

# SOFF: An OpenCL High-Level Synthesis Framework for FPGAs

Gangwon Jo\*, Heehoon Kim<sup>†</sup>, Jeesoo Lee<sup>†</sup>, and Jaejin Lee<sup>†</sup>

\**ManyCoreSoft*, Seoul 08826, Korea

Email: gangwon@manycorsoft.co.kr

<sup>†</sup>*Department of Computer Science and Engineering*

*Seoul National University*, Seoul 08826, Korea

Email: {heehoon, jeesoo}@aces.snu.ac.kr, jaejin@snu.ac.kr

**Abstract**—Recently, OpenCL has been emerging as a programming model for energy-efficient FPGA accelerators. However, the state-of-the-art OpenCL frameworks for FPGAs suffer from poor performance and usability. This paper proposes a high-level synthesis framework of OpenCL for FPGAs, called SOFF. It automatically synthesizes a datapath to execute many OpenCL kernel threads in a pipelined manner. It also synthesizes an efficient memory subsystem for the datapath based on the characteristics of OpenCL kernels. Unlike previous high-level synthesis techniques, we propose a formal way to handle variable-latency instructions, complex control flows, OpenCL barriers, and atomic operations that appear in real-world OpenCL kernels. SOFF is the first OpenCL framework that correctly compiles and executes all applications in the SPEC ACCEL benchmark suite except three applications that require more FPGA resources than are available. In addition, SOFF achieves the speedup of 1.33 over Intel FPGA SDK for OpenCL without any explicit user annotation or source code modification.

**Index Terms**—Accelerator architectures, FPGAs, high level synthesis, parallel programming, pipeline processing

## I. INTRODUCTION

As energy efficiency becomes one of the most important design goals for today’s high-performance computing (HPC) systems and datacenter servers, FPGAs are emerging as a new opportunity. Modern FPGAs from Intel and Xilinx provide millions of logic blocks and deliver the performance of several TFLOPS [27], [49]. Microsoft has adopted FPGAs in its datacenters to accelerate search engines [40], deep learning [37], data compression [19], etc. Amazon provides an EC2 compute instance that exploits FPGAs [1].

Recently, OpenCL (Open Computing Language) [32] has been considered as a promising programming model for FPGA accelerators. The two largest FPGA vendors, Intel (formerly

This work was supported in part by the National Research Foundation of Korea (NRF) grants (No. NRF-2016M3C4A7952587 and No. NRF-2019M3E4A1080386), by the BK21 Plus program for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU, No.21A20151113068) through NRF, and by the Institute for Information & communications Technology Promotion (IITP) grant (No. 2018-0-00581, CUDA Programming Environment for FPGA Clusters), all funded by the Ministry of Science and ICT (MSIT) of Korea. It was also supported in part by the Samsung Advanced Institute of Technology (SAIT) in Samsung Electronics Co., Ltd. ICT at Seoul National University provided research facilities for this study.

This work was done when Gangwon Jo was a Ph.D. student at Seoul National University.

Altera) and Xilinx, have started to release OpenCL frameworks for their own FPGAs [25], [48].

There are many advantages to having OpenCL as a language for high-level synthesis (HLS). First, every OpenCL kernel has spatial parallelism by its nature because many kernel threads can be executed simultaneously in a pipelined manner. Second, traditional hardware/software partitioning [7] is not required because an OpenCL application is already divided into compute-intensive kernels and the remaining part. Third, OpenCL excludes some high-level language features that are difficult to implement in FPGAs, such as recursion and function pointers. Fourth, it provides a three-level memory hierarchy that is naturally mapped to the memory hierarchy of modern FPGAs: external memory, embedded memory blocks, and registers. Finally, due to the success of GPGPU technology over the last decade, OpenCL has enjoyed a large software ecosystem. Thus, OpenCL-based HLS tools allow many existing applications to run on an FPGA without any modification.

However, it still requires significant effort from the research community to achieve high performance and good usability in OpenCL-based HLS. The state-of-the-art OpenCL frameworks for FPGAs are hindered by numerous bugs and unsupported features, require lots of explicit user annotations, and generate slow and inefficient circuits for real-world OpenCL applications. The most significant problem is that, no HLS technique for OpenCL kernels has been publicly proposed yet, while the internal implementation of commercial OpenCL frameworks remain black boxes. This hinders researchers from solving fundamental problems in OpenCL-based HLS.

This paper proposes a high-level synthesis framework of OpenCL for FPGAs, called *SOFF* (SNU OpenCL Framework for FPGAs). We present a complete datapath architecture for modern FPGAs to pipeline many threads in an OpenCL kernel. We also present a compilation process to automatically synthesize such a logic circuit. This paper extends the concept of run-time pipelining (Section II-A) proposed in previous studies on HLS and dataflow architectures to complete a compiler that covers all the features and corner cases in OpenCL. Especially, it describes a formal way to handle variable-latency instructions (Section IV-A and Section IV-B), complex control flows (Section IV-E), and synchronization primitives (Section IV-F) appeared in real-world OpenCL kernels.

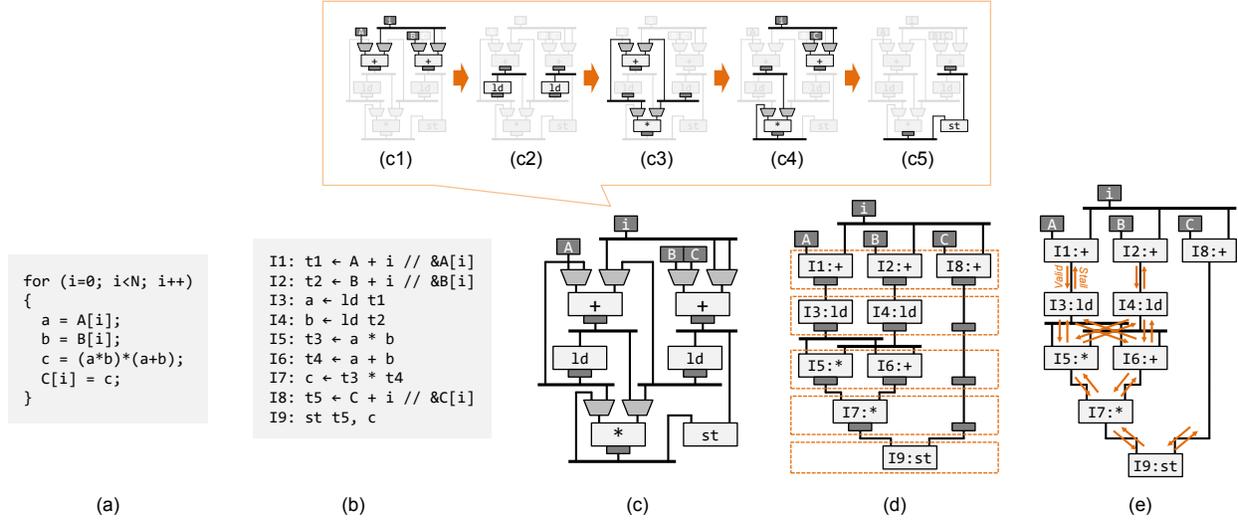


Fig. 1. (a) A for-loop, (b) instructions I1–I9 of the loop body, (c)–(e) three different datapath structures for executing the loop, and (c1)–(c5) a cycle-by-cycle process that executes a single loop iteration on the datapath in (c).

SOFF is the first OpenCL framework that correctly compiles and executes all applications in the SPEC ACCEL benchmark suite [11] except three applications that require more FPGA resources than are available. Moreover, it does not require any explicit user annotation and source code modification. The current version of SOFF is implemented and evaluated on an Intel Arria 10 FPGA. However, the proposed techniques are independent of a specific FPGA and can also be directly applied to other FPGAs such as Xilinx’s.

The contributions of the paper are as follows:

- It proposes techniques to automatically synthesize efficient datapaths for modern FPGAs from a given set of OpenCL kernels. The techniques include stall handling mechanisms and a deadlock prevention mechanism.
- It proposes adopting the control tree [44] as the intermediate representation for HLS to easily deal with complex control flows in a structured program.
- It proposes the design and implementations of OpenCL barriers and atomic operations in the datapath to correctly execute OpenCL kernels.
- It proposes an efficient memory subsystem for the datapath. The design is based on the characteristics of OpenCL kernels.
- The evaluation result of SOFF with SPEC ACCEL [11] and PolyBench [21] shows that it achieves the speedup of 1.33 over Intel FPGA SDK for OpenCL without any explicit user annotation or source code modification.

## II. BACKGROUND

In this section, we briefly introduce high-level synthesis (HLS) techniques and the OpenCL programming model.

### A. High-Level Synthesis

HLS techniques for FPGAs or other reconfigurable architectures that target imperative programming languages have

been proposed from the early 1990s [3], [6], [10], [16], [22], [23], [46], [47]. Despite their different target platforms and languages, their basic ideas are almost the same [7], [13], [20].

The primary goal of HLS is to generate a datapath for a given imperative program (usually a small piece of the entire application). An HLS tool places multiple functional units on the datapath (*allocation*). It determines when (*i.e.*, which clock cycle) each instruction in the target program begins and completes the execution (*scheduling*), and where (*i.e.*, on which functional unit) each instruction is executed (*binding*). Then, it connects the functional units by wires, buses, registers, and/or MUXes according to the result of scheduling and binding. In addition, it may generate multiple copies of the datapath as long as the capacity of the target FPGA allows, to execute multiple instances of the program (*e.g.*, threads and loop iterations) simultaneously. The datapaths are connected to other components such as memory controllers. Finally, a register-transfer level (RTL) description is delivered to a logic synthesis tool to reconfigure the FPGA.

There are three widely-used approaches to discover parallelism in an imperative program and to utilize many functional units in the datapath: *exploiting instruction-level parallelism (ILP)*, *compile-time pipelining*, and *run-time pipelining*.

1) *Exploiting ILP*: The first is to exploit ILP within a single basic block (or a variant such as a hyperblock [4] and a trace [17]). In every clock cycle, functional units in a datapath execute data-independent instructions in the same basic block in parallel. After all instructions in the current basic block are complete, the datapath executes the next basic block. Note that the same approach has been widely used for VLIW architectures [18].

Fig. 1 (c) illustrates a datapath that executes the for-loop in Fig. 1 (a). The loop body consists of nine instructions as shown in Fig. 1 (b). Suppose that the latency of every instruction is

a single cycle. As illustrated in Fig. 1 (c1)–(c5), the datapath computes  $\&A[i]$  and  $\&B[i]$  at the first cycle, loads  $A[i]$  and  $B[i]$  at the second cycle, computes  $t_3$  and  $t_4$  at the third cycle, computes  $c$  and  $\&C[i]$  at the fourth cycle, and finally stores  $c$  at the fifth cycle. Then, the next iteration of the loop begins its execution. However, this approach is hindered by limited instruction-level parallelism and time-varying instruction mixes of typical imperative programs. Moreover, it requires many MUXes because each functional unit needs to choose an appropriate input source for different instructions (e.g., the multiplier unit in Fig. 1 (c3) and (c4)). Such MUXes are costly in FPGAs [24].

2) *Compile-time pipelining*: On the other hand, if the target is a loop, multiple loop iterations can be executed in a pipelined manner instead of being executed one by one. Suppose that the loop body consists of a single basic block, and the latency of every instruction is known at compile time. If the loop has no loop-carried dependence, a pipelined datapath can be easily constructed by placing one functional unit for every instruction, connecting them according to data dependences between instructions, and inserting some registers between functional units to ensure that all possible paths from the entry to the exit have the same number of registers (i.e., all the paths are split into the same number of pipeline stages) [47].

Fig. 1 (d) shows an example of a pipelined datapath for the loop in Fig. 1 (a). Each of nine functional units corresponds to an individual instruction in Fig. 1 (b). MUXes are not required because each functional unit always receives input values from the same predecessors. Four registers are inserted between the functional units of  $I_8$  and  $I_9$  to split the datapath into five stages. As a result, the datapath can execute one loop iteration per cycle. If the loop has loop-carried dependences, or the capacity of the target FPGA is insufficient, more complex scheduling algorithms, such as modulo scheduling, can be used [5]. In this paper, we call this approach *compile-time pipelining* because every functional unit is assigned to exactly one of the pipeline stages at compile time.

3) *Run-time pipelining*: However, if the loop body contains a complex control flow, the datapath cannot be split into a fixed number of pipeline stages at compile time. This is also true for variable-latency instructions. In modern FPGAs, compute instructions usually have fixed latencies. Load/store instructions for embedded (on-chip) memory blocks, such as Intel’s M20K memory blocks and Xilinx’s block RAMs, can also have fixed latencies. However, many real-world applications need to use external memory (e.g., DRAM) to handle large data. One or more caches can be implemented in the FPGA using embedded memory blocks [9], [41] to improve the performance of external memory accesses. In this case, the latency of a load/store instruction may vary at run time, depending on whether the access is a cache hit or a miss. This is the reason why most previous compile-time pipelining techniques target innermost or perfectly nested loops, and do not employ a complex memory subsystem.

It is also possible to construct a pipelined datapath without

scheduling instructions at compile time. Instead, we can make each functional unit communicate with its neighbors using a handshaking protocol and execute an instruction whenever all of its operands are ready. This mechanism can be implemented either asynchronously (i.e., without the global clock) [2] or synchronously (i.e., with the global clock) [12], [30], [31]. We call this *run-time pipelining*. Fig. 1 (e) shows an example of such a datapath. Some recent high-level synthesis techniques adopt this approach to cover complex real-world applications [16], [41].

## B. OpenCL

OpenCL is a programming model for heterogeneous systems containing a general-purpose CPU and one or more accelerators, such as GPUs and FPGAs. An OpenCL application consists of two parts, a *host program* and a set of *kernels*. Compute-intensive and data-parallel tasks in the application are usually implemented as kernels and executed on an accelerator. The rest of the application becomes the host program and runs on the CPU. The host program is written in a typical high-level programming language, such as C, C++, Java, or Python. The kernels resemble usual C functions but are written in the OpenCL C programming language [33]. They take one or more arguments from the host program.

1) *Execution model*: The host program issues a kernel execution command to an accelerator using an OpenCL API function. Then, many instances of the kernel (i.e., kernel threads) are created and executed on the accelerator. Each kernel instance is called a *work-item*. In addition, work-items are equally divided into multiple small groups, and each group is called a *work-group*.

Each work-item has a unique *global ID*, and each work-group has a unique *work-group ID*. Similarly, each work-item has a unique *local ID* within its work-group. The host program needs to define an index space called an *NDRange* to execute the kernel. The *NDRange* determines the total number of work-items, the size of every work-group, and the rule for assigning an ID to each work-item. All work-items receive the same argument values. They are distinguished only by their IDs. OpenCL C provides built-in functions to obtain the current work-item ID.

2) *Memory hierarchy*: OpenCL provides three hierarchical memory address spaces for kernels – *global memory*, *local memory*, and *private memory*. The global memory is shared by the host program and all work-items. The host program invokes OpenCL API functions to allocate byte arrays (called buffers) in the global memory and to copy data between the main memory and the global memory. The host program can pass pointers to these arrays as kernel arguments. Work-items access the global memory through the pointers. On the other hand, the local memory is shared by work-items within the same work-group, and the private memory is private to each work-item. A kernel declares which of the two address spaces each local variable is allocated to. The host program cannot access the local memory and the private memory.

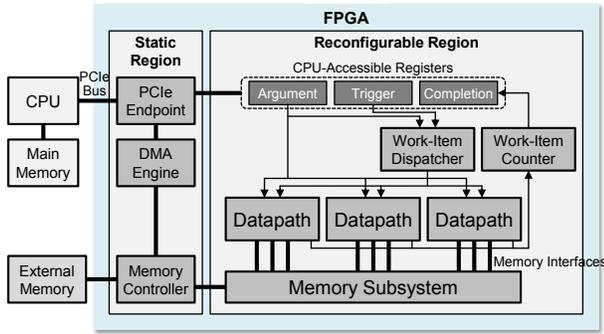


Fig. 2. The structure of the target platform and the block diagram of the circuits in the reconfigurable region.

3) *Memory consistency model*: OpenCL C uses a relaxed memory consistency model. All work-items can be executed independently except when the kernel uses a *work-group barrier* or an *atomic operation*. A work-group barrier (*i.e.*, a `barrier()` function) synchronizes all work-items within the same work-group and guarantees memory consistency between them. Note that it does not affect work-items in other work-groups. An atomic operation makes its result visible to all subsequent atomic operations from other work-items, even in different work-groups.

### C. Related Work

Many previous HLS techniques for OpenCL or similar programming languages (*e.g.*, CUDA) simply translate an OpenCL kernel into a sequential C program and deliver this to a traditional HLS tool [38], [39], [43]. They ignore the inherent spatial parallelism of OpenCL kernels and rely on traditional HLS tools to find new parallelism. Some of the previous studies synthesize one or more general-purpose cores in an FPGA and execute OpenCL kernels on the cores [8], [34]. This is not suitable for maximally utilizing the computing power of modern FPGAs.

Czajkowski *et al.* [15], [16] propose an OpenCL-to-FPGA compiler to generate a logic circuit that executes multiple work-items in a pipelined manner. They introduce basic-block modules that behave similarly to our basic pipelines in Section IV-B. They also mention deadlocks of inner loops very briefly in a single paragraph. However, they did not formally discuss the conditions of deadlocks nor propose an efficient prevention mechanism. Moreover, they did not address variable-latency instructions and work-group barriers, which are important issues in OpenCL-based HLS.

## III. OVERVIEW OF SOFF

This section describes the overall design and implementation of SOFF, including the target platform, circuit design, and compilation/execution flow.

### A. Target Platform

Fig. 2 illustrates the target platform of SOFF. It consists of a CPU and an FPGA connected by a PCIe bus. The FPGA

consists of logic blocks, DSP blocks, embedded memory blocks, and a programmable interconnect between them. In addition, it is connected to external memory that is larger and slower than the embedded memory. The external memory becomes the OpenCL global memory.

The FPGA is divided into a *static region* and a *reconfigurable region*. While the static region contains logic components that are commonly used for all OpenCL applications, the reconfigurable region contains a compiler-generated circuit dedicated to executing kernels in a single application. Once the target platform boots, the static region will not be changed anymore. On the other hand, the reconfigurable region is modified when a new application runs. Such partial reconfiguration is widely supported in modern FPGAs.

The static region contains a PCIe endpoint that communicates with a device driver running on the CPU. The PCIe endpoint enables the device driver to directly read and write some registers inside the reconfigurable region and to send DMA requests between the main memory and FPGA's external memory. The actual data transfer is done by a DMA engine in the static region. These features can be easily implemented using IP cores provided by the FPGA vendor.

### B. Reconfigurable Region Design

Fig. 2 also shows the block diagram of the circuits in the reconfigurable region. Suppose that the target OpenCL application has only one kernel. The reconfigurable region contains a *work-item dispatcher*, one or more copies of a *datapath*, a *memory subsystem*, and a *work-item counter*. Later in Section III-C, we will describe how the number of datapath copies is determined. The reconfigurable region also has an *argument register*, a *trigger register*, and a *completion register*. These registers are accessible by the device driver.

The values of kernel arguments and the definition of the NDRange (represented by seven integers) are assumed to be stored in the argument register. Once the trigger register is set to one, kernel execution begins. The work-item dispatcher distributes work-items to the datapaths by work-groups, *i.e.*, work-items in the same work-group are sent to the same datapath. It first assigns one work-group to each datapath. Then, it sends the IDs of every work-item in the work-group to the corresponding datapath, one by one, in every cycle unless the entry of the datapath is temporarily stalled. After all work-items in the work-group are sent, it assigns a new work-group to the datapath. This continues until all work-groups are assigned to the datapaths. Such work-group-level distribution helps work-group barriers and local memory to be easily implemented.

Each datapath executes the assigned work-items in a pipelined manner. It stores work-item private variables and intermediate values in its own registers. In addition, it exposes one or more memory interfaces to the memory subsystem. These interfaces are used to issue load/store requests to the global and local memory, and to receive the responses of load requests. The memory subsystem contains many embedded memory blocks that are used as the OpenCL local memory

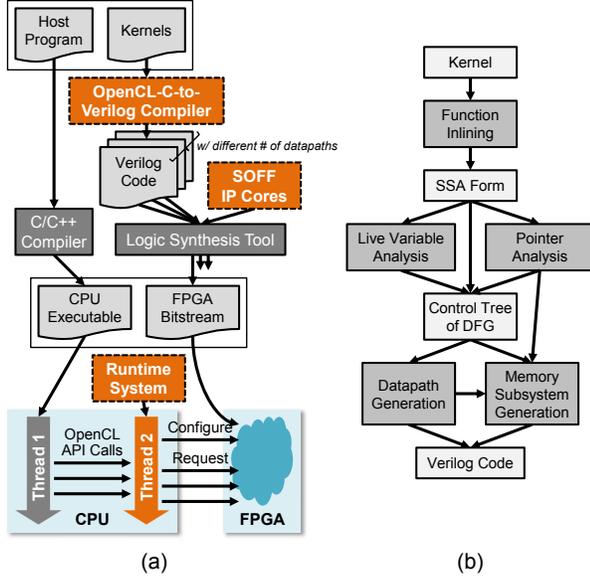


Fig. 3. (a) The process of compiling and executing an OpenCL application and (b) the detailed compilation flow. The dashed components in (a) are the contributions of the paper.

and as caches of the global memory. It is also connected to an external memory controller in the static region to handle cache misses. The internal implementations of the datapath and the memory subsystem are described in Section IV and Section V, respectively.

The work-item counter is incremented whenever a work-item finishes its execution in one of the datapaths. If the work-item counter reaches the total number of work-items in the NDRange, a cache flush signal is sent to the memory subsystem to write all dirty cache lines back to the external memory, and the completion register is set to one. The host program waits for the completion of the kernel execution by polling the completion register.

When the target OpenCL application has multiple kernels, the reconfigurable region accommodates multiple aforementioned circuits, one for each kernel, at the same time. It also contains a *kernel pointer register* that is accessible by the device driver. Only the circuit specified by the kernel pointer is enabled while all other circuits are disabled even if the trigger register is set to one. If the capacity of the target FPGA is not sufficiently large to contain multiple kernels, the reconfigurable region contains only the circuit for a single kernel. Then, SOFF modifies the reconfigurable region whenever the application launches a different kernel. Note that such partial reconfiguration takes only a few seconds in modern FPGAs.

### C. Compilation and Execution Flows

Fig. 3 (a) shows how an OpenCL application is compiled and executed by SOFF. First, the *OpenCL-C-to-Verilog compiler* of SOFF generates an RTL description of the reconfigurable region from the source code of all kernels in the

application. The result is written in Verilog and contains instances of many *SOFF IP cores*. The IP cores are basic building blocks of datapaths and memory subsystems. They have the same interface across different target FPGAs but may be implemented in a target-dependent manner (e.g., using IP cores provided by vendors). The RTL description and the SOFF IP cores are delivered to a logic synthesis tool such as Intel Quartus Prime [26] and Xilinx Vivado [50] to generate a bitstream for the reconfigurable region. Although OpenCL supports both online and offline compilation for kernels (i.e., kernels can be compiled either before or during run time), SOFF supports only the offline compilation because synthesizing a circuit may take several hours.

It is difficult to predict the maximum possible number of datapaths prior to the synthesis of the circuit. Thus, SOFF generates various RTL descriptions with different numbers of datapaths and attempts to synthesize all of them using the logic synthesis tool. It then chooses the one with the largest number of datapaths among those that are successfully synthesized.

1) *SOFF runtime*: The host program of an OpenCL application runs on a CPU together with the *runtime system* of SOFF. The runtime is a user-level library that implements OpenCL API functions invoked by the host program. It configures the reconfigurable region with the pre-built bitstream of the application. Then, it interacts with the device driver introduced in Section III-A to request data transfers between the main memory and the FPGA's global memory (i.e., the external memory) and to execute kernels on the FPGA. The runtime writes to the argument register, kernel pointer, and trigger register and checks the completion register for every kernel execution. It also has a simple memory allocator to manage the FPGA's global memory.

2) *Code generation*: Fig. 3 (b) illustrates the compilation flow in detail. First, each OpenCL kernel is represented as a static single assignment (SSA) form [14]. Every scalar variable, vector element, structure field, or array (which is treated as a big single variable) allocated in the private memory is replaced with an SSA variable unless its address is ever taken. All user-defined function calls in the kernel are inlined because it is difficult to implement function calls in an FPGA. A work-group barrier is regarded as a basic block leader (i.e., the first instruction of a basic block) and may split a sequence of instructions into two basic blocks (e.g., basic blocks B3 and B4 in Fig. 4 (a)).

Then, each basic block in the kernel is represented by a data flow graph (DFG). A DFG is an acyclic graph in which every node corresponds to an instruction and every edge corresponds to a data dependence between two instructions. For convenience, we introduce two arbitrary nodes – a *source* and a *sink* – in every DFG. The source produces all live-in SSA variables of the basic block while the sink consumes all live-out variables. Note that a DFG has been widely adopted as an intermediate representation in previous HLS techniques [7]. Fig. 4 (b) and (d) show DFGs for basic blocks B3 and B7 in Fig. 4 (a), respectively.

After constructing a DFG, SOFF adds edges to the DFG

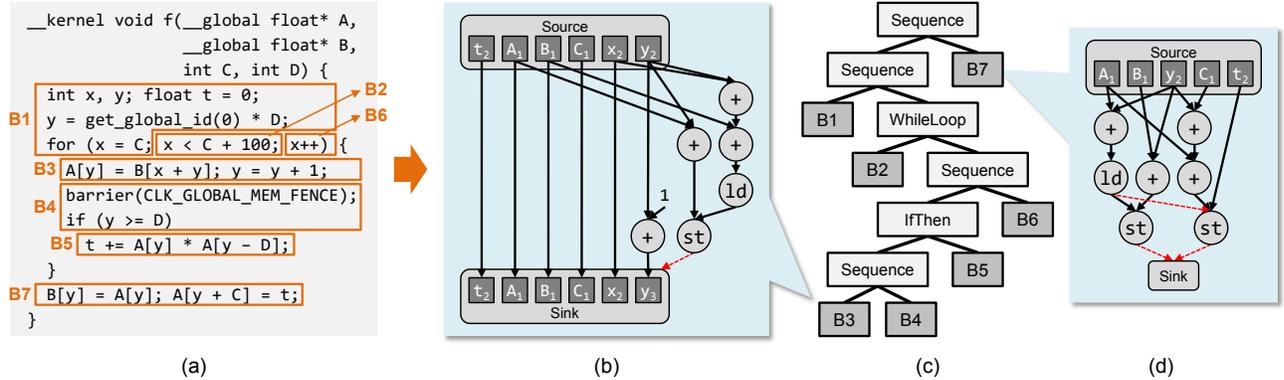


Fig. 4. (a) An OpenCL kernel consisting of seven basic blocks B1–B7, (b) the DFG for basic block B3, (c) the structure of the entire control tree, and (d) the DFG for basic block B7.

that represent possible anti- and output dependences between memory accesses because edges in the original DFG represent only true (flow) dependences between the nodes. The anti- and output dependence edges are treated as normal DFG edges that transfer the data of no size. SOFF conservatively assumes that two memory accesses refer to the same location if their addresses point to the same buffer [45]. For example, in Fig. 4 (d), an edge (in red) from the load node for  $A[y]$  to the store node for  $A[y + C]$  is inserted because they have a possible anti-dependence from the load to the store.

In addition, every memory access is connected to the sink node to ensure its completion, unless it has a subsequent data-dependent node. For example, the two stores in Fig. 4 (d) are connected to the sink with edges (in red). As a result, the DFG correctly represents the partial execution order of all the nodes it contains.

Finally, all basic blocks are hierarchically grouped as a control tree [35]. Fig. 4 (c) shows a control tree of the kernel in Fig. 4 (a). The root of the tree is the entire kernel and the leaves of the tree are individual basic blocks. Every node between the root and a leaf node represents a control-flow construct of a structured program, specifically, one of `Sequence`, `IfThen`, `IfThenElse`, `SelfLoop`, `WhileLoop`, `ProperInterval`, and `NaturalLoop` [44]. `ProperInterval` indicates a general single-entry, multiple-exit acyclic control-flow graph, and `NaturalLoop` indicates a general multiple-exit natural loop. These two nodes appear in a control tree if the target kernel contains `continue`, `break`, and `return` statements.

We assume that an OpenCL kernel is a structured program that can always be represented by a control tree (*e.g.*, it does not have any `goto`-like control-flow construct). This is true for almost all real-world OpenCL applications. The control tree will be exploited to recursively construct a pipelined datapath.

#### IV. DATAPATHS

In this section, we describe the architecture of individual datapaths.

##### A. Functional Units

As mentioned in Section II-A, functional units are the basic building blocks of a datapath. Each functional unit executes an individual instruction in the kernel. It *consumes* operands of the instruction and *produces* the result after several to tens of clock cycles. The implementations of functional units depend on the target FPGA and are provided as SOFF IP cores.

Every functional unit is fully pipelined to increase the operating clock frequency of the datapath and to take advantage of pipelined accesses to DRAM. Thus, it simultaneously executes the same instruction from different work-items in a pipelined manner. It consumes a different input in every clock cycle, and produces outputs in the order in which the inputs are given. We say a functional unit *holds* a work-item when the input of its instruction was consumed before the current clock cycle, and the output is produced in the current clock cycle or not yet produced (*i.e.*, either the result or an intermediate result for the instruction of the work-item is stored in the functional unit).

SOFF defines a *near-maximum latency*  $L_F$  for each functional unit  $F$ . If  $F$  has a fixed latency,  $L_F$  simply indicates that. Otherwise, the value of  $L_F$  is properly (empirically) determined so that most of the work-items can complete the corresponding instruction in less than  $L_F$  clock cycles. This value will be used to minimize functional unit stalls (Section IV-B) and to prevent deadlocks (Section IV-E).

##### B. Basic Pipelines

A *basic pipeline* is a circuit that executes a single basic block. It consumes live-in SSA variables of the basic block, and produces the corresponding live-out SSA variables after a while. It contains multiple functional units, one for each DFG node (*i.e.*, each instruction in the basic block). The connection between the functional units is isomorphic to the DFG. The output of a predecessor functional unit becomes an input to its successor functional unit.

The functional unit corresponding to a source node (mentioned in Section III-C) simply distributes the values of the live-in SSA variables to its successors. The functional unit

corresponding to a sink node aggregates the outputs of its predecessors and produces them as the output of the entire basic pipeline.

A basic pipeline can be implemented using the synchronous run-time pipelining scheme introduced in Section II-A. A functional unit consumes its inputs only when it verifies that all of its predecessors have produced their outputs. Similarly, a functional unit produces a new output only after it verifies that all of its successors have consumed its previous output. Until then, it continues to produce the previous output. For handshaking between predecessors and successors, the protocol proposed by Cortadella *et al.* [12] is used.

In the rest of the paper, a path from the source functional unit to the sink functional unit of a basic pipeline is called a *source-sink path*. All possible source-sink paths in a basic pipeline hold the same set of work-items in a certain clock cycle. It will be said that the basic pipeline *holds* these work-items.

### C. Handling Pipeline Stalls

We say a functional unit temporarily stalls when its pipeline is full and thus it cannot consume a new input in the current clock cycle. If a functional unit stalls, its predecessors may subsequently stall. There is at least one cycle delay between the stall of a functional unit and that of its predecessors (*i.e.*, the predecessors recognize the stall in the next clock cycle.) Without such a delay, a global stall signal, which is not scalable, needs to be introduced. Each functional unit contains an additional register to maintain its output when its successor begins to stall and it does not recognize the stall yet. A functional unit is restored from the stall when its output is consumed by all of its successors.

There are two types of stalls in SOFF: **Case 1** and **Case 2**. The **Case 1** stall occurs if a variable-latency instruction (*e.g.*, a memory load/store instruction) takes too long to finish. Then, the corresponding functional unit needs to hold more work-items as it consumes new inputs and finally, its pipeline becomes full.

To reduce **Case 1** stalls, SOFF enforces that every functional unit  $F$  never stalls when the number of work-items  $F$  holds is less than or equal to  $L_F$  (*i.e.*, the near-maximum latency). It is obvious that simple fixed-latency functional units (*e.g.*, adders and multipliers) meet this constraint. However, sophisticated variable-latency functional units and the memory subsystem need to be carefully implemented to satisfy such a constraint. To understand the deadlock prevention mechanism described in Section IV-E, keep in mind that  $F$  is designed to hold at least  $L_F + 1$  work-items when it stalls.

The **Case 2** stall occurs when two functional units, say  $F_1$  and  $F_2$ , share a common successor and one of them, say  $F_2$ , does not produce an output for a while. In this case,  $F_1$  cannot produce a new output until  $F_2$  finally produces an output. To reduce **Case 2** stalls, SOFF inserts some FIFO queues between functional units to make the sum of near-maximum latencies is the same on every source-sink path in the basic pipeline.

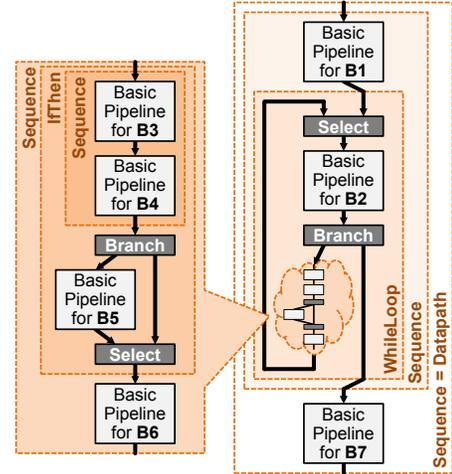


Fig. 5. A datapath for the OpenCL kernel in Fig. 4 (a).

The problem of adding a minimal amount of FIFO queues is formulated and solved by the integer linear programming (ILP) [36] in SOFF. Each variable in the ILP formulation represents the size of the FIFO queue between a pair of functional units.

### D. Hierarchical Datapath Generation

The entire datapath is constructed by hierarchically combining basic pipelines according to the structure of the control tree in Section III-C. For every non-leaf node in the control tree, SOFF merges smaller pipelines together, each of which comes from a different child node, with *glue logic* to form a larger pipeline. The role of the glue logic is to deliver a work-item from one pipeline to another, *i.e.*, to pass live variables of a work-item produced by one pipeline to the input of another pipeline.

Fig. 5 illustrates a datapath constructed from the control tree in Fig. 4 (c). It is built from seven basic pipelines corresponding to B1–B7. The dashed boxes represent compound pipelines corresponding to non-leaf nodes in the control tree (*i.e.*, **Sequence**, **IfThen**, and **WhileLoop** nodes).

The datapath contains *branch* and *select* glues between the pipelines. A branch glue delivers a work-item to one of its two successors, depending on whether the live-out condition value of its predecessor is zero or not. A select glue chooses only one work-item among several work-items coming from its predecessors and delivers it to its successor every clock cycle. SOFF uses the glue implementations proposed by Jacobson *et al.* [30].

Note that unlike functional units and basic pipelines, pipelines for non-leaf nodes may not produce outputs in the input order because different work-items can take different execution paths (*e.g.*, different branches, or a different number of loop iterations). Section IV-F will introduce new types of glue logic that restrict work-items from being freely reordered. This is necessary to consecutively deliver work-items in the same work-group to a work-group barrier. Also note that a

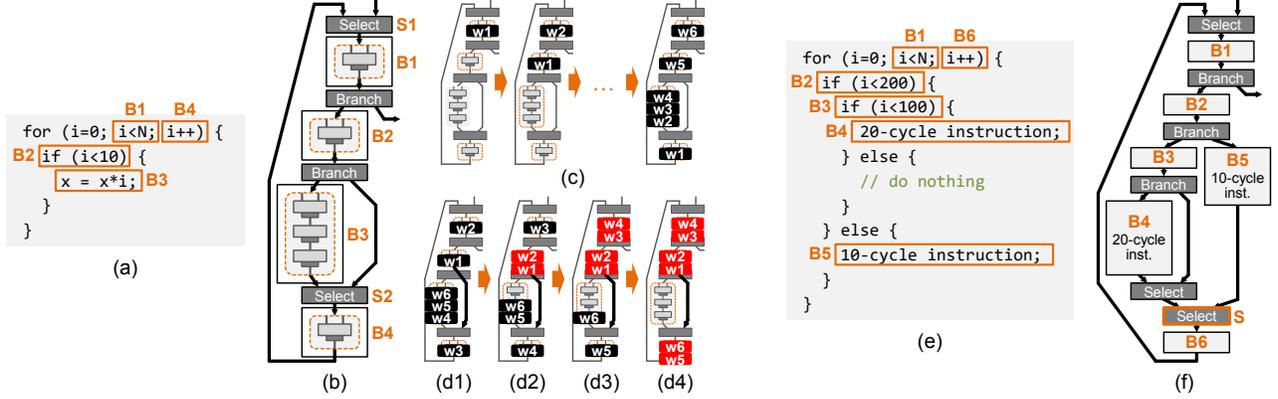


Fig. 6. An example of (a) a for-loop in a kernel, (b) a simplified pipeline to execute the loop, and (c), (d1)–(d4) a cycle-by-cycle process to cause the pipeline to be stuck in a deadlock; Another example of (e) a for-loop and (f) its pipeline whose deadlock cannot be avoided by assigning priorities to select glues at compile time.

partial pipeline of the datapath (*i.e.*, a pipeline that does not correspond to the root node) may execute the same work-item multiple times when the pipeline is in a loop or may never execute a particular work-item when the pipeline is in a branch.

### E. Deadlock Prevention

Each branch and select glue makes work-items move on regardless of the behavior of other glues. This may cause deadlocks in a loop. This section describes the conditions for a deadlock and introduces new types of glue logic to prevent the deadlock.

1) *Motivating examples:* Fig. 6 (b) illustrates a (simplified) pipeline to execute the for-loop in Fig. 6 (a). Every work-item takes the path B1-B2-B3-B4 during the first ten loop iterations, and then takes the path B1-B2-B4 in the remaining iterations.

Since the path B1-B2-B3-B4 has a latency of six, select glue S1 first puts six work-items w1–w6 into the loop body as shown in Fig. 6 (c). After w1 enters the eleventh iteration and arrives at B2 as shown in Fig. 6 (d1), select glue S2 needs to choose either the left-hand side or the right-hand side. Whenever S2 chooses a work-item from the left-hand side (*i.e.*, w4–w6), basic blocks B2, B1, and B4 stall in turn as shown in Fig. 6 (d2)–(d4). Finally, B1 waits for B2 to resume, B2 waits for B4, and B4 waits for B1. The loop results in a deadlock.

In this example, the deadlock can be easily avoided if S2 gives priority to B2 at compile time. Fig. 6 (e) and (f) show a more complex example in which deadlocks cannot be avoided by assigning priorities. Every work-item takes the path B1-B2-B3-B4-B6 during the first 100 iterations, the path B1-B2-B5-B6 during the next 100 iterations, and finally, the path B1-B2-B3-B6. If select glue S gives priority to the left-hand side, the loop is stuck in a deadlock at the 101st iteration. On the other hand, if S gives priority to the right-hand side, then the loop is stuck in a deadlock at the 201st iteration.

2) *Characteristics of basic pipelines:* Before introducing the deadlock prevention mechanism of SOFF, we first show some characteristics of individual basic pipelines.

**Definition.** A basic pipeline “strongly stalls” if it cannot consume a new input in the current clock cycle, and in addition, it can resume consuming inputs only after its output is consumed by a successor basic pipeline.

Fig. 7 (a) shows an example of a basic pipeline. It contains seven functional units, one for a DFG source node (Source), one for a DFG sink node (Sink), three for fixed-latency instructions (F1–F3), and two for variable-latency instructions (F4–F5).

If a Case 1 stall (introduced in Section IV-C) occurs at F4, a Case 2 stall subsequently occurs at F3, and then F1 and F2 also stall. As a result, the basic pipeline becomes unable to consume a new input, as shown in Fig. 7 (b1). In this case, however, the basic pipeline does not strongly stall because it can consume a new input two cycles after F4 produces the result of work-item w1, as shown in Fig. 7 (b2) and (b3), whether its output is consumed by a successor basic pipeline or not.

On the other hand, if a Case 1 stall occurs at F5, F1–F4 subsequently stall as shown in Fig. 7 (c). The basic pipeline also becomes unable to consume a new input. In this case, the basic pipeline strongly stalls because all the functional units can be restored from the stalls only after F5 produces an output and this is further consumed by a successor basic pipeline.

For a given source-sink path  $P$ , let  $l(P)$  be  $\sum_{F \in P} (L_F + 1)$  where  $F$  is a functional unit and  $L_F$  is its near-maximum latency. In addition, for a given basic pipeline  $B$ , let  $l_{min}(B)$  be the minimum of  $l(P)$  over all possible source-sink paths  $P$  in  $B$ . Our goal is to prove Theorem 1.

**Theorem 1.** For a given set of basic pipelines  $\mathbb{B}$ , if  $\mathbb{B}$  holds less than  $\sum_{B \in \mathbb{B}} l_{min}(B)$  work-items, at least one basic pipeline  $B \in \mathbb{B}$  does not strongly stall.

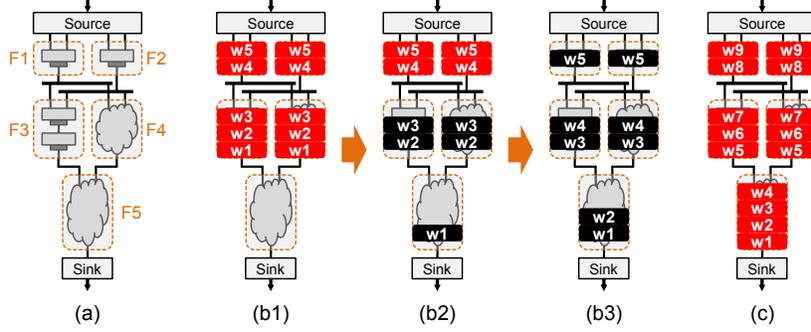


Fig. 7. (a) A basic pipeline, (b1)–(b3) an example that the basic pipeline cannot consume a new input but does not strongly stall, and (c) an example that the basic pipeline strongly stalls.

**Lemma 1.** *If a source-sink path  $P$  of a basic pipeline  $B$  holds less than  $l(P)$  work-items, there exists at least one functional unit  $F \in P$  that does not stall in the current clock cycle.*

*Proof.* Suppose that every functional unit  $F \in P$  stalls in the current clock cycle. As mentioned in Section IV-C, such a functional unit must hold at least  $L_F + 1$  work-items. Thus, the entire source-sink path  $P$  must hold at least  $l(P) = \sum_{F \in P} (L_F + 1)$  work-items. This contradicts the assumption.  $\square$

**Lemma 2.** *If a basic pipeline  $B$  holds less than  $l_{min}(B)$  work-items, there exists at least one functional unit  $F \in B$  such that (1)  $F$  does not stall in the current clock cycle, and (2)  $F$  is the source functional unit of  $B$  or every predecessor of  $F$  holds at least one work-item.*

*Proof.* Suppose an arbitrary source-sink path  $P$  in  $B$ .  $P$  holds exactly the same work-items than the entire  $B$  holds (*i.e.*, less than  $l_{min}(B)$  work-items). It follows that  $P$  holds less than  $l(P)$  work-items. By Lemma 1, there must be at least one functional unit  $F \in P$  that does not stall.

Starting from  $F$ , we check if  $F$  has a predecessor functional unit that does not hold any work-items, and of course, does not stall. If such a predecessor  $F'$  exists, we choose  $F'$  instead of  $F$  and repeat the same procedure.

Since  $B$  is acyclic and contains a finite number of functional units, we can finally reach a functional unit  $F$  such that it is the source functional unit or all of its predecessors hold one or more work-items.  $\square$

**Lemma 3.** *If a basic pipeline holds less than  $l_{min}(B)$  work-items,  $B$  never strongly stalls.*

*Proof.* This lemma means that, if a basic pipeline  $B$  holds less than  $l_{min}(B)$  work-items,  $B$  eventually consumes a new input even if its output is not consumed by a successor basic pipeline in the meantime.

We can always find a functional unit  $F \in B$  that satisfies the conditions in Lemma 2. If  $F$  is not the source functional unit of  $B$ , every predecessor of  $F$  will produce an output for the first work-item after a while. Then,  $F$  will consume an input by aggregating the outputs from its predecessors without

any intervention from other logic components. After that, we can choose a new functional unit again because  $B$  still holds the same number of work-items (*i.e.*, less than  $l_{min}(B)$  work-items).

Since  $B$  has a finite number of work-items and functional units, repeating such a process eventually makes the source functional unit of  $B$  to be chosen. At that time, the source functional unit does not stall, and the entire basic pipeline can now consume a new input.  $\square$

Theorem 1 easily follows from Lemma 3.

3) *Solution:* Assume the pipeline of a loop contains a cycle  $C$  of basic pipelines, branch glues, and select glues.  $C$  causes a deadlock if and only if (1) all basic pipelines of  $C$  strongly stall, and (2) every branch glue in  $C$  chooses a successor in  $C$ . Since the second condition is solely determined by the behavior of the OpenCL kernel and cannot be avoided, the first should not occur to prevent deadlocks. According to Theorem 1, SOFF prevents  $C$  from holding  $\sum_{B \in C} l_{min}(B)$  or more work-items.

It is very costly to control individual select glues at run time and thus to independently limit the number of work-items for every possible cycle in a loop pipeline. On the other hand, it is possible to find the cycle  $C$  with the minimum  $\{\sum_{B \in C} l_{min}(B)\} - 1$  and limit the total number of work-items in the loop pipeline to that minimum. However, this significantly lowers the utilization of the functional units in the loop if work-items usually take a longer execution path.

SOFF improves the latter. Let  $N_{max}$  and  $N_{min}$  be the maximum and the minimum of  $\{\sum_{B \in C} l_{min}(B)\} - 1$ , respectively. SOFF forces the number of work-items in the loop pipeline to be  $N_{max}$ , and at the same time, inserts a FIFO queue of size  $N_{max} - N_{min}$  at the loop back edge.

To enforce  $N_{max}$  work-items in the loop pipeline, SOFF attaches a *loop entrance* glue and a *loop exit* glue to the entry and exit of the loop pipeline, respectively. They share a work-item counter. If the counter reaches  $N_{max}$ , the loop entrance glue does not permit a new work-item to enter the loop pipeline until any work-item leaves the loop (*i.e.*, until the counter is decremented).

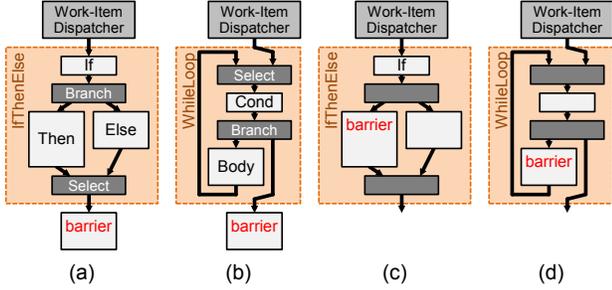


Fig. 8. Four possible control flow patterns between the work-item dispatcher and a work-group barrier.

## F. Synchronization Primitives

As described in Section II-B, OpenCL provides two types of synchronization primitives – work-group barriers and atomic operations. This section describes how functional units corresponding to these primitives are implemented in SOFF.

1) *Work-group barriers*: A functional unit for a work-group barrier is a special FIFO queue that consumes and produces all SSA variables live at the point of the barrier. It continues to store incoming live variables, and whenever all work-items in a specific work-group have arrived (*i.e.*, the number of work-items that arrive reaches the work-group size), it produces their live variables work-item by work-item. It assumes that all work-items in the same work-group arrive consecutively (*i.e.*, a work-item from a different work-group arrives only after all work-items from the previous work-group have arrived). Otherwise, it should have an indefinitely large storage to store live variables from many work-groups at the same time.

The work-item dispatcher described in Section III-B issues work-items in the same work-group consecutively. Thus, if the target OpenCL kernel is a straight-line code, the assumption above is satisfied for all work-group barriers. However, branches and loops between the entry of the datapath and a work-group barrier may change the order of work-items.

In the following examples, assume that there are eight work-items in total and they are grouped into two work-groups. Each work-item is called  $Xi$  where  $X$  denotes the work-group (one of **A** or **B**) and  $i$  denotes the local ID (one of 0, 1, 2, or 3).

We say a pipeline for a control tree node (*i.e.*, a part of the datapath) *preserves the work-group order* if the work-group of the  $i$ -th input is the same as that of the  $i$ -th output. For example, suppose that the order of the work-items that a pipeline consumes and produces are **A0**, **A1**, **B0**, **B1**, **B2** and **A1**, **A0**, **B1**, **B2**, **B0**. Then it preserves the work-group order. However, if the production order of the work-items is **A0**, **B0**, **B1**, **A1**, **B2**, it does not preserve the work-group order.

In Fig. 8 (a) and (b), a work-group barrier comes after a branch and a loop, respectively. In this case, we can make all work-items in the same work-group arrive at the barrier consecutively by preserving the work-group order in the branch and the loop. This can be recursively (inductively) implemented as follows:

- To make a branch (IfThen or IfThenElse) preserve the work-group order, SOFF makes its children (If, Then, and Else in Fig. 8 (a)) preserve the work-group order. In addition, it inserts a FIFO queue between the branch glue and the select glue. The branch glue enqueues the work-group ID of every incoming work-item. The select glue only delivers work-items from Then and Else whose work-group ID is the same as the first element of the queue. Since both Then and Else preserve the work-group order, either of them will produce a work-item whose work-group ID matches well.
- To make a single-entry, single-exit natural loop (SelfLoop or WhileLoop) preserve the work-group order, SOFF first checks whether the loop bound is an expression of kernel arguments and constant values (*i.e.*, all work-items iterate the loop the same number of times). If so, SOFF just makes its children (Cond and Body in Fig. 8 (b)) preserve the work-group order.
- Otherwise, SOFF attaches a *single work-group region (SWGR) entrance* glue and a *SWGR exit* glue to the entry and exit of the loop, respectively. Once a work-item enters the loop, these glues only permit work-items in the same work-group to enter the loop together, until the loop becomes empty. In this case, all descendants of the loop do not need to additionally preserve the work-group order. SOFF applies the same method to preserve the work-group order in ProperInterval and NaturalLoop.

In Fig. 8 (c), a work-group barrier is inside a branch. In this case, only the If pipeline needs to preserve the work-group order. Note that all work-items in the same work-group take the same branch according to the OpenCL programming model. Otherwise, the behavior of the OpenCL kernel is undefined.

In Fig. 8 (d), a work-group barrier is inside a loop. The problem is that the loop entrance glue may put only part of a work-group into the loop and cause the loop to be stuck into a deadlock. For example, assume work-items **B2** and **B3** are blocked by the loop entrance glue after six work-items **A0–A3**, **B0**, and **B1** enter the loop. Once **B0** or **B1** is enqueued to the barrier, the barrier stops permanently. Thus, in this case, SOFF replaces the loop entrance and exit glues by SWGR entrance and exit glues.

2) *Atomic operations*: Atomic operations are rarely used in real-world OpenCL applications because they hinder the performance of kernels on GPUs. Thus, SOFF conservatively implements functional units for atomic operations using locks.

All atomic operations that may access the same cache or the same local memory (will be described in Section V) share a set of 16 locks. At the beginning of the execution, a functional unit acquires the lock corresponding to the last four bits of its cache line address (*i.e.*,  $\text{lock}[(\text{addr} \gg 6) \% 16]$ ). At the end of the execution, it releases the lock. This is enough to minimize lock contention because typical OpenCL kernels have only a few atomic operations.

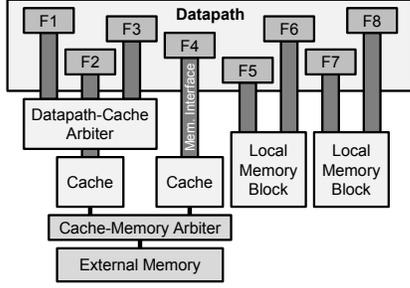


Fig. 9. An example of the memory subsystem. Functional units F1–F4 access the OpenCL global memory while F5–F8 access the OpenCL local memory.

## V. MEMORY SUBSYSTEM

This section describes the architecture of the memory subsystem. Fig. 9 illustrates an example block diagram of the memory subsystem used in SOFF. The role of the memory subsystem is to provide one memory interface to each functional unit that executes a memory load/store instruction (e.g., F1–F8 in Fig. 9). Unlike compute instructions, the functional units cannot execute memory load/store instructions by themselves. Instead, they issue memory access requests to the memory subsystem through the memory interfaces. Every memory interface in SOFF follows the Avalon Memory-Mapped (Avalon-MM) interface protocol [29].

### A. Caches

Functional units that access the global memory are connected to one or more caches in the memory subsystem. The caches are non-blocking in-order caches and thus can cooperate with (fully-pipelined) functional units well. SOFF uses simple direct-mapped, single-port caches because introducing complex caches may degrade the operating clock frequency of the entire circuit. The caches are further connected to FPGA’s external memory using a vendor-specific protocol (e.g., Avalon-MM or AXI [51]).

It is not feasible to handle all global memory access requests from numerous functional units in one single-port cache. Generally speaking, the more caches, the more memory access requests handled in every clock cycle. SOFF discovers two characteristics of OpenCL kernels to distribute irrelevant memory access requests to multiple separate caches.

- SOFF makes a separate cache for every OpenCL buffer. As mentioned in Section II-B, an OpenCL kernel accesses the global memory only through buffer pointers that are passed from the host program. If two memory accesses refer to different buffers, they never access the same memory location and need not share the same cache. SOFF chooses a proper cache for each functional unit according to the result of the pointer analysis.
- SOFF makes a separate cache for every datapath instance if the kernel does not contain an atomic operation. This is allowed because OpenCL does not guarantee memory consistency between different work-groups unless atomic operations are used, and all work-items in the same

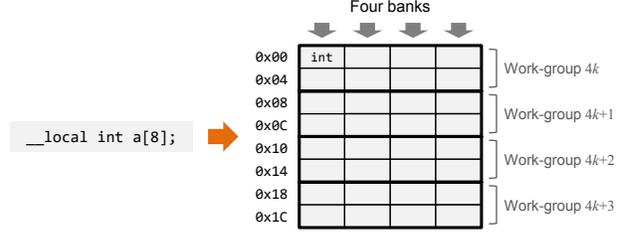


Fig. 10. An example of a local memory block for an array of eight `int` numbers.

work-group always run on the same datapath instance. In addition, this does not significantly degrade the cache hit ratio because different work-groups usually access different regions of the global memory.

If two or more functional units in the same datapath instance access the same buffer, they still need to share a single cache. In this case, a round-robin arbiter called a *datapath-cache arbiter* is inserted between the functional units and the cache. Functional units F1–F3 in Fig. 9 show an example.

As mentioned in Section IV-C, caches and datapath-cache arbiters should guarantee that every functional unit  $F$  across the memory interface never stalls while holding less than or equal to  $L_F$  work-items (i.e.,  $L_F$  pending memory access requests). Otherwise, they may cause additional pipeline stalls and deadlocks (Section IV-E).

### B. Local Memory Blocks

SOFF makes a *local memory block* for every variable or array allocated to the OpenCL local memory. Each functional unit that accesses a local memory variable is connected to the corresponding local memory block.

Fig. 10 illustrates a local memory block that stores an array of eight `int` numbers. SOFF determines the granularity of array accesses at compile time and makes the local memory block use 4-byte word addressing.

Unlike single-ported caches for the global memory, a local memory block provides a sufficient number of banks to enable all the connected functional units to concurrently access the block if no bank conflict occurs. Specifically, it creates  $2^{\lceil \log_2 N \rceil}$  banks where  $N$  is the number of connected functional units, and uses the last  $\lceil \log_2 N \rceil$  bits of the address to select a bank.

It is possible that different functional units in the datapath execute work-items in different work-groups at the same time. However, the local memory should be private to individual work-groups. Let  $L_{Datapath}$  be the maximum  $\sum_{F \in P} L_F$  among all possible paths  $P$  from the datapath entry to the exit. If the target OpenCL kernel uses the local memory, SOFF allows only  $\lceil L_{Datapath}/256 \rceil$  work-groups to enter the datapath together (i.e., the next  $\lceil L_{Datapath}/256 \rceil$  work-groups can enter the datapath after all the previous work-groups complete the execution). The number 256 is the most commonly occurring value for the work-group size.

TABLE I  
TARGET SYSTEMS

	System A	System B
FPGA board	Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA	Xilinx Virtex UltraScale+ FPGA VCU1525 Acceleration Development Kit
FPGA	Intel Arria 10 GX 10AX115N2F40E2LG	Xilinx XCVU9P-L2FSGD2104E
LUTs	1.150K logic elements	2.586K logic cells
DSPs	3,036 DSP blocks	6,840 DSP slices
Embedded memory	65.7 Mb	345.9 Mb
External memory	2x 4 GB DDR4	4x 16 GB DDR4
Interface	PCI Express Gen3 x8	PCI Express Gen3 x16
CPU	2x Intel Xeon Gold 6130 CPUs	
Main memory	256 GB	32 GB
OS	CentOS 7.6	Ubuntu 16.04
Logic synthesis tool	Quartus Prime Pro 17.1.1	Vivado Design Suite 2018.3
OpenCL framework	Intel FPGA SDK for OpenCL 17.1.1	SDAccel 2018.3

In addition, SOFF makes every local memory block store the variable of  $\lceil L_{Datapath}/256 \rceil$  different work-groups at the same time. For example, the local memory block in Fig. 10 has a capacity for four different work-groups, and the last two bits of the work-group ID are used as the first two bits of the address.

## VI. EVALUATION

In this section, we compare SOFF with two commercial OpenCL frameworks – Intel FPGA SDK for OpenCL [25] (Intel OpenCL) and Xilinx SDAccel [48].

### A. Experimental Setup

Table I describes two target systems **System A** and **System B**. We choose two state-of-the-art FPGA boards from Intel and Xilinx that officially support the vendors’ OpenCL frameworks. SOFF and Intel OpenCL run on **System A**, while Xilinx SDAccel runs on **System B**. Note that in terms of the number of logic cells and DSP slices, and the capacity of embedded memory, the Xilinx FPGA is much better than the Intel FPGA.

We use 19 applications in SPEC ACCEL [11] and 15 applications in PolyBench [21] as shown in Table II. The applications in SPEC ACCEL have various complicated features of OpenCL while the applications in PolyBench are quite simple. Since the benchmark applications mainly measure the execution time of OpenCL kernels, CPUs and other components in the two systems do not significantly impact the performance.

We implement the runtime system of SOFF (shown in Fig. 3) to support the Intel Arria 10 FPGA in **System A**. It uses an open-source OpenCL runtime system [42] as a code base, and also uses Intel’s Open Programmable Acceleration Engine library [28] as a device driver for the Intel FPGA. We also implement the SOFF IP cores for the Intel FPGA.

We empirically choose a proper near-maximum latency for every functional unit (e.g., 64 for global memory load/stores). We use global memory caches of size 64 KB as Intel OpenCL does on the same target FPGA. Note that the OpenCL-C-to-Verilog compiler of SOFF is independent of a specific FPGA

TABLE II  
APPLICATIONS USED.

Source	Applications	L <sup>†</sup>	B <sup>‡</sup>	A <sup>+</sup>	Intel	Xilinx	SOFF
SPEC ACCEL	101.tpacf	×	×	×	IA	CE	
	103.stencil					IA	
	104.lbm						
	110.fft						
	112.spmv						
	114.mriq				H		
	116.histo	×	×	×		CE	
	117.bfs	×	×	×		CE	
	118.cutcp	×	×				
	120.kmeans						
	121.lavamd	×	×		CE	CE	
	122.cfd				H		IR
	123.nw	×	×		H		
	124.hotspot	×	×		RE	CE	
	125.lud	×	×				
	126.ge						
	127.srad	×	×			CE	IR
	128.heartwall	×	×		CE	CE	IR
	140.bplustree	×	×		IA	IA	IR
Poly Bench	2dconv						
	3dconv						
	2mm					H	
	3mm						
	atax						
	bicg						
	gemm						
	gesummv						
	gramschm					H	
	mvt						
	syr2k					H	
	syrk						
	corr						
	covar					H	
	fdtd-2d					H	

<sup>†</sup>local memory; <sup>‡</sup>work-group barrier; <sup>+</sup>atomic operations.  
CE: compile errors; IA: incorrect answer; RE: run-time errors;  
H: hangs or takes too long; IR: insufficient FPGA resources.

and does not need to be modified when the target FPGA is changed.

### B. Functional Correctness

We try to compile and execute the benchmark applications using Intel OpenCL, Xilinx SDAccel, and SOFF. Table II summarizes the results. Intel OpenCL cannot execute 8 applications in SPEC ACCEL. Xilinx SDAccel cannot execute 9 applications in SPEC ACCEL and 5 applications in PolyBench. In particular, it yields compile errors in 7 applications because it does not support atomic operations, local memory accesses inside branches, and indirect pointers.

On the other hand, SOFF correctly executes 31 of 34 applications in SPEC ACCEL and PolyBench with no compile-time and run-time errors. The remaining three applications contain a kernel that fails to be compiled by SOFF, not because of any problem in SOFF, but because of the insufficient capacity of the target Intel FPGA. Even the circuit only for that kernel (i.e., not accommodating circuits for other kernels as described in Section III-B) does not fit into the target FPGA.

### C. Performance Comparison

Since both SOFF and Intel OpenCL can fully utilize the capacity of the target FPGA by executing work-items on multiple datapath instances, we mainly focus on the comparison of SOFF with Intel OpenCL.

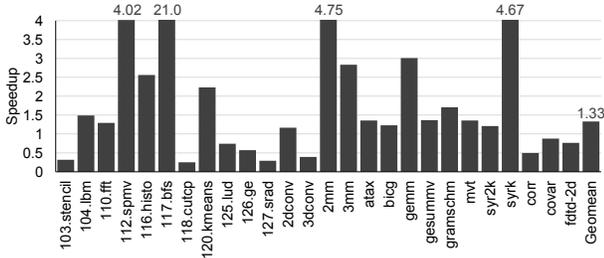


Fig. 11. The speedups of SOFF over Intel FPGA SDK for OpenCL.

Fig. 11 shows the speedups of SOFF over Intel OpenCL for the applications. SOFF automatically replicates the maximum number of datapath instances. For a fair comparison, we manually insert the `num_compute_units(N)` attribute in every application to also maximally replicate datapath instances in Intel OpenCL. SOFF outperforms Intel OpenCL in 17 of 26 applications, and achieves a speedup of 1.33 on average.

The static region and the IP cores of SOFF do not support the target Xilinx FPGA yet. Thus, just for reference, we compare SOFF with Xilinx SDAccel in indirect ways. First, we measure the execution time for running each application on the Intel FPGA using SOFF. Then, we compare this with the execution time required to run it on the Xilinx FPGA using Xilinx SDAccel. We call this comparison result Xilinx-vs-SOFF I.

Xilinx SDAccel uses only one datapath instance by default among tens of instances replicated in the target FPGA. The user needs to manually divide the target OpenCL kernel into multiple parts (in a data-parallel manner) to exploit the multiple datapath instances. With an optimistic assumption that Xilinx SDAccel achieves a linear speedup for every application using multiple datapath instances that it replicates, we extrapolate the execution time of each application and compare the result with the execution time required to run it on the Intel FPGA using SOFF. We call this comparison result Xilinx-vs-SOFF II.

As shown in Fig. 12 (a), Xilinx SDAccel is 25 times slower than SOFF despite using the better FPGA in System B. Fig. 12 (b) shows the speedup of SOFF over Xilinx SDAccel in the case of Xilinx-vs-SOFF II. Even with the optimistic assumption, SOFF is still 30% faster than Xilinx SDAccel.

We cannot analyze the reasons for the speedup or slowdown of individual applications in detail because Intel OpenCL and Xilinx SDAccel provide only limited information for the actual circuit layout. However, the results imply that the performance of the proposed framework and architecture is at least comparable to that of the state-of-the-art proprietary OpenCL frameworks and architectures.

## VII. CONCLUSIONS

In this paper, we propose how real-world OpenCL applications can be executed on FPGAs by the SOFF framework. The target FPGA contains one or more copies of the datapath. Each

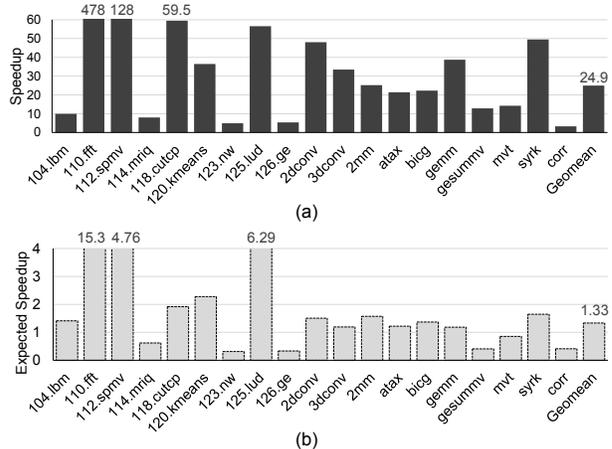


Fig. 12. (a) Xilinx-vs-SOFF I and (b) Xilinx-vs-SOFF II.

datapath executes multiple work-items in a pipelined manner. The datapath consists of basic pipelines. Each basic pipeline further consists of functional units, one for each instruction. Functional units communicate with each other according to the synchronous run-time pipelining scheme. SOFF reduces functional unit stalls, prevents deadlocks of inner loops, and preserves the work-group order for barriers, with the help of the control-tree-based intermediate representation and near-maximum latencies of functional units. The target FPGA also contains a memory subsystem that contains separate caches for different buffers and different datapaths to support as many memory requests as possible in every clock cycle.

We compare SOFF with the state-of-the-art proprietary OpenCL frameworks. The experimental results show that SOFF works correctly for a rich set of real-world applications and achieves comparable or better performance than the existing frameworks. We expect that SOFF can be a solid foundation for future studies on OpenCL-based HLS. The source code of SOFF can be downloaded from <http://aces.snu.ac.kr>.

## REFERENCES

- [1] Amazon, “Amazon EC2 F1 instances,” <https://aws.amazon.com/ec2/instance-types/f1/>.
- [2] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, “Spatial computation,” in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 14–26.
- [3] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, “The garp architecture and c compiler,” *IEEE Computer*, vol. 33, no. 4, pp. 62–69, 2000.
- [4] T. J. Callahan and J. Wawrzynek, “Instruction-level parallelism for reconfigurable computing,” in *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*, 1998, pp. 248–257.
- [5] T. J. Callahan and J. Wawrzynek, “Adapting software pipelining for reconfigurable computing,” in *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2000, pp. 57–64.
- [6] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski, “LegUp: High-level synthesis for FPGA-based processor/accelerator systems,” in *Proceedings of the 19th*

- ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2011, pp. 33–36.
- [7] J. M. P. Cardoso, P. C. Diniz, and M. Weinhardt, "Compiling for reconfigurable computing: A survey," *ACM Computing Surveys*, vol. 42, no. 4, pp. 13:1–13:65, 2010.
  - [8] E. Cartwright, S. Ma, D. Andrews, and M. Huang, "Creating HW/SW co-designed MPSoPCs from high level programming models," in *Proceedings of the 2011 International Conference on High Performance Computing & Simulation*, 2011, pp. 554–560.
  - [9] J. Choi, K. Nam, A. Canis, J. Anderson, S. Brown, and T. Czajkowski, "Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems," in *Proceedings of IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, pp. 17–24.
  - [10] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "Platform-based behavior-level and system-level synthesis," in *Proceedings of 2006 IEEE International SOC Conference*, 2006, pp. 199–202.
  - [11] S. P. E. Corporation, "SPEC ACCEL," <https://www.spec.org/accel/>.
  - [12] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *Proceedings of the 43rd Annual Design Automation Conference*, 2006, pp. 657–662.
  - [13] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.
  - [14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.
  - [15] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," in *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications*, 2012, pp. 531–534.
  - [16] T. S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D. P. Singh, and S. D. Brown, "OpenCL for FPGAs: Prototyping a compiler," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2012.
  - [17] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490, 1981.
  - [18] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983, pp. 140–150.
  - [19] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on FPGAs," in *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 52–59.
  - [20] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-level Synthesis: Introduction to Chip and System Design*. Norwell, MA, USA: Kluwer Academic Publishers, 1992.
  - [21] S. Grauer-Gray and L.-N. Pouchet, "PolyBench/GPU," <http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/GPU/>.
  - [22] Z. Guo, B. Buyukurt, W. Najjar, and K. Vissers, "Optimized generation of data-path from C codes for FPGAs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2005, pp. 112–117.
  - [23] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," in *Proceedings of 16th International Conference on VLSI Design*, 2003, pp. 461–466.
  - [24] S. Hadjis, A. Canis, J. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski, "Impact of FPGA architecture on resource sharing in high-level synthesis," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2012, pp. 111–114.
  - [25] Intel, "Intel FPGA SDK for OpenCL," <https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html>.
  - [26] Intel, "Intel Quartus Prime," <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html>.
  - [27] Intel, "Intel Stratix 10 fpgas overview," <https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html>.
  - [28] Intel, "Open programmable acceleration engine - documentation," <https://opae.github.io/>.
  - [29] Intel, "Avalon interface specifications," 2019, [https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl\\_avalon\\_spec.pdf](https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf).
  - [30] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers, "Synchronous interlocked pipelines," in *Proceedings of Eighth International Symposium on Asynchronous Circuits and Systems*, 2002, pp. 3–12.
  - [31] L. Josipovic, R. Ghosal, and P. Jenne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 127–136.
  - [32] Khronos Group, "OpenCL overview - the open standard for parallel programming of heterogeneous systems," <https://www.khronos.org/opencl/>.
  - [33] Khronos OpenCL Working Group, "The OpenCL specification," 2012, <https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf>.
  - [34] M. Lin, I. Lebedev, and J. Wawrzyniec, "Opencl: Low-power high-performance computing with reconfigurable devices," in *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, 2010, pp. 458–463.
  - [35] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
  - [36] K. G. Murty, *Linear Programming*. New York, NY, USA: John Wiley & Sons, 1983.
  - [37] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," Microsoft Research, Tech. Rep., 2015, <https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/>.
  - [38] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of platform architectures from OpenCL programs," in *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2011, pp. 186–193.
  - [39] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Proceedings of the 2009 IEEE 7th Symposium on Application Specific Processors*, 2009, pp. 35–42.
  - [40] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecture*, 2014, pp. 13–24.
  - [41] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig, "Performance and power of cache-based reconfigurable computing," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 395–405.
  - [42] Seoul National University, "SnuCL suite: OpenCL frameworks and tools for heterogeneous clusters," <http://snucl.snu.ac.kr>.
  - [43] K. Shagririthaya, K. Kepa, and P. Athanas, "Enabling development of OpenCL applications on FPGA platforms," in *Proceedings of the 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, 2013, pp. 26–30.
  - [44] M. Sharir, "Structural analysis: a new approach to flow analysis in optimizing compilers," *Computer Languages*, vol. 5, no. 3–4, pp. 141–153, 1980.
  - [45] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996, pp. 32–41.
  - [46] J. L. Tripp, K. D. Peterson, C. Ahrens, J. D. Poznanovic, and M. B. Gokhale, "Trident: An FPGA compiler framework for floating-point algorithms," in *Proceedings of 2005 International Conference on Field Programmable Logic and Applications*, 2005, pp. 317–322.
  - [47] M. Weinhardt and W. Luk, "Pipeline vectorization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 234–248, 2001.
  - [48] Xilinx, "SDAccel development environment," <https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html>.
  - [49] Xilinx, "Virtex UltraScale+," <https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html>.
  - [50] Xilinx, "Vivado design suite," <https://www.xilinx.com/products/design-tools/vivado.html>.
  - [51] Xilinx, "AXI reference guide," 2017, [https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_ref\\_guide/latest/ug1037-vivado-axi-reference-guide.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf).