2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-1016-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/HIPC53243.2021.00062

Exploring Thread Coarsening on FPGA

Mostafa Eghbali Zarch, Reece Neff, Michela Becchi
North Carolina State University
Raleigh, NC, USA
{meghbal,rwneff,mbecchi}@ncsu.edu

***Abstract*—Over the past few years, there has been an increased interest in including FPGAs in data centers and high-performance computing clusters along with GPUs and other accelerators. As a result, it has become increasingly important to have a unified, high-level programming interface for CPUs, GPUs and FPGAs. This has led to the development of compiler toolchains to deploy OpenCL code on FPGA. However, the fundamental architectural differences between GPUs and FPGAs have led to performance portability issues: it has been shown that OpenCL code optimized for GPU does not necessarily map well to FPGA, often requiring manual optimizations to improve performance.**

In this paper, we explore the use of thread coarsening – a compiler technique that consolidates the work of multiple threads into a single thread – on OpenCL code running on FPGA. While this optimization has been explored on CPU and GPU, the architectural features of FPGAs and the nature of the parallelism they offer lead to different performance considerations, making an analysis of thread coarsening on FPGA worthwhile. Our evaluation, performed on our microbenchmarks and on a set of applications from open-source benchmark suites, shows that thread coarsening can yield performance benefits (up to 3-4x speedups) to OpenCL code running on FPGA at a limited resource utilization cost.

Keywords— OpenCL, FPGA, high-level synthesis, compiler techniques, thread-coarsening, performance optimization

I. Introduction

Demands for high throughput and energy efficiency have led to an ever-increasing hardware heterogeneity in computer systems. Many supercomputers contain general purpose CPUs, GPUs and Intel many-core processors [1]. To further this trend, in the past few years there has been an increasing interest in using Field Programmable Gate Arrays (FPGAs) in data centers and high-performance computing clusters. Today major cloud computing services, such as Microsoft Azure [2] and Microsoft Web Services [3], offer FPGA-based computing instances.

Despite their compute capabilities and power efficiency, the wide adoption of FPGAs has been traditionally hindered by programmability issues. Programming with hardware description languages (HDL) is considered a specialized skill and requires logic design expertise. Hence, there have been significant efforts aimed to provide high-level synthesis (HLS) frameworks for FPGAs. In recent years, there has been a push towards the introduction of unified programming interfaces and languages allowing deployment of the same code on different hardware platforms seamlessly. This push has led to the definition of the OpenCL standard, initially targeting CPUs and GPUs, and of associated compilers and runtime libraries. Xilinx and Intel, the major FPGA vendors, are now providing their own OpenCL-to-FPGA toolchains, enabling programmers to deploy OpenCL code also on FPGA devices.

While OpenCL offers programming productivity, there is still a large performance gap between applications written in OpenCL and custom HDL versions of the same applications. This leaves room for much needed research and development aimed to improve existing OpenCL toolchains to fill the performance gap between OpenCL codes and custom HDL designs. Besides providing ease of programming, OpenCL allows easily porting applications from one hardware platform to another. However, performance portability is still a significant issue. Indeed, it has been shown that OpenCL code designed and optimized for GPU often performs poorly on FPGA [4]. To address this problem, several efforts have explored best practice optimizations, platform-agnostic and FPGA-specific compiler techniques operating directly on OpenCL source code and aimed to improve its efficiency on FPGA [4] [5] [6] [7] [8] [9] [10] [11].

This performance portability issue is due to the different architectural characteristics of GPUs and FPGAs and to the different kinds of parallelism they offer. While GPUs rely on their SIMD-like architecture to execute tens of thousands of threads simultaneously, FPGAs leverage pipelining to allow parallel execution of threads. In addition, synchronization primitives such as barriers and atomics are more efficiently supported on GPU than on FPGA, where barriers lead to pipeline flushes. Furthermore, current OpenCL-enabled FPGA boards have a lower global memory bandwidth than high-end GPUs. These factors suggest that the performance of OpenCL codes intended to run on FPGA can benefit from reducing the number of threads while exposing increased instruction-level parallelism, allowing the OpenCL-to-FPGA compiler to generate deeper and more efficient pipelines performing the work of multiple threads without requiring full logic replication.

In this work we explore thread coarsening – a compiler technique that consolidates the work of multiple threads into a single thread – on FPGA. Thread coarsening can be performed at the OpenCL level, allowing portability across platforms and compiler versions. This optimization allows reordering independent instructions within the consolidated threads, thus exposing instruction-level parallelism opportunities to the compiler and enabling memory accesses reordering. This technique has been extensively investigated on GPUs, CPU, and Intel Phi devices [12][13][14][15][16], showing modest performance benefits (1.1x-1.5x speedup). However, by inherently transforming SIMD parallelism into pipeline parallelism, thread coarsening can be more suitable for FPGAs.

2640-0316/21/$31.00 ©2021 IEEE
DOI 10.1109/HiPC53243.2021.00062

Our study makes the following contributions. First, we explore potential benefits and limitations or threads coarsening on FPGA and evaluate it on applications from the Rodinia and Pannotia benchmark suites [17][18]. We choose applications exhibiting different computation and memory access patterns and used in various domains. Our evaluation covers different ways of consolidating the work of multiple threads as well as different degrees of workload consolidation. Second, we compare the performance of thread coarsening with that of two other techniques to increase the amount of work performed concurrently by an FPGA kernel, namely, *pipeline replication* and *SIMD vectorization*. Third, to better understand the results, we design microbenchmarks with different code patterns. Our microbenchmarks allow exploring how factors such as memory access patterns and control flow divergence impact the performance of thread coarsening.

Our evaluation shows that, on FPGA, thread coarsening can lead to substantial performance benefits (up to 3.5x speedup on the benchmarks considered) at a limited resource utilization cost. The most significant factor hindering the performance of this optimization on FPGA is the presence of irregular memory access patterns in the kernel code. Furthermore, thread coarsening is more generally applicable than SIMD vectorization, and leads to performance comparable to pipeline replication at a reduced resource utilization cost. It is worth noting that thread coarsening, pipeline replication and SIMD vectorization are not mutually exclusive and can be combined.

II. Background

OpenCL is an open, cross-platform standard widely used to program heterogeneous platforms including multicore CPUs, GPUs, and FPGAs [19]. OpenCL compilers for FPGA extract pipeline parallelism from kernel code[20]. As an example, Fig. 1 shows a vector addition kernel and the corresponding hardware pipeline. The built-in function `_get_global_id` returns the thread-specific identifier, which each thread then uses to access a different element of arrays `a`, `b` and `c`. These arrays are stored in global memory (`_global` address space qualifier). The OpenCL-to-FPGA compiler instantiates load and store units to perform memory accesses and breaks the computation into stages. As can be seen, in this case, a 3-stage pipeline is created, with two load units for arrays `a` and `b` and one store unit for array `c`. At each clock cycle, a new thread will enter the pipeline, and different threads will be operating in parallel in different pipeline stages. Thus, the pipeline depth will be an indicator of the degree of parallelism of the kernel. The compiler can select different kinds of load-store units based on the nature of the memory accesses performed.

SIMD vectorization and *pipeline replication* are two mechanisms that can be used to increase hardware parallelism. On Intel platforms, these two optimizations can be enabled explicitly through the `num_simd_work_items` and `num_compute_units` keywords, respectively. SIMD vectorization allows multiple work-items (i.e., OpenCL threads) to execute in a SIMD fashion. Pipeline replication allows multiple work-groups to execute concurrently using different hardware pipelines. While SIMD vectorization shares control logic across SIMD vector lanes and allows the compiler to coalesce memory accesses, pipeline replication is less resource efficient and can lead to memory contention across hardware pipelines. However, SIMD vectorizations have several restrictions. Most notably, portions of a kernel in which work-items take different control paths (for example, due to work-item identifiers dependent branches) cannot be vectorized.

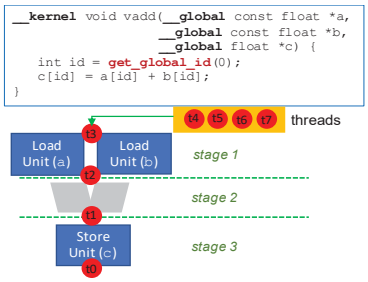

Fig. 1. Vector addition kernel and corresponding hardware pipeline.

III. Thread Coarsening on FPGA

A. Introduction to Thread Coarsening

Thread coarsening is a compiler technique that reduces the degree of multithreading of a parallel kernel by merging the work of multiple work-items into one work-item. This transformation increases the number of instructions each work-item executes, introducing opportunities for the compiler to apply additional optimizations to the code (such as instruction reordering). [12][13][14][15][16] On recent GPU architectures, the performance benefits of thread coarsening are modest, and they are mostly due to a reduction in the multithreading cost (e.g., kernel launch overhead) and to more efficient memory access patterns enabled by instruction reordering. On FPGA, in addition to allowing more efficient memory accesses, thread coarsening has the potential for generating hardware code that requires fewer load units, store units, arithmetic units, and hardware resources to handle control flow instructions across work-items. In addition, instructions belonging to different work-items are independent. Therefore, by assigning independent instructions to a single work-item, thread coarsening can expose more instruction level parallelism and lead to deeper hardware pipelines.

Despite its potential benefits, the use of thread coarsening is subject to tradeoffs. One drawback of this technique is that it can increase the number of resources used by the kernel, which can harm performance. For example, on GPU thread coarsening can lead to register pressure, thus limiting the number of active threads and the potential for memory latency hiding through multithreading. On FPGA, depending on the initial structure of the memory accesses, thread coarsening can increase the pressure on the memory units, resulting in more stalls for memory accesses.

Thread coarsening can be configured using two parameters: *coarsening type* and *coarsening degree*. The coarsening type determines the distribution of work to work-items, while the coarsening degree indicates the number of work-items consolidated into a single work-item. We consider two types of thread coarsening: *consecutive* and *gapped* coarsening. The former merges instructions from consecutive work-items into

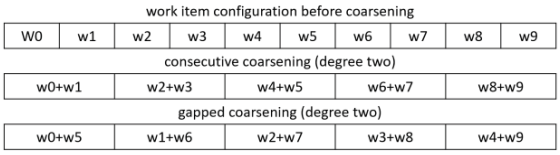

Fig. 2. Distribution of work to work-items before and after coarsening (coarsening degree of two).

one bigger work-item, while the latter divides the work-items into smaller evenly distributed groups and picks instructions from one work-item per group to form a larger work-item. Fig. 2 illustrates the work distribution resulting from these two types of thread coarsening with a coarsening degree of two.

In Fig. 3 we show how thread coarsening can be applied to an OpenCL kernel. The reference kernel (top of Fig. 3) is one-dimensional. The kernel loads values from global memory using global pointers (*in0* and *in1*), performs an arithmetic operation on them, and stores the results to a separate global memory location (*out0*). The code in the middle and the bottom of Fig. 3 shows the kernel after applying consecutive and gapped coarsening, respectively, with a degree of two. In both cases, instructions from two work-items are consolidated into a single work-item. In the code templates in Fig. 3, the handling of the work-item identifiers required to distinguish instructions from different work-items is highlighted in red. Instructions belonging to different work-items in the original kernel are highlighted using different colors (black and green). The names of the variables are extended (through "_0" or "_1") to distinguish the work-item of provenance in the original kernel.

B. FPGA-specific Considerations

As shown in Fig. 3, when performing thread coarsening, instructions originating from different work-items are interleaved. This has two implications. First, since instructions belonging to different work-items are independent, this method exposes instruction level parallelism, with the potential for deeper pipelines on FPGA. Second, memory operations are clustered together. Depending on the coarsening type and the memory access patterns of the original code, this might increase data locality, lead to better memory bandwidth utilization and, more generally, to more efficient memory accesses.

To better understand the effect of thread coarsening on kernel performance on FPGA, it is necessary to know how OpenCL-to-FPGA compilers handle memory instructions. Here, we refer to Intel's FPGA SDK for OpenCL Offline Compiler. Memory operations are handled through different types of load-store units (LSUs); the two most relevant are burst-coalesced and prefetching LSUs. The offline compiler determines which LSU type to instantiate based on inferred memory access patterns, types of memory available on the target device, and locality of memory accesses. Burst-coalesced LSUs buffer memory access requests until the largest possible number of requests can be sent to global memory at once. This type of LSUs can be instantiated with a separately assigned cache with a default size of 512Kb. The cache is assigned based on whether the memory access patterns are inferred to be data-dependent or repetitive. The compiler instantiates prefetching LSUs when it detects a contiguous read from a non-volatile global pointer. Burst-coalesced LSUs are more resource-

```
__kernel void multiplication(__global float * in0,
                             __global float * in1,
                             int N,  __global float * out0) {
  for (int gid=get_global_id(0); gid < N; gid+=get_global_size(0)){
    float r0= in1[gid];   float r1= in0[gid];
    float r2= r1*r0;
    out0[gid] = r2;
  }
}
```

```
__kernel void thc_multiplication_c(...) {
  for (int gid=get_global_id(0)*2; gid < N; gid+=get_global_size(0)*2){
    int gid_0= gid+0;   int gid_1= gid+1;
    float r0_0= in0[gid_0];   float r1_0= in1[gid_0];
    float r0_1= in0[gid_1];   float r1_1= in1[gid_1];
    float r2_0 = r1_0*r0_0;   float r2_1 = r1_1*r0_1;
    out0[gid_0] = r2_0;   out0[gid_1] = r2_1;
  }
}
```

```
__kernel void thc_multiplication_g(...) {
  int gapped_length = N / 2;
  for (int gid=get_global_id(0); gid < gapped_length; gid+=get_global_size(0)){
    gid_0= gid + gapped_length*0;   gid_1= gid + gapped_length*1;
    float r0_0= in0[gid_0];   float r1_0= in1[gid_0];
    float r0_1= in0[gid_1];   float r1_1= in1[gid_1];
    float r2_0 = r1_0*r0_0;   float r2_1 = r1_1*r0_1;
    out0[gid_0] = r2_0;   out0[gid_1] = r2_1;
  }
}
```

Fig. 3. Simple microbenchmark kernel with regular memory accesses before applying coarsening (top), with consecutive coarsening (middle) and with gapped coarsening (bottom).

intensive as they require more look-up tables (ALUTs), flip-flops (FFs), and possibly RAM blocks; however, they can provide better performance than prefetching LSUs.

For example, for the code in Fig. 3, the offline compiler assigns each load instruction a separate burst-coalesced LSU to handle the global memory access in the baseline code. After consecutive coarsening, the offline compiler makes all the memory accesses to the same global pointer be handled at once through a single 512-bit (8 floating-point values) width burst-coalesced LSU. On the other hand, the memory access pattern to the same pointer from gapped coarsening caused the offline compiler to create eight 32-bit (1 floating-point value) width burst-coalesced cached LSUs. This is because the offline compiler cannot find a pattern to coalesce memory accesses in the gapped coarsened kernel code and therefore it creates the same number of LSUs for each global pointer as the coarsening degree. Since one wider LSU is more efficient at accessing the same number of values from global memory than eight smaller LSUs, consecutive coarsening resulted in faster accesses to global memory compared to gapped coarsening. The reduction in the total number of memory accesses needed by the work-items is one of the key reasons why thread coarsening can improve the performance of kernels on the FPGA.

Another important factor that can affect the impact of thread coarsening on FPGA is work-item divergence. Work-item divergence prevents the offline compiler from coalescing memory accesses, applying code reordering, and performing other optimizations that require instructions being in the same basic block. We distinguish two types of divergence: *direct* and

```
for(int gid=get_global_id(0); gid < N; gid+=get_global_size(0)){
// Begin memory operations
   float r0= in7[gid];
   float r1= in6[gid];
   . . .
   float r7= in0[gid];
// Begin arithmetic operations
   float r8= r7+r3;
   float r9= r8+r5;
   . . .
   float r16= r15/r5;
   out0[gid] = r16;
}
```

```
// if-id
   if (get_global_id (0) %2 == 0) {
      ....
   }
// if-in
   if (get_global_id (0) %2 == 0) {
      ....
   }
// if-id
 for (int i=0; i<5; i++) {
    if (get_global_id (0) %2 == 0) {
        ....
    }
 }
```

Fig. 4. (a) Microbenchmark kernel baseline (b) Work-item divergence

indirect. Direct divergence originates when the condition of a control flow statement in the code depends on the work-item identifier. For instance, branches with a condition depending on the result of the *get_global_id* function call lead to direct divergence. Indirect divergence originates when the condition of a control flow statement or boundaries of a for-loop depend on a value that is loaded from global memory and can differ across work-items. In kernels with indirect work-item divergence, the offline compiler is unable to simplify the control flow graph for each work-item; therefore, it can result in fewer optimizations from the offline compiler after applying coarsening compared to direct divergence.

C. Methodology

We first evaluated the effect of thread coarsening on FPGA on a set of applications with different computation and memory access patterns from the Rodinia and Pannotia benchmark suites [17], [18]. Our observations on these applications showed several code features causing bottlenecks that warranted further exploration. To accomplish this, we created several microbenchmarks isolating each feature to gain further insight on how they individually affect the performance and resource utilization of consecutive and gapped coarsening. The main kernel features isolated within these microbenchmarks were: arithmetic intensity, nature of the memory access patterns (i.e., regular vs. irregular), memory access locality and divergence, and work-item divergence (conditional statements, for loops, and degree of divergence). When studying the effect of a specific code feature on the performance of thread coarsening, we kept all other features static by setting them to a default value. To generate microbenchmarks with realistic features, we determined each feature's default value by averaging the values observed for that feature in Rodinia and Pannotia benchmarks considered. For space limitation, in this paper, we focus on two aspects: memory access patterns and work-item divergence.

Baseline code – All microbenchmarks consist of a load phase, a computation phase, and a store phase (see baseline code in Fig. 4a). The load and store phases access the input and output arrays using configurable memory access patterns.

Memory access types – The microbenchmarks include both regular and irregular memory access patterns on arrays. For the regular memory access patterns (Fig. 5a), the data array is directly indexed using the work-item identifier. For the irregular ones, the data array is instead indirectly indexed via another array accessed using the work-item identifier. The indices in the intermediate array are generated based on the *irregularity*

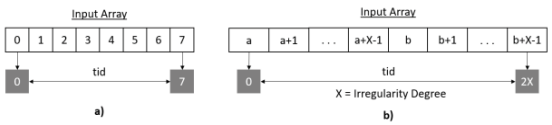

Fig. 5. Memory access patterns with direct and indirect indexing – (a) and (b), respectively.

degree parameter illustrated in Fig. 5b, where *a* and *b* (and, depending on the size of the array, possibly other values) are randomized starting indexes.

Work-Item Divergence – Work-item divergence microbenchmarks cover conditional statements, for-loops, and allow varying the degree of divergence among work-items. Conditional statement benchmarks use either the work-item identifier or a value in a data array, named *if-id* and *if-in* respectively, to determining whether to take a branch. The for-loop benchmarks use either an if-id configuration nested inside a for-loop with a constant bound (for-constant + if-id) or an if-in configuration nested inside a for loop with a bound reliant on a value in a data array (for-in + if-in). Examples of these code patterns are shown in Fig. 4b.

IV. Experimental Evaluation

A. Experimental Setup

Hardware – We run our experiments on an Intel programmable acceleration card with an Arria® GX FPGA. This board is equipped with two 4 GB DDR-4 SDRAM memory banks and 128 MB flash memory. The SDRAM memory can support a peak bandwidth of 34.1 GB/s. This FPGA includes 65.7 Mb of on-chip memory, 1150k logic elements (ALUTs), and 3036 digital signal processing (DSP) blocks. The host processor is an Intel Xeon® CPU E5-1607 v4 with a peak clock frequency of 3.1 GHz. We used Intel FPGA SDK for OpenCL version 19.4 with Ubuntu 18.04.5 LTS on the host system.

Benchmarks – We evaluated thread coarsening on two sets of benchmarks. The first set includes applications from Rodinia and Pannotia [18] benchmark suites. Table I summarizes the relevant characteristics of the applications and input datasets used and reports the execution time and resource utilization (in terms of logic elements, RAM blocks, and DSPs) of the baseline code (i.e., the unmodified OpenCL implementations from the benchmark suites). The second set includes our automatically generated microbenchmark kernels to evaluate the effects of different code features on thread coarsening performance.

Code variants – For all benchmarks, we generated thread-coarsened kernels using consecutive and gapped coarsening and coarsening degrees 2, 4, and 8. In addition, we tested pipeline replication (2, 4 and 8 hardware pipelines) and, whenever applicable, SIMD vectorization (with degrees 2, 4 and 8).

Evaluation metrics – For performance, we report the speedup over the original un-coarsened kernel with a single hardware pipeline (baseline). For resource utilization, we show the increase in the number of ALUTs and RAM blocks required by the thread-coarsened kernels over the baseline code.

TABLE I. CHARACTERISTICS OF RODINIA AND PANNOTIA BENCHMARKS USED IN THE EXPERIMENTS, INCLUDING EXECUTION TIME AND RESOURCE UTILIZATION OF THE ORIGINAL CODE (BASELINE)

Suite	Benchmark	Dwarves	Memory Access Pattern	Dataset Description	Execution Time (ms)	ALUTs	RAM Blocks	DSPs
Rodinia	Breadth-First Search (BFS)	Graph Traversal	Irregular	Input generator graph, #nodes=1M	172	31171	263	0
				coPapersCiteseer (copcs)	294			
	Hotspot	Structured Grid	Regular	Input generator, Size=2048	30300	14606	195	14
	Pathfinder	Dynamic Programming	Irregular	Input generator, Size=1000000×1000	567	13640	186	3
	LU Decomposition	Dense Linear Algebra	Regular	Input generator, Size=2048	13980	13640	486	25
	Back Propogation	Unstructured Grid	Regular	Input generator, Size=1048576	792	33224	510	13
	Gaussian Elimination	Dense Linear Algebra	Regular	Input generator, Size=256	1330	19724	277	11
	k-Nearest Neighbors	Dense Linear Algebra	Regular	Input generator, Size=8.3M	918	7856	75	7
Pannotia	Floyd-Warshall	Graph Traversal	Irregular	Pre generated, Size=512	1570	11457	168	3
	Page rank	Graph Traversal	Irregular	USA-road-d	627	29032	341	11
				coPapersCiteseer (copcs)	3140			

B. Experimental Results

Benchmark applications – Fig. 6 summarizes the performance and hardware resource utilization results from gapped and consecutive thread coarsening (*Gap* and *Con*, respectively), pipeline replication (*Pipe*), and SIMD vectorization (*SIMD*) for all considered benchmark applications. DSP utilization is not shown in these graphs since it scales linearly with the degree for all these optimizations. Additionally, Fig. 6 reports the average of the best speedup and the respective resource utilization for each method across all the benchmarks (rightmost bars). The number above each column indicates the degree (among 2, 4, and 8) that led to the best speedup. The missing results for SIMD vectorization are due to the inability of the compiler to vectorize kernels containing work-item identifier dependent branches (hence, SIMD vectorization averages are not included).

We make the following observations. First, on average pipeline replication performs slightly better than thread coarsening but at the cost of significantly higher resource utilization. Like SIMD vectorization, thread coarsening avoids control logic duplication. Second, while on two benchmarks (Hotspot and Backprop) SIMD vectorization yields the best speedup and resource utilization, this optimization is applicable only on kernels without complex control flows. Third, pipeline replication can achieve a slightly higher speedup (1.91x) among the tested benchmark applications compared to consecutive coarsening (1.56x) and gapped coarsening (1.7x). Fourth, when we compare the speedup from pipeline replication with the best speedup that can be achieved from applying thread coarsening (the best performing between gapped and consecutive), thread coarsening can achieve on average a 2x speedup across these benchmarks. In addition to a slightly better combined coarsening speedup, consecutive/gapped coarsening on average uses 34/41% fewer ALUTs and 22/32% fewer RAM blocks, respectively, compared to pipeline replication.

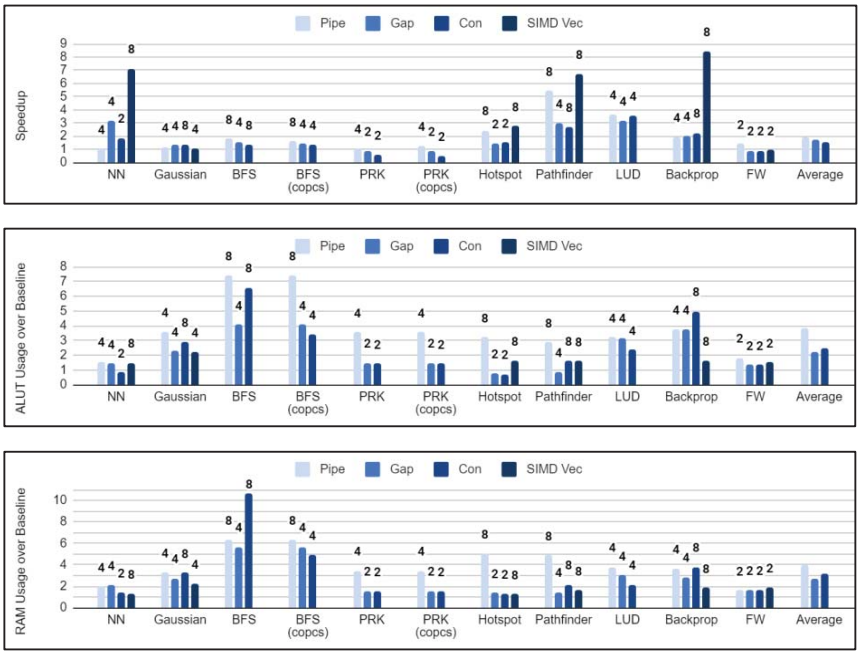

Fig. 6. Speedup increase (top), ALUT usage (middle), and RAM usage (bottom) compared to the baseline kernel from the best performing coarsening or pipeline degree (degree listed above each column).

On benchmarks exhibiting irregular memory access patterns (i.e., BFS and PageRank), pipeline replication outperforms thread coarsening, and gapped coarsening is preferable to consecutive coarsening. For these two algorithms, both run on two graph datasets, the input graph does not affect the performance trends. While graph applications report limited performance gains from pipeline replication, Pathfinder's performance scales with the replication degree (not shown in the graph). The low number of load/store units and high arithmetic intensity of Pathfinder keeps the memory bandwidth from being saturated and lets the pipeline replication speedup scale with the degree.

Regular applications exhibit different behaviors. Dense linear algebra applications (i.e., NN, LU decomposition, Gaussian Elimination) generally benefit from thread coarsening. For NN, gapped thread coarsening is the most effective optimization and provides a speedup up to

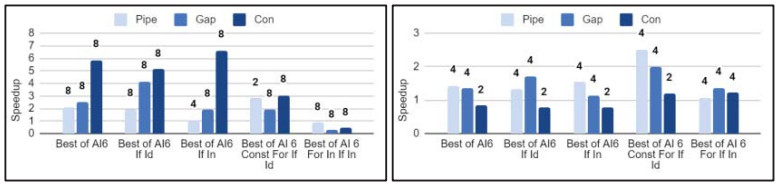

Fig. 7. Speedup comparison of microbenchmarks with regular (direct) memory accesses (left) and accesses and gapped coarsening provides a better performance improvement among kernels with indirect memory accesses.

~3x over the baseline code. On LU decomposition, thread coarsening, and pipeline replication exhibit similar performance and report a significant speedup (up to 3.5x). On Gaussian Elimination, while thread coarsening is the most effective implementation, it yields only a moderate speedup (about 30%) since this benchmark is dominated by memory accesses. In all three cases, going beyond coarsening/replication degree 4 does not bring further performance gain.

We also observed that pipeline replication shows better performance improvement than thread coarsening on kernels that include barrier synchronization (Pathfinder, Hotspot, Back Propagation, and LU decomposition) compared to the ones that do not have a barrier. Finally, these three optimization techniques are not mutually exclusive. For example, combining consecutive coarsening with degree 4 and pipeline replication with degree 2 on the Backprop kernel leads to a 3.2x speedup over the baseline, while the best speedup achieved using thread coarsening and pipeline replication alone are 2.1x and 2x (in both cases with degree 4), respectively.

Microbenchmarks – Fig. 7 shows results reported on microbenchmarks exhibiting different memory access patterns and control-flow divergence. The x-axis shows different code variants, where "AI *n*" means "Arithmetic Intensity with a degree of *n*" and the following words specify the type of control flow in the microbenchmark kernel (*if-id*, *if-in*, *for-constant+if-id*, and *for-in+if-in*). All these kernels access global memory arrays containing 64M elements. We report the best speedup for each optimization across the different degrees (2, 4, or 8).

Memory Access Type – In the presence of direct memory access patterns, the simple microbenchmark without branches and AI of six (AI6) achieves up to a 5.8x and a 2.1x speedup with consecutive coarsening and pipeline replication, respectively. We note that, as discussed in Section III.B, consecutive coarsening uses fewer ALUTs and RAM blocks than gapped coarsening of the same degree. When exploring the same kernel but with indirect memory accesses, gapped coarsening can only improve the performance of the kernel up to a 1.34x speedup whereas pipeline replication was able to achieve a 1.43x speedup. As for the benchmark applications, however, gapped coarsening required a smaller increase in the number of resources used to coarsen this kernel compared to pipeline replication codes of the same degree.

Work-Item Divergence – Except for the if-in version, adding control flow divergence resulted in a lower average speedup from consecutive thread coarsening in kernels with direct memory accesses. The offline compiler can take advantage of having more information regarding the work-item divergence in the constant-for+if-id kernel. Overall consecutive coarsening provides a better speedup for kernels with regular memory

V. Conclusion

In this work, we studied the impact of thread coarsening on the performance and resource utilization of OpenCL kernels running on FPGA. Our evaluation shows that thread coarsening can lead to performance comparable to pipeline replication at a reduced resource utilization cost, and is more generally applicable than SIMD vectorization. However, the benefits of thread coarsening can significantly decrease in the presence of irregular memory access patterns and control-flow divergence.

VI. Acknowledgements

This work was supported through NSF awards CNS-1812727 and CCF-1741683.

References

References

[1] Top 500 list. https://www.top500.org/

[2] Microsoft Azure FPGA. https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service.

[3] Amazon EC2 F1. https://aws.amazon.com/ec2/instance-types/f1/

[4] H. R. Zohouri et al, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in Proc. of SC16.

[5] K. Krommydas et al, "Bridging the performance-programmability gap for FPGAs via OpenCL: A case study with OpenDwarfs," in Proc. of FCCM 2016.

[6] A. Sanaullah, R. Patel and M. Herbordt, "An empirically guided optimization framework for FPGA OpenCL," in in Proc. of FTP 2018

[7] M. W. Hassan et al, "Exploring FPGA-specific optimizations for irregular OpenCL applications," in Proc. of ReConFig 2018.

[8] Q. Jia and H. Zhou, "Tuning stencil codes in OpenCL for FPGAs " in Proc. of ICCD 2016.

[9] K. Krommydas et al, "Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures," Journal of Signal Processing Systems, vol. 85, (3), pp. 373-392, 2016.

[10] Y. Luo et al, "Evaluating irregular memory access on OpenCL FPGA platforms: A case study with XSBench," in Proc. of FPL 2017.

[11] M. Nourian, M. E. Zarch and M. Becchi, "Optimizing complex OpenCL code for FPGA: A case study on finite automata traversal," in Proc. of ICPADS 2020.

[12] A. Magni, C. Dubach and M. F. P. O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in Proc. of SC 2013.

[13] A. Magni, C. Dubach and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in Proc. of PACT 2014.

[14] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in Proc. of SC 2008.

[15] N. Stawinoga and T. Field, "Predictable thread coarsening," ACM Transactions on Architecture and Code Optimization (TACO), vol. 15, (2), pp. 1-26, 2018.

[16] H. Wu and M. Becchi, "Evaluating thread coarsening and low-cost synchronization on intel xeon phi," in Proc. of IPDPS 2020.

[17] S. Che et al, "Rodinia: A benchmark suite for heterogeneous computing," in Proc. of IISWC 2009.

[18] S. Che et al, "Pannotia: Understanding irregular GPGPU graph applications," in Proc. of IISWC 2013.

[19] OpenCL. https://www.khronos.org/opencl/.

[20] Intel FPGA SDK for OpenCL: Programming Guide. https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html.