ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA

Zhenyuan Ruan* Tong He* Bojie Li†‡ Peipei Zhou* Jason Cong*
*University of California, Los Angeles †Microsoft Research ‡University of Science and Technology of China
{zainryan, tonghe, memoryzpp, cong}@cs.ucla.edu, v-bojli@microsoft.com

***Abstract*—In recent years we have witnessed the emergence of the FPGA in many high-performance systems. This is due to FPGA's high reconfigurability and improved user-friendly programming environment. OpenCL, supported by major FPGA vendors, is a high-level programming platform that liberates hardware developers from having to deal with the complex and error-prone HDL development. While OpenCL exposes a GPU-like programming model, which is well-suited for compute-intensive tasks, in many state-of-art systems that deploy FPGA, we observe that the workloads are streaming-like, which is communication-intensive. This mismatch leads to low throughput and high end-to-end latency.**

In this paper, we propose ST-Accel, a new high-level programming platform for streaming applications on FPGA. It has the following advantages: (i) ST-Accel adopts the multiprocessing programming model to capture the inherent pipeline-level parallelism of streaming applications while reducing the end-to-end latency. (ii) A message-passing-based host/FPGA communication model is used to avoid the coherency issue of shared memory, thus enabling host/FPGA communication during kernel execution. (iii) ST-Accel provides a high-level abstraction for I/O devices to support direct I/O device access that eliminates the overhead of host CPU and reduces the I/O latency. (iv) ST-Accel enables the decoupled access/execute architecture to maximize the utilization of I/O devices. (v) The host/FPGA communication interface is redesigned to cater to the demands of both latency-critical and throughput-critical scenarios. The experimental results on the Amazon AWS cloud and local machine show that ST-Accel can achieve 1.6X$\sim$166X throughput and 1/3 latency for typical streaming workloads when compared to OpenCL.

I. Introduction

The limitation of instruction-level parallelism and the failure of continuing Dennard scaling is forcing computer architects to switch from single-core CPU to multi-core CPU [1]. However, more recently, the occurrence of dark silicon signals the end of multi-core scaling [2]. Architects have to rediscover the computing solution for higher performance and energy efficiency. Among various alternative computing engines, such as graphics processing unit (GPU), many integrated core (MIC), field-programmable gate array (FPGA), and application-specific integrated circuit (ASIC), FPGA shows great promise given its flexible reconfigurabilty and high power/cost efficiency. Currently, FPGA has been deployed in large-scale data centers [3] and is also available in cloud service [4].

Having an efficient high-level programming platform is the key to massively harnessing the power of FPGA. Traditionally, FPGA developers use HDL (*e.g.*, Verilog, VHDL) to describe the FPGA logic. Although it has a strong programing flexibility that allows the RTL description, HDL is extremely hard to code and debug and also lacks the support of host-side integration. In order to improve developers' efficiency, in recent years both Xilinx and Intel FPGA have released their OpenCL-based high-level programming toolchains. OpenCL shares many concepts of GPGPU and exposes the single instruction multiple thread (SIMT) programming model, which perfectly matches the compute-intensive workloads [5]. The host/FPGA communication is handled by the OpenCL library, enabling the developers to concentrate on the computational logics. Equipped with OpenCL, developers are able to achieve considerable performance speed-ups while reducing a large amount of programing effort for compute-intensive applications, *e.g.*, [6], [7], [8], [9].

In recent years FPGA-based acceleration has been actively pursued in a number of research communities, ranging from architecture [10], [11], to network [12], to database [13], [14], and more recently to system [15]. After careful examination of these works (detailed in Section II and III), we surprisingly found that most of them (if not all) belong to streaming applications, which are communication-intensive rather than computational-intensive. Since OpenCL was originally developed to accelerate the computational-intensive workloads, there is a mismatch between these workloads and OpenCL. For example, according to the results in Section VI, the OpenCL implementation of the hash table lookup may even have up to two magnitudes of performance degradation compared with the HDL implementation. This motivates us to analyze the source of inefficiency. In Section III we summarize the limitations of OpenCL, which are the root of inefficiency when applying OpenCL to the streaming workloads, as follows.

1) Inappropriate memory model
2) FPGA limited to role of slave
3) Synchronous DRAM access interface
4) Inefficient host/FPGA communication design

FPGA inherently is not constrained by these limitations; they are actually introduced by the abstraction of OpenCL and can be avoided if we develop the design in HDL. This raises our question: *Could it be possible to enjoy the programming convenience brought by the high-level abstraction while incurring only a few performance penalty?*

To answer this question, we present ST-Accel, a high-level programming platform for streaming applications on FPGA. Our design goal is to find an appropriate level of abstraction that enables high-level programming while providing enough programming support to cater to the demands of streaming workloads. The key design points of ST-Accel are illustrated in Section IV. First, ST-Accel adopts the multiprocessing programming model to exploit the inherent pipeline-level parallelism of streaming workloads. Second, we choose the message passing model to avoid the coherency issue of shared memory in language level, enabling the host/FPGA communication during kernel execution. Third, ST-Accel enables FPGA to directly access the I/O devices to bypass the host and FPGA DRAM, which eliminates the host and FPGA DRAM overheads as well as reducing the data transfer latency. Fourth, the I/O interface is wrapped in a high-level fashion, and the requirements of its timing protocol are handled by the ST-Accel library. Last, since streaming workloads are sensitive to the data communication bandwidth and latency over PCIe, we design an efficient host/FPGA communication library. We implement zero-copy to eliminate the overhead of buffer copy during the data transferring and bypass the operating system kernel to minimize the data transferring latency. Thanks to the streaming based interface exposed by our library, the host-side processing and data transferring will automatically be pipelined to hide the latency of I/O data transfer. Further, to ease the burden of developers, a credit-based flow control mechanism is implemented. Developers only need to describe the data transfer logic, and are no longer worried about data loss due to unmatched data rates of two endpoints.

Equipped with ST-Accel, developers can achieve performance that nears the physical limitation for streaming workloads without sacrificing the programming simplicity. The experimental results on the Amazon AWS cloud and local machine are shown in Section VI. ST-Accel can achieve a nearly 50X reduction in latency, and it improves bandwidth by 5X for host/FPGA communication compared to SDAccel, an OpenCL implementation from Xilinx. To validate its effectiveness in real workloads, we select three representative applications: image processing (IP), hash table lookup (HTL) and network packets encryption (NPE). The significant end-to-end bandwidth (10.7x for IP, 166x for HTL and 1.6x for NPE) and latency improvements (1/3 for NPE) compared with the implementations of OpenCL.

II. Background and Related Work

A. Systems that Leverage FPGA to Accelerate Streaming Workloads

In recent years FPGA has frequently appeared in system designs of top-tier publications for the purpose of accelerating streaming workloads. In the database field, several core data operations—like select, aggregation, and sort—are standard streaming processing tasks. For these operations, [13] shows that FPGA can achieve significant advantages in terms of power consumption and parallel stream evaluation. Additionally, ExtraV [14] deploys FPGA to perform near-storage computing, which boosts the graph processing and outperforms all the other frameworks. In the network field, ClickNP [12] takes advantage of FPGA to perform in-network packet processing that achieves a 40 Gbps line rate with an ultra-low latency. Taking a look at the machine learning field, FPGA also demonstrates its potential on streaming document classification [10], latency-sensitive real-time inference [11], *etc.* More recently, in the system field, KV-Direct [15] leverages the reconfigurability and the inherent pipeline parallelism of FPGA to accelerate random local host memory access and remote atomic operations. After analyzing these workloads, we find that they share some common characteristics:

1) Communication-intensive rather than computational-intensive.
2) For the host/FPGA collaborative streaming workloads, FPGA and host need to communicate with each other continuously during kernel execution.
3) Host bypassing is required for better performance. FPGA should be able to directly talk to other resources, *e.g.*, NIC (network interface card), disk, host memory, *etc.*
4) Support for asynchronous I/O operations is required in order to maximize the utilization of I/O devices.
5) The throughput and latency of host/FPGA communication are critical for the end-to-end performance.

B. Design Optimizations for Streaming Applications on FPGA

There exists a large amount of work aimed at optimizing the mapping of streaming applications to FPGA under power, area, throughput and latency considerations. [16] proposed the stream folding algorithms which judiciously replicate the kernels to maximize the overall throughput under the area and latency constraints. Based on this, [17] further combines the idea of kernel selection, adopting slower yet resource-efficient implementations for non-critical kernels at the data flow path to optimize the area cost subject to the throughput constraint. Later, [18] optimizes the queue size between streaming kernels and combines it with previous kernel selection and replication techniques. More recently, [19] discusses the CPU-FPGA heterogeneous platform for streaming applications, presenting algorithms to decide kernel mappings on heterogeneous devices with power and latency constraints. These techniques are orthogonal to ST-Accel (which primarily focuses on the level of programming support) and can be leveraged by the users of ST-Accel to further improve their designs. Host pipe [20] supports communication between the FPGA kernel and the host kernel. It can achieve superior bandwidth compared with the current OpenCL flow. However, its API introduces the unnecessary kernel invocation for buffer copying, which adds the CPU overhead and end-to-end communication latency.

III. Limitations of OpenCL And Our Design Goals

A. Inappropriate Memory Model

OpenCL defines two memory categories: host memory and device memory. Device memory consists of four memory regions: global memory, constant memory, local memory and private memory. Global memory is defined to be the only device memory accessible to host. In the OpenCL execution flow, the host first prepares data at host memory and then transfers it into global memory. After that, the host launches the OpenCL kernel and cannot further access global memory during kernel execution in order to avoid coherency issues brought by simultaneous shared memory accesses. Finally, when the device finishes kernel execution, the host re-acquires the access right and reads the results back from global memory. In the OpenCL implementations of FPGA vendors, global memory is mapped to FPGA on-board DRAM and PCIe is used for host memory/global memory transferring. In order to observe the OpenCL specification, host and FPGA cannot talk to each other during kernel execution although they are physically connected via PCIe. With this constraint, the application's required host/FPGA collaboration cannot be implemented. In addition, for streaming applications, it is inappropriate for OpenCL to force the incoming host data to be stored at FPGA on-board DRAM for later accesses. The original design purpose is to provide data reuse. However, the streaming applications usually expose no or very low data locality. In this case, the above OpenCL mechanism is redundant and has two disadvantages: 1). Waste of on-board DRAM bandwidth and capacity; 2). Added end-to-end latency due to unnecessary DRAM accesses.

In summary, the OpenCL memory model is inappropriate for streaming applications. This abstraction layer should be redesigned to achieve two goals: 1). Provide the choice to bypass the on-board DRAM in host/FPGA communication; 2). Enable host/FPGA communication during kernel execution.

B. FPGA Limited to Role of Slave

OpenCL adopts the master/slave execution model in which host (master) offloads computing to FPGA (slave) and later retrieves data back from it. The host serves as a proxy when FPGA wants to communicate with other devices. This adds the unnecessary data path which increases the overall latency and burdens the host CPU. For example, in near-storage computing [21] when FPGA sits between host and PCIe SSD, disk data should be fetched and processed by FPGA directly and then transferred to host (dotted line in Figure 1). Although having the PCIe I/O pins physically, in OpenCL FPGA is wrapped as a slave which is only allowed to communicate with the host via limited OpenCL APIs. Thus, in this case, the data must be fetched by the host and then be dispatched to FPGA (solid line in Figure 1) which introduces the extra overhead compared with the primitive HDL design. This also limits the performance of applications like network processing and multi-FPGA acceleration when using OpenCL. Therefore, we need a high-level programming platform that allows FPGA to directly access I/O devices while still maintaining the programming simplicity.

C. Synchronous DRAM Access Interface

OpenCL exposes the global memory, *i.e.*, FPGA DRAM, in an array interface. The R/W operation to the array will be transformed into corresponding low-level signals sent to the DRAM controller. Though providing a simple programming experience, the array interface hides the critical fact that DRAM access has a long latency for FPGA, which is about 200 ns. Thus, for each DRAM access, the

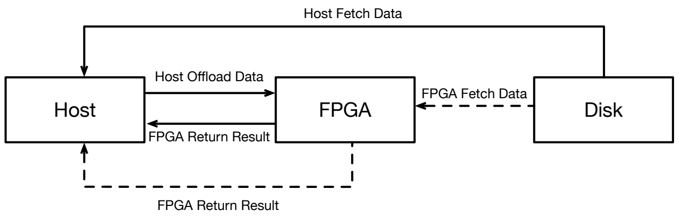
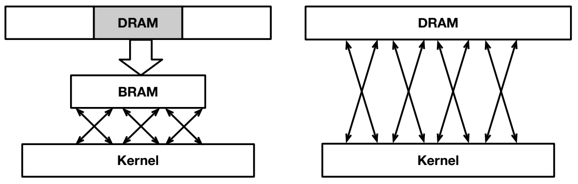


Fig. 1: Treating FPGA as a slave causes inefficient I/O device access.



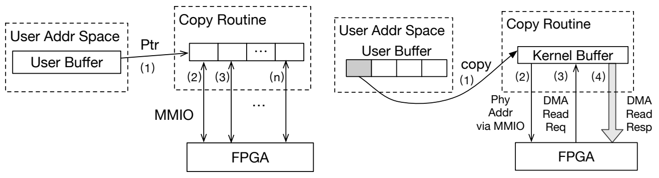
(a) The task with great data reuse. (b) Streaming task.

Fig. 2: Data access patterns of different types of workloads.

code snippet that relies on this access will be blocked for 50 cycles when running on a 250 MHz frequency. This will not be the problem for the computation-intensive tasks which have great data reuse. See Figure 2a for this kind of workload. The kernel first copies the data of slow on-board DRAM into the fast BRAM. Later, the kernel will access BRAM multiple times for computing. In this scenario, the latency to access DRAM will be amortized by following BRAM accesses; thus the overall performance will not be hurt. However, for the streaming workloads (Figure 2b), there is usually little or no data reuse. In this scenario, the stall cannot be amortized and is fatal to the overall performance. Thus, a new latency-aware I/O interface and programming paradigm should be proposed to hide the stall for performance consideration.

D. Inefficient Host/FPGA Communication Design

As we mentioned previously, streaming applications are communication-intensive rather than computation-intensive. As an example, we look at the case of using FPGA to accelerate database processing [13]. Due to the computational characteristics of database select or aggregation operation, the internal FPGA design is totally pipelined, and the overall performance is bounded by the data transferring rate. Therefore, the PCIe bandwidth for host/FPGA communication should be fully utilized for the performance consideration. What's more, for the most real-time streaming applications, there is a service-level agreement between the service providers and end users, which puts constraints on the communication latency.



(a) Host-side MMIO. (b) FPGA-side DMA.

Fig. 3: Two implementations that transfer host buffer to FPGA.

Traditionally, there are two different ways to implement the transferring routine: host-side MMIO (memory-mapped IO) and FPGA-side DMA. The mechanism of MMIO is simple; the driver maps the PCIe space of FPGA into a kernel space in host memory and any R/W operation on that memory region will be forwarded to FPGA. In host-side MMIO (Figure 3a), the transferring routine doesn't need to copy the host buffer. It simply traverses the host buffer and transfers each item via memory store instructions. However, the buffer copy is necessary for FPGA-side DMA (Figure 3b). In FPGA-side DMA, the transferring routine first copies a portion of the host buffer located in user space to a kernel space buffer which is contiguous in physical address. The kernel space buffer is usually small (less than 4 MB in Linux), and this explains why FPGA-side DMA can only transfer a small portion of host data each time. Later, the host program relays the physical address of this kernel buffer to FPGA via MMIO. After receiving the address, FPGA issues a DMA read request to fetch the data. Finally, after ensuring that FPGA has finished reading the buffer, the host can copy the next portion of the host buffer and start the next transfer process.

Despite the simplicity of host-side MMIO, it suffers performance problems. The payload size of each PCIe transfer is limited by the width of the bus interface unit of the CPU, which is currently 64 bytes [22]. According to the PCIe Gen3 specification, the header length of a transaction layer protocol packet is 30 bytes (when enabling 64-bit addressing and ECRC) [23]. Thus, host-sided MMIO can only achieve at most 64/(30+64)=68% throughput utilization. For FPGA-side DMA, the maximum data payload size is 128 or 256 bytes (bounded by the PCIe root complex). In this case, the theoretical utilization can be as high as 90%. However, the FPGA-side DMA implementation introduces an extra data copy, which adds unnecessary CPU overheads and increase the communication latency. This can be addressed by the zero-copy technique, namely mapping the kernel buffer into user space for direct access. Whereas the physically contiguous kernel buffer is a precious resource in OS whose size is usually smaller than the size of data to be transferred. Thus, an efficient buffer reuse mechanism without hurting the communication performance must be proposed. The above discussion reveals the fact that designing a high-performance communication library that performs well in term of both bandwidth and latency is challenging. As pointed out by a recent research paper [24], for an AlphaData FPGA board equipped with PCIe Gen3 x8 interface which has 6.8GB/s theoretical bandwidth, SDAccel, the Xilinx implementation of OpenCL, can only achieve 1.6GB/s bandwidth due to inefficient OpenCL buffer allocation and driver memory copy. Also, its communication latency is reported as high as 160 μs, even two magnitudes larger than the physical latency of the PCIe hard IP core, which is about 1 μs. This long latency partly comes from the kernel driver overheads.

Moreover, since the data processing rates of host and FPGA are usually different, the flow control mechanism must be implemented for the host/FPGA communication to prevent data loss. Our design goal is to provide an efficient host/FPGA communication library with flow control to achieve both near physically theoretical throughput and latency.

IV. Design

A. Multiprocessing Programming Model

ST-Accel adopts the multiprocessing programming model to exploit the inherent pipeline-level parallelism of streaming applications. The syntax of ST-Accel is based on C++. Actor and channel are two basic elements in ST-Accel.

a) Actor: Logically, an actor can be regarded as a process in the concepts of operating systems. An ST-Accel program can have multiple actors. Each actor has its private memory space and they execute concurrently but independently. Channels are used to connect different actors for exchanging data. An actor consists of the following blocks:

1) Channel declaration. This block declares the input and output channels connected to the actor.

2) Actor routine. The routine will read data from input channels and perform processing. The results will be written into output channels.
3) Termination condition. After finishing the execution of routine, the termination condition will be checked. If it is met, the actor will stop; otherwise the actor routine will be invoked again.
4) Persistent state. The persistent state is alive across different execution iterations of the actor routine.

The identifier *actor* declares an actor, see Listing 1 for an example. Each cycle, this actor will read data from the input channel, update the suffix sum (which is declared as a persistent state) and write the result of the suffix sum into the output channel. This actor will terminate when sum $\geq$ 128.

b) Channel: Logically, a channel can be conceived as a pipe in the concepts of an operating system that connects different processes. Users can indicate the data type and the depth when initializing a channel. The reader-side actor invokes a pull method to read data from the channel, while the writer-side actor calls push to write data into the channel. Push is designed to be blocking in case of data loss when the channel is full; pull is non-blocking to prevent users from constructing deadlock. Thus, users should always check the return value of pull to see whether the operation is successful.

```
actor suffix_sum(ST_Channel<int> in, ST_Channel<int> out)
{ /* Persistent state block */
  int sum = 0; // This is a persistent state
} { /* Actor routine block */
  int data; // This is a temporal state
  if (in.pull(&data)) { // Read data from the input
    sum += data; // Update the suffix sum
    out.push(sum); // Write the sum to the output
  }
} { /* Stop condition block */
  exit = (sum >= 128); // actor stops when sum >= 128
}
```

Listing 1: An example of the actor.

```
actor pcie_reader(ST_Channel<uint64_t> addr_chan,
      ST_Channel<ap_uint<512>> data_chan)
{ /* Empty persistent state block */ } {
  ST_Channel<PCIe_Read_Req> req_chan;
  ST_Channel<PCIe_Read_Resp> resp_chan;
  #pragma share_pcie_read req=req_chan resp=resp_chan
  PCIe_Read_Req req;
  if (addr_chan.pull(req.addr)) {
    req.num = 1;
    req_chan.push(req);
  }
  PCIe_Read_Resp resp;
  if (resp_chan.pull(resp)) {
    data_chan.push(resp.data);
  }
} { /* Stop condition block */
  exit = false;
}
```

Listing 2: An example to use PCIe read interface.

A complete ST-Accel program consists of three blocks: 1) actor definitions, 2) channel definitions, and 3) interconnects. The interconnect block is used to describe the channel connections between actors. The detailed syntax is omitted here due to the space constraint.

B. Message Passing Model for Host/FPGA Communication

The root cause of OpenCL preventing the host from accessing global memory (on-board DRAM FPGA) during kernel execution is to avoid the coherency issue. Since global memory can be accessed by both host and device, when they access global memory simultaneously, the memory coherency must be handled for correctness. For devices like the Intel HARP and AMD APU, the coherency is handled by the specialized hardware which is, however, not available in general devices. In order to cope with this, OpenCL stipulates that the host can only access global memory before or after the kernel execution. Thus the host-side access and the device-side access are separated, eliminating the coherency issue.

Instead of using the shared-memory model, ST-Accel adopts the message passing model for host/FPGA communication. In ST-Accel, host and device have separate memory space, and their accesses are limited to their own space. Logically, ST-Accel provides a bidirectional channel between host and FPGA for communication. Any data pushed into one end is guaranteed to be reliably transferred to the opposite end. Rather than directly writing to their memory, the incoming data is handled by the host or FPGA routine provided by the user. In this way, there is no simultaneous access to the same memory space. Thus, the coherency issue is at the language level. Additionally, in this design, the incoming host data are directly accessed by the FPGA kernel without going through on-board DRAM. This helps save the DRAM capacity and bandwidth as well as reducing the data transferring latency.

C. High-Level Abstractions for I/O Devices

I/O Device Name	Interface Type	Field
PCIe	PCIe_Read_Req	uchar num
		uint64_t addr
	PCIe_Read_Resp	ap_uint<512> data
	PCIe_Write_Req_Apply	uchar num
		uint64_t addr
	PCIe_Write_Req_Data	ap_uint<512> data
On-board DRAM	DRAM_Read_Req	uchar num
		uint64_t addr
	DRAM_Read_Resp	ap_uint<512> data
	DRAM_Write_Req_Apply	uchar num
		uint64_t addr
	DRAM_Write_Req_Data	ap_uint<512> data
Network	Net_Packet	bool sop
		bool eop
		ap_uint<5> len
		ap_uint<256> data

TABLE I: I/O interfaces in ST-Accel.

ST-Accel provides high-level abstractions to enable direct access to I/O devices. Table I presents the I/O interfaces defined in ST-Accel. These interfaces are wrapped into ST-Accel channels which support pipelined access. The host is bypassed, and I/O devices are directly accessed by FPGA through these interfaces. See Listing 2 for an example of using the PCIe read interface. The code defines an actor that contiguously issues PCIe read requests with addresses specified by channel addr_chan and forwards the read result into channel data_chan. The PCIe read request is wrapped into struct PCIe_Read_Req. The address field describes the PCIe address of the target device. The num field indicates the size of the PCIe read request: $size = num \times 512$ bit. The on-board DRAM shares the same interface with PCIe—thus the coding style to perform DRAM access is exactly the same.

Note, the read request and read response interfaces are separated in ST-Accel. This provides language-level support for constructing decoupled access/execute architecture [25] to maximize the utilization of I/O devices. See Algorithm 1 for an example of the hash table lookup. Assume that the hash table data is put in DRAM, and it is implemented in the chained fashion to solve collision. For each request, the kernel iterates the chain until it finds the matched entry. Due to the access latency of DRAM, the kernel will be blocked at line 3 and line 5 until the read response comes back. This leads to low I/O utilization and overall performance in OpenCL. In ST-Accel, thanks to the decoupled read request and response streaming interfaces, the decoupled access/execute architecture can be easily constructed to hide I/O latency. As presented in Figure 4, two actors called DRAM Reader and Matcher are defined; these act as the access processor and execute processor, respectively [25]. The DRAM

Reader is responsible for issuing the DRAM read request according to the hash table lookup request. After pushing the context information of this request into the Context channel, the DRAM Reader can immediately serve the next hash table lookup request and issue its corresponding DRAM request without stalling. When DRAM read response is ready, Matcher will read the context corresponding to this response from the Context channel and then perform key comparing. If the key is matched, the value will be sent to the Result channel. Otherwise, the address of the next chain will be sent to the DRAM reader via an Unmatched channel for reissuing the DRAM read request. The depth of Context and Unmatched channels are set to cover the DRAM read latency. The whole logic is fully pipelined without stalling. Thus, theoretically, it can issue DRAM read requests and serve DRAM read responses at every clock cycle; this maximizes the DRAM utilization and kernel throughput.

Algorithm 1 The pseudocode of chained hash table lookup.

```
1: procedure HASHTABLE-LOOKUP
2:     while $key \leftarrow$ GETREQUEST( ) do
3:         $hashline \leftarrow$ DRAM[HASH($key$)]
4:         while GETKEY($hashline$) $\neq key$ do
5:             $hashline \leftarrow$ DRAM[GETNEXT($hashline$)]
6:         SENDRESPONSE(GETVALUE($hashline$))
```

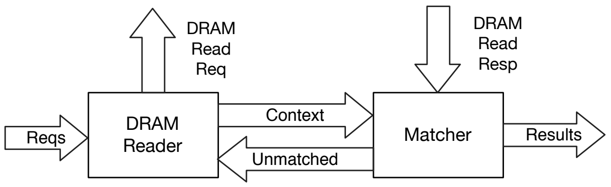


Fig. 4: Fully pipelined hash table lookup in ST-Accel.

D. Efficient Host/FPGA Communication Library with Flow Control

We identify the two types of data transferring between host and FPGA:

1) Control data transferring. Host and FPGA will periodically send commands or metadata to each other, which are used to control the computation. The data are small but latency-critical.
2) Input/output data transferring. This type of data are used in computations which are large but throughput-critical.

A single implementation of host/FPGA data transfer can not simultaneously meet both requirements. Thus, in ST-Accel, we provide two different communication interfaces for different scenarios (see Table II for the host-side APIs). The first two functions are used for exchanging small control data. So, to provide low latency, we implement them via host-side MMIO. In contrast, the input/output data transferring is implemented via FPGA-side DMA for better throughput. As we discussed at Section III-D, there are four issues to be solved to provide a reliable communication library that achieves near-physical performance:

1) Eliminate the overhead of buffer copy, a.k.a. zero-copy.
2) Pipeline the host-side processing and data transferring to overlap the latency of input/output data transferring.
3) Kernel bypass to minimize and stabilize the latency of control data transferring.
4) Implement flow control to avoid data loss.

The host-side DMA buffer must be physically contiguous since there is no hardware-supported virtual memory in general FPGA hardwares. Implement zero-copy is not trivial here. First, the volume of the data to be transferred is at GB or even TB scale, much larger than the maximum size of a single physically contiguous buffer (which is 4 MB in Linux). Thus, it's impossible to directly map

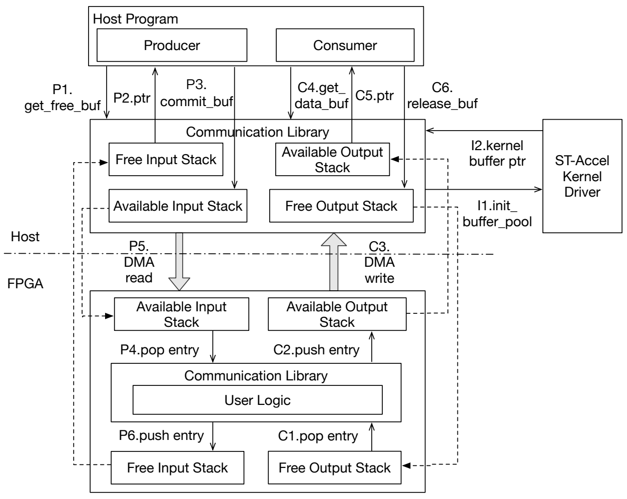


Fig. 5: The implementation of the host/FPGA communication library to do input/output data transferring.

void send_control_msg(uint32 msg)	Send a control message to FPGA.
uint32 receive_control_msg(uint32 tag)	Pull control message with specified tag from FPGA.
bool init_buf_pools(int Inum, int Onum)	Initialize data buffer pools with Inum input buffers and Onum output buffers.
uint32 *get_free_buf()	Request a free input buffer.
void commit_buf()	Transfer an input buffer to FPGA.
uint32 *get_data_buf()	Request an output buffer transferred from FPGA.
void *release_buf()	Release the most recently requested output buffer.

TABLE II: The host side APIs for host/FPGA communication.

the user data into a single physically contiguous memory region. Zero-copy buffer must be reused to transfer large amounts of data. However, this raises the question of synchronization and performance: a buffer can only be reused after being fully consumed — which involves stalling. ST-Accel solves these challenges via a credit-based flow control algorithm presented in Figure 5.

There are two steps I1 and I2 at the initialization stage. The init_buf_pools() is called to request the kernel driver to allocate the physically contiguous buffers for later DMA. The reason for allocating multiple input and output buffers is to pipeline the host-side computation and buffer data transmission (which will be discussed shortly). At I2, the kernel driver maps the kernel buffer into user space and returns their user space pointers and physical addresses which will be buffered at the communication library. Both host and FPGA have four stacks to maintain the status of the input and output buffers. A porter thread will be periodically invoked to move the entries of stacks via MMIO in the arrow directions shown in Figure 5 (dotted line). At the initialization stage, the host-side free input stack and the FPGA-side output stack are set to {0, 1, 2, ..., $Num_{input\_buf} - 1$} and {0, 1, 2, ..., $Num_{output\_buf} - 1$}, respectively. Other stacks are set to empty. Note that init_buf_pools is the only library function that involves the kernel call, and all the other functions are kernel-free. Since init_buf_pools is an one-time effort during initialization stage, all the following host/FPGA communications will bypass kernel to minimize and stabilize the communication latency.

The host program has two concurrent threads: producer and consumer. The producer thread is used for generating input data to FPGA, while the consumer thread is used for consuming the output data from FPGA. P1-P6 in Figure 5 present the flow for the producer to send FPGA data. Producer first applies a free input buffer via get_free_buf(). Later, the library routine pops an entry from the host-side free input stack and returns it. After that, the producer directly generates input data on this buffer. After filling up the buffer, producer calls commit_buf() to push the physical address of this buffer into a host-side available input stack. Then the producer can apply a new input buffer and generate new data. The host-side porter thread will move the stack entries into an FPGA-side available input stack in the background. When the FPGA user kernel wants to read input data from host, the hardware communication library will pop an entry from FPGA-side available input stack (P4) and issue a DMA read according to the address stored in this entry (P5). After receiving the data, the hardware communication library will push the aforementioned entry into the FPGA-side free input stack. Finally, this will be moved to the host-side free input stack by porter thread. C1-C6 in Figure 5 demonstrate a symmetrical path for the consumer to consume data from FPGA.

The input/output stacks decouple the data that is generating/-consuming from the data transferring; this enables the capability to overlap the transferring latency. Additionally, the flow control between host and FPGA is also achieved by the design of stacks. For example, when the data-generating rate of producer is faster than the FPGA kernel processing rate, the popping rate of the host-side free input stack will be faster than the pushing rate of the FPGA-side free input stack. This will lead the host-side free input stack to be empty and block the producer when calling get_free_buf(). The opposite case, when the FPGA kernel processing rate is faster, works symmetrically.

V. Implementation

A. The Work Flow of ST-Accel

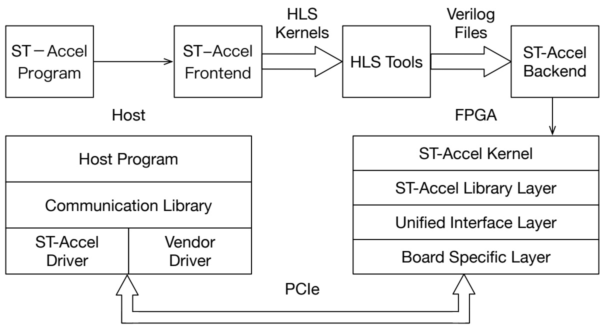


Fig. 6: The work flow of ST-Accel.

Our initial prototype for ST-Accel has about 3K lines of host-side code, 1.5K lines of Verilog code and 1K lines of HLS code. Figure 6 presents the work flow of ST-Accel. At the host-side, the communication library layer provides the runtime support for host/FPGA communication. The ST-Accel driver layer is responsible for allocating physically contiguous kernel buffers and mapping them into the user space for host programs. The vendor driver is in charge of MMIO register mapping to the FPGA PCIe space. The FPGA side architecture consists of four layers.

1) Board-Specific Layer. This layer should be provided by users who wish to adapt ST-Accel to their specific FPGA board. This layer initializes the IP core of FPGA I/O interfaces and exposes the I/O signals in vendor standard interface (AXI-4 for Xilinx and Avalon for Intel FPGA).
2) Unified Interface Layer. This layer provides a unified representation over the vendor standard interface. Thus the upper layer implementations will be compatible across vendors.
3) ST-Accel Library Layer. The signals from the FPGA hard IP core of the I/O interface should observe its specific timing protocol. Directly connecting the raw signals with the ST-Accel I/O interface will burden the high-level programmer. Thus, we introduce this layer to handle the timing protocol. An example of PCIe DMA signals is shown in Section V-B. In addition, the FPGA-side communication library is implemented at this layer.
4) ST-Accel Kernel Layer. The user-written ST-Accel program first goes through the ST-Accel front end. Each actor will be transferred into a HLS kernel, and the channel will be transferred into the HLS stream for Xilinx platform, or Intel OpenCL channel for Intel platform. Parsed by the HLS tool, the generated RTL files will be processed and glued by the ST-Accel back end. Finally, the output will be put into ST-Accel Kernel Layer to form a complete FPGA project.

B. Handling Timing Protocol Requirements of IO Interfaces

The signals from the IP core must observe its specific requirements of timing protocol. The ST-Accel Library Level is responsible for this in order to liberate the upper-level user from this cumbersome task. Due to the limited space, only the PCIe read/write interface will be discussed in this section.

Commonly, an actor can have an II (initiation interval) larger than 1 and might be stalled during execution due to a blocking push operation of channel. However, according to the specification of PCIe IP core (*e.g.* [26]), the user logic must be able to continuously receive incoming PCIe read completion (response) data or send outgoing PCIe write request data every cycle without stalling. In order to deal with this mismatch, the ST-Accel Library Layer provides two fully pipelined helper actors for PCIe read and PCIe write, respectively. PCIe write requests and corresponding write data are passed to the PCIe write helper first. Rather than directly forwarding these to the PCIe write channel connecting to PCIe IP core, the write helper first buffers them into its local double-ended circular buffer. After gathering the data belong to this write request completely, the helper will then start issuing PCIe write requests with data stored at the buffer. Since the helper actor of PCIe write is fully pipelined, it can guarantee the sending of valid signals, corresponding to the current write transaction, to PCIe IP core every cycle—thus observing the requirements of timing protocols. When the data rate of user kernels is larger than the date rate of the PCIe link, the circular buffer may overflow. To prevent this, the helper actor will stop reading data from the channel connecting to the user kernel when buffer is full. Since the push method of the ST-Accel channel is designed to be blocked, the user kernel will soon be blocked in this case, and we don't need to worry about data loss. Similarly, the helper actor of PCIe read also holds a circular buffer to store the incoming PCIe read response data. However, the logic for read is more complex since there is no backpressure in the PCIe link, and we need to perform flow control manually. In ST-Accel, the read helper will maintain a state λ indicating the remaining space of the circular buffer. Each time the reader helper receives the PCIe read request from the user kernel, it will first compare the read request size N with λ. When $N > \lambda$, a circular buffer overflow may occur in the worst case. Thus, the helper will block this request until $\lambda \geq N$, ensuring the reliable delivery.

C. Host/FPGA Communication Library

At host side, we set the kernel buffer size as 4 MB, the maximum size supported by Linux kernel call __get_free_pages(). In this way, the host-side communication routine will be called per 4 MB data, reducing the overhead of library call. However, for the FPGA-side implementation, simply setting the PCIe read/write granularity to 4 MB is not adequate. As we discussed previously in Section V-B, the library layer must buffer the PCIe read/write data in order to handle the requirements of timing protocol. This will lead to 8 MB BRAM usage in total, which is an extremely large consumption for FPGA. We measure the PCIe bandwidth in different request payloads of our testbed, shown in Figure 7. As we see, the PCIe read and write bandwidth stop increasing when payload is larger than 1024 bytes. Thus, in ST-Accel, the PCIe R/W request is split in 1024-byte granularity by the splitter actor in the library. In this case, a 1024-byte buffer is enough for PCIe write. However, it is more complex for PCIe read case. There is about 1 μs latency between issuing the PCIe read request and finally getting the response. Thus, in order to fully utilize the read bandwidth, we need to issue the next PCIe read request before we finish receiving the response data of the current request. Let T denote the granularity of the PCIe read requests; lat denote the PCIe round-trip latency; BW_{read} denote the PCIe read bandwidth. Then the splitter must issue the next read request after receiving $T - lat \cdot BW_{read}$ bytes of data for the current request. Let S denote the buffer size to store the read response data; C denotes the response data of the current requests that have been consumed by the user kernel. Then the prerequisite to issuing the next read request is $S - T + C \geq T$ which is equivalent to $S \geq 2T - C$. Therefore, for the PCIe read, the buffer size is set to $2T$. All of the above design relies on the the performance of PCIe bandwidth and latency, which are board-specific. In ST-Accel, we provide configuration files for users to adapt their specific boards.

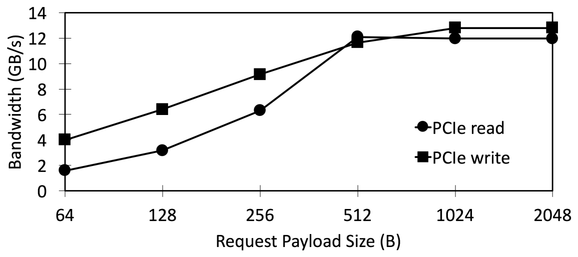


Fig. 7: The PCIe bandwidth in different request payloads.

VI. Evaluation

In this section we first evaluate the performance of the ST-Accel host/FPGA communication library and then select three typical streaming applications to compare the end-to-end performance between ST-Accel and OpenCL. The experiment in Section VI-B3 uses Altera Stratix V as the FPGA testbed and Altera OpenCL 16.1 as the OpenCL implementation. Other experiments are evaluated on the Amazon AWS cloud [4], which is equipped with the Xilinx Virtex UltraScale+ xcvu9p FPGA board and SDAccel v2017.1.

A. Host/FPGA Communication Library

The host/FPGA communication bandwidth of SDAccel is acquired by the microbenchmark from [24]. Note that we measure the end-to-end bandwidth, which takes the buffer allocation and memory copy overhead of OpenCL into account. The communication latency is the round-trip latency. The performance result and resource utilization are presented in Figure 8 and Table III, respectively. ST-Accel achieves about 1/50 RTT latency and near 4.6x bandwidth compared with SDAccel. The physical values in figures are measured by a Verilog testbench which directly connects signals from PCIe IP core, reflecting the physical capability of our testbed. As we see, ST-Accel makes full use of the physical capability.

	LUT	FF	BRAM	DSP
Used	4332	6573	10	0
Total	1182000	2364000	2160	6840
Utilization	0.37%	0.28%	0.46%	0%

TABLE III: Resource utilization of host/FPGA communication library.

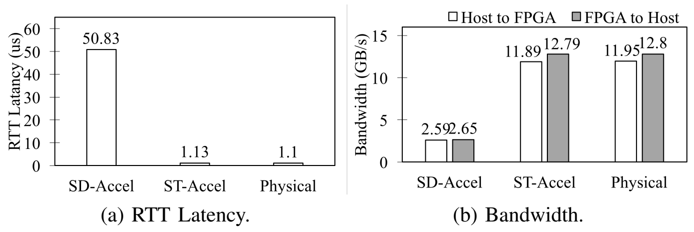


Fig. 8: Performance comparison of host/FPGA communication.

	Time (s)	Bandwidth (images/s)	Speedup
OpenCL	1.562	128	1
OpenCL pipelined	0.68	294	2.3
ST-Accel	0.146	1370	10.7

TABLE IV: Performance comparison of the image processing.

B. Applications

For fair comparison, we use the same core computational-logic implementation for ST-Accel and OpenCL. Thus, the lines of code (LOC) and FPGA resource utilizations of ST-Accel and OpenCL implementations are similar; this is omitted due to space constraints.

1) Image Processing: A wide range of image processing problems can be abstracted as the two-dimensional stencil computing, *e.g.* Gaussian blur, image noise reduction, *etc.* The results B of applying a $r \times r$ filter stencil F on $n \times m$ images G is defined as: $B[x,y] = \sum_{i=x-r}^{x+r} \sum_{j=y-r}^{y+r} F[i,j]G[i,j]$. In this section we perform a 3×3 stencil on 1080P (1920×1080) images. Each element in stencil and image array is a 4B quadruple (red, green, blue, alpha). F is stored in BRAM while G is stored in FPGA DRAM. Here, two-hundred 1080P images are processed and the end-to-end execution time is measured (see Table IV for results). The end-to-end execution time of OpenCL can be further broken down into three parts (see Table V). Thanks to the multiprocessing programming model, the communication library of ST-Accel will automatically overlap execution of the above three parts. In OpenCL we need extra coding effort to support overlapping. The inefficiency of OpenCL comes from other issues: 1). Inefficient host-FPGA transferring leads to long data copy time. 2). The memory model of OpenCL forces the host data to be stored in FPGA DRAM first, and then accessed by the kernel. Compared with ST-Accel, which directly receives data from PCIe, extra DRAM accesses add overhead. As the result, the computation time alone in the OpenCL flow is higher than the entire runtime of the ST-Accel flow.

2) Hash Table Lookup: Algorithm 1 is implemented here. We assume the data of hash table and lookup keys has been stored in the FPGA DRAM and BRAM, respectively. We synthesize three different test cases with 1, 2, 3 average DRAM access times per lookup request (marked as λ). The hash table entry is 64-bytes large, thus each hash table entry access will incur a 64-byte DRAM access. Our FPGA DRAM controller is measured to support up to 78 MOPS 64-byte DRAM access. Thus, for a benchmark with $\lambda = t$, the throughput will never exceed $78/t$ MOPS due to the DRAM constraint. The results of OpenCL and ST-Accel are presented in Figure 9. The upper bound of the throughputs calculated here are marked as *physical* in the figure. Since the ST-Accel implementation is fully pipelined (see Section IV-C), it can sustain the DRAM bandwidth as long as clock frequency is higher than 78 MHz (actually reaching 250 MHz in our design). However, the OpenCL implementation, can only issue one DRAM read request at a time and must wait for the read response

Host to FPGA Data Copy Time (s)	0.68
Kernel Computation Time (s)	0.232
FPGA to Host Data Copy Time (s)	0.65

TABLE V: Breakdown of the SDAccel end-to-end execution time.

before issuing the next read request due to the synchronous access interface. The DRAM RTT latency of our testbed is measured to be 50 cycles in 250 MHz clock frequency. Thus, the performance of OpenCL when $\lambda = 1$ should be $77890/50$ =1527 KOPS. However, since DRAM latency is unknown during compiling, and the HLS tool is feeble when dealing with pipelines of unknown latency, the generated pipeline structure is inefficient and cannot hide the latency of hash table access and key comparing. This further downgrades the throughput from 1527 KOPS to 460 KOPS.

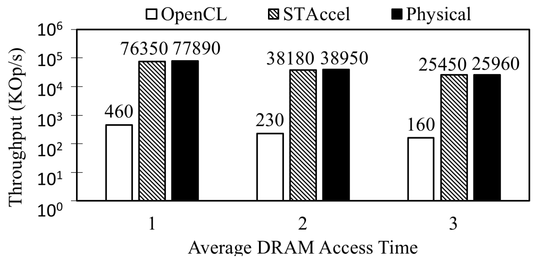


Fig. 9: Performance comparison of the hash table lookup.

3) Network Packets Encryption: The CPU tends to be the bottleneck when network packets require intensive computing [12]. Thus, in-network packet processing can be a good candidate for FPGA offloading [12], [27], [28]. In this section we choose the network packets encryption, which is used in secure networks (e.g., IPSec VPN) as the application used to compare OpenCL and ST-Accel. AES-256-CTR is selected as the encryption algorithm, and the 40 Gbps network is deployed for this test. For both OpenCL and ST-Accel, we use the same computing kernel implementation with two dedicated AES streams which can fully sustain the 40 Gbps line rate. The final end-to-end results are shown in Table VI. As discussed in Section III, FPGA is treated as a slave in OpenCL and the host acts as a proxy for FPGA/IO device communication. Thus, for the OpenCL case, the network packets should be first fetched by host and then forwarded to FPGA via PCIe. The overall throughput is bound by PCIe bandwidth, which explains the final 25.6 Gbps result. The involvement of host also adds latency. Theoretically, $Lat_{OpenCL} = Lat_{Net} + Lat_{FPGA} + Lat_{TCP/IP\ Stack} + Lat_{PCIe}$. For the ST-Accel case, FPGA directly accesses data from the network (via on-board QSFP port) and performs AES encryption. Thus, $Lat_{ST-Accel} = Lat_{Net} + Lat_{FPGA}$. In our testbed, the latency of the host TCP/IP stack and PCIe is measured as about 20 μs and 15 μs respectively, which explains the difference of end-to-end latency in Table VI. Another drawback brought by the master/slave programming model of OpenCL is the cost of three CPU cores to do packet forwarding. This can be avoided in ST-Accel.

	Throughput (Gbps)	Latency (us)	# CPU Core
OpenCL	25.6	60.3	3
ST-Accel	40.0	23.6	0

TABLE VI: Performance comparison of the network packets encryption.

VII. Conclusion

OpenCL for FPGA experienced success in improving the programming efficiency for compute-intensive tasks. However, as researchers expand the FPGA acceleration efforts into streaming-like applications, in many cases the current OpenCL support is inadequate and very inefficient. In this paper we studied the root causes of OpenCL limitations/inefficiency for the streaming workloads, arriving at ST-Accel, which addresses the limitations of OpenCL without compromising the programming simplicity. Using ST-Accel, we achieve one to two orders of magnitude performance improvement for several streaming applications. We believe that our work will enable more deployment of FPGAs for accelerating many more applications.

VIII. Acknowledgement

This work is partially supported by Huawei and Samsung under the Center for Domain-Specific Computing (CDSC) Industrial Partnership Program. Authors would like to thank Amazon for AWS credits donation, Xilinx for FPGA and software donation and Janice Wheeler for paper editing.

References

[1] J. Parkhurst *et al.*, "From Single Core to Multi-Core: Preparing for a new exponential," in ICCAD, 2006.
[2] H. Esmaeilzadeh *et al.*, "Dark Silicon and the End of Multicore Scaling," in ISCA, 2011.
[3] A. M. Caulfield *et al.*, "A Cloud-Scale Acceleration Architecture," in MICRO, 2016.
[4] "Amazon EC2 F1 Instances." https://aws.amazon.com/ec2/instance-types/f1/.
[5] J. E. Stone *et al.*, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in science & engineering*, 2010.
[6] C. Zhang *et al.*, "Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," in ICCAD, 2016.
[7] E. Pezzotti *et al.*, "FPGA-based Hardware Accelerator for Image Reconstruction in Magnetic Resonance Imaging," in FPGA, 2017.
[8] S.-H. Hung *et al.*, "A Platform-Oblivious Approach for Heterogeneous Computing: A Case Study with Monte Carlo-based Simulation for Medical Applications," in FPGA, 2016.
[9] D. Weller *et al.*, "Energy Efficient Scientific Computing on FPGAs Using OpenCL," in FPGA, 2017.
[10] W. Vanderbauwhede *et al.*, "A Hybrid CPU-FPGA System for High Throughput (10Gb/s) Streaming Document Classification," *SIGARCH Comput. Archit. News*, 2014.
[11] K. Guo *et al.*, "From Model to FPGA: Software-Hardware Co-Design for Efficient Neural Network Acceleration," HotChips, 2016.
[12] B. Li *et al.*, "ClickNP: Highly Flexible and High-performance Network Processing with Reconfigurable Hardware," in SIGCOMM, 2016.
[13] R. Mueller *et al.*, "Data Processing on FPGAs," *Proc. VLDB Endow.*, 2009.
[14] J. Lee *et al.*, "ExtraV: Boosting Graph Processing Near Storage with a Coherent Accelerator," *Proc. VLDB Endow.*, vol. 10, pp. 1706–1717, Aug. 2017.
[15] B. Li *et al.*, "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC," in SOSP, 2017.
[16] A. Hagiescu *et al.*, "A Computing Origami: Folding Streams in FPGAs," in DAC, 2009.
[17] J. Cong *et al.*, "Combining Module Selection and Replication for Throughput-Driven Streaming Programs," in DATE, 2012.
[18] J. Cong *et al.*, "Combining Computation and Communication Optimizations in System Synthesis for Streaming Applications," in FPGA, 2014.
[19] X. Wei *et al.*, "Throughput Optimization for Streaming Applications on CPU-FPGA Heterogeneous Systems," in ASP-DAC, 2017.
[20] K. Kang *et al.*, "Host pipes: Direct streaming interface between opencl host and kernel," In IWOCL, 2017.
[21] O. Arcas-Abella *et al.*, "Hardware Acceleration for Query Processing: Leveraging FPGAs, CPUs, and Memory," *Computing in Science Engineering*, 2016.
[22] L. Wang, "How to Implement a 64B PCIe* Burst Transfer on Intel Architecture." https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/pcie-burst-transfer-paper.pdf, 2013.
[23] J. Lawley, "Understanding Performance of PCI Express Systems WP350 (v1.2)." https://www.xilinx.com/support/documentation/white_papers/wp350.pdf, 2014.
[24] Y. k. Choi *et al.*, "A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms," in DAC, 2016.
[25] J. E. Smith, "Decoupled Access/Execute Computer Architectures," in ISCA, 1982.
[26] "Intel Arria 10 and Intel Cyclone 10 Avalon-ST Interface for PCIe User Guide." https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_a10_pcie_avst.pdf.
[27] N. Zilberman *et al.*, "NetFPGA SUME: Toward 100 Gbps as Research Commodity," *IEEE Micro*, 2014.
[28] M. Lavasani *et al.*, "Compiling High Throughput Network Processors," in FPGA, 2012.