FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs

Shuo Wang, Yun Liang*
Center for Energy-Efficient Computing and Applications (CECA), School of EECS, Peking University, China
{shvowang, ericlyun}@pku.edu.cn

Wei Zhang
Hong Kong University of Science and Technology
wei.zhang@ust.hk

ABSTRACT

The recent adoption of OpenCL programming model by FPGA vendors has realized the function portability of OpenCL workloads on FPGA. However, the poor performance portability prevents its wide adoption. To harness the power of FPGAs using OpenCL programming model, it is advantageous to design an analytical performance model to estimate the performance of OpenCL workloads on FPGAs and provide insights into the performance bottlenecks of OpenCL model on FPGA architecture. To this end, this paper presents FlexCL, an analytical performance model for OpenCL workloads on flexible FPGAs. FlexCL estimates the overall performance by tightly coupling the off-chip global memory and on-chip computation models based on the communication mode. Experiments demonstrate that with respect to RTL-based implementation, the average of absolute error of FlexCL is 9.5% and 8.7% for the Rodinia and PolyBench suite, respectively. Moreover, FlexCL enables rapid exploration of the design space within seconds instead of hours or days.

1. INTRODUCTION

Driven by the threat of dark silicon, energy-efficient accelerators such as FPGAs, GPUs, ASICs have emerged as mainstream ingredients of computer systems [1]. Among the accelerators, FPGAs can be reprogrammed to create customized pipelines with high parallelism for dedicated applications, providing orders of magnitude performance and energy benefits compared to general purpose processors. Consequently, FPGAs have become an increasingly popular vehicle for accelerating many workloads. For example, Microsoft and Baidu use FPGAs to accelerate the Bing search [2] and deep learning models [3], respectively.

While the benefits are clear, unfortunately, traditional FPGA development requires hardware design expertise in register transfer level (RTL) implementation, which is very tedious and time-consuming [4, 5]. High-level synthesis (HLS) lowers the FPGA programming barrier by using high level languages such as C/C++ [4, 5]. But, in reality, even with the HLS tools, programmers still need to spend enormous effort to optimize their applications for better performance [6]. In addition, FPGA architecture inherently prefers parallel codes, which can be replicated on FPGAs easily by creating multiple instances. However, using C/C++ programming model increases the difficulty in detecting the parallelism for HLS tools. In general, the programmers have to identify the parallelization opportunities manually and use directives to force the HLS tools to perform the parallelization.

More recently, FPGA vendors such as Xilinx and Altera propose to use parallel programming model OpenCL for FPGAs to improve the programmability and productivity. OpenCL, which is designed for heterogeneous computing, is applicable to a wide range of platforms. Using OpenCL for FPGAs has another important attribute that is it allows easy extraction of parallelism as different levels of parallelism are represented explicitly using work-item and work-group, etc.

Programming FPGAs using OpenCL model seems promising, but it brings new challenges. First, mapping OpenCL workloads onto FPGA architecture is so esoteric that it is very challenging for engineers to reason about the performance bottlenecks and fully harness the FPGA computing power. There are many optimization parameters at the application and architectural levels, forming a large design space with non-trivial performance trade-offs to explore. Second, the OpenCL-to-FPGA flow can be extremely slow for synthesizing a single solution, ranging from hours to days, making manual design space exploration infeasible.

In this paper, we propose FlexCL, an analytical performance model for OpenCL workloads on FPGAs. FlexCL takes the OpenCL kernel as input and outputs its performance on FPGAs. The key idea here is an accurate and rapid evaluation of OpenCL designs onto FPGA architecture through an analytical model. FlexCL achieves this by systematically modeling the effects of OpenCL-to-FPGA optimizations including pipeline, parallelization, and global memory access patterns. For computation model, it takes a bottom-up approach, by building the models for processing elements first, followed by models for compute units, and kernels. For global memory access, FlexCL models different access patterns. Finally, FlexCL seamlessly integrates the memory and computation models based on the communication mode.

FlexCL mainly uses static analysis, which is orders of faster than the lengthy RTL-based synthesis flow. More importantly, FlexCL is a highly trustworthy model as it yields accurate performance prediction. FlexCL will facilitate early design space exploration when mapping OpenCL workloads onto FPGAs and help the designers to quickly identify the solutions subject to a user defined performance

*Corresponding author

DAC'17, June 18–22, 2017, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3061639.3062251

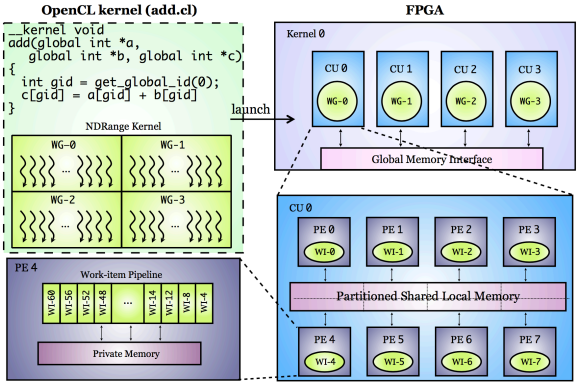


Figure 1: OpenCL mapping to FPGA. WG is work-group and WI is work-item.

constraint. FlexCL can also help to identify the performance bottlenecks on FPGAs, give code restructuring hints and make performance comparison across heterogenous architecture (GPUs v.s. FGPAs).

Experiments demonstrate that with respect to RTL-based implementation, the average of absolute error of FlexCL is 9.5% and 8.7% for Rodinia and PolyBench, respectively. FlexCL enables rapid exploration of optimization design space within seconds and identifies the solutions within 2.1% of the optimal solution.

2. BACKGROUND AND RELATED WORK

2.1 OpenCL Model to FPGA

An OpenCL program consists of a host program, which is in charge of one or more OpenCL devices, and a kernel program as shown in Figure 1. Host program dynamically invokes the OpenCL kernel executed on the accelerators (e.g. GPUs or FPGAs). In OpenCL, the basic unit of execution is a *work-item* and a group of work-items is bundled together to form a *work-group*. Multiple work-groups are combined to form a unit of execution called *NDRange*. For execution, the OpenCL program assumes that the underlying devices consisting of a number of compute units (CUs), which are further split into processing elements (PEs). When executing a kernel, work-groups are assigned to CUs, and work-items are assigned to PEs as shown in Figure 1. The number of work-groups in NDRange and the number of work-items in a work-group are specified by the programmer.

In this work, we use SDAccel flow to map OpenCL kernels onto Xilinx FPGA chips. SDAccel inherits almost the same OpenCL model from OpenCL standards. Figure 1 illustrates the mapping using *add.cl* kernel as an example. The PE corresponds to a pipeline of *add* operation which processes one work-item each time. A CU could replicate multiple PE instances in parallel, and the local memory indicated by the *__local* directive is shared among the PEs within the CU where it resides. The hardware *kernel* on FPGA is a complete implementation of software OpenCL kernel. The kernel could be configured using multiple CUs allowing multiple work-groups to be processed simultaneously. The global memory is a DRAM memory used to transfer data between host and FPGA device via PCI-e.

A rich set of OpenCL optimization options is exposed in SDAccel, which can be easily enabled by simply adding directives to the OpenCL source code. For example, the programmer can use *work_item_pipeline* directive to enable work-item pipelining. Although some directives used in SDAccel are specifically designed for Xilinx FPGA, the same optimization methodology is also applied to other FPGA-based OpenCL toolchain, such as Altera's OpenCL SDK.

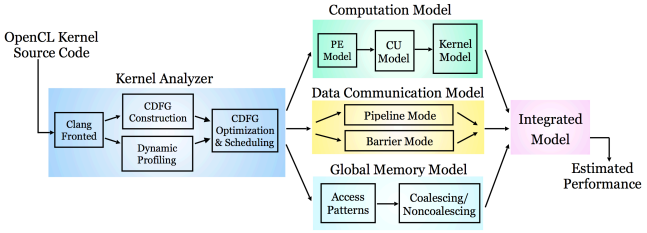


Figure 2: High-level Overview of FlexCL.

2.2 Related Work

Performance models and optimization strategies have been developed for ASIC or FPGA-based accelerators [6, 7, 8, 9, 10, 11, 12, 13]. Shao et al. [6] propose a pre-RTL performance modeling framework for ASICs. Their technique accepts trace generated from C program, not OpenCL program. Zhong et al. [14] propose an accurate performance estimation framework for HLS-based FPGA which enables faster design space exploration for accelerators compared to Xilinx HLS. But they use C program as input, not OpenCL and ignore the data communication between global memory and FPGA. Therefore, theses works cannot be directly applied to OpenCL-to-FGPA mapping due to the distinct OpenCL programming and optimization features.

Recently, OpenCL-based designs have been explored in [2, 15, 16]. [2] and [15] employ the OpenCL-based flow for the datacenter and CNN workloads using FPGAs respectively. However, their works focus on a specific workload and lack of a general performance model for accelerator design. Wang et al. [16] present an optimization framework for OpenCL programs for Altera FPGAs by employing a coarse-grained performance model. The optimization capability is fundamentally constrained by the model inaccuracy as it ignores important OpenCL-to-FPGA optimizations such as global memory access patterns, pipeline, parallelism, etc. Moreover, their step by step optimization approach examines a limited set of design space by assuming independence of different optimizations. Hence, such solution can easily lead to a solution stuck at local optima. The proposed FlexCL framework is accurate and can be also used to guide performance optimization for complex applications, such as iterative stencil algorithms [17].

3. PERFORMANCE MODEL

In this section, we start by introducing a high-level overview of FlexCL, followed by the details of each component.

3.1 High-Level Overview

Figure 2 presents the high-level flow of FlexCL. It takes the original kernel written in OpenCL as input and outputs its performance on FPGAs. FlexCL first employs Clang as the compilation frontend that transforms the OpenCL code to LLVM IR. After that, FlexCL performs kernel analysis, which analyzes the code structure, collect the operation latency, and retain other information about the program execution. Kernel analysis is essential as the subsequent computation and memory models of FlexCL are driven by detailed modeling of operation scheduling and data transfer. After that, FlexCL performs systematic computation model.

More clearly, it builds models for PEs, CUs and kernel. For the global memory model, it models eight global memory access patterns. Finally, FlexCL estimates the performance of OpenCL kernel on FPGAs by integrating the computation and memory models based on the way that computation communicates with global memory.

3.2 Kernel Analysis

The LLVM IR obtained using Clang frontend is first transformed into control data flow graph (CDFG), which serves as an abstract representation of the original OpenCL kernel. In CDFG, each node represents an IR operation and nodes are connected by data and control dependency edges. On FPGAs, each IR corresponds to an IP core [12, 13]. We obtain its operation latency through micro-benchmark profiling and associate each node with its latency. To facilitate the subsequent analysis, we simplify CDFG by merging nodes that belong to the same basic block and merging basic blocks with complex control dependencies such as loops. Figure 3(c) shows an example of simplified CDFG derived from OpenCL kernel, where the lines with arrows represent the data dependencies between basic blocks. Each basic block is implemented as a specific circuit on FPGA. This allows the basic blocks without data dependencies among each other to be executed in parallel.

The trip counts of loops and the global memory access trace are two important parameters for the subsequent computation and global memory model respectively. We use dynamic profiling to collect them for the cases where the static analysis fails. The global memory access trace, which is a sequence of indexes of accessed data array in global memory, is then transformed into realistic global memory accesses to different global memory banks based on the data mapping scheme. Then, the global memory accesses are categorized into different patterns to calculate the global memory access latency. It is needed to note that the profiling overhead is very small and takes at most a few seconds because only a few work-groups are profiled in practice.

3.3 Computation Model

Based on the OpenCL-to-FPGA mapping in Figure 1, we take a bottom-up approach to build the computation model. More concretely, we build the models for PE first, followed by models for CUs, and the entire kernel. During this process, different types of resources including local memory, and DSPs are modeled under their constraints. The current OpenCL-to-FPGA synthesis flow from Xilinx and Altera exposes a rich set of optimizations for users to customize the computation. Broadly speaking, we classify the optimizations into three categories, pipeline optimization (work-item and work-group pipeline), parallelism optimization (PE and CU parallelism), and on-chip memory optimization. In the computation model, we model all these optimizations.

3.3.1 Processing Element Model

Work-items are assigned to PEs for execution. We model the execution of a work-item using its CDFG. For each basic block, the execution latency is determined by two factors, data dependency between operations and resource constraints. In this work, we employ a resource-aware priority-ordered list scheduling algorithm [18, 19, 20, 21] to quickly estimate the execution latency of each basic block. The input of the algorithm is a CDFG of a basic block. The resource constraints considered include the number of memory access ports and DSPs. This algorithm schedules the operations using as soon as possible (ASAP) policy. The output of the algorithm is the execution latency of the basic block.

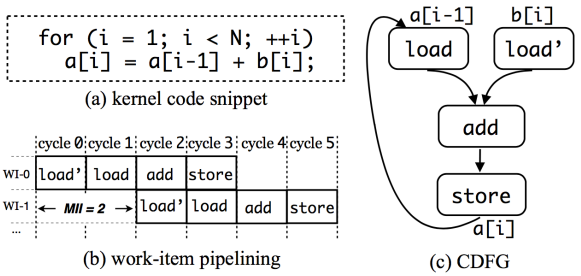


Figure 3: CDFG and work-item pipelining example.

Work-item pipeline aims to overlap the execution of multiple work-items in the same work-group in a pipeline manner for high throughput. The execution latency of work-items on a PE is determined by two factors: work-item initiation interval II^{wi}_{comp}, which is defined as the latency between the initiation of successive work-items, and the pipeline depth of the processing element D^{PE}_{comp}. We define N^{wg}_{wi} as the number of work-items in one work-group. Then, the execution latency of one work-group on a PE is computed as follows,

$$L^{PE}_{comp} = II^{wi}_{comp} \cdot (N^{wg}_{wi} - 1) + D^{PE}_{comp} \quad (1)$$

We derive II^{wi}_{comp} and D^{PE}_{comp} in two steps. In the first step, we compute the minimum initiation interval (MII). MII is defined as the lower bound of II^{wi}_{comp}, which depends on two factors, inter work-item data dependency and resource constraint [22, 23].

$$MII = \max(RecMII, ResMII) \quad (2)$$

where $RecMII$ is the recurrence constrained MII, and $ResMII$ is the resource constrained MII. $RecMII$ is introduced by inter work-item dependency. We derive $RecMII$ using the static data dependency method described in [22, 23]. As for $ResMII$, it is mainly limited by the on-chip local memory bandwidth and DSPs on FPGAs. Thus, we estimate $ResMII$ as follows,

$$ResMII = \max(ResMII_{mem}, ResMII_{dsp}) \quad (3)$$

where $ResMII_{mem}$ and $ResMII_{dsp}$ are the $ResMII$ constrained by the local memory bandwidth and DSPs, respectively. We compute $ResMII_{mem}$ as follows,

$$ResMII_{mem} = \max(\lceil \frac{N_{read}}{Port_{read}} \rceil, \lceil \frac{N_{write}}{Port_{write}} \rceil) \quad (4)$$

where N_{read} and N_{write} are the maximum number of read and write accesses to the local memory in the work-item pipeline. $Port_{read}$ and $Port_{write}$ are the number of available read and write ports of local memory in the PE, which can be computed by multiplying the number of banks of local memory and the number of ports per bank. Similarly, we can compute $ResMII_{dsp}$, which is the DSP-constrained MII.

For the second step, we employ the Swing Modulo Scheduling (SMS) algorithm [24] to estimate the II^{wi}_{comp} and D^{PE}_{comp}. The MII calculated from the previous step is used as the starting II^{wi}_{comp} value for SMS algorithm, and then SMS keeps refining the II^{wi}_{comp} until it satisfies all the resource constraints. SMS derives D^{PE}_{comp} by adding up all the latencies of the basic blocks along the critical path of the cor-

responding CDFG. The output of SMS algorithm is II^{wi}_{comp} and D^{PE}_{comp}.

Figure 3 (b) illustrates the work-item pipelining of the OpenCL code in Figure 3 (a). For this case, there is inter work-item data dependency, $II^{wi}_{comp} = MII = 2$ and $D^{PE}_{comp} = 6$.

3.3.2 Compute Unit Model

One CU can initiate multiple PEs to expose more parallelism by enabling *loop unrolling* pragma[1]. Similar to Equation 1, we compute the latency of CU as follows,

$$L^{CU}_{comp} = II^{wi}_{comp} \cdot \lceil \frac{N^{wg}_{wi} - N_{PE}}{N_{PE}} \rceil + D^{PE}_{comp} \quad (5)$$

where N_{PE} is the effective PE parallelism, which is constrained by the number of PEs P, and the local memory and DSP resources. The local memory and DSPs within a CU are shared by all its PEs, thus

$$N_{PE} = \min(P, \lceil \frac{Port_{read}}{N_{read} \cdot P} \rceil, \lceil \frac{Port_{write}}{N_{write} \cdot P} \rceil, \lceil \frac{Num_{dsp}}{N_{dsp} \cdot P} \rceil) \quad (6)$$

3.3.3 Kernel Computation Model

For a kernel with multiple CUs, the work-items are assigned to CUs at the granularity of work-group. Since there is no data dependency between work-groups in the OpenCL model, multiple work-groups can be processed on multiple CUs concurrently. Therefore, the computation latency of a kernel calculated as follows,

$$L^{kernel}_{comp} = L^{CU}_{comp} \cdot \lceil \frac{N^{kernel}_{wi}}{N^{wg}_{wi} \cdot N_{CU}} \rceil + C \cdot \Delta L^{schedule}_{comp} \quad (7)$$

where N^{kernel}_{wi} is the number of work-items contained in the kernel and N_{CU} is the effective CU parallelism, which is constrained by the number of CUs C, and work-group scheduling overhead $\Delta L^{schedule}_{comp}$.

$$N_{CU} = \min(C, \lceil \frac{L^{kernel}_{comp}}{\Delta L^{schedule}_{comp}} \rceil) \quad (8)$$

For an OpenCL kernel with multiple CUs, the work-groups are queued to be scheduled onto the idle CUs in a round-robin fashion as shown in Figure 1. Therefore, when the work-group scheduling overhead is considered, the ratio of L^{CU}_{comp} to $\Delta L^{schedule}_{comp}$ represents the maximum number of concurrent work-groups which is an upper bound for the effective CU parallelism.

3.4 Global Memory Model

In OpenCL model, global memory is the DRAM that resides off-chip. We estimate its access latency by modeling both DRAM architecture and access patterns [25].

Global memory has multiple banks and each bank is configured with a row buffer as cache in the bank. To reduce the bank conflicts, the data stored in the DRAM are arranged in byte-interleaved manner across all the banks. For each bank, the number of DRAM commands needed to handle a memory request varies for row-buffer hit or miss. If the memory request hits the row-buffer, only one read or write command is issued; otherwise, three DRAM commands are issued to that bank. Moreover, the memory access sequence also affects the latency of memory access latency. For example, the latency of a read request after a write is different from the read request after a read. To model the global memory access latency accurately, we model eight different global memory access patterns as shown in Table 1. The access sequence of data arrays within one work-item is profiled by executing a few work-groups of the kernel on CPU/GPU as discussed in Section 3.2. According to the byte-interleaved data mapping policy, we can get the global memory access patterns for each bank. The access latency of each global memory access pattern is profiled using micro-benchmarks.

Table 1: Global Memory Access Patterns And Parameters.

Global Memory Access Patterns	Access Latency	# Access
read(hit) access after read	ΔT^{hit}_{RAR}	N^{hit}_{RAR}
read(hit) access after write	ΔT^{hit}_{RAW}	N^{hit}_{RAW}
write(hit) access after read	ΔT^{hit}_{WAR}	N^{hit}_{WAR}
write(hit) access after write	ΔT^{hit}_{WAW}	N^{hit}_{WAW}
read(miss) access after read	ΔT^{miss}_{RAR}	N^{miss}_{RAR}
read(miss) access after write	ΔT^{miss}_{RAW}	N^{miss}_{RAW}
write(miss) access after read	ΔT^{miss}_{WAR}	N^{miss}_{WAR}
write(miss) access after write	ΔT^{miss}_{WAW}	N^{miss}_{WAW}

To fully utilize the global memory bandwidth, SDAccel will automatically coalesce the global memory accesses which are consecutive reads or writes. In this manner, the number of memory accesses is divided by a factor of coalescing degree $f = \frac{MemoryAccessUnitSize}{DataTypeBitWidth}$. For example, if there are 1024 consecutive global memory reads, the global memory access unit size is 512-bit and the accessed data type is *int* which is 32-bit, the number of memory accesses after memory coalescing should be $\frac{1024}{512/32} = 64$. The number of global memory accesses shown in the third column of Table 1 is the number after coalescing consecutive read or write accesses.

The global memory access latency of one work-item L^{wi}_{mem} is computed by summing the latency of different patterns within one work-item as follows,

$$\begin{aligned} L^{wi}_{mem} \quad = \quad & \Delta T^{hit}_{RAR} \cdot N^{hit}_{RAR} + \Delta T^{hit}_{RAW} \cdot N^{hit}_{RAW} \\ & + \Delta T^{hit}_{WAR} \cdot N^{hit}_{WAR} + \Delta T^{hit}_{WAW} \cdot N^{hit}_{WAW} \\ & + \Delta T^{miss}_{RAR} \cdot N^{miss}_{RAR} + \Delta T^{miss}_{RAW} \cdot N^{miss}_{RAW} \\ & + \Delta T^{miss}_{WAR} \cdot N^{miss}_{WAR} + \Delta T^{miss}_{WAW} \cdot N^{miss}_{WAW} \end{aligned} \quad (9)$$

3.5 Putting It All Together

Finally, we estimate the overall OpenCL kernel performance on FPGAs by integrating the computation and global memory model together based on the way that computation communicates with the global memory. The current OpenCL-to-FPGA synthesis flow allows the *barrier* and *pipeline* communication modes. We identify the communication mode by analyzing the OpenCL intrinsics.

Barrier Mode. In this mode, global memory access and computation operations are separated by barriers. In other words, there is no overlap between the computation and the global memory access. Thus, T_{kernel} is estimated by summing up the computation and memory latency as follows,

$$T_{kernel} = L^{wi}_{mem} \cdot N^{kernel}_{wi} + L^{kernel}_{comp}. \quad (10)$$

Pipeline Mode. In this mode, the global memory transfer is operated along with the computation in a pipeline manner. This could potentially overlap the computation

[1]Kernel vectorization enabled by using OpenCL vector types is also modeled based on PE parallelism, e.g. using 16 scalar PEs of *int* type to model one vectorized PE of *int16* vector type.

and memory operations leading to performance improvement. The work-items within the same work-group can be overlapped through pipelining, and thus the overall execution latency is calculated as,

$$T_{kernel} = (II_{wi} \cdot \lceil \frac{N_{wi}^{wg} - N_{PE}}{N_{PE}} \rceil + D_{comp}^{PE}) \cdot \lceil \frac{N_{wi}^{kernel}}{N_{wi}^{wg} \cdot N_{CU}} \rceil \quad (11)$$

where II_{wi} is work-item initiation interval after integrating global memory access and computation models, which is calculated as

$$II_{wi} = \max(L_{mem}^{wi}, II_{comp}^{wi}). \quad (12)$$

4. EXPERIMENT EVALUATION

4.1 Experiment Setup

All the experiments are conducted on Alpha Data's ADM-PCIE-7V3 board with a Xilinx Virtex-7 (XC7VX690T) FPGA and 16GB DDR3 memory with 8 banks and 1KB row-buffer size. The FPGA board is connected to the host via PCI-e 3.0 X8 interface. Xilinx SDAccel 2016.1 is used as the OpenCL SDK to synthesize the OpenCL kernel onto FPGAs. FlexCL is developed based on LLVM infrastructure, and Clang 3.4 is used as the OpenCL frontend. FlexCL is running on a server with Intel Core i7-4790 CPU.

We evaluate FlexCL using the entire benchmark suites Rodinia [26] and Polybench [27]. For each OpenCL kernel, we form a design space consisting of hundreds of design solutions by varying the parameters of optimizations, including work-group size, work-item and work-group pipeline, PE and CU parallelism, and data communication mode. For each solution, we evaluate the accuracy of FlexCL by comparing to two other techniques: System Run and SDAccel.

- System Run. Each kernel is synthesized to bitstream and implemented on FPGA using SDAccel. The performance obtained is the ground-true execution time, which is measured on the FPGA using the SDAccel runtime profiler.
- SDAccel. SDAccel also provides an HLS functionality, which synthesizes the OpenCL kernel to RTL design and gives a performance estimation in terms of cycles.

Finally, SDAccel and FlexCL estimate the performance in cycles. To obtain the performance in seconds, we have to multiply with the frequency. In this experiment, the frequency is set at 200MHz and all the benchmarks could be successfully synthesized.

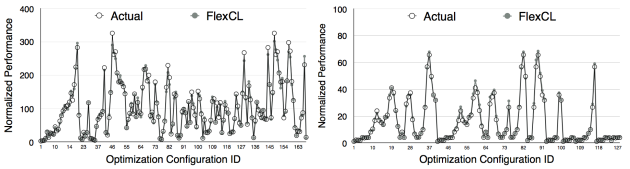
4.2 Accuracy Results

We first present the accuracy of FlexCL on Virtex-7, and then verify the robustness of FlexCL on a different platform.

Rodinia. Table 2 presents the performance estimation error and total estimation time for all the 45 kernels in Rodinia benchmark suite. The performance error in Table 2 is computed by comparing the estimation with the System Run. For each kernel, more than one hundred design solutions are tested. For each kernel, the performance error is the average error across all the design solutions. The estimation time is the total time for all the design solutions. Overall, FlexCL gives highly accurate prediction. It performs consistently well for all the kernels. The average performance estimation error of FlexCL is 9.5% for Rodinia suite.

Table 2: Performance Estimation Results of Rodinia.

Benchmark	Kernel Name	#Designs	Performance Estimation Error (%)		Total Design Exploration Time		
			SDAccel	FlexCL	System Run	SDAccel	FlexCL
backprop	layer	132	38.9	10.2	164 hrs.	80 mins.	19 secs.
	adjust	148	58.0	6.4	145 hrs.	65 mins.	35 secs.
bfs	bfs_1	128	71.2	8.2	59 hrs.	40 mins.	7 secs.
	bfs_2	164	84.9	6.7	83 hrs.	79 mins.	6 secs.
b+tree	findK	180	54.0	10.8	55 hrs.	67 mins.	12 secs.
	rangeK	128	34.2	9.2	47 hrs.	51 mins.	8 secs.
cfd	memset	132	68.4	10.3	98 hrs.	55 mins.	12 secs.
	initialize	132	71.3	12.2	84 hrs.	101 mins.	11 secs.
	compute	136	58.7	6.7	93 hrs.	76 mins.	15 secs.
	time_step	120	39.5	7.1	77 hrs.	69 mins.	7 secs.
dwt2d	compute	132	77.9	9.2	175 hrs.	126 mins.	16 secs.
	components	128	54.3	10.6	86 hrs.	80 mins.	10 secs.
	component	128	55.3	10.3	69 hrs.	74 mins.	15 secs.
	fdwt	128	59.5	14.2	180 hrs.	139 mins.	7 secs.
gaussian	fan1	128	64.0	12.4	146 hrs.	95 mins.	6 secs.
	fan2	128	46.1	16.1	135 hrs.	76 mins.	4 secs.
hotspot	hotspot	164	45.9	8.9	135 hrs.	108 mins.	19 secs.
hotspot3D	hotspot3D	128	58.2	7.8	65 hrs.	46 mins.	30 secs.
hybridsort	count	144	50.9	9.8	50 hrs.	119 mins.	12 secs.
	prefix	144	61.5	8.0	100 hrs.	106 mins.	12 secs.
	sort	144	58.9	8.6	70 hrs.	91 mins.	13 secs.
kmeans	center	132	52.6	13.4	171 hrs.	112 mins.	16 secs.
	swap	162	56.1	6.9	99 hrs.	108 mins.	14 secs.
lavaMD	lavaMD	128	56.4	11.4	140 hrs.	115 mins.	18 secs.
leukocyte	gicov	144	54.1	8.9	132 hrs.	145 mins.	8 secs.
	dilate	144	54.4	8.2	128 hrs.	137 mins.	5 secs.
	imgvf	144	58.5	7.5	123 hrs.	103 mins.	2 secs.
lud	diagonal	164	61.1	6.3	90 hrs.	95 mins.	10 secs.
	perimeter	164	50.0	10.8	103 hrs.	162 mins.	13 secs.
nn	nn	168	47.9	12.1	71 hrs.	34 mins.	8 secs.
nw	nw1	148	43.2	10.4	55 hrs.	56 mins.	19 secs.
	nw2	148	50.4	12.2	49 hrs.	37 mins.	13 secs.
particlefilter	find_index	128	52.5	6.0	96 hrs.	78 mins.	9 secs.
	normalize	128	54.8	11.4	125 hrs.	62 mins.	17 secs.
	sum	128	60.2	12.2	182 hrs.	49 mins.	12 secs.
	likelihood	128	44.7	8.6	109 hrs.	70 mins.	9 secs.
pathfinder	dynproc	148	76.3	13.2	109 hrs.	88 mins.	10 secs.
srad	extract	162	67.7	10.6	96 hrs.	151 mins.	13 secs.
	prepare	162	49.6	5.3	93 hrs.	113 mins.	15 secs.
	reduce	162	54.1	4.0	98 hrs.	91 mins.	11 secs.
	srad	162	82.5	10.6	104 hrs.	85 mins.	15 secs.
	srad2	162	61.1	7.6	109 hrs.	66 mins.	11 secs.
	compress	162	44.4	9.2	121 hrs.	78 mins.	12 secs.
streamcluster	memset	180	30.4	9.5	65 hrs.	81 mins.	7 secs.
	pgain	160	79.8	9.4	137 hrs.	103 mins.	9 secs.



(a) hotspot3D (b) nn

Figure 4: Performance estimation errors illustration.

SDAccel method fails to give estimation for some cases. In our experiments, for about 42% design solutions, SDAccel fails to return the results. It is due to several reasons. First, it lacks support for complex parallelism and memory access patterns. Second, it may take extremely long for certain cases. In our experiments, we stop the SDAccel process if the synthesis does not make any process after one hour. Hence, the average estimation error in Table 2 are computed for the surviving solutions only. However, the error of SDAccel is still high (30.4% - 84.9%) even we ruled out the failed cases. This high error is caused by a mixed effects including 1) underestimation of memory access latency, 2) conservative estimation of designs with relatively complex control dependency, and 3) ignorance of work-group scheduling overhead of multiple CUs.

Polybench. Compared with Rodinia benchmark suite, kernels in Polybench have simpler structures and are easy to analyze. Similarly, for each kernel in Polybench, we sweep different parameters, forming a design space with hundreds of design solutions. Then, we compare FlexCL with System Run results. The average absolute performance estimation error of FlexCL is 8.7%.

Estimation Error Analysis. Figure 4 plots the estimated performance by FlexCL and the actual performance for each design solution for *hotspot3D* and *nn* benchmarks.

We notice that FlexCL not only achieves low estimation error on average but also gives low estimation error for each design point. The inaccuracy of FlexCL are mainly from two sources, 1) estimation of IR operation latency and 2) estimation of memory access latency of each pattern. For the same IR operation, SDAccel may have multiple hardware implementation choices with different execution latencies. In the current toolchain, the hardware implementation can not be controlled by the programmer. In FlexCL, we address this problem by computing the average latency of an operation using micro-benchmarks. However, this might be different from the actual latency synthesized for real benchmark. As for memory access latency of each pattern in Table 1, similarly, is the average latency obtained through micro-benchmark profiling, which might be different from the actual latency.

Robustness Analysis. We also evaluate FlexCL on a different platform, NAS-120A development board with Xilinx KU060 FPGA, which uses the state-of-art UltraScale architecture. We use *HotSpot* and *pathfinder* from Rodinia as benchmarks, and the same design points explored previously are tested. The results show that the average performance estimation errors for *HotSpot* and *pathfinder* are only 9.7% and 13.6% respectively. This demonstrates that FlexCL is robust for different FPGAs platforms.

4.3 Design Space Exploration

We can use System Run and FlexCL to exhaustively explore the design space. Table 2 compares the exploration time of the two approaches. Compared to System Run, FlexCL accelerates the exploration process by more than 10,000X. System Run is slow because it has to go through the length synthesis process including logic synthesis, P&R, and bitstream generation. FlexCL enables rapid evaluation of design solutions using an analytical approach. It only involves lightweight static and dynamic analysis to analyze the OpenCL kernel. Overall, FlexCL explores the optimization design space within seconds and identifies the solutions within 2.1% of the optimal solution. Compared to the baseline unoptimized design, the best solution identified by FlexCL accelerates the performance by 273X on average.

Comparison. We can not directly compare with [16] as it fails to return absolute performance numbers and ignore important optimization features. Here, we compare FlexCL with exhaustive search and FlexCL using the heuristic search proposed in [16] for Polybench. The experimental results show that 96% design configurations found by FlexCL with exhaustive search are optimal while only 12% by [16] are optimal. Hence, an accurate performance model and systematic exploration are necessary for complex design space exploration.

5. CONCLUSION

OpenCL-based FPGA application design is still an emerging technology, and understanding the performance of OpenCL programming model on FPGA is difficult. To address this problem, this paper presents FlexCL, an analytical performance model for OpenCL workloads on flexible FPGAs. FlexCL leverages static and dynamic analysis to analyze the OpenCL kernels. It first develops systematic computation models. Then, it models different global memory access patterns. Finally, FlexCL estimates the overall performance by tightly coupling the memory and computation models based on the communication mode. Experiments demonstrate that with respect to RTL-based implementation, the average of absolute error of FlexCL is 9.5% and 8.7% for Rodinia and PolyBench suite, respectively.

6. ACKNOWLEDGMENTS

This work was supported by the Natural Science Foundation of China (No. 61672048). We thank all the anonymous reviewers for their feedback.

7. REFERENCES

[1] H. Esmaeilzadeh et al., "Dark Silicon and the End of Multicore Scaling," in *ISCA'11*, pp. 365–376, 2011.
[2] A. Putnam et al., "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in *ISCA'14*, pp. 13–24, 2014.
[3] J. Ouyang et al., "SDA: Software-Defined Accelerator for Large-Scale DNN Systems," in *HotChips'14*, 2014.
[4] J. Cong et al., "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *TCAD*, vol. 30, no. 4, pp. 473–491, 2011.
[5] Y. Liang et al., "High-level Synthesis: Productivity, Performance, and Software Constraints," *JECE*, 2012.
[6] Y. S. Shao et al., "Aladdin: A Pre-RTL, Power-performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures," in *ISCA '14*, pp. 97–108, 2014.
[7] B. So et al., "A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems," in *PLDI*, pp. 165–176, 2002.
[8] S. Bilavarn et al., "Design Space Pruning Through Early Estimations of Area/Delay Tradeoffs for FPGA Implementations," *TCAD*, vol. 25, no. 10, pp. 1950–1968, 2006.
[9] B. C. Schafer et al., "Divide and Conquer High-level Synthesis Design Space Exploration," *TODAES*, vol. 17, no. 3, pp. 29:1–29:19, 2012.
[10] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with High-Level Synthesis," in *DAC*, 2013.
[11] N. K. Pham et al., "Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis," in *DATE*, pp. 157–162, 2015.
[12] J. Villarreal et al., "Designing modular hardware accelerators in c with rocc 2.0," in *FCCM'10*.
[13] F. Vahid et al., "Warp Processing: Dynamic Translation of Binaries to FPGA Circuits," *Computer*, vol. 41, pp. 40–46, July 2008.
[14] G. Zhong et al., "Lin-analyzer: A High-level Performance Analysis Tool for FPGA-based Accelerators," in *DAC'16*, 2016.
[15] N. Suda et al., "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks," in *FPGA*, pp. 16–25, 2016.
[16] Z. Wang et al., "A Performance Analysis Framework for Optimizing OpenCL Applications on FPGAs," in *HPCA'16*, pp. 97–108, 2016.
[17] S. Wang et al., "A Comprehensive Framework for Synthesizing Stencil Algorithms on FPGAs using OpenCL Model," in *DAC'17*.
[18] A. Aho et al., *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.
[19] J. Cong and Z. Zhang, "An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation," in *DAC'06*, pp. 433–438, 2006.
[20] Z. Zhang and B. Liu, "SDC-based Modulo Scheduling for Pipeline Synthesis," in *ICCAD '13*, pp. 211–218, 2013.
[21] A. Canis et al., "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *FPGA '11*, pp. 33–36, 2011.
[22] T. M. Lattner, "An Implementation of Swing Modulo Scheduling with Extensions for Superblocks," Master's thesis, UIUC, 2005.
[23] B. R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," in *MICRO'94*, pp. 63–74, 1994.
[24] J. Llosa et al., "Swing module scheduling: a lifetime-sensitive approach," in *PACT'96*, pp. 80–86, 1996.
[25] H. Choi et al., "Memory Access Pattern-aware DRAM Performance Model for Multi-core Systems," in *ISPASS'11*.
[26] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC'09*, pp. 44–54, 2009.
[27] S. G.-G. et al., "Auto-tuning a high-level language targeted to GPU codes," in *InPar'12*.