Exploring FPGA-specific Optimizations for Irregular OpenCL Applications

Mohamed W. Hassan[1], Ahmed E. Helal[1], Peter M. Athanas[1], Wu-Chun Feng[1,2], and Yasser Y. Hanafy[1]

[1]Electrical & Computer Engineering, Virginia Tech, Blacksburg, VA, USA
[2]Computer Science, Virginia Tech, Blacksburg, VA, USA
{*mwasfy,ammhelal,athanas,wfeng,yhanafy*}@*vt.edu*

***Abstract*—OpenCL is emerging as a high-level hardware description language to address the productivity challenges of developing applications on FPGAs. Unlike traditional hardware description languages (HDLs), OpenCL provides an abstract interface to facilitate high productivity, enabling end users to rapidly describe the required computations, including parallelism and data movement, to create custom hardware accelerators for their applications. However, these OpenCL-realized accelerators are unlikely to make efficient use of the reconfigurable fabric without adopting FPGA-specific optimizations, particularly for irregular OpenCL applications. Consequently, we explore the FPGA-specific optimization space for OpenCL applications and present insights on which optimization techniques improve application performance and resource utilization. Exploring this optimization space will enable end users to harness the computational potential of the FPGA.**

While these optimizations are general and applicable to any application, the expected performance gain and resource-utilization efficiency vary depending on the application characteristics. Specifically, hardware profilers are used to analyze the limitations of OpenCL application kernels and to guide the development of FPGA-optimized implementations. In particular, we pursue the more challenging problem of irregular OpenCL applications, which suffer from workload imbalance, unpredictable control flow, and irregular memory-access patterns. Experiments using representative kernels from the graph traversal, combinational logic, and sparse linear algebra application domains show that FPGA-specific optimizations can improve the performance of irregular OpenCL applications by up to 27-fold in comparison to the architecture-agnostic OpenCL code from the OpenDwarfs benchmark suite.

***Index Terms*—FPGA, High-Level Synthesis, OpenCL, OpenDwarfs, Irregular Applications, Performance Optimization, Hardware Profiling**

I. INTRODUCTION

FPGAs have been used to accelerate a wide spectrum of applications, due to their superior power efficiency over general-purpose architectures such as CPUs and GPUs. However, these performance and power gains come at the cost of complex programming with hardware description languages (HDLs). OpenCL compilers for FPGAs were introduced to address this problem [1], [2]. Unlike HDLs, OpenCL provides an abstract machine model and high-level programming approach for reconfigurable architectures [3], [4], making it easier for end users to develop custom hardware accelerators for their applications and benefit from the power efficiency of FPGAs. Moreover, OpenCL employs a hierarchical memory structure with strong support for parallel execution. Hence, the parallelism can be specified at different granularity levels and data movement can be easily manipulated, enabling OpenCL compilers to potentially generate efficient hardware units and data paths on FPGAs.

The OpenCL programming model targets heterogeneous systems with different types of accelerators, including CPUs, GPUs, Intel MICs, DSPs, and FPGAs. While OpenCL provides *functional* portability across these accelerators, *performance* portability is not guaranteed. In particular, generic (architecture-agnostic) OpenCL kernels are unlikely to make efficient use of the FPGA resources, which leads to performance degradation. This has been shown for multiple application domains such as dense linear algebra, structured grid, unstructured grid, dynamic programming, and N-Body [5]–[12]. In addition, existing OpenCL code that targets CPUs and GPUs is not directly applicable to FPGAs, due to the different hardware capabilities and execution models.

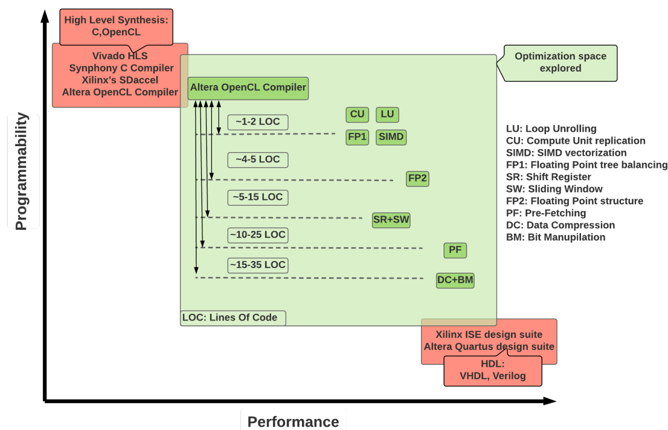

Fig. 1: Programmability vs. performance spectrum for FPGAs.

Figure 1 illustrates the performance versus programmability spectrum and shows the design-space exploration and how FPGA-specific optimizations can be used to enhance the performance with little impact on OpenCL programmability. While the figure is not drawn to scale, it shows the additional programmability overhead (in terms of the average lines of code used to apply the optimization) with respect to the expected performance gain. The optimization techniques depicted in Figure 1 are explained later in Section III.

Irregular applications typically achieve a small fraction of the peak performance on general-purpose architectures due to their workload imbalance, unpredictable control flow, and irregular memory-access patterns. As a consequence, they have the potential to benefit from acceleration using custom hardware architectures. However, when targeting irregular applications, identifying which optimization (or combination of optimizations) to use to enhance the performance and resource utilization on FPGAs is challenging. This study aims to guide non-expert users to the appropriate FPGA-specific optimizations for irregular application domains, including graph traversal, combinational logic, and sparse linear algebra.

Figure 2 demonstrates the benefits of applying such optimizations to generate efficient accelerators on FPGA architectures. For a set of irregular OpenCL kernels, our optimized FPGA realizations achieve

This work was supported in part by NSF I/UCRC IIP-1266245 via CHREC.

978-1-7281-1968-7/18/$31.00 ©2018 IEEE

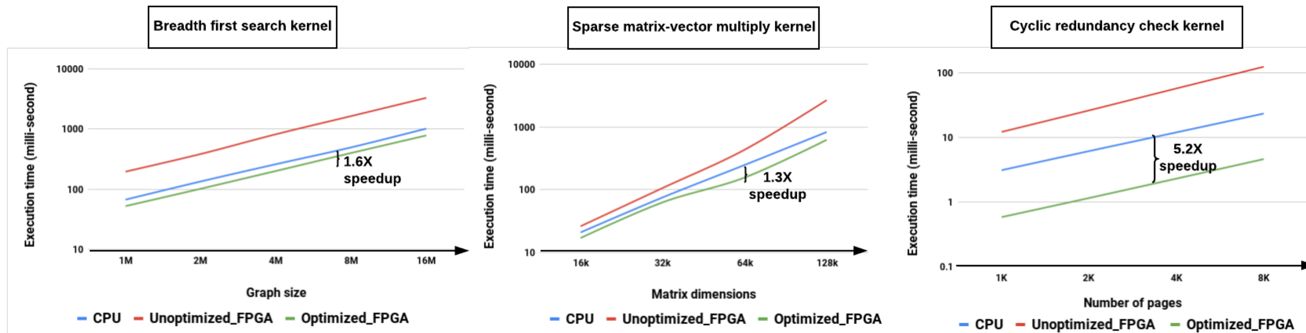

Fig. 2: The performance of irregular OpenCL kernels on CPU and FPGA architectures. The optimized FPGA execution uses a deeply-pipelined, compute unit running at 200-260 MHz, while the CPU platform consists of 16 compute units running at 3.5 GHz.

1.3-5.2× speedup over the corresponding parallel execution on a 16-core CPU, while running at an order-of-magnitude slower frequency.

The following are the contributions of this work:

- *Identification of and insight on the FPGA-specific optimizations for OpenCL kernels.* The identified optimizations apply to any user application; however, the expected performance gain and resource utilization depend on the characteristics of the application kernels. Furthermore, end users need insight on which optimizations to use for their target applications.
- *Profiling and analyzing of the OpenCL kernels to identify the execution bottlenecks and, in turn, guide the FPGA optimization of irregular codes.* We use Intel/Altera's hardware profiler to facilitate the analysis and optimization of irregular application kernels from the OpenDwarfs benchmark suite.
- *A detailed study of the FPGA-specific optimizations for representative irregular applications, namely graph traversal, combinational logic, and sparse linear algebra applications*. The results show that the FPGA-specific optimizations improve performance by an order of magnitude when compared to the architecture-agnostic OpenCL code from OpenDwarfs.

II. Related Work

Czajkowski et al. [3] demonstrated the Altera's OpenCL compiler using four well-known applications: Monte Carlo Black-Scholes (MCBS), matrix multiplication (SGEMM), finite difference (FD), and particle simulation (Particles). To enhance the performance of dynamic programming on FPGAs, Settle introduced OpenCL pipes [4], which improves the performance by 1.5× and 9.9× in comparison to the GPU and CPU implementations, respectively, while providing energy savings of up to 26-fold. Both endeavors achieved high utilization of FPGA resources with low clock frequency (less than 200 MHz). Moreover, the FPGA-specific implementations differed from their GPU-based counterparts. While the GPU implementation used SIMD-like parallelism, the FPGA implementation adopted a MIMD-like execution where each thread executed a distinct operation on a set of data items.

In [5], Zohouri et al. evaluated the performance of six regular benchmarks from the Rodinia suite using the Altera OpenCL SDK on a Stratix V FPGA. The original OpenCL implementations followed the bulk synchronous parallel (BSP) execution model, targeting GPU-like architectures with massive multi-threaded execution. Unfortunately, this approach can degrade FPGA performance due to barrier synchronization points that dictate flushing the pipeline, effectively halving the pipeline throughput. The authors in [5] reached the conclusion that FPGA-specific optimizations must be applied to the OpenCL kernels to yield efficient, high-performance hardware designs. In particular, they outlined five main FPGA-specific optimization techniques: compute unit replication, vectorization (or ”SIMD-iztion”), loop unrolling, shift registers, and sliding windows. These optimizations improved the performance by up to two orders of magnitude compared to the BSP OpenCL kernels and achieved 3.4× better power efficiency when compared to the NVIDIA K20c GPU. While [5] focused on regular benchmark applications, our work addresses irregular applications, which are more challenging.

The work published in [5] was extended in [12] to evaluate the performance of three different design methodologies for FPGAs: general-purpose manycore system (30 Nios II soft-core processors), FSM-based architecture using LegUp HLS tool (MIMD architecture with focus on lower latency), and Intel's FPGA SDK for OpenCL (deeply-pipelined architectures with focus on higher throughput). The experiments showed that the FSM and soft-core implementations have scalability issues that are mainly related to cache conflicts and capacity misses. This issue was partially solved using a multi-banked cache design. However, the OpenCL implementations still outperformed both approaches across all the applications with up to two orders-of-magnitude speedup.

Other work [6], [13] used the OpenDwarfs benchmark suite to evaluate the performance of the OpenCL programming model on a Stratix V FPGA using the Altera OpenCL SDK. Kernels from regular application domains were tested, such as N-body methods, structured grids, unstructured grids, and dense linear algebra. Unlike Rodinia, the OpenDwarfs suite provides architecture-agnostic OpenCL kernels rather than GPU-specific (i.e., GPU-biased) implementations. These kernels were used as the baseline for comparison on the CPU, GPU, Intel MIC, and FPGA architectures. The authors explored FPGA-specific optimization techniques that exploit different parallelism levels as well as minimizing data movement across the memory hierarchy. It was also reported that the architecture-agnostic OpenCL kernels yielded inefficient hardware designs, which further suggests the need for FPGA-specific optimizations.

Static and dynamic analyses were used in [8] to build an analytical performance model for the key architectural features of FPGAs under the OpenCL programming model. This tool can predict the performance of OpenCL kernels with different combinations of FPGA-specific optimizations. This greatly helps in guiding the code-tuning process for performance purposes. On the other hand, the framework in [7] aims to achieve scalable execution of memory-bound applications, such as AES encryption, on multiple FPGAs. In particular, six Stratix V FPGAs were used to demonstrate the scalability on a high-performance backplane. The authors reported that

SIMD vectorization provided better FPGA resource utilization and significantly less on-chip memory usage than the kernel-replication approach. The authors reported three-fold improvement in throughput per watt over the CPU implementation using a single FPGA, while four FPGAs yielded a five-fold improvement.

Finally, XSBench, a proxy application for Monte Carlo simulation, was used in [11] to evaluate the performance of OpenCL applications with irregular memory access on FPGAs on an Intel Arria 10 FPGA platform. The authors applied three different optimizations and evaluated their effect on performance. A fused multiply-add unit was integrated into the design. The BRAMs were used to implement a constant cache along with data pre-fetching and packing techniques. The final optimization used vector data types and stored them in private memory. Applying these optimizations delivered a 50% improvement in energy efficiency, while sacrificing 35% of the performance compared to an Intel Xeon CPU with eight cores.

In summary, previous studies showed the need for applying FPGA-specific optimizations to OpenCL kernels to generate efficient custom hardware accelerators. However, the expected performance and efficiency is highly dependent on the characteristics of the target application. In addition, OpenCL kernels with BSP execution models, which achieve high performance on GPUs, generate extremely inefficient FPGA designs. While previous work focused on regular OpenCL kernels, this paper attacks the problem of optimizing the *irregular* OpenCL applications on FPGAs.

III. FPGA Design-Space Exploration

We categorize the FPGA-specific optimization space as follows: (1) exploiting parallelism at different levels, (2) optimizing floating-point operations, and (3) minimizing data movement across the memory hierarchy. By default, FPGA OpenCL compilers exploit pipeline parallelism, which, in turn, generally achieves higher throughput than data parallelism or task parallelism, due to the limited resources on FPGAs, which restrict the number of concurrently active work items.

A. Parallelism Optimizations

There are two main OpenCL execution models on FPGAs: multi-threaded execution and single-task execution. Multi-threaded execution attempts to expose the maximum parallelism by executing multiple threads concurrently, if possible. On the other hand, single-task execution exploits pipeline parallelism and runs the work items (i.e., units of computation) sequentially as a single task.

Loop Unrolling. Unrolling loops improves performance by decreasing the number of loop iterations executed and, in turn, the number of branches. However, there is a trade-off between the loop unrolling factor and the extra hardware cost incurred.

Kernel Vectorization. Vectorization enables multiple work items to execute in a single-instruction, multiple-data (SIMD) fashion. This technique achieves *higher computational throughput* and automatically performs *memory coalescing*. The SIMD approach vectorizes the data path of the kernel while keeping a single control logic path shared across the SIMD lanes. Therefore, backward branches with thread ID dependencies prohibit this optimization technique, as they can serialize the execution process.

Compute Unit Replication. Generating multiple compute units, where data and control paths are replicated, fully parallelizes the kernel execution. This optimization divides the workload on the available compute units which can mitigate the limitations of the SIMD approach, namely the thread ID dependency problem. However, compute unit replication uses more hardware resources than the SIMD approach. It also increases the stress on the global memory bandwidth, as more load/store units would be competing for accessing the global memory.

B. Floating-Point Optimizations

The floating-point operations in a specific kernel may not be balanced, leading to pipeline stalls and higher hardware cost [14]. The Altera OpenCL Compiler provides command-line options to optimize the floating-point operations using balanced trees. Moreover, removing the floating-point rounding operations and conversions, whenever possible, introduces hardware savings.

Floating-Point Accumulator. The newer FPGA platforms, such as Altera's Arria 10, include a floating-point accumulator that performs the accumulations in a single cycle; however, only single work-item kernels that perform accumulation in a loop without branching can leverage this feature. Modifications are required in the kernel code for the compiler to infer the use of the accumulator structure.

C. Data Movement Optimizations

Shift Registers (SR) and Sliding Windows (SW). Several computational kernels, such as sparse matrix-vector multiplication (SPMV), have loop-carried data dependencies. On FPGA architectures, cross-iteration dependencies may increase the initiation interval of the loop, where the next iteration is stalled until the dependency is resolved. To relax this cross-iteration dependency, the loop body is modified to employ shift registers with a sliding-window technique, which resolves this problem by eliminating the pipeline stalls.

Data Compression (DC) and Bit Manipulation (BM). The OpenCL standard instantiates Boolean variables as 32-bit integers. Programming bitwise operations and masks allows single-bit on-chip memory (BRAM/register) access by the OpenCL code.

IV. A Case Study for Irregular Applications

In our experiments, we use the Intel FPGA Dynamic Profiler for OpenCL to analyze the execution profile of the architecture-agnostic (generic) OpenCL kernels from the OpenDwarfs suite [15]. Analyzing the execution profile pinpoints the bottlenecks of the execution pipeline. This study applies the above optimizations, both in isolation and in combination, to the target OpenCL kernels and evaluates the resulting performance, which typically outperforms the architecture-agnostic and GPU-optimized OpenCL implementations on FPGA architectures. The FPGA resource utilization is also considered in evaluating the hardware cost of each optimization technique. Finally, the performance analysis of the different optimizations provides key insights into which optimizations to use for each target application and how to apply such optimizations to address the execution bottlenecks and to achieve the required performance gain. The optimized kernels are available at *https://github.com/vtsynergy/OpenDwarfs*.

Test Platform. The experiments use an Altera Arria 10 1150-GX FPGA connected to two 4-GB DDR3 memory with peak bandwidth of 25 GB/s. The FPGA attaches to the host machine via PCIe 8x 3.0 interface. The host includes an Intel Xeon E5-2637 CPU and runs Ubuntu 14.04 along with Altera OpenCL SDK version 16.0.

A. Graph Traversal

Breadth-first search (BFS) is used by the OpenDwarfs suite as a representative kernel for the graph traversal dwarf. The target graphs are undirected and unweighted in the form $G = (V, E)$, where V is the set of vertices or nodes and E is the set of edges connecting them. To avoid processing a node more than once, a Boolean visited array is used. As such, the graph is traversed in levels, where all nodes at each level are explored before the next level is processed. The final output is the cost C, which represents the shortest distance

TABLE I: BFS optimizations and resource utilization.
[MT]: Multi-Threaded, [ST]: Single Task

Optimization	Description	Frequency	Logic utilization	BRAM
Generic	Architecture agnostic OpenDwarfs kernel	248 MHz	29%	20%
MT-LU8	Loop unrolling factor 8	205 MHz	37%	37%
MT-PE2	Compute unit replication (2 PEs)	204 MHz	32%	24%
MT-PE4	Compute unit replication (4 PEs)	202 MHz	35%	42%
ST-Regular	Simple conversion to single work item by inserting outer `for-loop`	201 MHz	28%	20%
ST-NoSync	Eliminate host side synchronization by inserting outer `while-loop`	212 MHz	29%	20%
ST-Mem	Use bit manipulation on integer arrays to implement Boolean arrays	160 MHz	27%	74%

from the source node to each visited node on the graph. The time complexity is $O(V + E)$.

The original OpenCL kernel (from OpenDwarfs) is multi-threaded, executing each graph level update in a separate kernel launch. After computing a level of the graph, synchronization with the host is required, paired with a new kernel launch for the new graph level. Our hardware profiling pinpointed two major bottlenecks. First, the cross-iteration dependencies stalled the execution pipeline for more than 800 clock cycles in *each* iteration. This high initiation interval of the loop is caused by (a) the serial execution of the `for-loop`,where loop pipelining optimization isn't applied by default due to unrelieved cross iteration dependency and (b) the host side synchronization step between kernel invocations. Second, the global memory access pattern for five different arrays is inefficient, lowering the bandwidth efficiency of data transfer to an average of 13% of the peak memory bandwidth. Table I shows the different optimization techniques employed to address these bottlenecks, along with their operating frequency and logic utilization.

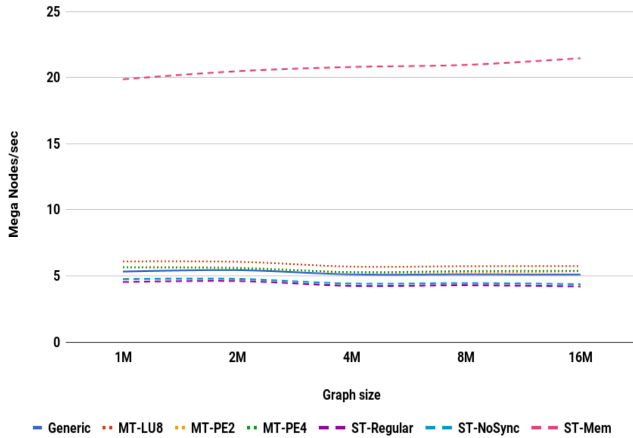

Fig. 3: The performance of BFS (nodes processed per second) across different graph sizes for multiple optimization techniques.

1) Multi-Threaded Execution: The global memory access bottleneck and kernel launch overhead are the main reasons that none of the multi-threaded optimization techniques yielded any significant performance gains. Compute unit replication does not address the memory access bottleneck or solve the pipeline stall problem. Hence, the performance improvement was 6% at most, as shown in Figure 3. Loop unrolling enhanced the performance by a maximum of 14%, due to memory coalescing which increases the bandwidth efficiency. The combination of loop unrolling with compute unit replication is unnecessary as neither have the potential to address the pinpointed bottlenecks. The multi-threaded code does not leverage data-level parallelism (vectorization), due to the loop-carried data dependencies and the thread ID dependent branching in its inner and outer loop. So, SIMD optimization was not applied to this application kernel.

2) Single Task Execution: In the single work-item execution model, multiple optimizations were tested to enhance performance. The "ST-Regular" implementation fully pipelines the `for-loop` without modifying the global synchronization scheme. On the other hand, "ST-NoSync" avoids synchronizing with the host, which was on average 5% of the execution time, and moves all computations to the device. The Altera OpenCL compiler was not able to pipeline the outer `while-loop` in this implementation, as cross iteration dependency is critical for functional correctness. Figure 3 shows a slight decrease in performance for "ST-Regular" and "ST-NoSync", as these optimizations do not address the memory access bottleneck, which has the most impact on performance.

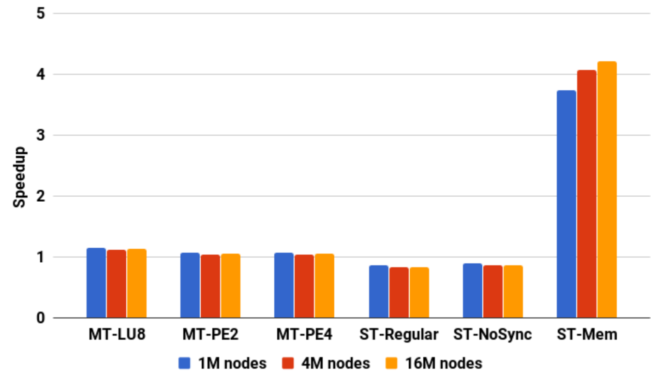

Fig. 4: BFS speedup across the different optimization techniques. The baseline is the OpenDwarfs, architecture-agnostic OpenCL code.

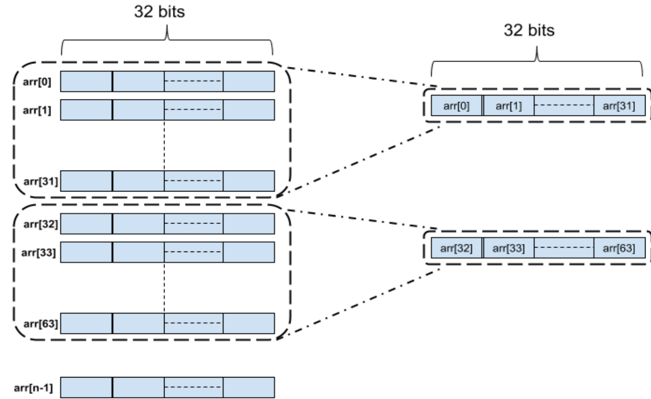

Fig. 5: Boolean array data compression.

The following step was to optimize the memory access operations by moving the Boolean arrays to the local on-chip memory (BRAMs) in "ST-Mem". The OpenCL standard supports Boolean variables; however, they are treated as 32-bit integers with a constant value of "0" or "1", which wastes the on-chip memory. Therefore, integer arrays are used, as shown in Figure 5, where each integer represents 32 Boolean flags that can be accessed through a series of bitwise manipulations (shift, AND, OR and XOR operations). This imple-

TABLE II: SPMV optimizations and resource utilization.
[MT]: Multi-Threaded, [ST]: Single Task

Optimization	Description	Frequency	Logic utilization	BRAM
Generic	Architecture agnostic OpenDwarfs kernel	255 MHz	26%	21%
MT-LU16	Loop unrolling factor 16	225 MHz	33%	31%
MT-LU32	Loop unrolling factor 32	211 MHz	39%	46%
MT-PE2	Compute unit replication (2 PEs)	245 MHz	28%	25%
MT-PE4	Compute unit replication (4 PEs)	230 MHz	31%	36%
MT-PE2-LU16	Compute unit replication (2 PEs) + Loop unrolling factor 16	203 MHz	41%	47%
MT-PE2-LU32	Compute unit replication (2 PEs) + Loop unrolling factor 32	174 MHz	54%	77%
MT-PE4-LU16	Compute unit replication (4 PEs) + Loop unrolling factor 16	164 MHz	57%	80%
ST-PF-SR-LU8	Pre-fetching+SR and sliding window+Loop unrolling factor 8	205 MHz	41%	50%
ST-PF-SR-LU12	Pre-fetching+SR and sliding window+Loop unrolling factor 12	192 MHz	51%	66%
ST-PF-SR-LU8-LU4	Pre-fetching+SR and sliding window+LU8(outer loop)+LU4(inner loop)	166 MHz	57%	69%

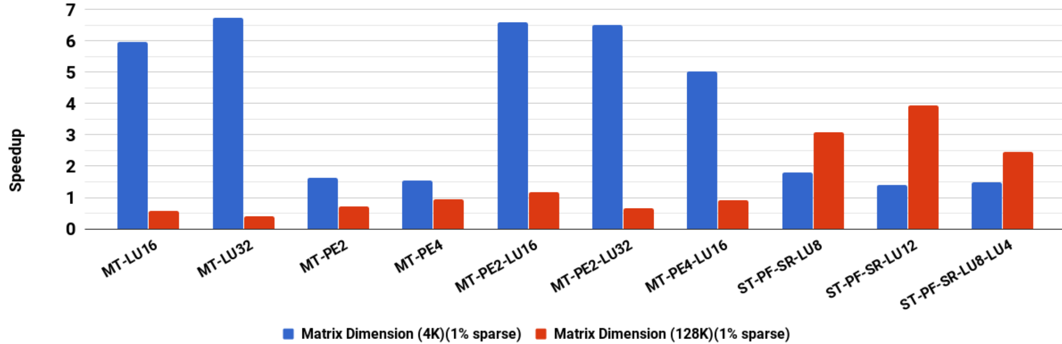

Fig. 6: SPMV speedup for small and large matrix sizes.

mentation enables fast Boolean checking which yielded 4× speedup as shown in Figure 4. However, due to the limited on-chip memory, this approach can only support graphs of sizes up to 32M nodes.

For smaller graphs with size up to 512K nodes, the algorithmic refactoring showed great performance. The kernel was modified to use a local-memory queue instead of the Boolean mask. The new unvisited nodes are inserted into the FIFO queue, and one node is popped in each iteration, which greatly reduces the total number of iterations. However, this evaluation targets large-scale graphs with at least 1M vertices; hence, this approach was excluded from the results.

B. Sparse Linear Algebra

The OpenDwarfs suite includes SPMV (sparse matrix-vector multiplication) as a representative kernel of sparse linear algebra. While computation across the rows of the input sparse matrix (outer loop) are independent, the operations required to compute a single output element (inner loop) have data dependencies. A series of memory accesses are required by each iteration of the outer loop to retrieve the indices of non-zero elements of each sparse row, and read the respective values of these elements and the corresponding elements of the input vector. On hardware architectures with limited memory bandwidth, such memory operations introduce a global memory bottleneck. The hardware profiler showed that bandwidth efficiency is limited to 55% at the bottlenecked inner loop. Moreover, the number of iterations in the inner loop is input dependent (i.e., depends on the sparsity pattern of the input matrix).

1) Multi-Threaded Execution: The multi-threaded version of SPMV exploits different parallelism levels: task-level (compute unit replication), and instruction-level (loop unrolling). However, the multi-threaded code does not leverage data-level parallelism (vectorization), due to the loop-carried data dependencies and the thread ID dependent branching in the inner loop.

Figures 6 and 7 show the effect of the different optimizations on the SPMV performance in comparison with the baseline OpenCL code,

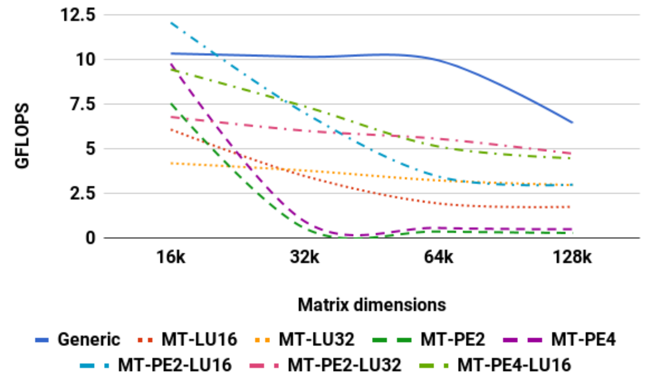

Fig. 7: The performance of SPMV for the multi-threaded optimizations. The baseline is the OpenDwarfs, architecture-agnostic code.

while Table II shows the resource utilization of each implementation. For the small input data, these optimizations showed a maximum speedup of 6.8× (matrix size 4K in Figure 6). "MT-LU32" provides the best performance only with 13% more logic utilization than the baseline kernel, but with double the on-chip memory usage. Compute unit replication has limited performance improvement, it achieved at most 1.7× speedup by "MT-PE2" and "MT-PE4". This is mainly due to the contention on the limited global memory bandwidth. Combining both optimizations of multiple compute units and loop unrolling shows comparable performance to simply just unrolling the loop, but with much higher hardware cost. "MT-PE4" increases the logic utilization by 30% and the on-chip memory usage by 60%.

Figures 6 and 7 show the limited scalability of the multi-threaded execution model for SPMV. Scaling up the input matrix puts the multi-threaded SPMV at a great disadvantage, due to the limited global memory bandwidth. In fact, these optimizations do not address the issue of the bandwidth efficiency of the inner loop. So, as the

matrix size increases these optimization add more stress to the global memory access and the performance decreases and becomes slower than the baseline code.

2) Single Task Execution: The single task (work item) execution of SPMV throttles the parallelism (concurrent work items) to reduce the contention on the limited FPGA resources, specifically the global memory bandwidth. However, due to the extra unused FPGA resources, the single task code can leverage advanced techniques to minimize the data movement across the memory hierarchy, such as caching (pre-fetching), shift registers, and enhanced floating-point units. The caching optimization pre-fetches the data into the private memory (BRAM) and maximizes the data reuse. The shift register (SR) optimization uses a sliding window technique to alleviate the loop-carried dependencies in the SPMV inner loop, which enables the compiler to efficiently pipeline the inner loop with successive iterations initiated into the pipeline every clock cycle. Finally, the code was modified to allow the compiler to infer floating-point accumulator (see section III-B) to further enhance the performance of the inner loop.

These techniques relieved the inner loop contention on global memory access increasing the bandwidth efficiency to 100% (according to the hardware profiler). However, the hardware profiler showed that this execution model limits the efficiency of the store unit that writes the final result at the end of each outer loop iteration to 20%. Nevertheless, the effect of this limitation has much less impact on performance, since the number of store operations is significantly less than the number of load operations. The number of load operations is $O(NZE)$, where NZE is the (number of Non Zero Elements). The number of store operations is $O(R)$, where R is the (number of matrix Rows).

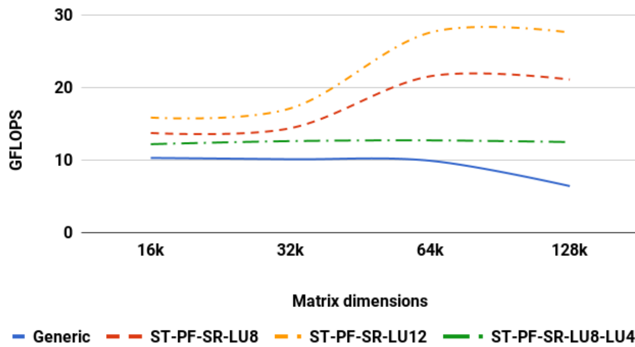

Fig. 8: The performance of the single-task optimizations for SPMV. The baseline is the OpenDwarfs, architecture-agnostic OpenCL code.

Figures 6 and 8 show the performance of the single task execution compared to the baseline and the multi-threaded versions. Unrolling the outer loop and minimizing the data movement in "ST-PF-SR-LU12" showed a speedup of 4× over the baseline kernel. Moreover, single task execution has scalable performance and sustainable speedup, as depicted in Figure 8, with input matrix size up to 128K. Unrolling the inner loop in "ST-PF-SR-LU8-LU4" didn't provide performance advantages, due to inefficient pipeline structure in addition to the input dependent number of inner loop iterations (loop bounds are not constants). The Altera OpenCL Compiler might fail to meet scheduling because it cannot unroll this nested loop structure easily, resulting in a high II (number of stall clock cycles before issuing the next loop iteration) [2], [14]. In summary, the results showed the importance of alleviating the contention on the limited FPGA global memory bandwidth and inferring an efficient pipeline structure to attain scalable performance.

C. Combinational Logic

The OpenDwarfs adopts cyclic redundancy check (CRC) as a representative kernel of combinational logic applications, which rely on bitwise logic operations. This application domain is amenable to acceleration using FPGA architectures with fine-grain logic fabric. The CRC kernel computes the 32-bit CRC code of a set of input data pages (packets) using the "Slice-By-8" algorithm developed by Intel [16], [17]. The CRC32 generation process consists of a single table lookup, bitwise and shift operations for each byte. The hardware profiler showed that there is 67 clock cycles of stalls in the pipeline execution for each loop iteration, due to inefficient loop structure.

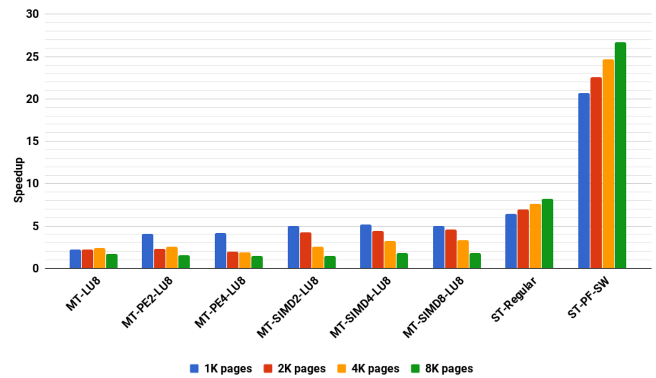

Fig. 9: CRC speedup across different optimization techniques. The baseline is the OpenDwarfs, architecture-agnostic OpenCL code.

1) Multi-Threaded Execution: Figure 9 shows the performance of the different multi-threaded versions in comparison with the baseline, architecture-agnostic code. The results show that fully-unrolled loops in "MT-LU8" yield around 2× speedup compared to the baseline with minimal hardware cost (6% logic utilization increase).

Exploiting task-level parallelism in "MT-PE2-LU8 and MT-PE4-LU8" achieves 4-5× speedup with additional it hardware cost of 14%-30% for two and four processing elements (PEs). However, Figure 9 shows that the compute unit replication approach does not provide scalable speed up, due to the contention on the global memory access. In particular, as the number of input data pages increases, beyond 1K, the performance degrades and converges to a consistent 2× speedup.

SIMD vectorization shows a speedup of 6× over the baseline in "MT-SIMD2-LU8"; however, increasing the number of vector lanes does not improve the performance over using two SIMD lanes, while incurring up to 52% additional area overhead. The increased hardware cost for using more vector lanes suggests that using two lanes would be the best option. SIMD vectorization works well with this specific kernel, as there is no thread ID dependent, backward branching.

Although the multi-threaded execution model shows some performance gain, it can be noticed in Figure 10 that, as the input size grows larger, the performance advantages degrade. The above multi-threaded execution versions suffer from a major bottleneck: the limited global memory bandwidth, where multiple threads in flight are competing for global memory access. Therefore, as the size of input data increases (more than 4K data pages), the performance takes a severe hit and the speedup decreases to 1.5-2× over the baseline.

2) Single Task Execution: Moving to the single task execution model alleviated the problem of having multiple threads competing for the limited global memory bandwidth. Figure 9 shows that the single task CRC versions achieved scalable speedup with the growing

TABLE III: The CRC optimizations and their resource utilization.
$[MT]$: Multi-Threaded, $[ST]$: Single Task

Optimization	Description	Frequency	Logic utilization	BRAM
Generic	Architecture agnostic OpenDwarfs kernel	270 MHz	26%	18%
MT-LU8	Loop unrolling factor 8	236 MHz	32%	23%
MT-PE2-LU8	Compute unit replication (2 PEs) + LU8	230 MHz	40%	31%
MT-PE4-LU8	Compute unit replication (4 PEs) + LU8	230 MHz	56%	47%
MT-SIMD2-LU8	SIMD (2 vector lanes) + LU8	227 MHz	39%	29%
MT-SIMD4-LU8	SIMD (4 vector lanes) + LU8	224 MHz	52%	44%
MT-SIMD8-LU8	SIMD (8 vector lanes) + LU8	203 MHz	78%	89%
ST-Regular	Simple conversion to single work item by inserting outer `for-loop`	148 MHz	41%	30%
ST-PF-SW	Pipelining `for-loop` by pre-fetching+relaxing cross iteration dependency	231 MHz	26%	18%

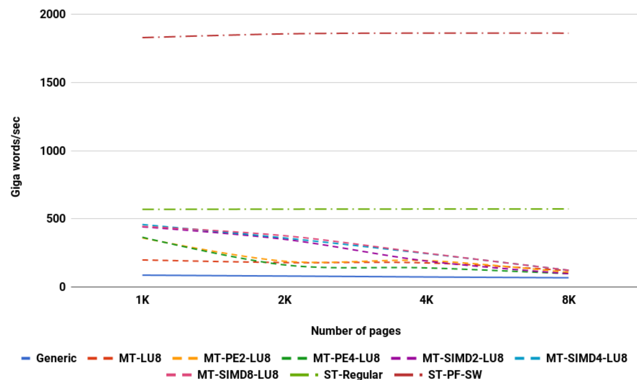

Fig. 10: The performance (words processed per second) of CRC across different optimization techniques.

input size. Moreover, Figure 10 shows that ”ST-Regular” and ”ST-PF-SW” process a constant number of words per second, that does not decrease. The execution profile pulled from the hardware profiler shows that the efficient pipeline execution with no stalls allows high bandwidth efficiency for each loop iteration. Memory accesses are pipelined and latency is efficiently hidden by the computation in each loop iteration, sustaining scalable performance.

Hence, unlike the multi-threaded execution model, the single task execution is scalable, i.e., its speedup improves as the input data size increases. After inspecting the FPGA execution profile, further advanced optimizations were not needed, as the computational loops are efficiently pipelined with a loop iteration issued every two clock cycles (according to the compiler optimization report).

D. Discussion

The results across three representative irregular OpenCL applications showed that the limited global memory bandwidth of the FPGA architecture hinders the scalability of the multi-threaded OpenCL execution model. When the input data set is small enough to not increase the contention on the global memory access, the multi-threaded execution model can provide significant gains (e.g., up to 6.8× performance gains in SPMV kernel). However, with larger input data, the performance of the multi-threaded kernels is severely affected which leads to saturated and limited speedup. On the other hand, the single task execution model resolves the global memory bottleneck and enables streamed memory access to/from the main memory without competition between multiple threads.

Figure 2 shows the FPGA performance relative to the execution on a multi-core CPU platform. The CPU platform includes an Intel Xeon E5-2637 with 16 compute units running at 3.5 GHz, and it has a memory bandwidth of 80 GB/s and a cache size of 15 MB. The performance of the architecture-agnostic OpenCL kernels on FPGA (Unoptimized_FPGA) is an order of magnitude slower than the CPU execution. Even though the significantly higher frequency and larger number of compute units of the CPU platform put the FPGA platform at a great disadvantage, applying the FPGA-specific optimizations yields sustainable speedups over the CPU execution. While the optimized FPGA implementation runs at a frequency range of 200-260 MHz and uses a single deeply-pipelined compute unit, it achieves 5.2×, 1.3× and 1.6× speedup for the CRC (ST-PF-SW), SPMV (ST-PF-SR-LU12), and BFS (ST-Mem) kernels, respectively, compared to the CPU execution.

The experiments show that applying FPGA-specific optimizations to the architecture-agnostic OpenCL code can significantly enhance the performance. Exploring the aforementioned optimization space, code patterns were identified according to the hardware profiler and the optimization reports. The effects of the optimizations on these code patterns are analyzed aiming to help in two aspects. First, providing guidelines and best practices for the development of new OpenCL kernels with similar code patterns towards the best performance. Second, guiding the future work of automating the optimizations process of architecture-agnostic OpenCL kernels. Below is a list of the identified code patterns along with their relative FPGA-specific optimizations.

- *OpenCL kernels with Boolean data structures that reside in the global memory* can be optimized using data compression techniques and bit-mask arrays to reduce the memory usage and to be able to place such arrays in on-chip BRAMs, allowing fast Boolean array look up.
- *Kernels that use conditional statements depending on a global memory read transaction* should be handled using pre-fetching of the conditional variable to the on-chip local memory to enable fast conditional checking.
- *Using DEF-USE chain analysis, loop carried dependencies can be detected*, and then the performance can be improved by relaxation using shift registers and sliding window operation, or by elimination using temporary on-chip storage before offloading the results to the corresponding output.
- *Floating-point accumulation can be easily detected in the code* and modified for optimization by balancing the floating-point operations tree and/or inferring floating point accumulation structures (see section III-B).
- *Loop unrolling factor is critical for performance*. The unrolling factor should be closely coupled to the expected number of iterations of the loop. Unrolling a loop with a higher than necessary value would waste space (area) and time (frequency), which might lead to performance degradation.

V. Conclusion

In this paper, the FPGA-specific optimization space is explored for the OpenCL programming model, with a specific focus on the irregular applications that suffer from workload imbalance, fine-grain bitwise operations, dynamic control flow, and scattered memory access pattern. Applying such optimizations enables the OpenCL kernels to deliver both functional *and* performance portability on the FPGA architectures by synthesizing more efficient hardware designs. In addition, hardware profiling was used to pinpoint the execution bottlenecks and to guide the optimization process. A detailed analysis of the FPGA-specific optimizations on the target application domains is provided to guide the end users to extract high performance from the energy-efficient reconfigurable architectures.

The experiments showed the potential of the single task execution model to resolve the contention on the shared FPGA resources and demonstrated *scalable* speedup on the three tested application domains. Specifically, the Breadth first search (BFS), cyclic redundancy check (CRC), and sparse matrix-vector multiplication (SPMV) applications achieved up to 4.2×, 27×, and 6.8× speedup, respectively, over the architecture-agnostic kernels from the OpenDwarfs benchmarks suite.

While significant performance improvements were obtained using the FPGA-specific optimizations of the original algorithms, previous studies [18]–[20] showed that algorithmic refactoring can result in multiplicative performance gain. As such, there are many opportunities to expand the current work by analyzing and modeling the inherent characteristics of the different algorithms [21], [22] to guide the algorithmic innovation and refactoring of the irregular applications to better match the capabilities of the FPGA platforms.

References

[1] Xilinx. (2015). The xilinx sdaccel development environment, [Online]. Available: https://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf.

[2] Altera. (2015). Altera sdk for opencl programming guide, [Online]. Available: https://www.altera.com/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

[3] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on fpgas," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 531–534.

[4] S. O. Settle, "High-performance dynamic programming on fpgas with opencl," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2013, pp. 1–6.

[5] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, 35:1–35:12.

[6] K. Krommydas, A. E. Helal, A. Verma, and W. Feng, "Bridging the performance-programmability gap for fpgas via opencl: A case study with opendwarfs," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 198–198.

[7] S. Gao and J. Chritz, "Characterization of opencl on a scalable fpga architecture," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, IEEE, 2014.

[8] Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing opencl applications on fpgas," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 114–125.

[9] K. Krommydas, W. Feng, M. Owaida, C. D. Antonopoulos, and N. Bellas, "On the characterization of opencl dwarfs on fixed and reconfigurable platforms," in *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 2014, pp. 153–160.

[10] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *IEEE International Symposium on Workload Characterization (IISWC'10)*, ser. IISWC '10, 2010, pp. 1–11.

[11] Y. Luo, X. Wen, K. Yoshii, S. Ogrenci-Memik, G. Memik, H. Finkel, and F. Cappello, "Evaluating irregular memory access on opencl fpga platforms: A case study with xsbench," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[12] A. Podobas, H. R. Zohouri, N. Maruyama, and S. Matsuoka, "Evaluating high-level design strategies on fpgas for high-performance computing," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[13] A. Verma, A. E. Helal, K. Krommydas, and W.-C. Feng, "Accelerating workloads on fpgas via opencl: A case study with opendwarfs," Virginia Polytechnic Institute and State University, Tech. Rep., 2016.

[14] Altera. (2015). Altera sdk for opencl: Best practices guide, [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf.

[15] K. Krommydas, W. Feng, C. D. Antonopoulos, and N. Bellas, "Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures," *J. Signal Process. Syst.*, vol. 85, no. 3, pp. 373–392, Dec. 2016, ISSN: 1939-8018.

[16] M. E. Kounavis and F. L. Berry, "A systematic approach to building high performance software-based crc generators," in *IEEE Symposium on Computers and Communications (ISCC'05)*, 2005, pp. 855–862.

[17] Intel. (2014). Slicing-by-8, crc, [Online]. Available: https://sourceforge.net/projects/slicing-by-8/.

[18] A. E. Helal, A. M. Bayoumi, and Y. Y. Hanafy, "Parallel circuit simulation using the direct method on a heterogeneous cloud," in *Proceedings of the 52nd Annual Design Automation Conference*, ACM, 2015, p. 186.

[19] K. Hou, W. Liu, H. Wang, and W.-c. Feng, "Fast segmented sort on gpus," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17, Chicago, Illinois: ACM, 2017, 12:1–12:10, ISBN: 978-1-4503-5020-4.

[20] K. Hou, H. Wang, W. Feng, J. S. Vetter, and S. Lee, "Highly efficient compensation-based parallelism for wavefront loops on gpus," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 276–285.

[21] A. E. Helal, W.-c. Feng, C. Jung, and Y. Y. Hanafy, "Automatch: An automated framework for relative performance estimation and workload distribution on heterogeneous hpc systems," in *Workload Characterization (IISWC), 2017 IEEE International Symposium on*, IEEE, 2017, pp. 32–42.

[22] A. E. Helal, C. Jung, W.-c. Feng, and Y. Y. Hanafy, "Commanalyzer: Automated estimation of communication cost and scalability on hpc clusters from sequential code," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2018, pp. 80–91.