A Code Transformation to Improve the Efficiency of OpenCL Code on FPGA through Pipes

Mostafa Eghbali Zarch
North Carolina State University
Raleigh, NC, USA
meghbal@ncsu.edu

Michela Becchi
North Carolina State University
Raleigh, NC, USA
mbecchi@ncsu.edu

ABSTRACT

Over the past few years, there has been an increased interest in using FPGAs alongside CPUs and GPUs in high-performance computing systems and data centers. This trend has led to a push toward the use of high-level programming models and libraries, such as OpenCL, both to lower the barriers to the adoption of FPGAs by programmers unfamiliar with hardware description languages, and to allow to deploy a single code on different devices seamlessly. Today, both Intel and Xilinx offer toolchains to compile OpenCL code onto FPGA. However, using OpenCL on FPGAs is complicated by performance portability issues, since different devices have fundamental differences in architecture and nature of hardware parallelism they offer. Hence, platform-specific optimizations are crucial to achieving good performance across devices.

In this paper, we propose a code transformation to improve the performance of OpenCL codes running on FPGA. The proposed method uses pipes to separate the memory accesses and core computation within OpenCL kernels. We analyze the benefits of the approach as well as the restrictions to its applicability. Using OpenCL applications from popular benchmark suites, we show that this code transformation can result in higher utilization of the global memory bandwidth available and increased instruction concurrency, thus improving the overall throughput of OpenCL kernels at the cost of a modest resource utilization overhead. Further concurrency can be achieved by using multiple memory and compute kernels.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**.

KEYWORDS

OpenCL, FPGA, high-level synthesis, compiler techniques, pipes, performance optimization

ACM Reference Format:
Mostafa Eghbali Zarch and Michela Becchi. 2023. A Code Transformation to Improve the Efficiency of OpenCL Code on FPGA through Pipes. In *20th ACM International Conference on Computing Frontiers (CF '23), May 9–11, 2023, Bologna, Italy*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3587135.3592210

1 INTRODUCTION

Over the past several years, there has been an increasing trend toward using heterogeneous hardware in single machines and large-scale computing clusters. This trend has been driven by demands for high performance and energy efficiency. Initially, heterogeneity has mostly involved using GPUs and Intel many-core processors alongside multi-core CPUs [6]. More recently, due to their compute capabilities and energy efficiency, the trend has evolved to include Field Programmable Gate Arrays (FPGAs) [20] in high-performance computing clusters and data centers. Today, Microsoft Azure and Amazon Web Services include FPGAs in their compute instances [2][3].

Hardware heterogeneity involves significant programmability challenges. Without a unified programming interface, not only are users required to become familiar with multiple programming frameworks, but they also need to understand how to optimize their code to various hardware architectures. To address this challenge, the Khronos group has introduced a unified programming standard called OpenCL, which is intended for accelerated programming across different architectures [4]. This programming model initially targeted CPUs and GPUs. At the same time, programming FPGAs using low-level hardware description languages (HDLs) has traditionally been considered a specialized skill. To facilitate the adoption of FPGAs, vendors have spent substantial resources on the design and the development of OpenCL-to-FPGA toolchains, including runtime libraries and compilers allowing the deployment of OpenCL code on FPGA. Intel and Xilinx, two major FPGA vendors, are now providing their own OpenCL-to-FPGA development toolchain and runtime system [5] [11].

Although OpenCL allows portability and productivity, it does not guarantee performance portability [24]. Specifically, due to architectural differences across devices, an OpenCL code tailored to one platform often performs poorly on a different one. For instance, when porting an OpenCL application from GPU to FPGA, three main architectural differences affect performance portability. First, these devices offer a different form of parallelism. FPGAs leverage deep pipelines to exploit parallelism across OpenCL work-items alongside spatial parallelism, while GPUs rely on massive SIMD execution of threads (or work-items). Second, the off-chip memory bandwidth of current FPGA boards is much lower than that offered by high-end GPUs, which results in inefficient memory operations and low overall application performance. Third, while GPUs provide relatively efficient support for synchronization primitives like barriers and atomic operations, barriers on FPGAs result in a full pipeline flush, leading to significant performance degradation. Several papers have aimed to improve the efficiency of existing OpenCL code (often tailored to GPUs) on FPGA through platform-agnostic and specific compiler optimizations and scheduling techniques [18][12][10][23].

In this work, we explore and evaluate the use of the feed-forward design model to improve the performance of OpenCL code on FPGA. The proposed model splits each kernel into two kernels - a *memory* and *compute kernel* - connected through pipes. At a high level, the model aims to increase the memory bandwidth utilization, reduce the memory units' congestion, and maximize the instructions concurrency within the application. We show that the feed-forward design model allows the offline compiler to generate designs with more efficient memory units and increased instruction parallelism, leading to better performance with a low resource utilization overhead. A simplified version of this scheme has been explored in [25] on simple micro-kernels, in most cases leading to performance degradation over the original single work-item version of the code. In this work, we show that, when generalized and applied to more complex kernels with irregular compute and memory access patterns, this technique can lead to significant speedups over the single work-item version of the code.

Our exploration is structured as follows. First, based on recommendations from Intel's OpenCL-to-FPGA documentation [1], we convert SIMD-friendly code into serial code (i.e., a single work-item kernel). Second, we split each kernel into two kernels (*memory* and *compute* kernels), thus separating global memory reads/writes from the rest of the instructions inside the kernel. In order to minimize the data communication latency, we connect these kernels through pipes. We study the effect of loop-carried dependencies in the original code on the applicability of the method and its performance benefits. Lastly, we explore increasing the concurrency by having multiple versions of *memory* and *compute* kernels working on different portions of the data. In our experiments, we first compare the performance and resource utilization of the original kernels and the versions using the feed-forward design model. We then evaluate the effect of increasing the number of memory and compute kernels on performance and resource utilization.

In summary, this work makes the following contributions:

- A systematic code transformation method that separates the memory accesses and core computation within an OpenCL kernel. The proposed approach aims to improve memory bandwidth utilization and maximize instruction concurrency.
- An analysis of the effect of different classes of loop-carried dependencies on the applicability and benefits of the proposed code transformation method.
- An evaluation on a set of applications from popular benchmark suites [9][8]. Our experiments show performance improvements from our method from 30% up to 86× over baseline code at the cost of a modest resource utilization overhead.
- A study based on synthetic microbenchmarks to evaluate how the applications' compute and memory access patterns affect the performance improvement achievable through the proposed code transformation.

2 BACKGROUND

2.1 OpenCL for FPGA

OpenCL allows programmers to write platform-agnostic programs and deploy them on a wide range of OpenCL compatible devices. An OpenCL application consists of two types of code: host code and device code. The host code is responsible for data allocation on the host machine and accelerators (devices), communication setup and data transfer between host and devices, configuration of the accelerators, and launching the device code on them. The device code contains the core compute kernels, is written to execute on one or multiple platforms, and is often parallelized. In OpenCL terminology, a kernel consists of multiple *work-items* evenly grouped in *work-groups*. When deployed on GPU, work-items correspond to threads and work-groups to thread-blocks.

OpenCL kernels can be in two forms: *NDRange* or *single work-item*. NDRange kernels consist of multiple work-items, distinguishable through their local and global identifiers, launched by the host code for parallel execution. This model is widely used for programming CPUs and GPUs; on FPGAs, concurrent execution of work-items is enabled through pipeline parallelism. Single work-item kernels have a serial structure, with only one work-item launched by the host code. The single work-item model is preferred when the NDRange version of the kernel presents fine-grained data sharing among work-items. Single work-item kernels are often recommended by FPGA vendors [11], partially because writing the same kernels in NDRange fashion might require expensive atomic operations or synchronization mechanisms to ensure correctness. In cases that the OpenCL application is in NDRange form like the baseline implementation of the benchmarks in [8] and [9], programmers can construct the single work-item version by embedding the body of the NDRange baseline kernel within a nested loop. The outer and inner loops must have the work-group and work-item sizes as the loop iteration count, respectively.

Major FPGA vendors - such as Intel and AMD/Xilinx - currently provide OpenCL-to-FPGA SDKs to facilitate FPGA adoption by a wide range of programmers with different skills. However, the automatic generation of FPGA code often incurs performance portability issues, especially when the OpenCL code was originally optimized for a different device, such as a GPU. To bridge the performance gap between FPGAs and other devices, it is critical to understand the performance limiting factors on FPGA, and design FPGA specific optimizations.

2.2 Use of load/store units on FPGA

In order to understand the effect of compiler optimizations and scheduling techniques on memory operations, it is essential to know how OpenCL-to-FPGA compilers implement memory operations using load/store units (LSUs). In the rest of the paper, we refer to Intel's OpenCL-to-FPGA SDK as the offline compiler. The offline compiler can instantiate several types of LSUs depending on the inferred memory access patterns of the memory operations. Two pieces of information used to determine the LSU type to be used are the memory region accessed (i.e., global versus local memory) and the types of LSUs available on the target FPGA platform. There are three LSU types available to the Intel's offline compiler: *burst coalesced*, *prefetching*, and *pipelined* LSUs. Burst coalesced

LSUs are often used as the default type. This type of LSU is the most resource-hungry memory module, and it is designed to buffer memory requests until the largest possible burst of data read/write requests can be sent to the global memory. Prefetching LSUs leverage a FIFO to read large blocks of data from global memory and aim to keep the buffer full of valid data. This type best fits memory operations with a sequential memory access pattern. For local memory accesses, the offline compiler typically instantiates pipelined LSUs, which submit memory requests in a pipeline manner as soon as they are received. In some cases, the offline compiler uses pipelined LSUs as an alternative for global memory accesses, resulting in slower but more resource-efficient memory units.

There are differences between terminologies and representations that different vendors (Xilinx/Intel) use for the OpenCL-to-FPGA memory model. However, they both represent the same support for memory instructions. For example, similar to a burst coalesced LSU functionality, Xilinx leverage burst transfers automatically using AXI burst transfers where applicable to improve the performance[21]. However, unlike Intel SDK, programmers cannot request a specific type of burst memory read, and they need to write the code in a way that causes the compiler to infer a burst memory read.

2.3 OpenCL pipes and channels

Pipes - OpenCL applications consisting of multiple kernels require efficient mechanisms for inter-kernel communication. Using global memory for this purpose requires race-free global memory accesses or the use of atomic operations and barriers, which can be inefficient on FPGA. The OpenCL standard provides a mechanism to pass data between kernels, called “pipes”. Essentially, pipes represent ordered sequences of data items. Each pipe has separate write and read endpoints, allowing an OpenCL kernel to write to one endpoint of the pipe while another kernel reads from the other endpoint. By allowing concurrent execution of interconnected kernels, pipes enable pipeline parallelism across kernels. Both FPGA vendors (Intel/Xilinx) support pipe features as part of their OpenCL-to-FPGA which makes this work applicable to FPGAs from both vendors. It is worth mentioning that, in OpenCL, the host and device(s) can also communicate through pipes, a feature not used in this paper.

Channels - Intel provides an OpenCL extension called “channels” as a mechanism for data communication between kernels. Programmers can define the depth of the channels as an input attribute. The offline compiler considers this input to be the minimum depth of that specific channel, and may increase the depth of the channels in two situations: first, if there is a need to balance the reconverging paths through multiple kernels; and second, to achieve better loop pipelining [1].

3 IMPLEMENTING THE FEED-FORWARD DESIGN MODEL

In this section, we first motivate our work (§ 3.1). Second, we provide a high level overview of how to implement the feed-forward design model on FPGA (§ 3.2) and illustrate it through an example (§ 3.3). We then analyze the impact of loop-carried dependencies on the applicability and benefits of the model (§ 3.4). At last, we present our approach to systematically transform a generic kernel to use this design model, we elaborate on the strengths and weaknesses of the method, and we propose some optimizations to it (§ 3.5). The proposed transformation is performed on the OpenCL device code. The transformed code is then fed to the offline compiler to generate HDL code, which is then deployed on FPGA by using the vendor's synthesis, placing and routing tools.

Figure 1 shows the OpenCL-to-FPGA compilation and synthesis flow with the proposed feed-forward design transformation. As can be seen, the proposed transformation is applied to the original OpenCL code. The transformed OpenCL code is then fed to the vendor's HLS compilation toolchain (Intel/Xilinx). After generating the HDL code, the toolchain generates the FPGA bitstream. Finally, the compiled host C/C++ code invokes kernels compiled in the FPGA bitstream to accelerate parts of the application on the FPGA.

3.1 Motivation

Global memory accesses are known to be one of the main performance bottlenecks for OpenCL kernels implemented on FPGAs. Wang et al. [19] measured the memory bandwidth of sequential and random memory accesses for different variable types and concluded that random memory accesses within a kernel can limit the memory bandwidth achieved drastically. In addition, they observed that severe lock and memory bandwidth overhead limit the throughput of many of the OpenCL kernels they considered in their analysis. Optimizing memory accesses in OpenCL code is not a trivial task. While Intel's SDK gives the programmer some level of control over the type of load/store units (LSUs) instantiated by the offline compiler to handle memory instructions, selecting the optimal LSU for each memory operation requires a good understanding of the hardware and insights into the offline compiler's operation.

By studying the memory analysis reports of the offline compiler for a set of real and synthetic OpenCL kernels with various memory access patterns, we identified two significant factors affecting the kernel's performance: (i) the type and configuration of the LSUs instantiated by the compiler to handle global memory instructions, and (ii) the presence of dependencies on global memory instructions and of loop-carried dependencies on local variables. The offline compiler associates to each loop an *initiation interval* (II), which represents the number of clock cycles between the launch of successive loop iterations. In the presence of loop-carried dependencies, the offline compiler serializes the loop's execution, resulting in a high initiation interval and, consequently, low throughput.

In our study, we noticed that the offline compiler takes a conservative approach when identifying the memory dependencies in kernels. This is mainly due to the following reasons. First, the offline compiler cannot guarantee that the device finishes global store instructions before load instructions to the same location in the same or different iterations of a loop. Second, it does not gather information from the host code at compile time. This is a common shortcoming among different vendors. Xilinx's compiler also fails to identify memory dependencies in similar scenarios. This conservative approach limits the optimizations performed, negatively affecting the performance of memory instructions [21].

One advantage of the feed-forward design model is that it exposes information on the characteristics of the memory operations

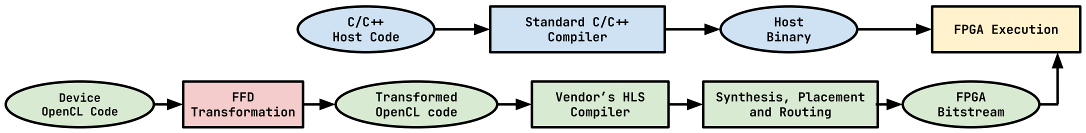


Figure 1: OpenCL-to FPGA compilation and synthesis flow with the proposed feed-forward design transformation

and data dependencies within a kernel to the offline compiler. Importantly, using the feed-forward design model implies the absence of unsafe loop-carried dependencies between load and store instructions on global memory. We will elaborate on unsafe loop-carried dependencies in Section 3.4. In addition, using the feed-forward model can increase the concurrency of memory instructions using multiple load units. In turn, this technique can result in synchronization-free kernels with high memory bandwidth utilization.

3.2 The feed-forward design model on FPGA

In the feed-forward design model, the computation is broken down into two kernels: a *memory kernel* and a *compute kernel*. The former is responsible for loading values from global memory, and the latter performs computation on the loaded data and stores the result back to global memory. In order to allow for efficient implementation, these two kernels should be connected through a hardware mechanism that does not involve the use of global memory, allowing the second kernel to avoid global memory loads. As explained in Section 2, programmers can use *pipes/channels* to establish this communication. The compute kernel can be further split in two kernels: the first performing computation, and the second issuing global memory writes. However, we verified that using a three-kernel model does not add performance benefits. Therefore, in the reminder of this paper we focus on the two-kernel (memory & compute) model.

Previous work has explored the use of pipes to connect multiple kernels that are already part of an application, creating efficient kernel pipelines [13]. Our study uses pipes for a different purpose, and also targets applications that consist of a single kernel. Specifically, our proposed transformation *splits* existing kernels to enable the use of the feed-forward model, and then uses pipes to connect the generated sub-kernels. Other work has explored the use of pipes on simple hand-written kernels, with the goal of improving the efficiency of their memory operations [25][19]. However, the performance advantages reported are limited, partially due to the simplicity and the regular memory access patterns of those kernels. Our goal is to propose a general code transformation method that allows applying the feed-forward execution model to diverse OpenCL kernels with regular or irregular memory access patterns, and with simple or complex control flows.

While the use of the feed-forward design model can improve the memory bandwidth utilization of OpenCL code deployed on FPGA, there are limitations to its applicability on iterative applications. Specifically, loop-carried dependencies can prevent or affect the application of this design model. However, not all loop-carried dependencies are problematic, and the code transformation method we propose can actually handle some of them. In Section 3.4, we categorize loop-carried dependencies and discuss their effect on applying the feed-forward design model to existing OpenCL kernels.

```
for (int tid = 0; tid < max_iter; tid++) {
    a = input0[tid];
    b = input1[tid];
    c = a * a + b * b;
    output0[tid] = c;
}
```

(a) Baseline kernel

```
for (int tid = 0; tid < max_iter; tid++) {
    a = input0[tid];
    write_channel_intel(c0, a)
    b = input1[tid];
    write_channel_intel(c1, b);
}
```

(b) Memory kernel (feed-forward model)

```
for (int tid = 0; tid < max_iter; tid++) {
    a = read_channel_intel(c0);
    b = read_channel_intel(c1);
    c = a * a + b * b;
    output0[tid] = c;
}
```

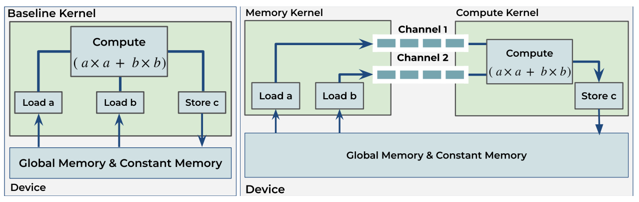
(c) Compute kernel (feed-forward model)

Listing 1: Feed-forward design model example

3.3 Feed-forward code transformation example

Listing 1 shows the application of the feed forward design model on a simple single work-item kernel. The proposed method applies also to NDRange kernels. However, Intel recommends using single work-item kernels when using channels and pipes [1]. The example in Listing 1 uses syntax from the Intel OpenCL-to-FPGA SDK.

In the example code, the baseline kernel performs two global memory loads (pointers *input0* and *input1*), calculates the sum of the squares of the values read, and writes the results to a different global memory location (pointer *output0*). Applying the feed-forward design model will split the baseline kernel in two kernels. The *memory* kernel contains the instructions required to load values from global memory. These include load instructions and computation that affects the control path of load instructions and the memory addresses they access. The *compute* kernel contains all the computation and store instructions. Data transfers between these two kernels are performed through channels, and global memory loads within the



(a) Baseline kernel **(b) Transformed design**

Figure 2: Illustration of the kernels in Listing 1

```
1 for (int tid = 5; tid < max_iter; tid++) {
2     r = 0;
3     for (int i = 0; i < 5; i++) {
4         a = input0[tid-i];
5         r += a;
6     }
7     output0[tid] = r
8 }
```

(a) Baseline kernel

```
1 for (int tid = 5; tid < max_iter; tid++) {
2     for (int i = 0; i < 5; i++) {
3         a = input0[tid-i];
4         write_channel_intel(c0, a);
5     }
6 }
```

(b) Memory kernel (feed-forward model)

```
1 for (int tid = 5; tid < max_iter; tid++) {
2     r = 0;
3     for (int i = 0; i < 5; i++) {
4         a = read_channel_intel(c0);
5         r += a;
6     }
7     output0[tid] = r
8 }
```

(c) Compute kernel (feed-forward model)

Listing 2: Data loop-carried dependency example

compute kernels are replaced by reads from assigned channels. Figure 2 illustrates the general hardware structure of the original and transformed kernels.

3.4 Loop carried dependencies analysis

Loop-carried dependencies are a key factor affecting the applicability of the proposed feed-forward code transformation method and its performance benefits. Here, we categorize loop-carried dependencies in *data loop-carried dependencies* (DLCDs) and *memory loop-carried dependencies* (MLCDs), and study the impact of these dependencies on the proposed code transformation method.

Data LCD - A Data LCD (DLCD) occurs when a local variable updated in one loop iteration is read in a different iteration (RAW dependency). The offline compiler serializes loops containing DLCDs. Listing 2 shows an example of DLCD. Updating variable *r* at line 5 of Listing 2a creates a DLCD on this variable for the inner loop. We note that this DLCD does not affect the control flow path of the global memory load instruction at line 4 or the memory address accessed by it (i.e., the *input0* array's index). However, in the presence of this dependency, compilers will serialize the inner loop. Serialization prevents pipeline parallelism and reduces memory bandwidth utilization, ultimately limiting kernel performance.

Transforming the kernel to the feed-forward model using our proposed code transformation often removes DLCDs from the memory kernel. In these cases, after the transformation, the loop with the DLCD becomes part of the *compute* kernel, which is free from global memory instructions. Removing DLCDs from the *memory* kernel allows the offline compiler to schedule load instructions more efficiently. While the DLCD is still present in the *compute* kernel, the serialized loop in the *compute* kernel will only load data from channels, which have low latency and high throughput - an advantage over the baseline version. Listings 2b and 2c show that the DLCD in the loop in Listing 2a is present only in the *compute* kernel, allowing the offline compiler to schedule the memory instructions in the *memory* kernel in a pipelined manner. We note that DLCDs cannot be removed from the *memory* kernel when they affect the control flow of load instructions or the memory addresses that they access. In those cases, while the feed-forward transformation is still safe to perform, it might not benefit the kernel's performance. Kernels with this characteristic can still benefit from using multiple *memory & compute* kernels, as discussed in Section 3.5.

```
1 for (int tid = 1; tid < max_iter; tid++) {
2     a = output[tid-1];
3     b = input[tid];
4     output[tid] = a * b;
5 }
```

Listing 3: Unsafe memory loop-carried dependency example

Memory LCD - A Memory LCD (MLCD) occurs when a global memory location updated in one loop iteration is read in a different iteration (RAW dependency). In the presence of MLCDs, the compiler needs to schedule the load and store instructions serially to guarantee correctness. We categorize MLCDs into *safe* and *unsafe* dependencies. Using the feed-forward design model on kernels with safe MLCDs guarantees correctness, while using this transformation on kernels with unsafe MLCDs can affect correctness and needs additional considerations.

An *unsafe* MLCD happens when a global load instruction on iteration j loads the value stored in the same location in iteration i, where: $j - i > 0$ or $j - i$ is unknown at compile time. Otherwise, we consider the MLCD to be *safe*. Unsafe MLCDs require manual intervention on the baseline code to be removed, thus allowing the feed forward code transformation while preserving correctness.

Programmers can use two methods to identify loop-carried dependencies within their kernels. First, OpenCL-to-FPGA compilers generate report files highlighting all the loop-carried dependencies inside each kernel. Programmers can parse these report files to extract loop-carried dependencies. Second, programmers can generate the LLVM IR for each kernel (for example, using the Clang compiler). Having the LLVM IR, programmers can create the data dependence graphs for each kernel using the LLVM analysis tool. In LLVM, dependence graphs include all the dependencies between instructions. Each cycle in the dependence graph indicates a loop-carried dependency. In this work, we used the first approach to extract loop-carried dependencies for each kernel in the code.

There are two common cases when unsafe MLCDs can be either disregarded or removed (i.e., made safe) through standard code transformations. The first case is when the distance between j and i is a function of the kernel input(s), and analyzing the kernel inputs or knowing their characteristics can determine this distance. In these cases, programmers can use their knowledge of the inputs to determine whether an unsafe MLCD at compile time is safe at runtime. The second case is when the distance between j and i is a positive integer known at compile time. Programmers can use shift registers to change these unsafe MLCDs to DLCDs [1]. However, the presence of an MLCD with a positive distance still hinders leveraging spatial parallelism with multiple *memory* and *compute* kernels, which we discuss in Section 3.5.

Listing 3 shows an example of an unsafe MLCD in which the statement at line 2 depends on the result of the execution of the statement at line 4 in the previous iteration (RAW with a distance of one). Using shift registers will remove the MLCD on the global memory pointer and move the dependency to a variable in either local or shared memory.

```
1  for (int tid = 0; tid < max_iter; tid++) {
2    if (input0[tid] == -1) {
3      start = input1[tid];
4      if (tid + 1 < max_iter)
5        end = input1[tid+1];
6      else
7        end = max_end;
8      r = 0;
9      for (int j = start; j < end; j++) {
10       if (input2[j] == -1)
11         r = r + input3[j];
12     }
13     output1[tid] = r;
14   }
15 }
```

(a) Baseline

```
1  for (int tid = 0; tid < max_iter; tid++) {
2    in0 = input0[tid];
3    write_channel_intel(c0, in0);
4    if (in0 == -1) {
5      start = input1[tid];
6      write_channel_intel(c1, start);
7      if (tid + 1 < max_iter)
8        end = input1[tid+1];
9      else
10       end = max_end;
11     write_channel_intel(c2, end);
12     for (int j = start; j < end; j++) {
13       in2 = input2[j];
14       write_channel_intel(c3, in2);
15       if (in2 == -1) {
16         in3 = input3[j];
17         write_channel_intel(c4, in3);
18       }
19     }
20   }
21 }
```

(b) Feed-forward design memory kernel

```
1  for (int tid = 0; tid < max_iter; tid++) {
2    in0 = read_channel_intel(c0);
3    if (in0 == -1) {
4      start = read_channel_intel(c1);
5      end = read_channel_intel(c2);
6      r = 0;
7      for (int j = start; j < end; j++) {
8        in2 = read_channel_intel(c3);
9        if (in2 == -1)
10         r = r + read_channel_intel(c4);
11     }
12     output1[tid] = r;
13   }
14 }
```

(c) Feed-forward design compute kernel

Listing 4: Feed-forward transformation example

3.5 Code transformation method

Listing 4 illustrates our proposed code transformation method on a more complex kernel. Our method consists of the following steps:

1. Identify instructions that read from global memory (lines 2, 3, 5, 10, & 11 of the baseline kernel in Listing 4a).
2. Use the LLVM analysis tool or the offline compiler report to find all the MLCDs inside the kernel. Note that the list of loop-carried dependencies identified by the offline compiler is provided in the compiler's report. Unless resolved (using inferring shift registers or runtime analysis), unsafe MLCDs prevent the safe application of the method.
3. Allocate a local variable for each load instruction used in the condition of a conditional statement or a loop and replace all its usages with the local variable (similar to lines 2 & 13 in Listing 4b).
4. Copy the baseline kernel into two different kernels, namely, the *memory* kernel and *compute* kernel (Listings 4b & 4c).
5. Define a channel for each global load instruction using the proper data type. If the same data item is loaded repeatedly in the baseline kernel, only assign a single channel to it.
6. Add to the *memory* kernel a write-to-channel instruction for each read from global memory unless the loaded value is only used as an index by another load instruction (lines 3, 6, 11, 14, & 17 of Listing 4b). Coherently with step (5), if a data item is loaded repeatedly in the baseline kernel, write the corresponding channel only once.
7. In the *compute* kernel, replace instances of reading from global memory with a read from the assigned channel (lines 2, 4, 5, 8, and 10 in Listing 4c).
8. Use manual or automatic compiler analysis to mark all the instruction that does not affect the loads from global memory.
9. Remove instruction marked in the previous step from the *memory* kernel.
10. Remove from the resulting kernels empty control flow paths and values not used (i.e., apply a dead code elimination pass).
11. Instantiate *memory kernel* and *compute kernel* multiple times to increase concurrency and adjust the main loop trip counts accordingly (more details provided below).
12. Repeat step (10).
13. Replace the baseline kernel launch inside the host code with invocations of the *memory* and *compute* kernels on separate queues.

We note that this technique can generate a *memory kernel* with a simplified control flow graph (CFG) compared to the original (baseline) kernel. A less complex CFG results in fewer stalls for global memory reads, hence leading to a higher memory bandwidth utilization. This, in turn, can lead to better overall performance.

Enabling feed-forward design model with multiple memory and compute kernels - The most significant advantage of our proposed technique is enabling the feed-forward design model to increase the memory bandwidth utilization for load instructions. This approach can achieve more performance advantages by increasing concurrency among memory operations. In the feed-forward design model, data moves from the memory kernel to the compute kernel in one or multiple words. For each word written on a pipe by the memory kernel, the compute kernel will process the data and free up the memory space assigned to the channel. Having multiple memory and compute kernels can potentially increase the global memory bandwidth achieved by the application, increasing performance at a limited resource utilization overhead.

As mentioned earlier, safe LCDs allow applying the proposed code transformation while preserving correctness. However, there are two cases where safe LCDs hinder using multiple *memory* and *compute* kernels to improve the performance of an OpenCL kernel.

Table 1: Benchmarks applications and datasets

Benchmark	Dwarves	Pattern	Dataset
Breadth-First Search (BFS)	Graph Traversal	Irregular	#nodes=2M
Hotspot (HS)	Structured Grid	Regular	Size=8192
k-Nearest Naighbors (NN)	Dense Linear Algebra	Regular	Size=8.3M
Hotspot 3D (HS-3D)	Structured Grid	Regular	Size=8192
Needleman-Wunsch (NW)	Dynamic Programming	Regular	Size=8192
Back Propogation (BP)	Unstructured Grid	Regular	Size=12.8M
Floyd-Warshall (FW)	Graph Traversal	Irregular	Size=512
Maximal Independent Set (MIS)	Graph Traversal	Irregular	Size=1.58M
Page Rank (PR)	Graph Traversal	Irregular	Size=1.58M
Graph Coloring (GC)	Graph Traversal	Irregular	Size=1.58M

The first case is when a DLCD is present in the outermost loop of a single work-item kernel. The second case is when an unsafe MLCD is resolved using shift registers.

Having multiple memory and compute kernels requires making various decisions regarding their number, the load balancing mechanism, and the buffer management scheme to be adopted. Kernel replication adds concurrency while increasing complexity and resource utilization. Intel recommends limiting the number of channels used in the design, as they can add complexity and limit overall performance. In addition, having a large number of kernels reading data from global memory concurrently can increase memory congestion and result in poor global memory bandwidth utilization. We explored using a single memory kernel with multiple compute kernels and vice versa. Results indicate that having separate memory and compute kernels communicating directly yields higher concurrency than having one memory kernel send data to multiple compute kernels.

Different load balancing mechanisms can be used to distribute the work across multiple memory/compute kernels. These mechanisms can be classified as either static or dynamic. Unlike dynamic mechanisms, static load balancing schemes do not consider the system's state when making decisions. Many dynamic load balancing schemes require busy-wait or feedback mechanism implementations involving kernels polling on non-blocking channels. This form of busy-wait can result in performance degradation on FPGA. This work uses static load balancing to connect memory and compute kernels. When using multiple memory/compute kernels, we partitioned the input workload consecutively and equally across different memory-compute pairs. By using this approach, we achieved two goals. First, we avoided busy waits and feedback paths between the memory and compute kernels. Second, we preserved the existing regular memory accesses within the application.

4 EXPERIMENTAL EVALUATION

4.1 Experimental setup

Hardware and Software Setup - We ran our experiments on an Intel programmable acceleration card (PAC) with an Arria® GX FPGA. This board contains two 4 GB DDR-4 SDRAMs memory banks with a maximum bandwidth of 34.1 GB/s, and 128 MB of flash memory. This FPGA includes 65.7 Mb of on-chip memory, 1150k logic elements (ALUTs), and 1518 digital signal processing (DSP) blocks. On the host side, the machine is equipped with an Intel Xeon® CPU model E5-1607 v4 with a maximum clock frequency of 3.1 GHz. We used the Intel FPGA SDK for OpenCL version 19.4 and Ubuntu 18.04.6 LTS.

Benchmarks - We evaluated our method on widely used open-source benchmarks from Rodinia [9] and Pannotia [8]. These benchmark suites contain applications from different domains, and have been used in previous work on OpenCL for FPGA [23] [14][24] [13]. Table 1 summarizes the main characteristics of these applications, including the nature of their memory access patterns (column 3).

In the second set of experiments, we used automatically generated microbenchmark kernels to evaluate the impact of the kernel characteristics (namely, memory access patterns and branch divergence) on the efficacy of the proposed method. We will elaborate on the features of these microbenchmarks in section 4.2.1.

Performance Metrics - For performance, we report the speedup over the original single work-item version of each benchmark. For resource utilization, we report the logic utilization and use of block RAMs (BRAMs) of each implementation. The logic utilization represents an estimate of the number of half-adaptive logic modules (ALMs) used by the compiler to deploy the OpenCL kernels on FPGA. ALMs are hardware logic blocks. The simplest version of ALMs contains a lookup table (LUT) and a register. The compiler reports the logic utilization as a percentage of the total number of half ALMs on the FPGA board. To evaluate improvements in memory bandwidth utilization and instruction parallelism resulting from the feed-forward design transformation, we also report the total memory bandwidth utilization and initiation intervals of the main loops before and after applying the proposed code transformation.

4.2 Experimental results

Single memory/single compute kernels - Table 2 shows the performance impact of applying the feed-forward design model to the considered benchmark applications when using a single memory and a single compute kernels. In all cases, we used the baseline code without any optimizations in order to isolate the impact of the feed-forward model on performance. We report the best results from running the same experiments using channels with three depths: 1, 100, and 1000. We recall that the depth parameter passed to the offline compiler indicates the minimum depth of that specific channel, but the compiler might increase the depth to improve efficiency. Our experiments showed that the channel depth parameter's setting does not affect performance significantly, proving that the Intel compiler does a good job of adjusting the channel depth.

As shown in Table 2, among the benchmarks we explored, BFS, FW, BP, MIS, and NW benefit significantly from applying the feed-forward model. For all these benchmarks, the main driver for the speedup is the removal of safe LCDs on variables located in the device's global memory. To better understand the effect of the feed-forward design on the generated hardware design, we include in the table the initiation interval (II) and memory bandwidth utilization of the designs. We recall that the II indicates the number of clock cycles between two consecutive loop iterations and is an indicator of the efficiency of the hardware pipeline generated by the compiler. The high II in the baseline code of the five applications denotes serialization inside the implementation as the offline compiler finds LCDs inside the kernels. This serialization results in

Table 2: Resource utilization and throughput comparison of the feed-forward design (FFD) model against the baseline code

Benchmark	Baseline Execution Time (ms)	FFD Speedup	Logic Utilization		BRAM		Frequency		Main Loop II		Memory Bandwidth (MB/s)	
			Baseline	FFD	Baseline	FFD	Baseline	FFD	Baseline	FFD	Baseline	FFD
BFS	6422	13.84	18.63	20.00	578	596	253	267	166	1	273	3687
HS	22553	0.85	16.14	17.25	517	522	313	257	1	1	8655	7340
NN	18.27	1.03	16.04	16.46	376	376	289	298	1	1	3468	3463
HS-3D	31967	0.88	16.45	17.95	542	536	294.63	258.13	1	1	10586	9315
NW	26036	50.95	16.10	18.86	506	407	267.23	266.02	283	1	660	2232
BP	140601	44.54	24.67	26.68	674	646	275.7	263.57	416	1	17087	21603
FW	41760	64.95	16.21	16.47	482	465	266	301	285	1	629	3129
MIS	2166	6.47	21.77	24.44	803	807	226	247	>=1	1	207	2116
PR	8430	0.96	20.43	22.52	703	709	272	261	1	1	2566	2283
GC	453	1.02	17.78	19.48	651	656	246	253	1	1	3345	3490

sub-optimal memory bandwidth utilization. After applying the proposed transformation, the offline compiler generates designs with an II of one, indicating a fully pipelined implementation. Hence, the transformed codes enjoy better throughput and improved memory bandwidth utilization over the baseline.

For FW, the unsafe loop-carried dependencies detected by the offline compiler resulted in a large initiation interval (II) of 285 for the main loop inside the kernel. As often done on global memory loads, the offline compiler used the burst coalesced LSU type to implement load instructions, resulting in low memory bandwidth for instructions with regular memory access patterns. The use of the feed-forward design model had two effects. First, resolving those unsafe loop-carried dependencies allowed converting the main loop to a fully pipelined loop with an II of 1. Second, it enabled the offline compiler to use a prefetching LSU for one of the three global load operations with a regular memory access pattern and increased the maximum global memory bandwidth of the kernel from 630 MB/s to 3130 MB/s. These changes resulted in a speedup up to 65× compared to the baseline kernel.

BP benefits from the feed-forward implementation similarly. In the original kernel, the performance is limited by the main loop, which exhibits an II of 416. In the feed-forward version of the kernel, the same loop is fully pipelined with an II of one. This decrease in the II also increased the maximum global memory bandwidth used by the kernel and resulted in a significant speedup of 44× over the baseline version of the kernel. Similar trends were observed on the other three benchmarks (BFS, MIS, and NW) benefiting from the feed-forward model.

We should note that the baseline version of NW carries an unsafe MLCD inside the main loop of the kernel. However, this MLCD is entangled to a load instruction in iteration *K*, which is dependent on a store instruction in iteration *K-1*. In this case, this MLCD can be resolved in the baseline kernel using a local variable in the private memory of the device. Storing the dependency value at the end of each iteration can remove the existing loop-carried dependency. As a result, the kernel can read the same value at the beginning of the next iteration (except the first iteration). Adding this private variable results in a single work-item baseline kernel with no MLCDs. Then, applying the feed-forward design model allows achieving a 50× speedup by decreasing the II of the main loop and increasing the global memory bandwidth of the kernel.

The feed-forward model reports minor performance improvements on NN and GC, and small performance degradation on HS, HS-3D, and PR (see Table 2). For these applications, the Intel profiler's report reveals the presence of one or multiple LSUs with high "occupancy". The LSU occupancy is an indicator of the percentage of the execution time where the LSU is not stalled. A high LSU occupancy (i.e., low number of memory stalls) is desirable, and indicates that the kernel is already making good use of the memory bandwidth and memory accesses do not need further optimization.

As reported in Table 2, our code transformation introduces only a modest resource utilization overhead. On average, our method only increases the logic utilization by 9% and even has a positive effect on BRAM usage on some benchmarks (at the expense of using more registers). The profiling and resource utilization reports from the offline compiler show that our method can optimize the load units from both the throughput and resource utilization point of view. This limits the overhead associated with transforming one kernel to two kernels. Moreover, channels are implemented using on-chip memory, hence, they introduce a very small overhead on resource utilization.

Multiple memory/compute kernels - The feed-forward design model enables using multiple memory and compute kernels to increase concurrency, at the cost of a resource utilization overhead. Figure 3 shows the speedup and resource utilization overhead from using two memory and two compute kernels (M2C2) over a single memory/single compute kernel implementation (M1C1). The results show an average speedup of 39% over the M1C1 baseline with only a 31% average increase in logic utilization and a 26% average increase in the number BRAMs used. In the case of PR and BP, the profiling data from the M1C1 version indicate highly optimized memory operations with high global memory bandwidth utilization. This characteristic hinders further performance improvements from using multiple memory and compute kernels. Further, we explored using four memory and four compute kernels (M4C4) for benchmarks that benefited from the M2C2 transformation. Except for a performance improvement of ≈20% on HS and BFS and of ≈15% on GC, we did not observe additional performance benefits. Using more memory and compute kernels to work concurrently on different portions of the data increases the memory bandwidth utilization of the application. However, the added concurrency leads to increased contention on the memory units and introduces more stalls and lower occupancy for the LSUs inside the kernel. Our experiments show that M4C4 implementations have, on average, 68% more stalls on LSUs than M2C2 kernels. Taking the associated resource utilization overhead into consideration, M2C2 results in the best multiple memory/compute kernels configuration. When

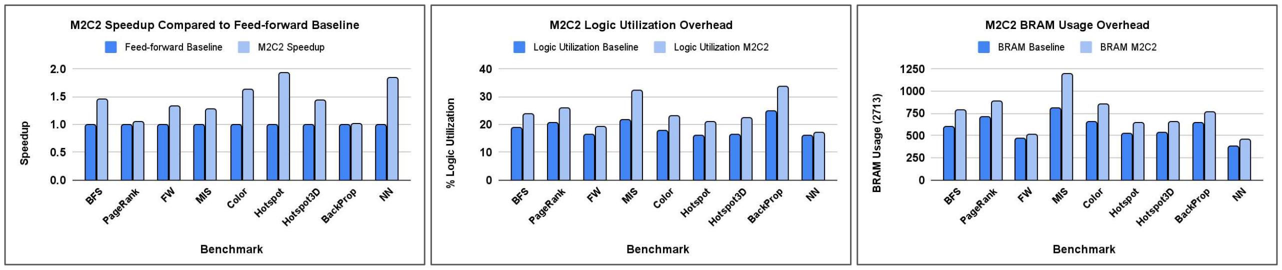


Figure 3: M2C2 speedup and resource utilization overhead compared to a single memory/single compute implementation

comparing the M1C1 and M2C2 code versions, we did not identify a uniform trend in the clock frequency. However, as expected, M4C4 implementations had lower frequencies than M2C2 ones.

4.2.1 Experiments on Microbenchmarks. We designed two sets of automatically generated microbenchmarks to explore the impact of two code features on the performance of the feed-forward design model. The first feature is the access pattern of the load instructions, and the second is the control flow divergence in the main loop of the single work-item kernel. The first set of microbenchmarks targets memory access patterns. We use two kernels with no control flow divergence across main loop iterations, *eight* load instructions from global memory, and *eighty* arithmetic operations (i.e., the arithmetic intensity of 10). These two kernels only differ in the behavior of their load instructions. The first benchmark in this set, called *M AI10 R*, has load instructions with regular memory access patterns, and the second one, called *M AI10 IR*, has load instructions with irregular memory access patterns.

The second set of microbenchmarks targets the presence of control flow divergence within the main loop in a single work-item kernel. To this end, we designed two kernels with the same characteristics as the first set; however, we added an inner loop alongside a conditional statement inside the inner loop to add divergence to the first set of microbenchmarks. To further show the impact of the feed-forward design model on kernels with DLCD, we added a reduction operation inside the inner loop to add data dependencies across different iterations of the inner loop. We also decreased the number of arithmetic operations inside the kernels to increase the control flow divergence impact on the execution time of the kernel. The first microbenchmark, called *M AI6 for-if R*, has load instructions with regular, and another, called *M AI6 for-if IR*, has load instructions with irregular memory access patterns.

Table 3 shows the impact of the feed-forward design model with two memory kernels and two compute kernels on these sets of microbenchmarks. The results suggest that kernels containing load instructions with regular memory access patterns would often benefit more from the feed-forward design model. This comparison indicates that higher memory contention for irregular load instructions in the feed-forward design with multiple memory kernels leads to lower memory bandwidth utilization. Moreover, the feed-forward design model improves performance for kernels with control flow divergence and DLCD. The baseline version of these microbenchmarks has a low memory bandwidth utilization due to a more complex control flow and the presence of DLCD. As we recall from Section 3, using the feed-forward design model removes the DLCD from the memory kernel and improves the performance of the design on the FPGA. Furthermore, having multiple memory and compute kernels increases the concurrency among memory instructions. These changes result in significantly higher memory bandwidth utilization and better execution time.

5 RELATED WORK

Zohouri et al. [24] and Nourian et al. [14] studied several optimization techniques on applications from Rodinia benchmark suite and finite automata traversal, respectively, while focusing on performance and power consumption. Their analysis confirms the performance portability gap while porting a GPU-optimized OpenCL implementation to FPGA and indicates a need for FPGA-specific optimizations. Krommydas et al. [12] performed a similar analysis on several OpenCL kernels investigating pipeline parallelism on single work-item kernels, manual and compiler vectorization, static coalescing, pipeline replication, and inter-kernel channels. Hassan et al. [10] explored FPGA-specific optimizations in their work. Their benchmarks were chosen from irregular OpenCL applications suffering from unpredictable control flows, irregular memory accesses, and work imbalance among work-items. Their work exploits parallelism at different levels, floating-point optimizations, and data movement overhead across the memory hierarchy.

There have been several efforts targeting the decoupling of memory accesses and computations on FPGA kernels [16] [7] [17]. Purkayastha et al. [16] explored the impact of decoupling part or all of the memory accesses from the computation. Their solution achieves more than 2× speedup for a subset of the considered OpenCL applications. While their automated process to detect decouplable variables can ease the transformation for programmers, following the steps we proposed, programmers can decouple memory instructions from computation in more cases, resulting in performance improvement in a larger set of applications. Charitopoulos et al. [7] explored decoupling memory accesses from execution on FPGA devices. They showed that using separate fetch units that access memory for load/store operations can result in performance improvements of 2× compared to the optimized HLS for three target applications in simulation. Unfortunately, no on-board comparison for HLS code is provided.

Several previous works have tried to leverage pipes to improve the performance of their implementations by increasing the concurrency among the instructions. Purkayastha et al. [15] applied pipes only to global variables whose access pattern and control flow is known at compile time, restricting the memory accesses that are optimized. Given this restriction, they don't require a complex data transformation that handles loops and complex control flows inside

Table 3: M2C2 Speedup and resource utilization comparison for microbenchmarks

Benchmark	Baseline Execution Time (ms)	Speedup	Logic Utilization Baseline	Logic Utilization M2C2	BRAM Baseline	BRAM M2C2
M AI10 R	232	1.55x	17.69	25.39	612	892
M AI10 IR	440	1.00x	17.91	24.60	817	1215
M AI6 for-if R	10780	1.90x	18.13	25.39	664	892
M AI6 for-if IR	11500	1.84x	17.71	24.35	799	1161

the memory kernel. Sanaullah et al. [18] proposed an empirically guided optimization framework for OpenCL to FPGA. They leveraged channels to convert a single kernel implementation to multiple kernels, each working as a separate processing element. In their work, they used channels for data communication among kernels. However, their analysis indicates that using channels in their implementation can result in lower performance, mainly due to the data dependency among kernels and the need for synchronization. Wang et al. [22] leveraged using task kernels and channels to design a multi-kernel approach to reduce the lock overhead. Mainly their work was focused on data partitioning workload. Yang et al. [19] used channels to implement a molecular dynamic application.

In a more recent work, Liu et al. [13] proposed a compiler scheme to optimize different types of multi-kernel workloads. They introduced a novel algorithm to find an efficient implementation for each kernel to balance the throughput of a multi-kernel design. Additionally, they explored bitstream splitting to separate multiple kernels into more than one bitstream to enable more optimizations for individual kernels.

6 CONCLUSION

In summary, in this work we explored using the feed-forward design model to improve the performance of OpenCL kernels on FPGA. We proposed a code transformation to apply this model to existing OpenCL kernels. We evaluated the benefits and limitations of our method, as well as its applicability in the presence of different classes of loop-carried dependencies.

Our results emphasize the importance of LCDs (especially on global memory) and their impact on performance. In particular, we showed that avoiding MLCDs results in efficient pipelined implementations. Our study shows that OpenCL-to-FPGA compilers take a conservative approach when identifying MLCDs within kernels, leading to serial loop execution even for loops that could be pipelined. Using our code transformation, programmers can improve the performance of their FPGA applications in two ways. First, they can remove false loop-carried dependencies, resulting in a speedup up to 64× over the baseline code. Second, they can leverage this transformation to exploit spatial parallelism on the FPGA by increasing the number of memory and compute kernels. Our experimental results show that this can further improve the performance of applications by up to 93% at the cost of a modest resource utilization overhead. The proposed code transformation can be integrated in a source-to-source compiler. However, to allow the applicability of the method on a broader set of kernels, it is important to incorporate code annotations to identify unsafe MLCDs that can be disregarded or removed at run-time (but cannot be handled at compile time).

ACKNOWLEDGEMENT

The research is supported by NSF award CNS-1812727.

REFERENCES

[1] 2021. Intel FPGA SDK for OpenCL Best Practice Guide. https://www.intel.com/content/www/us/en/docs/programmable/683521/21-4/introduction-to-pro-edition-best-practices.html

[2] 2022. Amazon EC2 F1. https://aws.amazon.com/ec2/instance-types/f1/ ID: doc:605062b38f08a4e6d25943f1; M1: Web Page.

[3] 2022. Microsoft Azure FPGA. https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service

[4] 2022. OpenCL. https://www.khronos.org/opencl/

[5] 2022. OpenCL Memory Model. (2022). https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc

[6] 2022. Top 500 list. https://www.top500.org/ ID: doc:605059a18f08377acf72d2ac; M1: Web Page.

[7] George Charitopoulos, Charalampos Vatsolakis, Grigorios Chrysos, and Dionisios N. Pnevmatikatos. 2018. A Decoupled Access-Execute Architecture for Reconfigurable Accelerators. In *Proceedings of the 15th ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '18)*. Association for Computing Machinery, New York, NY, USA, 244–247. https://doi.org/10.1145/3203217.3203267

[8] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *- 2013 IEEE International Symposium on Workload Characterization (IISWC)*. 185–195. https://doi.org/10.1109/IISWC.2013.6704684 ID: 1.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *- 2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797 ID: 1.

[10] M. W. Hassan, A. E. Helal, P. M. Athanas, W. Feng, and Y. Y. Hanafy. 2018. Exploring FPGA-specific Optimizations for Irregular OpenCL Applications. In *- 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–8. https://doi.org/10.1109/RECONFIG.2018.8641699 ID: 1.

[11] Intel. 2022. Intel FPGA SDK for OpenCL Pro Edition: Programming Guide. https://tinyurl.com/2p87v84n

[12] K. Krommydas, A. E. Helal, A. Verma, and W. Feng. 2016. Bridging the Performance-Programmability Gap for FPGAs via OpenCL: A Case Study with OpenDwarfs. In *- 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 198. https://doi.org/10.1109/FCCM.2016.56 ID: 1.

[13] Ji Liu, Abdullah-Al Kafi, Xipeng Shen, and Huiyang Zhou. 2020. *MKPipe: A Compiler Framework for Optimizing Multi-Kernel Workloads in OpenCL for FPGA*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3392717.3392757

[14] Marziyeh Nourian, Mostafa Eghbali Zarch, and Michela Becchi. 2020. Optimizing Complex OpenCL Code for FPGA: A Case Study on Finite Automata Traversal. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. 518–527. https://doi.org/10.1109/ICPADS51040.2020.00073

[15] Arnab A Purkayastha, Sai Raghavendran, Jhanani Thiagarajan, and Hamed Tabkhi. 2019. Exploring the Efficiency of OpenCL Pipe for Hiding Memory Latency on Cloud FPGAs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2019.8916236

[16] Arnab A. Purkayastha, Samuel Rogers, Suhas A. Shiddibhavi, and Hamed Tabkhi. 2020. LLVM-based automation of memory decoupling for OpenCL applications on FPGAs. *Microprocessors and Microsystems* 72, 102909. https://doi.org/10.1016/j.micpro.2019.102909

[17] Arnab A Purkayastha and Hamed Tabkhi. 2021. Design Study on Impact of Memory Access Parallelism for Cloud FPGAs. In *2021 IEEE 34th International System-on-Chip Conference (SOCC)*. 254–259. https://doi.org/10.1109/SOCC52499.2021.9739477

[18] A. Sanaullah, R. Patel, and M. Herbordt. 2018. An Empirically Guided Optimization Framework for FPGA OpenCL. In *2018 International Conference on Field-Programmable Technology (FPT)*. 46–53. https://doi.org/10.1109/FPT.2018.00018

[19] Zeke Wang, Bingsheng He, and Wei Zhang. 2015. A study of data partitioning on OpenCL-based FPGAs. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2015.7293941

[20] Zeke Wang, Bingsheng He, Wei Zhang, and Shunning Jiang. 2016. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 114–125. https://doi.org/10.1109/HPCA.2016.7446058

[21] Xilinx. 2022. AXI Burst Transfers On Vitis. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/AXI-Burst-Transfers

[22] Chen Yang, Jiayi Sheng, Rushi Patel, Ahmed Sanaullah, Vipin Sachdeva, and Martin C. Herbordt. 2017. OpenCL for HPC with FPGAs: Case study in molecular electrostatics. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–8. https://doi.org/10.1109/HPEC.2017.8091078

[23] Mostafa Eghbali Zarch, Reece Neff, and Michela Becchi. 2021. Exploring Thread Coarsening on FPGA. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 436–441. https://doi.org/10.1109/HiPC53243.2021.00062

[24] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. 2016. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *- SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 409–420. https://doi.org/10.1109/SC.2016.34 ID: 1.

[25] Hamid Reza Zohouri and Satoshi Matsuoka. 2019. The Memory Controller Wall: Benchmarking the Intel FPGA SDK for OpenCL Memory Interface. 11–18. https://doi.org/10.1109/H2RC49586.2019.00007