Bridging the Performance-Programmability Gap for FPGAs via OpenCL: A Case Study with OpenDwarfs

Konstantinos Krommydas*, Ahmed E. Helal†, Anshuman Verma†, and Wu-Chun Feng*†
Department of Computer Science*,
Department of Electrical and Computer Engineering†,
Virginia Tech,
Email: {kokrommy, ammhelal, anshuman, 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 has come at the significant expense of programmability, i.e., the performance-programmability gap. In particular, FPGA developers use hardware design languages (HDLs) to implement the application data path and to design hardware modules for computation pipelines, memory management, synchronization, and communication. This process requires extensive low-level knowledge of the target FPGA architecture and 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. However, this significantly improved programmability can come at the expense of performance; that is, there still remains a performance-programmability gap. 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 the OpenDwarfs benchmark suite, with its 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 the optimized OpenDwarfs, along with their potential re-factoring, to bridge the performance gap from programming in OpenCL versus programming in a HDL.

***Index Terms*—OpenDwarfs; FPGA; OpenCL; GPU; GPGPU; 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, computational fluid dynamics in oil and gas, and packet processing in network routers and firewalls. However, this performance has come at the significant expense of programmability, i.e., the performance-programmability gap. In particular, FPGA programmers use a hardware design language (HDL) to implement the application data path and to design hardware modules for computation 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 [1], [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 FPGA 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. 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 [3], 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 dwarfs contain optimizations that favor a specific architecture over another.

First, we assess the performance gap between fixed and reconfigurable architectures by characterizing the performance of benchmarks in the OpenDwarfs benchmark suite on multi-core CPUs, GPUs, Intel MIC and FPGAs. Unsurprisingly, we find that architecture-agnostic OpenCL kernels result in inefficient hardware designs on FPGAs. Next, 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 an example case from the 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.

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

TABLE I: OpenDwarfs Benchmarks Used

Dwarf	Benchmark	Input data
N-body Methods	GEM	nucleosome 80 1 0
Structured Grid	SRAD	2048x2048 FP matrix, 128 iterations
Dense Linear Algebra	LUD	2048x2048 FP matrix
Unstructured Grid	CFD	missile.domn.0.2M

II. Motivation and Background

OpenCL is a portable and standard programming model for heterogeneous systems that typically consist of a hierarchical array of processing elements and memory structure. At the 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 for FPGAs, OpenCL is explicitly parallel, which allows OpenCL-FPGA compilers to automatically generate a many-core 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 applications characteristics and improve performance and power efficiency [4], [5]. Although OpenCL's abstract machine model allow 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 a CPU-like, a GPU-like or an application-specific architecture.

III. A Case Study with OpenDwarfs

A. OpenDwarfs 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 presents the OpenDwarfs subset considered in this study and their input datasets and/or parameters, and 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), and the Altera OpenCL SDK v14.0. Figure 2 shows the performance of the OpenDwarfs on both the fixed and reconfigurable architectures. The FPGA resources utilization for GEM, SRAD, LUD and CFD is 29.7%, 45.54%, 38.57% and 85.64%, respectively. Unsurprisingly, the architecture-agnostic implementation of the OpenCL kernels results in inefficient hardware designs on FPGAs.

TABLE II: Test Architectures' Specifications

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

1) GEM: N-body algorithms are characterized by all-to-all computations within a set of particles. In GEM, the electrostatic surface potential of a biomolecule is calculated as the sum of charges contributed by all atoms in the biomolecule 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. 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 large number of PEs, and being sensitive to data locality. Hence, GPUs with their massive number of PEs achieve the best performance.

2) SRAD: In structured grids algorithms, computation proceeds as a series of update steps to a regular grid data structure of two or more dimensions. SRAD is a structured grids application that attempts to eliminate speckles (i.e., locally correlated noise) from images, following a partial differential equation approach. SRAD, similar to all stencil computations, is a regular, memory-bound algorithm, where each grid cell can be updated independently. However, synchronization is required before proceeding to the next grid update step. Therefore, it requires large number of PEs and high memory-bandwidth that renders it suitable for GPU architectures.

3) LUD: Dense linear algebra algorithms are characterized by computations on dense vectors (1D) and matrices (2D data). In LU decomposition, an NxN matrix is reduced to the upper and lower triangular matrices in n steps. In every step k, the

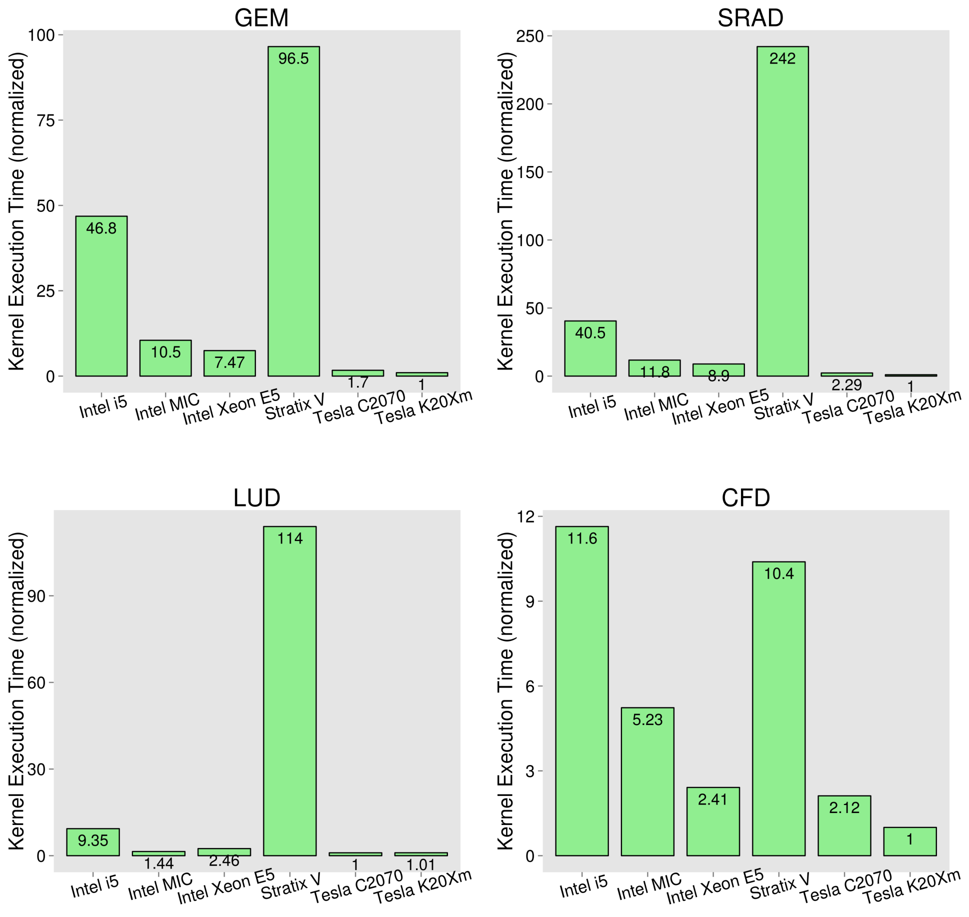

Fig. 1: Architecture-agnostic kernel performance

effect of the kth pivot is applied to the trailing matrix of order N-k x N-k. Therefore, the parallelism decreases in every step, and there is global synchronization between the factorization steps. LUD is a regular synchronization-bound and memory-bound algorithm, if the matrix can't fit in the on-chip memory. Hence, it needs efficient global synchronization and is sensitive to the memory bandwidth. Although GPUs achieve the best performance, due to their inefficient global synchronization are on par with CPU-like architectures.

4) CFD: Unstructured grids algorithms use irregular data structure to keep track of the location and neighborhood of the points on unstructured grid, and perform a series of update steps on this unstructured data. CFD is a solver for the three-dimensional Euler equations for compressible flow that use the finite-volume method. Similar to SRAD, each cell in the grid can be updated independently and synchronization is required before proceeding to the next grid update step. However, CFD suffers from uncoalesced memory accesses due to the use of irregular data-layout that significantly impacts GPU performance. As such, the performance gap between GPUs and other architectures is less than SRAD.

TABLE III: GEM Kernel Implementations' Features

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

B. *FPGA Optimizations and Insights*

To improve the performance of OpenCL kernels on FPGAs, we can exploit different parallelism levels: task, data (SIMD vectorization) and pipeline parallelism. We can minimize memory accesses by controlling data movement across the memory hierarchy levels, and coalescing memory accesses. Since FPGAs have limited hardware resources and memory bandwidth, it is imperative that we analyze different combinations of these optimization techniques to identify the best and generate the most efficient (performance, resource utilization) hardware design for all dwarfs. In the context of this work,

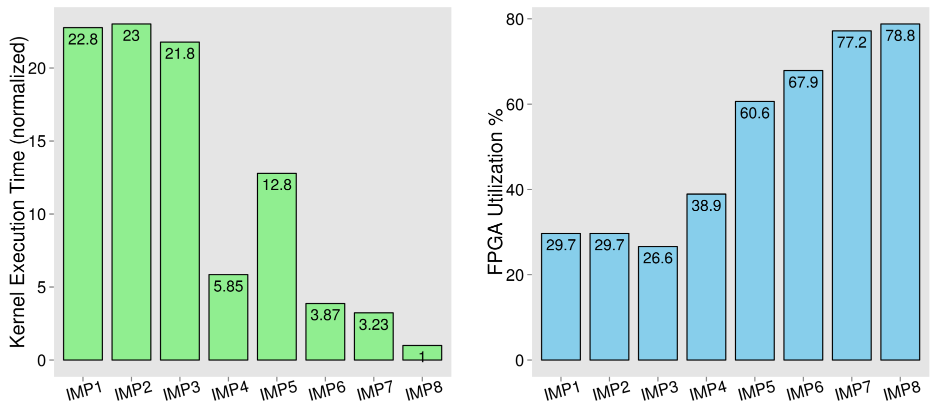

Fig. 2: Optimized GEM Kernel implementations

we start exploring the large FPGA-oriented optimization space and attempt to provide some preliminary insights.

1) Use of restrict/const keywords and kernel vectorization: An optimization strongly suggested by Altera [6] 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 unecessary 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 2) 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.

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 2). 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 [6]. Developers should consider the aforementioned limitations when compiling using the resource-driven optimization option.

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 2), the reduced resource utilization of the refactored algorithm allows SIMD length of 16, whereas the original one accomodated up to 8 (logical elements being the limiting factor). This translates to an 1.22x faster execution of the former compared to the latter.

IV. Conclusion

In this paper we attempted to identify the performance and programmability gap in employing OpenCL to program FPGAs. We start with the unoptimized implementations of certain applications from the OpenDwarfs benchmark suite and subsequently explore part of the large FPGA-targeted optimization space for one of these applications. In the longer term, we seek to identify these optimizations that are suitable for common computation and communication patterns as identified by the dwarfs categorization and subsequently fold them back into an OpenCL compiler for FPGAs.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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*.

[5] 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.

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