Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs

Jiantong Jiang[1⋆] Zeke Wang[2⋆] Xue Liu[1*] Juan Gómez-Luna[2]
Nan Guan[3] Qingxu Deng[1] Wei Zhang[4] Onur Mutlu[2]

[1] Department of Computer Science and Engineering, Northeastern University, China
[2] ETH Zürich, Switzerland
[3] Department of Computing, Hong Kong Polytechnic University, Hong Kong
[4] Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong

ABSTRACT

FPGA vendors provide OpenCL software development kits for easier programmability, with the goal of replacing the time-consuming and error-prone register-transfer level (RTL) programming. Many studies explore optimization methods (e.g., loop unrolling, local memory) to accelerate OpenCL programs running on FPGAs. These programs typically follow the default OpenCL execution model, where a kernel deploys multiple work-items arranged into work-groups. However, the default execution model is *not* always a good fit for an application mapped to the FPGA architecture, which is very different from the multithreaded architecture of GPUs, for which OpenCL was originally designed.

In this work, we identify three other execution models that can better utilize the FPGA resources for the OpenCL applications that do not fit well into the default execution model. These three execution models are based on two OpenCL features devised for FPGA programming (namely, single work-item kernel and OpenCL channel). We observe that the selection of the right execution model determines the performance upper bound of a particular application, which can vary by two orders magnitude between the most suitable execution model and the most unsuitable one. However, there is no way to select the most suitable execution model other than empirically exploring the optimization space for the four of them, which can be prohibitive.

To help FPGA programmers identify the right execution model, we propose Boyi, a systematic framework that makes automatic decisions by analyzing OpenCL programming patterns in an application. After finding the right execution model with the help of Boyi, programmers can apply other conventional optimizations to reach the performance upper bound. Our experimental evaluation shows that Boyi can 1) accurately determine the right execution model, and 2) greatly reduce the exploration space of conventional optimization methods.

KEYWORDS

FPGA; OpenCL; Execution Model; Programmability

⋆: Equal contribution. *: Corresponding author.

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02…$15.00
https://doi.org/10.1145/3373087.3375313

ACM Reference Format:
Jiantong Jiang, Zeke Wang, Xue Liu, Juan Gómez-Luna, Nan Guan, Qingxu Deng, Wei Zhang, Onur Mutlu. 2020. Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs. In *2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3373087.3375313

1 INTRODUCTION

Bulk Synchronous Parallel (BSP) programming languages (e.g., OpenCL [26], CUDA [36]) are successfully employed to program compute devices that feature a large number of cores, such as GPUs. Since FPGAs are inherently parallel, OpenCL has naturally been proposed for FPGA programming, in an attempt to ease effort spent on programming hardware. OpenCL can deliver increased programmability and lower the learning curve for programming hardware, by abstracting away the complexity of direct hardware programming, i.e., register-transfer level (RTL) programming. Intel FPGA SDK [21] and Xilinx SDAccel [32] both support OpenCL. Several research proposals [12, 23, 25, 37, 41, 44, 46, 47, 50, 52, 54, 59, 61, 63] explore optimizations for *conventional OpenCL* kernels on FPGAs, where the conventional OpenCL kernel is the NDRange kernel [21], which employs multiple *work-items* (i.e., threads in OpenCL terminology) grouped into *work-groups* to express parallelism. The NDRange kernel is the default execution model in OpenCL. However, the performance of conventional NDRange kernels running on FPGAs is far from optimal in many cases, because this execution model *cannot* always represent the FPGA architecture in an efficient way.

In this work, we identify four different *execution models* for OpenCL on FPGA, which show different levels of suitability for different workloads mapped to the FPGA architecture:

NDRange (NDR). This is the default OpenCL execution model, widely employed in GPU programming. Different kernels in the same application communicate via the off-chip global memory.

Single Work-item (SWI). Unlike NDR, this execution model uses one single work-item, and relies on the offline compiler to extract pipeline parallelism at compile time. Same as NDR, different kernels communicate with each other via the global memory.

NDRange with Direct Kernel-to-kernel Communication (NDR+C). In this execution model, communication between two kernels can be directly done via a FIFO called the *OpenCL channel* [21], which reduces the off-chip memory traffic.

Single Work-item with Direct Kernel-to-kernel Communication (SWI+C). This execution model combines the SWI execution model with direct kernel-to-kernel communication via OpenCL channels.

Our **key observation** is that the selection of the execution model has a first-order effect on the performance upper bound of

an application running on an FPGA. Figure 1 shows the speedups obtained with the most suitable and the most unsuitable execution models for eight applications over the baseline, which employs NDR.[1] We observe that 1) the average performance difference between the most suitable and the most unsuitable execution model is 200.2×, and 2) the most suitable execution model differs between different applications (not shown in Figure 1). Thus, finding the right execution model for an application is clearly important. The first observation is inline with previous works that compare the default NDR execution model to the other execution models [2, 6, 24, 34, 49, 53, 56, 67].

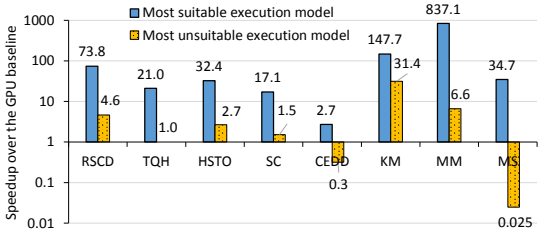


Figure 1: Speedup of the most suitable and the most unsuitable execution model over the baseline implementation for eight applications (see Section 6 for our methodology).

Unfortunately, programmers do not have an easy way to determine a priori what is the most suitable execution model for a particular application. Implementing four versions of an application based on the four execution models is challenging, time-consuming, and error-prone. For example, efficiently implementing an application with direct kernel-to-kernel communication requires a deep understanding of the FPGA architecture and design methodology [53]. In general, for each execution model, we can apply multiple conventional optimization methods with different configurations (e.g., loop unrolling factors). Given that synthesizing a single configuration takes a long time (e.g., a few hours), exploring the whole design space is prohibitively expensive.

Our **goal** is to alleviate the burden on FPGA programmers by systematically and automatically identifying the most suitable execution model for a given application. To this end, we present ***Boyi***,[2] a systematic framework that analyzes the characteristics of an OpenCL application (typically, a baseline implementation for GPUs) and determines the most suitable execution model on an FPGA. Boyi operates in two steps. First, Boyi identifies three patterns that are widely-used in OpenCL kernels for GPUs: 1) *Atomic Operation* (AO), 2) *Multi-Pass Scheme* (MPS), and 3) *Kernel-to-Kernel Communication* (KKC). Second, Boyi predicts the best-performing OpenCL execution model based on the presence or absence of those three OpenCL patterns. As a result, Boyi allows an OpenCL programmer to determine the most suitable execution model, without requiring FPGA background or any effort on the part of the programmer.

Our experimental evaluations show Boyi's effectiveness in 1) accurately predicting the most suitable OpenCL execution model for a given application, which determines the performance upper bound of the application on the FPGA, and 2) dramatically reducing the size of the exploration space of conventional optimizations. Boyi predicts accurately the most suitable execution model for 10 of 11 tested applications, and reduces the exploration space of conventional optimizations by 3.4×. For example, for SC [13], the number of optimization combinations decreases by a factor of 5.9, after Boyi determines the right execution model. We believe that Boyi can enable higher programmer productivity, as it eases the first and most important step of the optimization process of OpenCL programs, which is the selection of the most suitable execution model. Our work is thus complementary to other studies that focus on conventional optimizations [12, 23, 25, 37, 41, 44, 46, 47, 50, 52, 54, 59, 61, 63]. Boyi source code and the performance results of our experiments are freely available at [22].

The main contributions of this paper are as follows:

- We identify four OpenCL execution models, which have different levels of suitability for different workloads mapped to an FPGA. We connect these four execution models to three typical OpenCL patterns that are present in OpenCL kernels for GPUs.
- We present Boyi, a systematic framework that automatically determines the most suitable execution model for a given application, without requiring any effort from the OpenCL programmer.
- We evaluate the effectiveness of Boyi and show that Boyi 1) chooses the right execution model, which determines the performance upper bound that conventional optimizations can achieve, for 10 of 11 tested applications, and 2) reduces the exploration space of conventional optimizations by up to 5.9×.

2 BACKGROUND

In this section, we first provide an overview of the conventional OpenCL execution model. Second, we describe two major conventional optimizations for OpenCL kernels mapped to FPGAs. Third, we present two new OpenCL features on FPGAs.

Conventional OpenCL Programming Model. Conceptually, OpenCL [26] divides a problem into equal partitions, with the granularity of a *work-item*, which represents the basic unit of execution. Work-items are organized into *work-groups* and multiple work-groups compose a three-dimensional index space called *NDRange*. The NDRange dimensions are defined in host code. The *NDRange kernel* is the default OpenCL execution model on FPGAs.

Conventional Optimizations. Next, we describe two FPGA-specific optimizations that are typically applied to the NDRange kernel. We refer the reader to the related work [21, 54, 66] for other conventional optimizations.

Multiple Compute Units (CUs). If the FPGA has enough hardware resources, this optimization replicates the kernel pipeline to generate multiple compute units, enabling more parallelism.

Loop Unrolling (LU). In a kernel pipeline with many loop iterations, some loop iterations can be on the critical path due to load imbalance. Unrolling the loop increases the pipeline throughput at the expense of more hardware resources.

New OpenCL Features on FPGAs. Intel OpenCL SDK [21] includes two new features that can increase the performance upper bound of OpenCL applications mapped to an FPGA.

Single Work-Item (SWI) Kernel. The SWI kernel follows a sequential model like C programs. It deploys only one work-group with one single work-item. In the SWI kernel, parallelism is implicit. The

[1] On top of each execution model, we apply conventional optimizations (e.g., loop unrolling, local memory). We refer the reader to Section 6 for details about the baseline and our experimental methodology.

[2] Boyi is a hero in Chinese mythology who *assists* Yu the Great to control the Great Flood [1].

OpenCL SDK extracts pipelined parallelism at compile time based on dependency analyses. The SWI kernel resembles the traditional deep-pipeline nature of an FPGA.

OpenCL Channel. This feature enables direct communication between two OpenCL kernels without accessing global memory. The OpenCL channel implements FIFO buffers between the kernels, thereby reducing memory traffic. The OpenCL channel resembles direct on-chip communication on an FPGA.

3 OVERVIEW OF BOYI

The conventional NDRange kernel (Section 2) represents the default OpenCL execution model. By leveraging the emerging OpenCL features, we identify three more execution models on FPGA: SWI, NDR+C, and SWI+C. As we observe the first-order influence of the execution model on the performance upper bound of an OpenCL application (see Figure 1), we aim for an automatic detection of the most suitable execution model for any application.

We present Boyi, a systematic framework that assists OpenCL programmers to determine the most suitable execution model for an OpenCL application. Figure 2 depicts the different components of Boyi. Boyi takes the source code of the OpenCL kernels and the C/C++ host code as inputs, and outputs the most suitable execution model for the target OpenCL application.

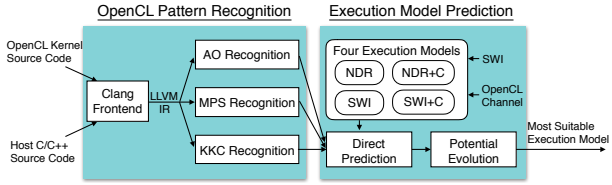


Figure 2: Overview of Boyi.

Boyi consists of two main components: 1) the OpenCL pattern recognition, and 2) the execution model prediction. First, the OpenCL pattern recognition (Section 4) analyzes the input OpenCL kernel and host codes and looks for the presence or absence of three OpenCL patterns: AO, MPS and KKC. These three patterns appear frequently in NDRange kernels. Boyi relies on the LLVM framework [29] to perform code analysis, transformation, and optimization on the LLVM intermediate representation (IR) assembly language. Second, the execution model prediction (Section 5) determines the most suitable execution model. There is a direct prediction and a potential evolution to SWI+C for SWI.

4 OPENCL PATTERN RECOGNITION

We present the first component of Boyi, OpenCL pattern recognition. First, we discuss the properties of three common OpenCL patterns. Second, we describe our LLVM-based OpenCL pattern recognition mechanism, which identifies the three OpenCL patterns in the target OpenCL application.[3]

4.1 Three OpenCL Patterns

For each OpenCL pattern, we identify several issues when the pattern is mapped to an FPGA, and discuss potential optimization directions based on the four possible OpenCL execution models.

4.1.1 Atomic Operation (AO). NDRange kernels require the use of atomic instructions to avoid data races, when multiple work-items try to update the same memory location [14, 15, 39, 48]. Since the hardware for atomic instructions has greatly improved in recent GPU generations (e.g., NVIDIA Pascal [35]), the AO pattern works efficiently on GPUs [11]. As an example, we use the histogram calculation in Listing 1. Each work-item (*tid*) updates one bin of the shared *hist* atomically.

```
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: AO-based histogram calculation.

Issues on FPGAs. We identify three issues for the AO pattern on FPGAs. First, implementing atomic operations on an FPGA is relatively complex, as it requires a large amount of FPGA resources [21]. Second, if an OpenCL kernel contains the AO pattern, *all memory transactions* (i.e., both atomic and non-atomic memory accesses) have to enter the atomic module, which detects and resolves all potential address conflicts among the in-flight memory transactions. This increases the latency of *all* memory accesses, and also potentially leads to lower memory bandwidth. Third, the clock frequency of an OpenCL kernel implementation with the AO pattern on FPGAs is slightly lower than that of an OpenCL kernel without the AO pattern [21]. This leads to lower performance. Due to these three issues, we conclude that the AO pattern is rarely a good fit for FPGAs.

Potential on FPGAs. The SWI OpenCL execution model allows us to avoid the AO pattern on FPGAs, without compromising programmability. Since the SWI kernel uses one single work-item, it avoids any data races and skips the need for atomic instructions. The SWI execution model on FPGAs exploits pipelined parallelism (a good fit for FPGAs), not thread-level parallelism, which is commonplace on GPUs (but not a good fit for FPGAs). We can convert the AO-based histogram calculation in Listing 1 into an SWI kernel, which does not need atomic instructions, as shown in Listing 2.

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

Listing 2: SWI-based histogram calculation.

4.1.2 Kernel-to-Kernel Communication (KKC). The communication between *producer* kernels and *consumer* kernels in the default OpenCL execution model has to use global memory. In particular, the producer kernel writes intermediate data to global memory, and the consumer kernel reads the data from global memory, as shown in Figure 3a. The KKC pattern is suitable for GPUs, since their multithreaded architecture can hide the latency of global memory accesses.

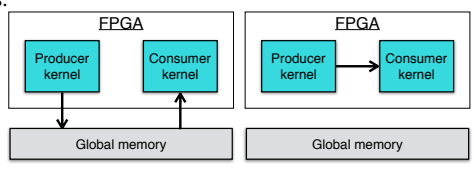


Figure 3: Kernel-to-kernel communication without (a) and with (b) OpenCL channels.

Issues on FPGAs. Since the memory bandwidth of FPGAs is typically much smaller than that of GPUs, KKC via global memory is usually not a good fit for FPGAs.

[3] Boyi can be extended to recognize more OpenCL patterns. We leave this extension for future work.

Potential on FPGAs. With OpenCL channels, a producer kernel can directly send intermediate data to a consumer kernel at the register level (i.e., FIFO queues), without using global memory, as illustrated in Figure 3b. This approach has two main advantages. First, it can eliminate a large amount of global memory accesses. Second, producer and consumer kernels can execute concurrently, if there are enough hardware resources. Thus, not only intra-kernel parallelism (i.e., pipeline parallelism) but also inter-kernel parallelism (i.e., concurrent kernel execution) can be exploited.

4.1.3 Multi-Pass Scheme (MPS). In the default OpenCL execution model, the only way to communicate intermediate results across different work-groups is via the global memory. To ensure memory consistency, this communication requires kernel termination, which represents a global synchronization point. As a result, multi-pass algorithms, where data is scanned (i.e., read from/written to global memory) multiple times, are a common way of implementing OpenCL applications. On GPUs, this multi-pass pattern works efficiently, since GPUs contain many compute units or streaming multiprocessors (SMs), which can run many work-groups in parallel, thereby hiding the latency of global memory accesses. An example of the multi-pass pattern is the parallel prefix sum [18]. Listing 3 shows a pseudocode of the MPS-based prefix sum as implemented in OpenCL. This implementation requires three steps to compute the prefix sum of input array `in` of size `N` and store the output in array `out`. In step 1, `B` work-groups (WGs) execute concurrently. Each WG `b` computes the prefix sum (kernel *prefix_sum_wg*) of its own part of `in`, which starts at address `&in[N*b/B]` and contains `N/B` input elements. Each WG stores its local sum into `local_sum`. In step 2, one single WG computes the prefix sum `local_sum` and stores it into `pre_bsum`. In step 3, each WG adds the corresponding scalar value `pre_bsum[b]` to its part of `out` to produce the final prefix sum.

```
//Step 1: Each WG b computes the prefix sum of its part of in
#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[N*b/B], out[N*b/B], pre_bsum[b], N/B);
```

Listing 3: MPS-based prefix sum.

Issues on FPGAs. MPS-based implementations require off-chip global memory accesses (i.e., loading inputs and storing intermediate results) in each pass of the algorithm. On GPUs, MPS-based implementations rely on a powerful memory subsystem that provides high global memory bandwidth. For example, an NVIDIA Tesla P100 GPU achieves a global memory bandwidth of up to 732GB/s [35], and its SMs can leverage this bandwidth due to their multithreaded architecture. However, the typical global memory bandwidth in an FPGA board is much smaller (e.g., 18GB/s in the Terasic DE5a-Net [45] that we use in our evaluation[4]). As a result, MPS is usually not a desirable pattern on FPGAs.

[4]Some recent FPGA cards, e.g., the Xilinx Alveo U280 [60], feature high bandwidth memory (HBM) with up to 460GB/s bandwidth. However, leveraging that bandwidth is extremely challenging as it requires a very careful and time-consuming hardware design effort.

Potential on FPGAs. The SWI execution model can implement multiple-pass algorithms with a single-pass approach, as illustrated in Listing 4. The single-pass approach reduces the global memory traffic significantly, because it reads from `in` and writes to `out` only once, not twice as the multi-pass algorithm.

```
out[0] = 0;
for (t=1; t<N; t++)  //for loop instead of multiple work-items
  out[t] = out[t-1] + in[t]; //prefix sum of element t
```

Listing 4: SWI-based prefix sum.

4.2 LLVM-based OpenCL Pattern Recognition

In this section, we describe Boyi's LLVM-based OpenCL pattern recognition mechanism, which identifies the presence or absence of three OpenCL patterns (Section 4.1) without programmer intervention. The pattern recognition mechanism consists of nine built-in LLVM passes that analyze OpenCL kernels and host code to automatically find the patterns.

4.2.1 AO Recognition. The recognition of the AO pattern is straightforward, as each atomic instruction in an OpenCL kernel is converted to an atomic LLVM IR instruction. Accordingly, we develop an LLVM analysis pass called *HasAO* (see Figure 4(a)), based on the *llvm::ModulePass* class [29], to find all AO instructions in OpenCL kernels.

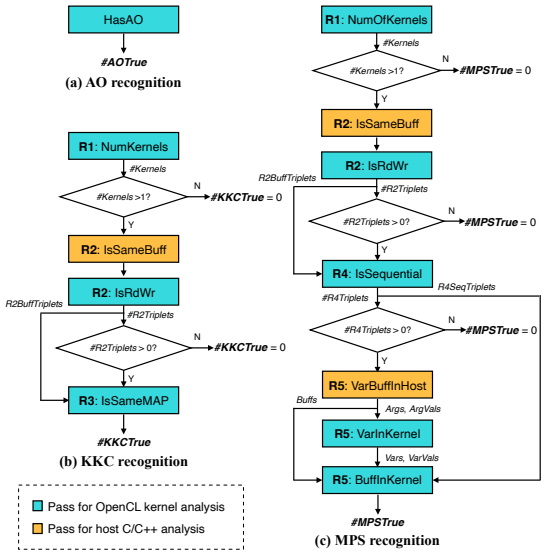


Figure 4: OpenCL pattern recognition: identifying three OpenCL patterns using nine LLVM built-in analysis passes.

4.2.2 KKC Recognition. Figure 4(b) depicts the flowchart for KKC recognition. An OpenCL application contains the KKC pattern (i.e., *#KKCTrue* = 1), if the application code satisfies exactly three requirements (R1, R2, and R3). We describe R1, R2, and R3 next.

R1: The OpenCL Application Has Two or More OpenCL Kernels. The KKC pattern can only exist when the number of kernels (*#Kernels*) of the OpenCL application is greater than one. Accordingly, we develop an LLVM pass called *NumKernels*, based on the *llvm::ModulePass* class, which obtains the value of *#Kernels*. If *#Kernels* = 1, *#KKCTrue* = 0.

R2: A Consumer Kernel Reads from the Same Buffer Object (in Global Memory) as a Producer Kernel Writes to. This requirement of the KKC pattern establishes a data dependence between two kernels. This data dependence stems from two facts:

1) two kernels access the same buffer object in global memory, and 2) one kernel (the producer) writes to this buffer object before another kernel (the consumer) reads from the same buffer object. Accordingly, we design two LLVM passes (*IsSameBuff* and *IsRdWr*) that capture appropriate candidate buffer-kernel triplets (*buffer*, *k1*, *k2*), which satisfy R2. *k1* and *k2* are the candidate producer kernel and the candidate consumer kernel, respectively. The two LLVM passes analyze both OpenCL kernels and host code. Next, we briefly present the design details of these two passes.

The first LLVM pass, *IsSameBuff*, parses the *clSetKernelArg* functions [26] in the host code to determine all references to each buffer object in global memory. More precisely, this pass determines whether two kernels *k1* and *k2* access the same *buffer*.

The second LLVM pass, *IsRdWr*, examines the OpenCL kernel codes to determine whether *k1* writes data to *buffer* before *k2* reads data from the same *buffer*.

After these two passes, we obtain a list of candidate triplets (*R2BuffTriplets*). The number of candidate triplets is *#R2Triplets*.

R3: The Producer and the Consumer Kernels Access the Buffer Object with the Same Memory Access Pattern (MAP). KKC exists if the candidate consumer kernel can directly consume the data generated by the candidate producer kernel, which is only possible when the two kernels have the same MAP to the common buffer. The same MAP occurs when work-items with the same ID in the two kernels access the same buffer elements in the same order. If this happens, the communication between two kernels can be done via an OpenCL channel, instead of via global memory. If not, the two kernels have to communicate via global memory, which implies that the first kernel needs to terminate before the second kernel starts to guarantee memory consistency, i.e., the two kernels have to run sequentially. To recognize whether the two kernels *k1* and *k2* of a candidate triplet from *R2BuffTriplets* have the same MAP to the buffer object *buffer*, we develop an LLVM pass called *IsSameMAP*.

IsSameMAP recognizes regular and irregular MAPs. To this end, this pass checks 1) whether the address of a store instruction to *buffer* in *k1* is the same as the address of a load instruction from *buffer* in *k2*, and 2) whether the execution conditions for both instructions are the same in both kernels. The execution conditions are related to the location of the instructions in the code, e.g., in a sequential structure, in a loop structure, or in a branch structure. The control flow structure should be the same in both kernels. For a loop structure, the loop control logic and the initial index value should be the same in both kernels. For a branch structure, the branch condition should be the same in both kernels. If a candidate triplet satisfies R3, KKC is possible between the producer kernel *k1* and the consumer kernel *k2* (i.e., *#KKCTrue* = 1).

4.2.3 MPS Recognition. Figure 4(c) shows the flowchart for MPS recognition. An OpenCL application contains the MPS pattern (i.e., *#MPSTrue* = 1), if the application code satisfies four requirements (R1, R2, R4, and R5). R1 and R2 are the same requirements that KKC has. Thus, we can use the same three passes (*NumKernels* from R1, and *IsSameBuff* and *IsRdWr* from R2) explained in Section 4.2.2 to obtain a list of candidate triplets *R2BuffTriplets*. Then, we check whether any triplets in *R2BuffTriplets* satisfy R4 and R5. We describe these two requirements next.

R4: Two Kernels Access the Same Buffer Object with Different MAPs. In a multi-pass algorithm, the second kernel can access intermediate data generated by the first kernel only after the first kernel finishes writing to the buffer object. Thus, the two kernels execute sequentially. Accordingly, we develop an LLVM pass, *IsSequential*, that identifies if two kernels need to execute sequentially due to different MAPs to the same buffer object. Essentially, R4 is the opposite of R3, since R4 does not allow the concurrent execution of two kernels. Therefore, the triplets from *R2BuffTriplets* that satisfy R4 are the triplets that do not satisfy R3. For MPS, the candidate triplets are in a list *R4SeqTriplets* with a number of *#R4Triplets* candidate triplets. If the *R4SeqTriplets* list is empty, MPS does not exist in the application code.

R5: Intermediate Buffer Objects are Not Needed when the OpenCL Application is Mapped to One Work-Group. A multi-pass algorithm requires multiple passes because kernel termination is the only way to communicate across different work-groups in the default NDR execution model. In the NDR execution model, the number of work-groups can be either 1) a function of the dataset size (i.e., each work-group processes one portion of the dataset), or 2) an arbitrary number (set by the user), e.g., if the NDR kernel uses thread coarsening [33] (i.e., each work-group runs multiple iterations/portions until the whole dataset is processed). A multi-pass algorithm can be implemented as a single-pass approach, if one single work-group produces correct results (i.e., it processes the entire dataset), which requires that the work-group can access all intermediate results of one pass of the multi-pass algorithm in the next pass. Thus, the key idea in R5 is to analyze the multi-pass algorithm when NDRange contains one single work-group. If all intermediate results of one pass of the multi-pass algorithm are visible to the work-group in the next pass, the intermediate buffer objects are not needed. The intermediate data can be kept in registers of the work-group, and the single-pass approach is feasible. To perform this analysis, we use symbolic execution [27] to determine whether the intermediate buffer objects in *R4SeqTriplets* are still required when *#WG* is set to one. In particular, we can transform a multi-pass algorithm into a single-pass approach if, each buffer object *buffer* satisfies one of the following two cases: 1) the size of *buffer* becomes one, when *#WG* is 1 (**Case C1**), or 2) the second kernel *k2* does not need to access *buffer* (**Case C2**). If so, the MPS pattern (Section 4.1.3) exists. We develop three LLVM passes (*VarBuffInHost*, *VarInKernel* and *BuffInKernel*) to recognize R5, as shown in Figure 4(c).

The first LLVM pass, *VarBuffInHost*, analyzes numeric values and buffer objects in the host code when *#WG* = 1. *VarBuffInHost* parses the *clSetKernelArg* functions for each kernel invocation in the host code, to identify arguments that are a function of *#WG*. In particular, we employ symbolic execution to examine two types of candidate arguments: 1) numeric values, and 2) buffer objects in global memory. For each numeric value (e.g., *int*, *long*), *VarBuffInHost* generates a pair (*Arg*, *ArgVal*), where *Arg* is the argument and *ArgVal* is its value when *#WG* = 1. For example, if the size of an argument *len* is 2×*#WG*, *VarBuffInHost* generates the pair (*len*, 2). These pairs are further processed in the second pass. For each buffer object (i.e., *cl_mem*), *VarBuffInHost* checks whether its size becomes one when *#WG* = 1, i.e., the size of an argument of type

cl_mem is *#WG*. If so, the buffer object satisfies C1. It is added to a buffer pool, *Buffs*, that the third pass uses.

The second LLVM pass, *VarInKernel*, analyzes variables in the OpenCL kernels when *#WG* = 1. *VarInKernel* consists of two steps. First, *VarInKernel* analyzes the OpenCL kernels of the application to find variables of size that is a function of *get_num_groups* (built-in function that returns the number of work-groups *#WG* of an OpenCL kernel) [26]. For each of these variables, *VarInKernel* creates a pair (*Var*, *VarVal*), where *Var* is the variable and *VarVal* is its size when *#WG* = 1. Second, *VarInKernel* identifies variables in the OpenCL kernels that take the value from candidate numeric arguments from the first pass, i.e., pairs (*Arg*, *ArgVal*). For them, *VarInKernel* creates the corresponding pairs (*Var*, *VarVal*).

The third LLVM pass, *BuffInKernel*, uses the outcomes of the previous two passes to check if buffer objects satisfy C1 or C2. *BuffInKernel* analyzes the OpenCL kernels to determine whether the intermediate buffer objects in *R4SeqTriplets* are still required when the variables *Var* from the candidate pairs (*Var*, *VarVal*) have the values *VarVal* (with *#WG* = 1). Essentially, the intermediate buffers are not required if they satisfy either C1 or C2. The buffer objects in *Buffs* from the first pass satisfy C1. The remaining buffer objects (RBOs) from *R4SeqTriplets* that are not in *Buffs* may satisfy C2 or not. *BuffInKernel* examines whether the load/store instructions to the RBOs make sense for execution with the variable pairs (*Var*, *VarVal*) (with *#WG* = 1). For example, if a memory operation becomes $array[index]$+=0, the corresponding load/store instructions do not need to be executed since $array$ is not modified. In such cases, the RBOs satisfy C2.

5 EXECUTION MODEL PREDICTION

In this section, we first describe four OpenCL execution models that we identify. We analyze their programmability, compute parallelism, and memory traffic. Second, we present Boyi's execution model prediction.

5.1 Four OpenCL Execution Models

We present four execution models (NDR, SWI, NDR+C, and SWI+C), which have different degrees of programmability, compute parallelism, and memory traffic (see Table 1). We use histogram calculation (HSTO [13]) as our running example. For each execution model, we show pseudocode with the best combination of conventional optimizations.

	NDR	SWI	NDR+C	SWI+C
Programmability	2	3	1	1 to 2
Compute Parallelism	3	1	3	2 to 3
Memory Traffic	1	3	2	3

Table 1: Programmability, compute parallelism, and memory traffic of four OpenCL execution models. "3" stands for the best score (i.e., high programmability, high compute parallelism, or low memory traffic) and "1" indicates the worst score, according to our experience and analysis.

5.1.1 NDRange (NDR) Execution Model. The NDR execution model employs the NDR kernel, which is the conventional OpenCL kernel, widely-used in GPU programming. The NDR kernel exploits thread-level parallelism. Optimization techniques that are used on GPUs can also apply to FPGAs, e.g., the use of on-chip *local* memory for data reuse.

Programmability. Similar to other parallel programming languages, programming NDR kernels naturally requires more effort from OpenCL programmers than programming sequential code, since programmers need to control the execution of many work-items. For example, in the NDR-based histogram calculation of Listing 5, a programmer needs to use atomic instructions to guarantee memory consistency inside each work-group (WG). Though more challenging than sequential programming, the learning curve of the NDR execution model is much lower than that of RTL programming. We rate the programmability of the NDR execution model as *moderate (2)* in Table 1.

```
NDR-based kernel: __attribute((num_compute_units(6))) //number of CUs
int gid = get_global_id(0);   //global work-item ID
int tid = get_local_id(0);    //local work-item ID
int gs  = get_global_size(0); //global size (#work-items in NDRange)
int ls  = get_local_size(0);  //local size (#work-items in WG)
//Step 1: each WG computes the local histogram on its own data.
for (t = gid; t < N; t += gs) {
  int d   = data[t];          //fetch the data from global memory
  int h_d = hash(d);          //compute the hash index
  atomic_add(&l_hist[h_d],1);//atomically add to the local histogram
}
barrier(CLK_LOCAL_MEM_FENCE); //intra-WG synchronization
//Step 2: atomically add the local histogram to the global histogram
for (pos = tid; pos < BINS, pos += ls)
  atomic_add(&hist[pos],l_hist[pos]); //atomically add l_hist to hist
```

Listing 5: NDR-based histogram calculation.

Compute Parallelism. The NDR kernel expresses parallelism using a large number of work-items that run in multiple compute units inside specialized hardware synthesized on an FPGA. The number of compute units is limited only by the available hardware resources. For example, Listing 5 uses 6 compute units. Thus, the NDR execution model can potentially achieve *high (3)* compute parallelism, as shown in Table 1.

Memory Traffic. The NDR execution model resorts to MPS and/or AO patterns to guarantee memory consistency across work-groups (MPS) or inside a work-group (AO). Both patterns entail high memory traffic requirements. MPS performs communication across kernels via global memory, thereby requiring global memory traffic to read inputs and write intermediate results. AO does not allow the OpenCL compiler to coalesce multiple memory requests to generate large burst memory transactions, thereby requiring memory traffic for each request. We rate the memory traffic of the NDR execution model as *high (1)* in Table 1.

5.1.2 Single Work-item (SWI) Execution Model. The SWI execution model is based on sequential programming, as it employs one single work-item.

Programmability. As the SWI kernel uses one single work-item, it is easy to program. Listing 6 shows the SWI-based histogram calculation, where we do not need atomic instructions as in Listing 5. Instead, it uses multiple local histograms (32 in Listing 6) to achieve more parallelism with loop unrolling. We rate the programmability of the SWI execution model as *high (3)* in Table 1.

Compute Parallelism. The SWI kernel relies on an offline compiler to extract pipelined parallelism. To guarantee correct execution of memory accesses, the compiler imposes an initiation interval *II*, which is the number of cycles between the start of consecutive pipeline iterations. For example, in Listing 6 *II* = 2, which means that a new iteration starts every 2 cycles. In this case, these two cycles stem from the updates to the local histograms *l_hist*, where one read and one write (i.e., 2 cycles) per update are needed. Due to the need for such ordering and delay between pipeline iterations, we rate the compute parallelism of SWI as *low (1)* in Table 1.

```
SWI-based kernel:
//Step 1: compute 32 local histograms
for(i = 0; i < N/32; i++) { //II = 2
  int16 d0=((__global int16*)data)[i*2]; //read the data from memory
  int16 d1=((__global int16*)data)[i*2+1];
  #pragma unroll
  for(j = 0; j < 16; j++) {
    h_d_0j = hash(d0.sj);   //compute the hash index
    h_d_1j = hash(d1.sj);
    l_hist_0j[h_d_0j] += 1; //compute the local histogram
    l_hist_1j[h_d_1j] += 1;
  }
}
//Step 2: accumulate all 32 local histograms to the global histogram
for(pos = 0; pos < BINS; pos++)
  hist[pos] =l_hist_00[pos] +...+ l_hist_0F[pos]
            +l_hist_10[pos] +...+ l_hist_1F[pos];//add to global hist
```

Listing 6: SWI-based histogram calculation.

Memory Traffic. The SWI execution model has significantly lower memory traffic requirements than NDR. First, it can leverage a single-pass approach, which entails less memory traffic than the multi-pass approach. Second, it does not need atomic instructions because there is only one single work-item (see Listing 6). We rate the memory traffic of SWI as *low (3)* in Table 1.

5.1.3 NDRange+Channel (NDR+C) Execution Model. The NDR+C execution model divides a large NDR kernel into multiple small NDR kernels connected via OpenCL channels.

```
Channel int C_IN[16]; //16 channels for inter-kernel communication

Data_in kernel:
for (t = gid; t < N/16; t += gs) { //gs = global size
  int16 d = ((__global int16*)data)[t]; //read data from memory
  #pragma unroll
  for(c = 0; c < 16; c++)
    write_channel_intel (C_IN[c], d.sc); //write channel (Intel SDK)
}

Data_out kernel k (k = 0,1,...,15):
//Step 1: each WG computes the local histogram on its own data
for(t = gid; t < N/16; t += gs) {
  int d = read_channel_intel (C_IN[k]); //read data from channel
  int h_d = hash(d);                    //compute the hash index
  atomic_add(&l_hist[h_d],1); //atomically add to the local histogram
}
barrier(CLK_LOCAL_MEM_FENCE); //intra-WG synchronization
//Step 2: atomically add each local histogram to the global histogram
for (pos = tid; pos < BINS, pos += ls)
  atomic_add(&hist[pos],l_hist[pos]);//atomically add to global hist
```

Listing 7: NDR+channel-based histogram calculation.

Programmability. Programming with OpenCL channels increases the programming difficulty. First, the producer kernel has to produce the data flow in the same order as it is expected by the consumer kernel. This is challenging because work-items can execute out-of-order due to load imbalance and, thus, the data flow may not be predictable. Second, we have to explicitly instantiate each involved kernel. Listing 7 shows the NDR+C-based histogram calculation, which instantiates multiple NDR kernels to work concurrently (one `Data_in` kernel and 16 `Data_out` kernels). The `Data_in` kernel reads 16 integers from global memory per cycle, and dispatches one integer to each `Data_out` kernel via an OpenCL channel. We rate the programmability of NDR+C as *low (1)* in Table 1.

Compute Parallelism. The OpenCL channel allows producer/-consumer NDR kernels to execute concurrently, leading to high compute parallelism. Listing 7 shows one producer kernel and 16 consumer kernels that can run concurrently. We rate the compute parallelism of NDR+C as *high (3)* in Table 1.

Memory Traffic. OpenCL channels can potentially reduce the high memory traffic of the NDR execution model. However, the NDR+C execution model still needs to use costly atomic instructions. Thus, the memory traffic of NDR+C is *moderate (2)* in Table 6.

5.1.4 Single Work-item+Channel (SWI+C) Execution Model. The SWI+C execution model employs OpenCL channels to connect multiple SWI kernels that can run concurrently (see Listing 8).

Programmability. The programmability of an SWI kernel is lower when using OpenCL channels. Thus, we rate the programmability of SWI+C as *low (1)* to *moderate (2)* in Table 1.

Compute Parallelism. Even though one SWI kernel has low compute parallelism, OpenCL channels can significantly increase the compute parallelism by allowing multiple producer/consumer SWI kernels to execute concurrently. Listing 8 divides the SWI kernel into one producer kernel (`Data_in`), which reads from memory, and one consumer kernel (`Data_out`), which calculates the histogram. It uses two channels for the two input elements processed in each iteration. Therefore, SWI+C's overall compute parallelism is *moderate (2)* to *high (3)* in Table 1.

```
Channel int16 C_IN[2];

Data_in kernel:
for(i = 0; i < N/32; i++) {
  int16 d0=((__global int16*)data)[i*2]; //read data from memory
  int16 d1=((__global int16*)data)[i*2+1];
  write_channel_intel(C_IN[0], d0); //write channel (Intel SDK)
  write_channel_intel(C_IN[1], d1);
}

Data_out kernel:
//Step 1: compute 32 local histograms
for(i = 0; i < N/32; i++) {
  int16 d0 = read_channel_intel(C_IN[0]);//read data from channel
  int16 d1 = read_channel_intel(C_IN[1]);
  #pragma unroll
  for(j = 0; j < 16; j++) {
    h_d_0j = hash(d0.sj);   //compute the hash index
    h_d_1j = hash(d1.sj);
    l_hist_0j[h_d_0j] += 1; //compute the local histogram
    l_hist_1j[h_d_1j] += 1;
  }
}
//Step 2: accumulate all 32 local histograms to the global histogram
for(pos = 0; pos < BINS; pos++)
  hist[pos] =l_hist_00[pos] +...+ l_hist_0F[pos]
            +l_hist_10[pos] +...+ l_hist_1F[pos];//add to global hist
```

Listing 8: SWI+channel-based histogram calculation.

Memory Traffic. The low memory traffic of the SWI execution model can be further reduced by using OpenCL channels in the SWI+C execution model. Therefore, we rate the memory traffic of the SWI+C execution model as *low (3)* in Table 1.

5.2 Execution Model Prediction

We present the second component of Boyi, execution model prediction. First, we present the direct prediction of the execution model, which is based on the presence or absence of the three OpenCL patterns. Second, we discuss the potential evolution of the SWI execution model to SWI+C to achieve higher compute parallelism.

5.2.1 Direct Prediction. We can directly predict the most suitable execution model for a target OpenCL application (see "Direct prediction" column in Table 2), based on the presence or absence of the three OpenCL patterns (the three leftmost columns of Table 2). For each pattern combination, the direct prediction stems from the potential implementations on FPGAs using the two new OpenCL features, as we discussed in Section 4.1. Essentially, OpenCL applications with the AO and/or MPS patterns benefit from the SWI execution model, while OpenCL applications with the KKC pattern benefit from OpenCL channels.

AO	MPS	KKC	Direct prediction	Potential SWI evolution
N	N	N	NDR	-
Y	N	N	SWI	SWI+C
N	Y	N	SWI	SWI+C
Y	Y	N	SWI	SWI+C
N	N	Y	NDR+C	-
Y	N	Y	SWI+C	SWI+C
N	Y	Y	SWI+C	SWI+C
Y	Y	Y	SWI+C	SWI+C

Table 2: Directly predicted execution model (based on the presence or absence of AO, MPS, and KKC patterns) and potential SWI evolution.

5.2.2 Potential Evolution of SWI to SWI+C. If the direct prediction for an OpenCL application is SWI, the execution model can potentially evolve to SWI+C (see the "Potential SWI evolution" column in Table 2) for higher compute parallelism, if the application satisfies two conditions (S1 and S2). The evolution of the SWI execution model consists of dividing an SWI kernel into multiple smaller SWI kernels, which can run concurrently, connected via OpenCL channels. Doing so increases compute parallelism at the expense of higher programming complexity.

S1: Sufficient FPGA Resources Available. The evolution can only happen if there are sufficient FPGA resources available. The SWI+C execution model instantiates more than one SWI kernel connected via OpenCL channels, which requires more FPGA resources than a single SWI kernel.

S2: The SWI Kernel is Compute-Bound and $II > 1$. The evolution can only happen when the original SWI kernel, before applying conventional optimizations, is 1) compute-bound and 2) its II is greater than 1. First, the SWI kernel should be compute-bound, because the goal of the evolution is to increase the compute parallelism, which is in general not effective for a memory-bound kernel, as it is already bottlenecked by memory traffic. Second, II should be greater than 1, because $II = 1$ means that the hardware is fully pipelined. Thus, no potential improvement is possible in terms of pipelined parallelism.

To determine whether an SWI kernel is compute-bound, we propose a cost model that estimates the number of compute cycles (C^{comp}) and memory cycles (C^{mem}) of the SWI kernel. The SWI kernel is compute-bound if C^{comp} is greater than or equal to C^{mem}, as shown in Equation 1. For the estimation of C^{comp} and C^{mem}, we focus on the analysis of loops, as they typically represent the main body of a kernel. First, we present the cycle count estimation for an entire SWI kernel. Second, we describe how we estimate the compute and memory cycles for each loop.

$$C^{comp} \geq C^{mem} \tag{1}$$

Estimation of Kernel Cycles. We assume that an OpenCL kernel consists of n outer loops that are executed sequentially. We estimate C^{comp} and C^{mem} as the sum of the estimated numbers of computation cycles (C_l^{comp} for loop l) and memory cycles (C_l^{mem} for loop l), respectively, for all outer loops, as shown in Equation 2.[5]

$$C^{comp} = \sum_{l=1}^{n} C_l^{comp}, \quad C^{mem} = \sum_{l=1}^{n} C_l^{mem} \tag{2}$$

Estimating C_l^{comp}. We estimate C_l^{comp} as the trip count LTC_l of loop l multiplied by its initiation interval II_l, as shown in Equation 3. This gives us the number of cycles for all loop iterations.

[5]We do not consider nested loops in this paper, but our work can be easily extended to consider them.

$$C_l^{comp} = LTC_l \times II_l \tag{3}$$

Estimating C_l^{mem}. We estimate C_l^{mem} as the sum of memory cycles required by all sequential and random memory accesses, as Equation 4 shows. C_i^{seq} and C_i^{rand} stand for the number of cycles for all sequential and random memory transactions in loop l, respectively.

$$C_l^{mem} = C_l^{seq} + C_l^{rand} \tag{4}$$

We estimate C_l^{seq} (or C_l^{rand}) as the total number of sequential (or random) memory transactions MT_l^{seq} (or MT_l^{rand}) in loop l multiplied by the number of cycles CPT^{seq} (or CPT^{rand}) per sequential (or random) memory transaction, as Equation 5 shows. We obtain CPT^{seq} and CPT^{rand} with microbenchmarks. In particular, we measure the throughput of a kernel with many back-to-back sequential (or random) memory accesses. CPT^{seq} (or CPT^{rand}) is the inverse of the throughput. Our measured $CPT^{seq} = 1.0$ (or $CPT^{rand} = 2.0$) means that one sequential (or random) memory transaction finishes every cycle (or every two cycles).

$$C_l^{seq} = MT_l^{seq} \times CPT^{seq}, \quad C_l^{rand} = MT_l^{rand} \times CPT^{rand} \tag{5}$$

We estimate MT_l^{seq} as the loop trip count (left factor in Equation 6) multiplied by the number of sequential memory transactions in one loop iteration (right factor in Equation 6). We assume that the body of loop l has N_l^{seq} sequential memory instructions and the s-th sequential memory instruction references B_s bytes each time, where $1 \leq s \leq N_l^{seq}$. Since sequential memory accesses are coalesced, i.e., they merge into a single wide memory transaction, we divide the number of bytes by the wide transaction size (or memory burst size, which is up to $\#BS$=64 bytes in our FPGA board).

$$MT_l^{seq} = LTC_l \times \sum_{s=1}^{N_l^{seq}} \frac{B_s}{\#BS} \tag{6}$$

We estimate MT_l^{rand} as the loop trip count LTC_l multiplied by the number of random memory transactions N_l^{rand} (random accesses cannot be coalesced), as Equation 7 shows.

$$MT_l^{rand} = LTC_l \times N_l^{rand} \tag{7}$$

6 EVALUATION

This section presents our experimental setup (Section 6.1), compares the performance of 11 OpenCL applications using the four execution models (Section 6.2), and validates Boyi's execution model prediction (Section 6.3).

6.1 Experimental Setup

Hardware Configuration. We run our experiments on a Terasic DE5a-Net board [45] with an Intel Arria 10 GX FPGA (10AX115N2F45E1SG) and 8GB 2-bank DDR3 device memory. We employ Intel OpenCL SDK version 16.1 [21].

Workloads. We use with 11 OpenCL applications, listed in Table 3. Seven applications are from the Chai benchmark suite [4, 13], which is originally designed for GPUs. The GPU implementation of each application serves as baseline for our comparisons.

6.2 Performance Comparison

We analyze the performance impact of different execution models and conventional optimizations.

Application	Description	AO	MPS	KKC	Datasets
BFS [13]	Breadth-First Search	Y	N	N	NY, NE, UT
RSCD [13]	RANSAC	Y	N	Y	2000 iterations
TQH [13]	Task Queue System	Y	N	N	Basket
HSTO [13]	Histogram	Y	N	N	256 bins
SC [13]	Stream Compaction	Y	N	N	50%
PAD [13]	Padding	Y	N	N	200*199
CEDD [13]	Canny Edge Detection	N	N	Y	Peppa, Maradona, Paw
KM [5]	KMeans	N	N	N	25600 points, 8 features
MM [20]	Matrix Multiplication	N	N	N	A: 2k*1k, B: 1k*1k
MS [20]	Mandelbrot Set	N	N	N	640*800, 2000 iterations
PS [18]	Prefix Sum	N	Y	N	4194304 points

Table 3: Benchmarks, OpenCL patterns, and datasets.

Exploring Optimization Combinations. For each OpenCL application, we implement versions with the four execution models and multiple combinations of conventional optimizations (see Table 4). We apply the conventional optimizations following the step-by-step approach proposed in [54]. Our exploration of the optimization space considers all possible combinations of optimization factors (e.g., number of CUs, unrolling factor) until either the FPGA resources are exhausted or the performance saturates. To illustrate our optimization exploration, we use KMeans as an example. Figure 5 shows, for each execution model, the speedup of a subset of optimization combinations over the baseline GPU code [5]. The x-axis shows the optimization combinations for the four execution models. We observe that different execution models yield a performance difference as large as 4.7×. In particular, the best optimization combination under the NDR execution model achieves a speedup of 147.7× over the GPU baseline, while the best optimization combination under the SWI+C execution model achieves 31.4× speedup.

Comparison of Execution Models. Table 4 shows the number of optimization combinations for the four execution models and the maximum speedup achieved with the best combination for each execution model over the baseline. We make three observations. First, different execution models result in significant performance differences. For example, for HSTO, the most suitable execution model (SWI+C) obtains a speedup of 32.4× over the baseline, while the least suitable execution model (NDR) achieves only 2.7×. Second, different OpenCL applications prefer different execution models. For example, TQH favors SWI+C, while MM favors NDR. Third, finding the most suitable execution model is critical and nontrivial, as tens of optimization combinations should be tested for each application and execution model.

App	Number of combinations				Maximum speedup			
	NDR	SWI	NDR+C	SWI+C	NDR	SWI	NDR+C	SWI+C
BFS	17	7	7	9	1.9	**3.1**	1.2	**3.1**
RSCD	25	10	24	46	15.8	4.6	**73.8**	39.7
TQH	9	15		23	1.1	1.3		**21.0**
HSTO	13	37	11	29	2.7	5.1	16.9	**32.4**
SC	15	34		10	1.5	4.5		**17.1**
PAD	10	10		14	1.2	1.6		**4.8**
CEDD	57	15	22	7	2.7	0.3	**2.7**	0.4
KM	33	11	10	18	**147.7**	32.8	136.4	31.4
MM	25	9		6	**837.1**	13.3		6.6
MS	7	6		7	**34.7**	0.02		3.2
PS	26	20		12	15.8	**44.4**		**46.2**

Table 4: Exploration of optimization combinations and maximum speedup for each execution model. An empty slot indicates that the particular execution model cannot be implemented due to limitations of the OpenCL channel.

Effect of New OpenCL Features. As discussed in Section 4.1, the two new OpenCL features have the potential to reduce memory traffic. Figure 6 shows the memory traffic (a) and execution time (b) of four applications with the four execution models (using the best optimization combination). We make two observations. First, the SWI execution model provides a dramatic reduction in memory traffic over the NDR execution model (Figure 6a). SWI kernels employ single-pass approaches, which require less global memory accesses than the multi-pass approaches employed by NDR kernels. Second, even though SWI and SWI+C execution models require the same memory traffic, their performance difference is significant (Figure 6b). The SWI execution model cannot always provide enough compute parallelism to the application. OpenCL channels allow higher compute parallelism (concurrent kernel execution), leading to higher performance on compute-bound applications. We conclude that the two new features can greatly improve the performance of OpenCL programs on FPGAs.

Comparison to Previous Work. Table 5 shows the comparison of our implementations (with the best optimization combination) of 6 applications with previous implementations on FPGAs [19, 20]. We observe that our implementations achieve 1.5-38.3× higher throughput, validating the effectiveness of our Boyi approach. The speedups are mainly due to the use of SWI kernels and OpenCL channels, which are not used in [19, 20].

App	Boyi (ms)	Previous work (ms)	Speedup
RSCD	0.8	28.9 [19]	38.3
TQH	66.9	150.6 [19]	2.3
HSTO	38.8	487.9 [19]	12.6
CEDD	161.9	237.8 [19]	1.5
MM	9.1	34.3 [20]	3.8
MS	27.2	944.1 [20]	34.7

Table 5: Execution time of our Boyi-driven implementations for 6 applications and comparison to previous implementations [19, 20] on FPGAs.

6.3 Execution Model Prediction

We validate that Boyi can predict the most suitable execution model for each OpenCL application. Table 6 shows the predicted execution model, as well as the actual most suitable execution model that achieves the highest performance in our experiments. We make two observations. First, Boyi execution model prediction is accurate for all workloads except RSCD. In RSCD, the only atomic instruction[6] is executed after one loop with many iterations (5922 for our dataset), so it has negligible impact on the overall performance. Thus, the existing AO pattern does not affect performance. As a result, the predicted execution model does not match the actual most suitable execution model. We leave the study of such corner cases for future work. Second, Boyi's potential SWI evolution (Section 5.2.2) significantly increases the prediction accuracy. Table 6 shows that four out of five correct "SWI+C★" predictions are due to potential evolution. Potential evolution provides speedups of up to 16.2× over the SWI execution model.

7 RELATED WORK

To our knowledge, Boyi is the first systematic study to help OpenCL programmers find the most suitable OpenCL execution model (i.e., NDR, SWI, NDR+C, SWI+C) for a particular application on FPGAs. After determining the most suitable execution model, programmers can apply conventional optimizations for OpenCL on FPGAs.

Conventional Optimization Methods. Previous works [3, 16, 23, 25, 30, 37, 41–44, 46, 47, 50, 51, 55, 57–59, 61–63, 65] focus on improving the performance of applications on FPGAs. However,

[6]The original version of RSCD [13] uses two atomic instructions. We refactored the code and got rid of one atomic instruction that was on the critical path.

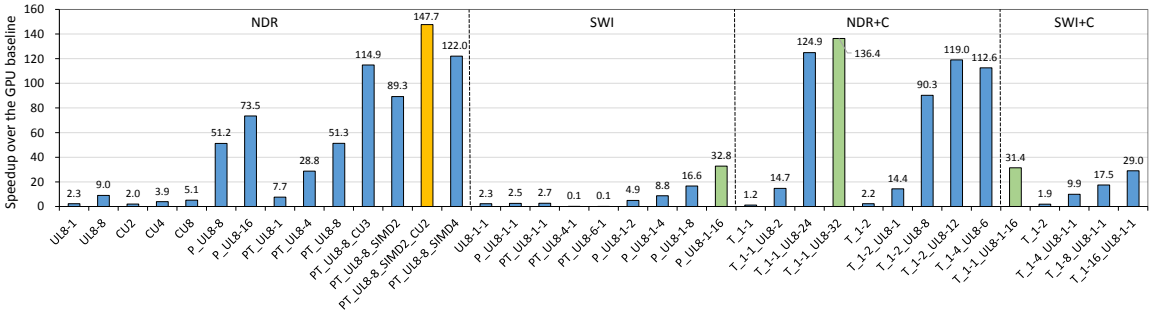
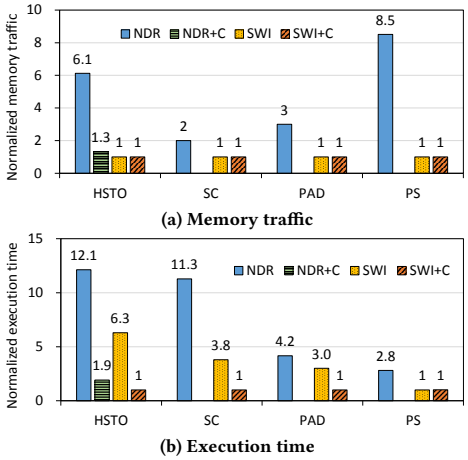


Figure 5: Speedup of a subset of optimization combinations over baseline for KM. "CUx" indicates x CUs, "SIMDx" indicates the kernel vectorization factor x, "ULx-y-z" indicates the loop unrolling factors x,y,z of the inner, middle and outer loops, respectively. "1-x" indicates a multi-kernel design with one producer kernel and x consumer kernel(s). "P" indicates the use of private memory for the input feature array and "T" indicates the transposition of the input feature array for a more regular memory access pattern.



(a) Memory traffic

(b) Execution time

Figure 6: Comparison of four execution models.

Application	AO	MPS	KKC	Actual	Prediction	Speedup of SWI+C over SWI
BFS	Y	N	N	SWI	SWI	
RSCD	Y	N	Y	NDR+C	SWI+C	
TQH	Y	N	N	SWI+C	SWI+C★	16.2
HSTO	Y	N	N	SWI+C	SWI+C★	6.3
SC	Y	N	N	SWI+C	SWI+C★	3.8
PAD	Y	N	N	SWI+C	SWI+C★	3.0
CEDD	N	N	Y	NDR+C	NDR+C	
KM	N	N	N	NDR	NDR	
MM	N	N	N	NDR	NDR	
MS	N	N	N	NDR	NDR	
PS	N	Y	N	SWI	SWI	

Table 6: Actual and predicted most suitable execution models for each application. "SWI+C★" indicates the potential evolution of SWI. The "Speedup" column shows the speedup of SWI+C over SWI due to potential SWI evolution.

these works are limited to High Level Synthesis (HLS) codes or conventional OpenCL NDR kernels, which usually fail to exploit the full potential of an FPGA. In contrast, our work is a systematic study on the suitability of four OpenCL execution models. Three of these models are based on two new OpenCL features (i.e., SWI kernel, OpenCL channel) that are intended to fully leverage the FPGA compute capability.

Optimizations with Two New OpenCL Features. Previous works [2, 6, 24, 34, 49, 53, 56, 67] employ the two new OpenCL features to accelerate OpenCL applications on FPGAs. However, these works focus on specific applications. Thus, their findings cannot directly generalize to other OpenCL applications. In contrast, our work provides an automatic tool to decide on the OpenCL execution model, and thus the use of the two new OpenCL features.

Optimization Frameworks. Prior works propose a number of performance frameworks and auto-tuning tools [7–10, 17, 28, 31, 38, 40, 52, 54, 64, 66]. For example, Wang et al. [54] present a performance analysis framework to identify bottlenecks of OpenCL kernels on FPGAs. However, they only cover analytical models for the conventional NDRange kernel. In [40], the authors present seven empirically-guided code optimization versions, which include the usage of the two new OpenCL features, for optimizing OpenCL kernels on FPGAs. They do not explicitly determine which code version to use for a given application. Our work provides a systematic framework to assist programmers on the selection of the most suitable execution model for an application.

8 CONCLUSION

The conventional OpenCL execution model on FPGAs cannot fully harvest the performance potential of FPGAs. To address this problem, we identify three other execution models and design Boyi, a systematic framework that determines the most suitable execution model for an application, based on the presence or absence of three OpenCL patterns. Our experimental results show that Boyi's predicted execution model matches very accurately the actual most suitable execution model for a given application. We believe Boyi can greatly alleviate the programming burden on FPGA workload optimization exploration. Although we demonstrate Boyi with the Intel OpenCL SDK, Boyi can also be applied to the Xilinx OpenCL tool. We believe Boyi is the first step towards automated code transformations that convert the conventional OpenCL execution model to other execution models. We freely release Boyi [22] so that future work can build on it seamlessly.

Acknowledgement. We thank Intel FPGA Academic Program who denoted Terasic's DE5a-Net FPGA board and licenses for our research. This work was supported by the National Key R&D Program of China (2017YFC0805005 and 2018YFB1702000), the Joint Funds of the National Natural Science Foundation of China (U1908212), and the National Natural Science Foundation of China (61871107 and 61602104). Onur Mutlu and Juan Gómez-Luna acknowledge support from the SAFARI Group's industrial partners, especially Facebook, Google, Huawei, Intel, Microsoft, SRC, and VMware.

REFERENCES

[1] Yi (husbandman). https://en.wikipedia.org/wiki/Yi_(husbandman). Accessed: 2019-12-09.

[2] M. S. Abdelfattah, A. Hagiescu, and D. Singh. Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL. In *IWOCL*, 2014.

[3] Y. Afsharnejad, A. Yassine, O. Ragheb, P. Chow, and V. Betz. HLS-based FPGA acceleration of light propagation simulation in turbid media. In *HEART*, 2018.

[4] L.-W. Chang, J. Gómez-Luna, I. El Hajj, S. Huang, D. Chen, and W.-m. Hwu. Collaborative computing for heterogeneous integrated systems. In *ICPE*, 2017.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[6] X. Chen, R. Bajaj, Y. Chen, J. He, B. He, W. Wong, and D. Chen. On-the-fly parallel data shuffling for graph processing on OpenCL-based FPGAs. In *FPL*, 2019.

[7] Y. T. Chen and J. H. Anderson. Automated generation of banked memory architectures in the high-level synthesis of multi-threaded software. In *FPL*, 2017.

[8] J. Cong, Z. Fang, Y. Hao, P. Wei, C. H. Yu, C. Zhang, and P. Zhou. Best-effort FPGA programming: A few steps can go a long way. In *Arxiv*, 2018.

[9] J. Cong, P. Wei, C. H. Yu, and P. Zhang. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *DAC*, 2018.

[10] N. Engelhardt, C. D. Hung, and H. K. So. Performance-driven system generation for distributed vertex-centric graph processing on multi-FPGA systems. In *FPL*, 2018.

[11] S. Garcia de Gonzalo, S. Huang, J. Gómez-Luna, S. Hammond, O. Mutlu, and W. Hwu. Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on GPUs. In *CGO*, 2019.

[12] Q. Gautier, A. Althoff, and R. Kastner. Spector: An OpenCL FPGA benchmark suite. In *FPT*, 2016.

[13] J. Gómez-Luna, I. El Hajj, V. Chang, Li-Wen Garcia-Flores, S. Garcia de Gonzalo, T. Jablin, A. J. Pena, and W.-m. Hwu. Chai: Collaborative heterogeneous applications for integrated-architectures. In *ISPASS*, 2017.

[14] J. Gómez-Luna, J. González-Linares, J. Benavides, and N. Guil. An optimized approach to histogram computation on GPU. *Machine Vision and Applications*, 2013.

[15] J. Gómez-Luna, J. González-Linares, J. Benavides, and N. Guil. Performance modeling of atomic additions on GPU scratchpad memory. *TPDS*, 2013.

[16] Y. Guan, Z. Yuan, G. Sun, and J. Cong. FPGA-based accelerator for long short-term memory recurrent neural networks. In *ASPDAC*, 2017.

[17] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W. Hwu, and D. Chen. FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge. In *DAC*, 2019.

[18] M. Harris. Parallel prefix sum (scan) with CUDA. Technical report, Nvidia, https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda, 2007.

[19] S. Huang, L.-W. Chang, I. El Hajj, S. Garcia de Gonzalo, J. Gómez-Luna, S. R. Chalamalasetti, M. El-Hadedy, D. Milojicic, O. Mutlu, D. Chen, and W.-m. Hwu. Analysis and modeling of collaborative execution strategies for heterogeneous CPU-FPGA architectures. In *ICPE*, 2019.

[20] Intel. Intel SDK for OpenCL Design Examples. 2018.

[21] Intel. Intel SDK for OpenCL Optimization Guide. 2018.

[22] J. Jiang. Boyi (FPGA2020 version). https://doi.org/10.5281/zenodo.3575234, Dec. 2019. Zenodo.

[23] Z. Jin and H. Finkel. Optimizing an atomics-based reduction kernel on OpenCL FPGA platform. In *IPDPSW*, 2018.

[24] T. Kenter, J. Förstner, and C. Plessl. Flexible FPGA design for FDTD using OpenCL. In *FPL*, 2017.

[25] T. Kenter, G. Mahale, S. Alhaddad, Y. Grynko, C. Schmitt, A. Afzal, F. Hannig, J. Förstner, and C. Plessl. OpenCL-based FPGA design to accelerate the nodal discontinuous galerkin method for unstructured meshes. In *FCCM*, 2018.

[26] Khronos group. The OpenCL specification, July 2019.

[27] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 1976.

[28] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *FPGA*, 2019.

[29] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. 2004.

[30] Q. Li, X. Zhang, J. Xiong, W. Hwu, and D. Chen. Implementing neural machine translation with bi-directional GRU and attention mechanism on FPGAs using HLS. In *ASPDAC*, 2019.

[31] Y. Liang, S. Wang, and W. Zhang. FlexCL: A model of performance and power for OpenCL workloads on FPGAs. *TC*, 2018.

[32] Loring Wirbel. Xilinx SDAccel, a unified development environment for tomorrow's data center. 2014.

[33] A. Magni, C. Dubach, and M. O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *SC*, 2013.

[34] V. Mirian and P. Chow. Exploring pipe implementations using an OpenCL framework for FPGAs. In *FPT*, 2015.

[35] NVIDIA. NVIDIA Tesla P100. White paper, 2016.

[36] NVIDIA. CUDA C programming guide v. 10.0, November 2019.

[37] H. Peng, X. Zhang, and L. Huang. An energy efficient approach for C4.5 algorithm using OpenCL design flow. In *FPT*, 2017.

[38] A. Powell, C. Bouganis, and P. Y. K. Cheung. High-level power and performance estimation of FPGA-based soft processors and its application to design space exploration. *JSA*, 2013.

[39] N. Ramanathan, J. Wickerson, F. Winterstein, and G. A. Constantinides. A case for work-stealing on FPGAs with OpenCL atomics. In *FPGA*, 2016.

[40] A. Sanaullah, R. Patel, and M. C. Herbordt. An empirically guided optimization framework for FPGA OpenCL. In *FPT*, 2018.

[41] S. Sridharan, P. Durante, C. Faerber, and N. Neufeld. Accelerating particle identification for high-speed data-filtering using OpenCL on FPGAs and other architectures. In *FPL*, pages 1–7, 2016.

[42] N. K. Srivastava, S. Dai, R. Manohar, and Z. Zhang. Accelerating face detection on programmable SoC using C-based synthesis. In *FPGA*, 2017.

[43] J. Su, N. J. Fraser, G. Gambardella, M. Blott, G. Durelli, D. B. Thomas, P. H. W. Leong, and P. Y. K. Cheung. Accuracy to throughput trade-offs for reduced precision neural networks on reconfigurable logic. In *ARC*, 2018.

[44] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. B. K. Vrudhula, J. Seo, and Y. Cao. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *FPGA*, 2016.

[45] Terasic. *DE5-Net User Manual*, 2018.

[46] L. D. Tucci, K. O'Brien, M. Blott, and M. D. Santambrogio. Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL. In *DATE*, 2017.

[47] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside. Resource elastic virtualization for FPGAs using OpenCL. In *FPL*, 2018.

[48] G. van den Braak, J. Gómez-Luna, H. Corporaal, J. M. González-Linares, and N. Guil. Simulation and architecture improvements of atomic operations on GPU scratchpad memory. In *ICCD*, 2013.

[49] D. Wang, K. Xu, and D. Jiang. PipeCNN: An OpenCL-based open-source FPGA accelerator for convolution neural networks. In *FPT*, 2017.

[50] H. Wang, M. Zhang, T. Prabu, and O. Sinnen. FPGA-based acceleration of FDAS module using OpenCL. In *FPT*, 2016.

[51] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang. C-LSTM: enabling efficient LSTM using structured compression techniques on FPGAs. In *FPGA*, 2018.

[52] S. Wang, Y. Liang, and W. Zhang. FlexCL: An analytical performance model for OpenCL workloads on flexible FPGAs. In *DAC*, 2017.

[53] Z. Wang, B. He, and W. Zhang. A study of data partitioning on OpenCL-based FPGAs. In *FPL*, 2015.

[54] Z. Wang, B. He, W. Zhang, and S. Jiang. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *HPCA*, 2016.

[55] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang. Relational query processing on OpenCL-based FPGAs. In *FPL*, 2016.

[56] Z. Wang, J. Paul, B. He, and W. Zhang. Multikernel data partitioning with channel on OpenCL-based FPGAs. *TVLSI*, 2017.

[57] Z. Wang, S. Zhang, B. He, and W. Zhang. Melia: A MapReduce framework on OpenCL-based FPGAs. *TPDS*, 2016.

[58] X. Wei, Y. Liang, and J. Cong. Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management. In *DAC*, 2019.

[59] D. Weller, F. Oboril, D. Lukarski, J. Becker, and M. B. Tahoori. Energy efficient scientific computing on FPGAs using OpenCL. In *FPGA*, 2017.

[60] Xilinx. *Xilinx Alveo U230, product brief*, 2019.

[61] C. Yang, J. Sheng, R. Patel, A. Sanaullah, V. Sachdeva, and M. C. Herbordt. OpenCL for HPC with FPGAs: Case study in molecular electrostatics. In *HPEC*, 2017.

[62] Y. Yang, Q. Huang, B. Wu, T. Zhang, L. Ma, G. Gambardella, M. Blott, L. Lavagno, K. A. Vissers, J. Wawrzynek, and K. Keutzer. Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded FPGAs. In *FPGA*, 2019.

[63] J. Zhang and J. Li. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *FPGA*, 2017.

[64] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He. Comba: A comprehensive model-based analysis framework for high level synthesis of real applications. In *ICCAD*, 2017.

[65] R. Zhao, W. Song, W. Zhang, T. Xing, J. Lin, M. B. Srivastava, R. Gupta, and Z. Zhang. Accelerating binarized convolutional neural networks with software-programmable FPGAs. In *FPGA*, 2017.

[66] 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.

[67] H. R. Zohouri, A. Podobas, and S. Matsuoka. Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. In *FPGA*, 2018.