latency, after which there are diminishing returns.*

D. Evaluation

Our implementation of 3D-Stencil algorithm without any unrolling achieves the near-maximum possible throughput for a single pipeline. A pipeline operating at 228.4 Mhz and working on 8-bytes of data imposes a bandwidth requirement of $228.4 * 8 = 1.872$ GBps for computation of one point of grid in each cycle. Our implementation utilizes 6.678% of memory bandwidth or $1.709 GBps$, translating to an efficiency of 91.32% of theoretical throughput. This design performs stencil computation for one point per cycle, and each computation performs six floating point operations translating into a performance of 1.25 GFlops. This floating point performance increases to 7 GFlops with an unroll factor of 8. Figure 6 shows the speed-up attained using unrolling for heights of 2048, 16384, and 131072 elements. This speed-up does not scale linearly with unrolling because a larger unrolling factor results into increased stalls introduced by memory banks.

VI. Related Work

Many approaches have been proposed to address the challenges of programming FPGAs by abstracting the hardware implementation details using high-level programming models. Traditional high-level synthesis (HLS) tools, such as Catapult C, AutoPilot, Handel-C and C-to-Silicon, use a variation of C/C++ with non-standard and special language extensions [5].

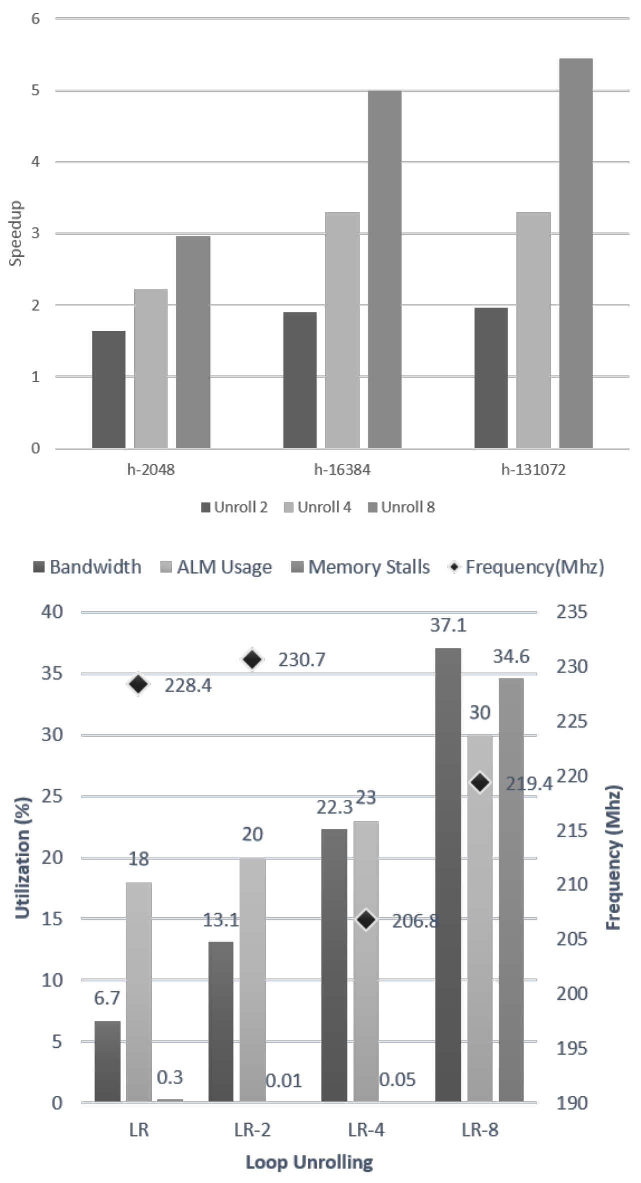

Fig. 6: Speedup and Profiling Information For Unroll

Moreover, these C-like programming models are inherently sequential and lack the specifications for concurrency and data transfer, which usually result in poor hardware design [11]. Unlike traditional HLS approaches, OpenCL is a standard and portable programming model. In addition, it is explicitly parallel, which allows software developers to control the parallelism at different granularity levels, and manage data movement explicitly through the memory hierarchy. Hence, OpenCL has the potential to generate efficient hardware designs that matches the applications characteristics [2].

After the release of Altera OpenCL compiler in 2013, which is the first OpenCL-to-FPGA compiler to pass Khronos conformance test, several research studies investigated the use of this HLS framework in diverse application domains including option pricing [1], information filtering [12], embedded systems [13], DSP and image processing [14], [15], [13].

Inggs et al. [1], used an options pricing benchmark (with Black-Scholes and Heston models) to evaluate the performance of several HLS tools including Altera OpencL. The authors analyzed several optimization techniques such as pipelining, task-level parallelism and data blocking, and showed that FPGAs achieve two order of magnitude speedup over sequential CPU, when programmed with optimized OpenCL implementation. In [12], Chen et al. demonstrated an efficient FPGA design of a document filtering algorithm, implemented using optimized OpenCL kernels. The authors explored the OpenCL optimization space on FPGAs, and showed that their FPGA implementation outperform CPUs and GPUs by a factor of 5 in terms of performance per watt.

In [14], the authors studied the performance and programmability of OpenCL in comparison with VHDL using three image processing kernels. The results showed that OpenCL improve the productivity by 6 folds, while consuming up to 70% more resources than VHDL designs. Chen et al. [15] evaluated CPUs, GPUs and FPGAs using fractal video compression application implemented in OpenCL. The authors showed that with application-specific optimizations, FPGAs achieved 3x and 114x speedup over GPU and CPU respectively.

CHO [13] is an OpenCL benchmark suite for FPGAs that contains workloads implemented using the single work-item programming model. Although CHO kernels are drawn from real world applications, they are limited to DSP and embedded computing domain, which is the traditional domain for reconfigurable computing architectures. In addition, since the benchmark suite is intended to evaluate the effectiveness of FPGA HLS tools, the authors don't provide optimized kernels.

Unlike previous approaches, we addresses the challenges of programming FPGAs using OpenCL in high-performance computing (HPC) and scientific computing workloads, which are the traditional domain for many-core accelerators such as GPUs and Intel MIC.

VII. Conclusion and Future Work

In this work we discuss the performance-programmability gap in employing OpenCL to program an FPGA. We started with GPU-favorable implementations of certain OpenDwarfs applications, ported them to the FPGA without architectural considerations and explored compiler optimizations to accelerate them. Further, we explored the refactoring a 3D Stencil algorithm, implemented a highly efficient design using OpenCL, and showcased the performance. We show that OpenCL programming model can indeed be used to design efficient accelerators. We exhibit the potential of reconfigurable architectures with high programmability and no-so-detailed necessary knowledge of digital design for an end user. In future work we plan to design accelerators using OpenCL for a broader set of algorithms from the OpenDwarfs benchmark suite, identify appropriate optimizations based on common computation and communication patterns, as identified by the dwarfs categorization, and subsequently incorporate them into an OpenCL compiler for FPGA.

References

[1] G. Inggs, S. Fleming, D. Thomas, and W. Luk, "Is High Level Synthesis Ready for Business? An Option Pricing Case Study," in *FPGA Based Accelerators for Financial Applications*. Springer International Publishing, 2015, pp. 97–115.

[2] S. Windh, X. Ma, R. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. Najjar, "High-Level Language Tools for Reconfigurable Computing," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 390–408, March 2015.

[3] T. S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D. P. Singh, and S. D. Brown, "OpenCL for FPGAs: Prototyping a Compiler," in *International Conference on Reconfigurable Systems and Algorithms 2012*.

[4] K. Krommydas, W.-c. Feng, C. D. Antonopoulos, and N. Bellas, "OpenDwarfs: Characterization of Dwarf-Based Benchmarks on Fixed and Reconfigurable Architectures," *Journal of Signal Processing Systems*, pp. 1–20, 2015.

[5] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 18–25, July 2009.

[6] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh, "From OpenCL to High-Performance Hardware on FPGAS," in *FPL*, Aug 2012.

[7] K. O. W. Group *et al.*, "The OpenCL Specification, Version: 2.0 Document Revision: 22," *URL http://www. khronos. org/registry/cl/specs/opencl-1.0*, vol. 29, 2014.

[8] *Altera SDK for OpenCL: Best Practices Guide*, Altera, 2015. [Online]. Available: http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf

[9] K. Sano, Y. Hatsuda, and S. Yamamoto, "Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 695–705, 2014.

[10] S. O. Settle, "High-Performance Dynamic Programming on FPGAs with OpenCL," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, 2013, pp. 1–6.

[11] S. A. Edwards, "The Challenges of Synthesizing Hardware from C-Like Languages," *IEEE Design Test of Computers*, vol. 23, no. 5, pp. 375–386, May 2006.

[12] D. Chen and D. Singh, "Invited paper: Using OpenCL to Evaluate the Efficiency of CPUS, GPUS and FPGAS for Information Filtering," in *2012 22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 5–12.

[13] G. Ndu, J. Navaridas, and M. Luján, "CHO: Towards a Benchmark Suite for OpenCL FPGA Accelerators," in *Proceedings of the 3rd International Workshop on OpenCL*, ser. IWOCL '15. New York, NY, USA: ACM, 2015, pp. 10:1–10:10. [Online]. Available: http://doi.acm.org/10.1145/2791321.2791331

[14] K. Hill, S. Craciun, A. George, and H. Lam, "Comparative Analysis of OpenCL vs. HDL with Image-Processing Kernels on Stratix-V FPGA," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2015, pp. 189–193.

[15] D. Chen and D. Singh, "Fractal Video Compression in OpenCL: An Evaluation of CPUs, GPUs, and FPGAs as Acceleration Platforms," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2013, pp. 297–304.