Go-beyond OpenCL or Not: Optimizing OpenCL Kernel on FPGAs

ABSTRACT

For a broad range of applications, FPGA architecture allows a massive bit-level parallelism to be exploited to achieve high performance and energy efficiency. Therefore, the parallel nature of FPGA inherently matches the parallel programming language, e.g., OpenCL. For example, FPGA vendors such as Intel have released OpenCL SDK for FPGAs so that FPGA programmer can now leverage OpenCL to program FPGA, instead of tedious, time-consuming and error-prone register transfer level (RTL). Accordingly, a wealth of literature have explored multiple optimization methods (e.g., loop unrolling) to accelerate conventional OpenCL (i.e., NDRange kernel) on FPGAs. However, conventional OpenCL cannot always represent FPGA architecture in an efficient manner, since OpenCL is originally designed for GPUs. Accordingly, Intel OpenCL SDK provides two go-beyond OpenCL features (single work-item kernel and OpenCL channel) to better utilize FPGA resources. However, how to leverage go-beyond OpenCL features is still unclear to the conventional OpenCL programmer. In this paper, we reduce the problem (go-beyond OpenCL features or not) to the problem of bridging the gap between three typical OpenCL patterns and four execution models (aware of go-beyond OpenCL features). Essentially, the OpenCL programmer can easily determine whether to employ go-beyond OpenCL features, based on whether the interested OpenCL code contains any OpenCL patterns or not. Experimental result shows that, 1)choosing the suitable execution model can lead to up to three orders of magnitude performance speedup over the most unsuitable execution model; 2)we can determine the right execution model via three OpenCL patterns plus an analytical model. Therefore, we argue the execution model should be considered as the first step to decide, as it decides the potential of other optimization combinations can reach.

ACM Reference Format:
. 2019. Go-beyond OpenCL or Not: Optimizing OpenCL Kernel on FPGAs. In *Proceedings of ACM conference (DAC'19).* ACM, New York, NY, USA, 7 pages.

1 INTRODUCTION

The parallel programming languages, e.g., OpenCL, are based on BSP (Bulk Synchronous Parallel) model and are widely adopted by GPUs, which features a massive number of cores for the computation task. Due to the fact that FPGA is embarrassingly parallel, it is a natural attempt to use the existing parallel programming languages to program FPGA. For instance, Intel has launched an end-to-end OpenCL SDK to program FPGAs [4] such that the complexity from direct hardware programming (e.g., Verilog) is abstracted away. OpenCL delivers increased programmability and lower learning curve with respect to RTL. Accordingly, plenty of research work [5, 9, 12**?**] are aim at optimizing the *conventional OpenCL* kernels on FPGAs, where the conventional OpenCL kernel is the NDRange kernel which employs a multi-thread approach to accelerate the computing task.

Despite the preliminary success from directly adopting conventional OpenCL programs on FPGAs, we still identify a significant gap from achieving near-optimal performance on FPGAs due to the factor that the conventional OpenCL kernel cannot always represent FPGA resources in an efficient way. In particular, it is not able to fully utilize two FPGA-aware features, referred as go-beyond OpenCL features for FPGAs.

F1: Single Work-Item (SWI) Kernel. Intel OpenCL SDK provides a new programming model for FPGAs: *single work-item kernel.* It relies on the off-line compiler to explore the pipelined parallelism at the compilation time. Therefore, it only contains only one work item to do the computation, indicating significantly different from data-parallel programming model of conventional OpenCL kernel.

F2: Direct Kernel-to-Kernel Communication. The communication between two kernels can be done via a FIFO (i.e., OpenCL channel) on FPGAs, not via memory subsystem like GPUs. It can potentially reduce memory traffic and achieve more parallelism by concurrently executing multiple OpenCL kernels on FPGAs.

Unfortunately, there is no comprehensive study to guide the OpenCL programmer how to optimize the OpenCL kernels on FPGAs in the context of go-beyond OpenCL features, so the burden of choosing the right OpenCL optimization methods (e.g., go-beyond OpenCL features) still falls on OpenCL programmers without any rule-of-thumb guidelines. Therefore, we ask the following question:

Can we explicitly determine whether to use go-beyond OpenCL or not for the conventional OpenCL programmer?

In this paper, we make the following contributions to answer the question by connecting three typical OpenCL patterns and four high-level execution models. As such, the OpenCL programmer can directly decide whether to use go-beyond OpenCL or not (in terms of execution model), based on three well known OpenCL patterns.

C1: Three Typical OpenCL Patterns on FPGAs (Section 4). We identify three typical patterns from OpenCL kernels on GPUs: atomic operation, multi-pass scheme and kernel-to-kernel communication, which are worth to be heavily revisited on FPGAs, since their performance can be significantly improved with the help of go-beyond-OpenCL features enabled by FPGAs. Since these patterns are well understood by the conventional OpenCL programmers, they can serve as a really nice starting point from OpenCL side.

C2: Four High-level Execution Models on FPGAs (Section 5). We identify four high-level OpenCL execution models: NDRange,

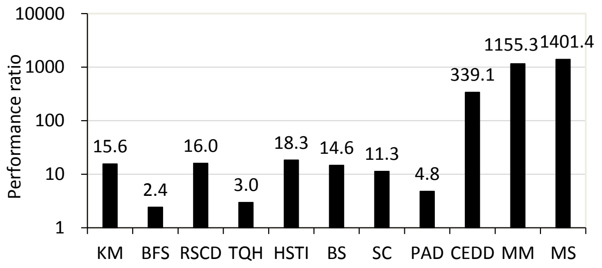


Figure 1: Performance difference between the most suitable and unsuitable execution models.

SWI, NDRange+Channel and SWI+Channel, which are allowed by the OpenCL SDK for FPGAs. Choosing the right execution model actually determines the performance upper bound of the further optimization process. Figure 1 shows that the performance ratio of the most suitable to the most unsuitable execution models, where the best optimization combination has already been employed for each execution model. The exact execution models used by each application are not important here. We refer the interested reader to Section 7 for more details. We can observe that the performance difference can achieve three orders of magnitude. It shows the importance of choosing the right execution model, which should serve as the first and the most important step to determine for optimizing OpenCL kernel on FPGAs.

C3. Intensive Experiment to Verify the Connection (Sections 6, 7). We implement and optimize 11 OpenCL applications on a Terasic's DE5A-Net Arria 10 FPGA board. We make our best effort[1] to explore optimization combinations for all the four execution models for each OpenCL application. Based on the experimental result, we can observe that the real execution model (which has the best performance) can match the predicted execution model, which is determined by three OpenCL patterns.

This paper can serve as a complementary guideline for the existing optimization efforts [5, 9?]. Besides, we argue that determining the execution model should be considered as the first step to decide, as it determines the potential of further optimization combinations can reach.

Future Directions. To the best of our knowledge, this paper is the first attempt to guide the conventional OpenCL programmer to leverage go-beyond OpenCL or not, and we believe it opens a few exciting research directions so that it will attract more and more people to devote to FPGA acceleration. One future direction is to provide an end-to-end performance analytical model to predict the performance of go-beyond OpenCL kernels. Until now, the literature only cover the analytical model for the NDRange kernel [5, 9]. Another interesting future direction is to design an automatic optimization process for each execution model with the help of performance analytical model. As such, more and more OpenCL programmer without any FPGA background can take advantage of computing power of FPGAs.

[1]Based on our five years' experience on OpenCL-based FPGAs, we are pretty sure that our implementation have typically reached the near-optimal performance for each execution model. We will make all the related source code open-sourced.

Limitations. This work has two limitations. First, we manually try plenty of optimization combinations for each execution model. Second, for a few applications, we cannot successfully implement all the four execution models. In particular, the result is not correct.

2 BACKGROUND

2.1 Traditional OpenCL Programming Model

Intel provides the OpenCL SDK to abstract the hardware complexities from the traditional FPGA design. Different from viewing FPGAs as pure hardware resources like LUTs, DSP blocks and memory blocks, the OpenCL SDK regards the FPGA as a large-scale parallel architecture. Typically, OpenCL conceptually divides a problem into equally loaded partitions, in terms of *work-item* which represents the basic unit of execution. A set of work-items are organized into a *work-group* and multiple work-groups are combined to form a three-dimensional index space called *NDRange*. *NDRange kernel* is the default OpenCL kernel model which achieves the pipelined parallelism on FPGAs. We introduce two main optimization methods, which are widely adopted by normal and go-beyond OpenCL kernels. We refer the reader to the related work [9, 12] for more optimization methods on FPGAs.

Multiple Compute Units (CUs). Suppose the FPGA has sufficient hardware resource, the kernel pipeline can be replicated to generate multiple CUs to achieve more computing parallelism.

Loop Unrolling (UL). If the kernel pipeline contains a large number of loop iterations, the loop iterations could potentially be the critical path due to its unbalance. Unrolling the loop can increase the pipeline throughput at the expense of more hardware resources consumption only for the loop.

2.2 Go-beyond OpenCL Features on FPGAs

Single Work-Item (SWI) Kernel. In addition to the standard NDRange kernel model, Intel OpenCL SDK also supports a SWI kernel, which follows a sequential model like C programming, thus executes the kernel in only one CU that contains only one work-item. In the SWI kernel, the parallelism is implicit, and the OpenCL SDK will instead determine pipelined parallelism at the compilation time based on the dependency. The SWI kernel more closely matches the traditional deep-pipelined nature of FPGAs.

OpenCL Channel. Intel OpenCL SDK provides the *OpenCL channel* feature, which can be used to pass data between two OpenCL kernels (typically SWI) and synchronize the kernels with high efficiency and low latency. The channel allows the kernels to communicate directly with each other via on-chip FIFO buffers.

3 DESIGN OVERVIEW

In this section, we illustrate how the next three sections organized. In Section 4, we identify three typical OpenCL patterns, which are well understood by the conventional OpenCL programmer. In Section 5, we generally analyze the characteristics of four OpenCL execution models, which takes into account the go-beyond OpenCL features enabled on FPGAs. In Section 6, we explicitly bridge the gap between OpenCL patterns and execution models on FPGAs. Choosing the right execution model determines the performance upper bound of the interested OpenCL application can reach.

4 THREE OPENCL PATTERNS

When migrating conventional OpenCL programs, which is originally designed for GPUs, we identify three OpenCL patterns which can be significantly improved: atomic operation, multi-pass scheme and communication approach. In the following, we analyze the issue and potential optimization direction of each pattern on FPGAs.

4.1 Atomic Operation (AO)

For NDRange OpenCL, the atomic operation is required to guarantee the consistence of OpenCL kernel, when multiple work items try to update the same memory location [6?]. It provides the improved programmability that GPU programmers nowadays leverage, since the hardware for the atomic operation has greatly improved in recent GPU generations (e.g., NVIDIA Kepler, Maxwell, Pascal). Take as an example the simplified histogram, whose main body is illustrated in Listing 1.

```
int tid = get_global_id(0);//global work item
int d   = data[tid];        //fetch the data from memory
int h_d = hash(d);          //compute the hash index
atomic_add(&hist[h_d],1);   //atomically add to hist
```

Listing 1: Atomic-based histogram

Issues on FPGAs. We have identified three issues regarding atomic operation on FPGAs. First, the mechanism of atomic operation is relatively complex, so massive FPGA resources are required to implement it, leading to a noticeable resource overhead on FPGAs. Second, with atomic operation in our OpenCL kernel, all the memory transactions, including both atomic and normal memory transactions, have to enter the atomic module to guarantee the correct execution, as atomic and normal memory transactions from OpenCL kernel can reference the same memory location. Therefore, each memory transaction has a longer memory access latency, leading to potentially lower memory bandwidth. Third, the frequency of OpenCL kernel with atomic operation is slightly lower than that without atomic operation, leading to lower performance. We conclude that the atomic operation is not able to fit well on FPGAs.

Potential on FPGAs. In order to achieve good performance of OpenCL kernel on FPGAs, one potential direction is to get rid of atomic operation. Fortunately, Intel OpenCL SDK [4] supports a new execution model: SWI kernel. Essentially, it contains only one work item during the kernel execution, so there is no conflict from multiple work items, indicating that no atomic operation is required. Instead, it exploits the pipelined parallelism, not the thread-level parallelism which is widely exploited by GPUs. The previous histogram application is converted into the SWI kernel as shown in Listing 2.

```
for (t=0; t<size; t++) { //for loop instead of work item
  int d   = data[t];      //fetch the data from memory
  int h_d = hash(d);      //compute the hash index
  hist[h_d] += 1;         //accumulate to hist
}
```

Listing 2: SWI-based histogram

4.2 Multi-Pass Scheme (MPS)

The multi-pass scheme is widely adopted to leverage a massive amount of cores in CPUs/GPUs to accelerate parallel algorithm, e.g., database operators [3, 10, 11]. Besides, it can also improve the cache locality on GPUs, e.g., gather/scatter [2], so as to increase the overall parallelism at the cost of more memory traffic. Take the simplified parallel prefix sum [1] as an example, we perform the prefix sum on the input array *in* of size *N* and store the output in the array *out*, as illustrated in Listing 3. In Step 1, B work groups (WGs) executes concurrently, each WG performs the prefix sum (kernel *prefix_sum_wg*) on its own part of data of starting address *in[N*b/B]* and of length *N/B*. Meanwhile, each WG stores its local sum to *local_sum*. In Step 2, we employ one WG to compute the prefix sum on *local_sum* and store to *pre_bsum*. In Step 3, each WG adds the scalar value *pre_bsum* to the vector and then produces the final prefix sum. In summary, the total memory traffic is *4N+4B*.

```
//Step 1: each WG computes the prefix sum on its own data
#pragma parallel in B work groups
for (b = 0, b < B, b++) {
  local_sum[b] = prefix_sum_wg(out[N*b/B],in[N*b/B],N/B);
}

//Step 2: one WG computes the prefix sum on "local_sum"
prefix_sum_wg(pre_bsum, local_sum, B);

//Step 3: out@[b*N/B] += pre_bsum[b]
#pragma parallel in B work groups
for (b = 0, b < B, b++) {
  vec_add(out[b*N/B], out[b*N/B], pre_bsum[b], N/B);
}
```

Listing 3: MPS-based prefix sum

Issues on FPGAs. We observe that the inevitable requirement for the widespread adoption of multi-pass scheme is powerful memory subsystem, as it always requires multiple times more memory traffic to realize the algorithm. It works pretty well on GPUs, whose memory bandwidth is almost an order of magnitude larger than the same-generation FPGA. For example, the memory bandwidth of Nvidia Tesla P100 GPU is able to reach 732GB/s, while the memory bandwidth of our tested FPGA board is 18GB/s. Therefore, it suffers from its severe performance issue, even though the FPGA can provide massive thread-level parallelism.

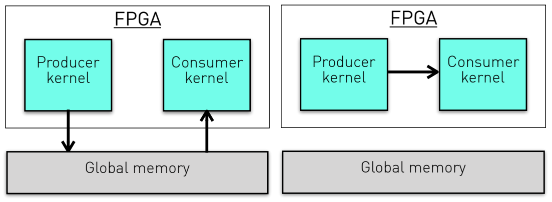
Potential on FPGAs. Since Intel OpenCL SDK for FPGAs supports the new single work-item kernel, the OpenCL kernel is allowed to be implemented in a single-pass approach, as illustrated in Listing 4. Therefore, the memory traffic is reduced to a minimum level (i.e., accessing the array *in/out* once, not twice) just as the single-threaded implementation running on a single CPU core.

```
out[0] = 0;
for (t=1; t<n; t++) { //for loop instead of work item
  out[t] = out[t-1] + in[t]; //dependency in out
}
```

Listing 4: SWI-based prefix sum

4.3 Kernel-to-Kernel Communication (KKC)

On GPUs, the typical communication approach between producer/consumer kernels is done via global memory. In particular, after the producer kernel writes the data to the global memory, the consumer kernel reads the data from global memory, as illustrated in Figure 2a. It is a common communication approach to exchange information among Stream Processors (SPs) in GPUs, as there is no physical communication path between any two SPs. This communication approach works well on GPUs since GPUs can execute each kernel relatively fast due to its massive thread-level parallelism



(a) Without channel **(b) With channel**

Figure 2: Kernel-to-kernel communication.

	NDRange	SWI	NDRange+Channel	SWI+Channel
Programmability	Moderate	High	Low	Low
Computing Parallelism	Moderate	Low	High	High
Memory Traffic	High	Low	Moderate	Low

Table 1: General comparison of four execution models

and powerful memory subsystem. Suppose each OpenCL kernel is fully optimized, the overall performance of producer/consumer kernels is maximized. This communication approaches widely used in accelerating various applications, e.g., database [3, 11].

Issue on FPGAs. The memory bandwidth is relatively low on FPGAs, e.g., 18GB/s on our FPGA board, the communication via external memory can be expensive, compared with that on GPUs.

Potential on FPGAs. We can use OpenCL channel via which the producer kernel can directly send data to the consumer kernel at the register level (i.e., FIFO), without any memory traffic involved between the producer and consumer kernels, as illustrated in Figure 2b. Besides, both kernels, each of which has the dedicated hardware resources to implement, execute concurrently, so not only intra-kernel parallelism (i.e., pipelined) is exploited, but also inter-kernel parallelism (i.e., multiple kernels).

5 FOUR OPENCL EXECUTION MODELS

Enabled by two go-beyond features of OpenCL on FPGAs, we identify four execution models in the context of OpenCL kernel on FPGAs. In this section, we generally analyze each execution model, in terms of programmability, computing parallelism and memory traffic. The comparison is highlighted in Table 1.

5.1 NDRange

The NDRange execution model directly employs the NDRange kernel, which is the conventional OpenCL programming mode and which is widely adopted in GPU programming. It explores a massive thread-level parallelism. The optimization methods which are used by GPUs can also apply to FPGAs, e.g., shared memory.

Programmability. The OpenCL programmer needs to be aware of the activity of each thread to guarantee the correct execution. Therefore, it is much more difficult than the traditional sequential programming language, e.g., C. However, there is plenty of tutorial on how to program NDRange kernel, especially for GPUs, so the programmability of NDRange execution model is moderate.

Computing Parallelism. NDRange kernel relies on massive amount of work items to explore the parallelism from the specialized hardware generated from the NDRange kernel. Therefore, we conclude that it can achieve a moderate computing parallelism.

Memory Traffic. NDRange kernel may employ the multi-pass scheme to implement the parallel algorithm, leading to potentially multiple times more memory traffic. We conclude that the requirement of memory traffic is high.

AO	MPS	KKC	Direct prediction	Potential evolution
N	N	N	NDRange	NDRange
Y	N	N	SWI	SWI+Channel
N	Y	N	SWI	SWI+Channel
Y	Y	N	SWI	SWI+Channel
N	N	Y	NDRange+Channel	NDRange+Channel
Y	N	Y	SWI+Channel	SWI+Channel
N	Y	Y	SWI+Channel	SWI+Channel
Y	Y	Y	SWI+Channel	SWI+Channel

Table 2: Prediction rule

5.2 Single Work-item (SWI)

The SWI execution model directly employs the SWI kernel, which follows a sequential programming model.

Programmability. Programming with SWI kernel is just like C programming. Therefore, it is relatively easy to program. We conclude that the programmability of SWI kernel is high.

Computing Parallelism. SWI kernel relies on the off-line compiler to explore the parallelism. In order to guarantee the consistence, the compiler is always conservative, leading to a low computing parallelism.

Memory Traffic. SWI kernel can leverage a one-pass scheme (less memory traffic) to implement the parallel algorithm, instead of the multi-pass scheme (more memory traffic). We conclude that the SWI kernel requires a low memory traffic.

5.3 NDRange + Channel

The NDRange + Channel execution model employs an OpenCL channel to connect two NDRange kernels such that the data communication between two kernels can be done via on-chip FIFO, rather than global memory.

Programmability. Besides the programming difficulty from NDRange kernel, we still need to be aware of the constraint from OpenCL Channel. In particular, the producer kernel has to produce the exact data flow which the consumer kernel wants. To make things worse, work items can execute out-of-order due to the different workload of each work item, leading to unexpected data flow from NDRange kernel. We conclude that the programmability of NDRange+Channel execution model is low.

Computing Parallelism. Besides the high computing parallelism from NDRange kernel, OpenCL channel allows producer/consumer kernels to execute concurrently, leading to a high computing parallelism.

Memory Traffic. NDRange kernel requires a high memory traffic. The OpenCL channel can potentially reduce the memory traffic to a certain extent. We conclude that the NDRange + Channel execution model requires a moderate memory traffic.

5.4 SWI + Channel

The SWI + Channel execution model employs OpenCL channels to connect multiple SWI kernels such that multiple SWI kernels can work together to implement a parallel algorithm. The side effect of OpenCL channel is to relax the dependency of SWI kernel.

Programmability. SWI kernel has a high programmability, since its programming model is sequential. However, the OpenCL Channel incurs some constraint to guarantee the correct design. We conclude that its programmability is moderate.

Computing Parallelism. Even though the SWI kernel can achieve low computing parallelism, OpenCL channel allows producer/consumer kernels to execute concurrently to significantly increase the computing parallelism, leading to a high computing parallelism.

Memory Traffic. SWI kernel requires a low memory traffic. The OpenCL channel might further reduce the memory traffic, so we conclude that its requirement of memory traffic is low.

6 BRIDGE THE GAP BETWEEN PATTERNS AND EXECUTION MODELS

In this section, we explicitly bridge the gap between three OpenCL patterns and four execution models. Typically, we can directly predict the right execution model based on three patterns (Subsection 6.1). However, there is an exception about SWI, which can only achieve low computing parallelism (Table 1), indicating that the computing potential of FPGA may not be fully harvested. SWI can evolve to SWI+Channel for higher computing parallelism. Accordingly, we explicitly determine whether SWI needs to evolve or not(Subsection 6.2).

6.1 Direct Prediction

Based on whether the targeted OpenCL application has any patterns (i.e., the first three columns of Table 2), we can directly predict the execution model, as illustrated in the "Direct prediction" of Table 2. Typically, the OpenCL application with AO and MPS can benefit from SWI kernel, while KKC can benefit from OpenCL channel, as discussed in Section 4.

6.2 Potential Evolution of SWI

In this subsection, we determine whether SWI should be evolved to SWI+Channel for higher computing parallelism, as shown in the column "Potential evolution" of Table 2. The evolution should satisfy two conditions.

S1: Enough FPGA resources. SWI+Channel instantiates more than one SWI kernels connected by Channel and then requires significantly more FPGA resources to implement.

S2: SWI kernel is compute-bound. The evolution happens only when the SWI kernel is compute-bound. We present a simplified analytical model to predict the targeted SWI kernel is compute-bound or memory-bound. In particular, the computing time T_{comp} is larger than memory time T_{mem}, as shown in Equation 1.

$$T_{comp} > T_{mem} \tag{1}$$

The computing time T_{comp} is evaluated as shown in Equation 2.

$$T_{comp} = \frac{LTR * II}{\#Freq}, \tag{2}$$

where the first part $LTR * II$ is the estimated number of cycles, equal to be the loop trip count LTC multiplied by the initiation interval II. II refers to the number of cycles the SWI kernel needs to proceed each iteration ($II \geq 1$), determined by the OpenCL SDK. The second part $\#Freq$ is the frequency of the SWI kernel obtained from synthesis report.

The memory time T_{mem} is evaluated to be the number of memory traffic MT divided by the memory bandwidth $\#MB$, as illustrated in Equation 3, where $\#MB$ is determined by the calibrations [8], e.g., 18GB/s on our FPGA board.

$$T_{mem} = \frac{MT}{\#MB} \tag{3}$$

Application	Source	AO	MPS	KKC	Datasets
BFS	Chai [?]	Y	N	N	NY, NE, UT
RSCD		Y	N	Y	2000 iterations
TQH		Y	N	N	Basket
HSTI		Y	N	N	256 bins
BS		N	N	N	4*4
SC		Y	Y	N	50%
PAD		Y	Y	N	1000*999
CEDD		N	N	Y	Peppa, Maradona, Paw
KM	Rodinia [7]	N	N	N	25600 points, 8 features
MM	Intel demos	N	N	N	A: 2048*1024, B: 1024*1024
MS		N	N	N	640*800, 2000 iterations

Table 3: Experimental datasets with OpenCL patterns

7 EXPERIMENT

In this Section, we firstly present the experimental setup, secondly we conduct intensive experiment to validate our observations.

7.1 Experimental Setup

Hardware configuration. We conduct our experiments on a Terasic DE5a-Net board with an Altera Arria 10 GX FPGA and 8GB 2-bank DDR3 device memory. We design our kernels using Altera OpenCL SDK version 16.1. The FPGA board is connected to the host via a x8 PCI-e 3.0 interface.

Workloads. We carry out our experiment with 11 OpenCL applications, as shown in in Table 3. For example, eight applications are from Chai benchmark [?], which is highly optimized for GPUs.

Comparison Methodology. Our evaluations mainly validate two hypotheses. First, different execution model leads to a significant performance difference (Subsection 7.2). Second, we can use three OpenCL patterns to determine the right OpenCL execution model (Subsection 7.3).

7.2 Impact of Execution Model

In this subsection, we mainly validate that the right execution model leads to a significant performance speedup for the OpenCL application when mapping to FPGAs. In particular, we first try our best effort[2] to explore the optimization combinations for each execution model of each OpenCL application, second quantitatively compare the highest performance speedup that each execution model can achieve.

Exploring optimization combinations. For each OpenCL application, we explore a broad range of optimization combinations for each execution model such that its performance potential has been harvested. Due to the space limitation, we take the application RSCD as an example. Figure 3 shows for each combination model, the performance speedup of each optimization combination over the baseline, which is optimized GPU code. X-axis is the optimization combination under all the four execution models. We can make two observations. First, different execution model yields an order of magnitude performance difference. In particular, the best optimization combination under "NDRange+Channel" can achieve 43.4X, while 2.72X for the best combination under "SWI". Second, the same optimization method can yield significantly different performance

[2]Based on our more than five years' experience in programming FPGA with OpenCL, we are pretty sure that we have already found the near-optimal optimization combination for each execution model if available.

Application	Number of combinations				Maximum speedup			
	NDR	SWI	NDR+C	SWI+C	NDR	SWI	NDR+C	SWI+C
KM	30	14	12	17	147.67	8.76	136.41	28.96
BFS	17	1	1	4	1.85	2.95	1.22	2.96
RSCD	22	10	24	39	9.31	2.72	43.4	20.42
TQH	1	15		20	1	1.28		2.98
HSTI	9	36	9	28	2.67	1.06	9.49	19.71
BS	20	17	18	8	3.01	10.77	3.34	44.06
SC	15	34		9	1.52	4.51		17.13
PAD	11	10		14	1.15	1.6		4.8
CEDD	55	9	25	2	2.95	0.01	2.72	0.02
MM	25	3		1	837.12	0.73		0.72
MS	7	6		7	34.72	0.02		3.17

Table 4: Space exploration of optimization combinations

Application	AO	MPS	KKC	Real	Prediction
BFS	Y	N	N	SWI+Channel	SWI+Channel
RSCD	Y	N	Y	NDRange+Channel	SWI+Channel
TQH	Y	N	N	SWI+Channel	SWI+Channel
HSTI	Y	N	N	SWI+Channel	SWI+Channel
BS	N	N	N	SWI+Channel	NDRange
SC	Y	Y	N	SWI+Channel	SWI+Channel
PAD	Y	Y	N	SWI+Channel	SWI+Channel
CEDD	N	N	Y	NDRange	NDRange+Channel
KM	N	N	N	NDRange	NDRange
MM	N	N	N	NDRange	NDRange
MS	N	N	N	NDRange	NDRange

Table 5: Prediction of execution model

speedup under different execution models. For example, loop unrolling "UL50" under "SWI" can only achieve 1.26X performance speedup, while 43.4X under "NDRange+Channel".

Speedup comparison of execution models. Table 4 demonstrates for each execution model, the number of optimization combinations we have tried and maximum performance speedup over the baseline application which is directly imported from each source, e.g., Chai. We can make two observations. First, different execution model indeed results in significant performance speedup difference. Take HSTI as an example, the right execution model (SWI+Channel) can have 19.54X performance speedup over the baseline which is the optimized GPU code, while the inappropriate execution model (SWI) can only achieve 1.01X speedup. Second, different OpenCL application requires different execution model to achieve the best performance. For example, the best execution model for BFS is SWI+Channel, while NDRange is most suitable for MM. We conclude that it is critical to decide the right execution model when optimizing OpenCL application on FPGAs.

7.3 Prediction of Right Execution Model

In this subsection, we mainly validate that for each OpenCL application, the prediction of execution model matches the empirical result. Essentially, we can predict the right execution model, which can produce best performance, based on three OpenCL patterns (Section 6). Table 5 shows the predicted execution model, as well as the real execution model which can achieve the best performance in our real experiment. We can observe that the prediction of execution model almost matches the real experimental result.

8 CONCLUSION

Despite the preliminary success of programming FPGA with OpenCL, direct adoption of OpenCL cannot fully harvest the performance potential of FPGAs, since the architecture of FPGAs is significantly different from that of GPUs, for which OpenCL is originally designed. In particular, the OpenCL programmer should be aware of two go-beyond OpenCL features to explore the performance potential of FPGAs. However, the OpenCL programmer still requires an end-to-end guideline about how to leverage these two features. In this paper, we bridge the gap between three typical OpenCL patterns and four execution models (aware of go-beyond OpenCL features). Experimental result shows that the right execution model can yield three order of magnitude performance difference.

Acknowledgement. We thank Intel which has generously donated Terasic's DE5A-Net Arria 10 FPGA board for our research.

REFERENCES

[1] M. Harris. Parallel Prefix Sum (Scan) with CUDA. Technical report, Nvidia, https://www.mimuw.edu.pl/ ps209291/kgkp/slides/scan.pdf, 2007.
[2] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *SC*, 2007.
[3] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *TODS*, 2009.
[4] Intel. Intel SDK for OpenCL Optimization Guide. 2018.
[5] Y. Liang, S. Wang, and W. Zhang. Flexcl: A model of performance and power for opencl workloads on fpgas. *TC*, 2018.
[6] N. Ramanathan, J. Wickerson, F. Winterstein, and G. A. Constantinides. A case for work-stealing on fpgas with opencl atomics. In *FPGA*, 2016.
[7] J. M. D. T. J. W. S. S. L. S. Che, M. Boyer and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.
[8] Z. Wang, B. He, and W. Zhang. A study of data partitioning on opencl-based fpgas. In *FPL*, 2015.
[9] Z. Wang, B. He, W. Zhang, and S. Jiang. A performance analysis framework for optimizing opencl applications on fpgas. In *HPCA*, 2016.
[10] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang. Relational query processing on opencl-based fpgas. In *FPL*, 2016.
[11] S. Zhang, J. He, B. He, and M. Lu. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *VLDB*, 2013.
[12] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. Evaluating and optimizing opencl kernels for high performance computing with fpgas. In *SC*, 2016.

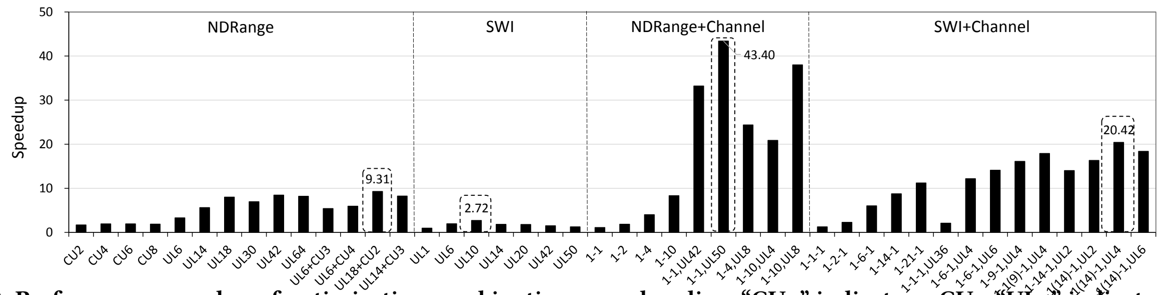


Figure 3: Performance speedup of optimization combinations over baseline. "CUx" indicates x CUs, "ULx" indicates the loop whose unroll factor x. x-y(z)-w indicates XXX.