Applying Models of Computation to OpenCL Pipes for FPGA Computing

Nachiket Kapre
University of Waterloo
200 University Ave W.
Waterloo, Ontario, Canada N2L 3G1
nachiket@uwaterloo.ca

Hiren Patel
University of Waterloo
200 University Ave W.
Waterloo, Ontario, Canada N2L 3G1
hiren.patel@uwaterloo.ca

ABSTRACT

OpenCL pipes offer a powerful construct for synthesizing multi-kernel FPGA applications with inter-kernel communication dependencies. The communication discipline between the FPGA kernels is restricted to producer-consumer style patterns supported with on-chip FPGA FIFOs. While this provides few restrictions on the usage, the OpenCL compiler is unable to provide guarantees on buffering capacity or schedulability of the connected kernels. Without these guarantees, an OpenCL developer may over-provision hardware resources or assume pessimistic timing during scheduling. We propose imposing a communication discipline inspired from models of computation (e.g.Ptolemy) such as synchronous dataflow (SDF), and bulk synchronous (BSP). These models offer a restricted subset of communication patterns that enable implementation tradeoffs and deliver performance and resource guarantees. This is useful for OpenCL developers operating within the constraints of the FPGA device. We provide a preliminary analysis of our proposal and sketch programmer and compiler responsibilities that would be needed for integrating these features into the FPGA OpenCL environment.

CCS CONCEPTS

• Hardware → Reconfigurable logic and FPGAs; **• Theory of computation →** *Parallel computing models*;

KEYWORDS

OpenCL, FPGA, Models of Computation

ACM Reference format:
Nachiket Kapre and Hiren Patel. 2017. Applying Models of Computation to OpenCL Pipes for FPGA Computing. In *Proceedings of IWOCL '17, Toronto, Canada, May 16-18, 2017,* 4 pages.
DOI: http://dx.doi.org/10.1145/3078155.3078163

1 INTRODUCTION

Modern FPGAs span the computing landscape from low-end embedded devices, and mid-range closely-coupled CPU-FPGA devices, to high-end, at-scale deployment in datacenters. For certain workloads, FPGAs deliver superior performance, and energy efficiency compared to the alternatives. They can also be reconfigured as needed to support varying demands of user applications. However, FPGAs have traditionally been difficult to configure as programmers describe their applications as low-level *circuits* rather than high-level software programs. However, at a fundamental level, hardware circuits are specialized parallel programs. This means, it should be possible to convert a high-level parallel computation into a circuit, if certain restrictions are imposed. While reasoning about parallelism is difficult even for traditional architectures, the adoption of OpenCL makes this task more structured, and also delivers functional portability across platforms. OpenCL [4] is a framework for describing portable parallel computations that can be implemented across a variety of devices such as CPUs, GPUs, DSPs, and FPGAs. OpenCL allows SIMD/SIMT style of parallelism to be expressed easily but arbitrary parallel computations are not supported. This makes it easier to describe correct parallel programs and sacrifices some expressive freedom for portable implementations. The rapid acceptance of OpenCL in the traditional computing landscape opens the door to ease programmer burden for exotic hardware platforms such as FPGAs as well. Both Xilinx [8] and Intel [3], the two large FPGA vendors, ship OpenCL compilers.

While circuits implemented on FPGAs are parallel, they differ from traditional architectures in one fundamental way – the datapaths and organization of hardware components on the FPGA can be completely customized to the application. Modern FPGAs provide millions of configurable Lookup Tables, hundreds of high-throughput specialized integer and even single-precision floating-point DSPs, hundreds on small, distributed on-chip SRAMs (scratchpads), and millions of configurable high-speed wires to transport data across the chip. Implementations of computation that expose *spatial* parallelism work particularly well with FPGAs. Spatial parallelism exposes communication dependencies directly to the FPGA fabric in the form of wires. On traditional architectures, this data reuse pattern is supported through expensive multi-ported register files, and coherent caches with large overheads. With OpenCL, we can harness the configurability, and rich connectivity on FPGAs by generating custom datapaths within an OpenCL kernel, and using OpenCL pipes to connect multiple kernels together.

In this position paper, we discuss the use of OpenCL pipes for FPGA design using models of computation. A Model of Computation [6] (MoC) provides the semantics of concurrent execution of computation components, and their interactions between themselves. Employing MoCs in the design process can offer advantages in the form of analytical methods to compute resource bounds, run-time guarantees, proofs of correctness, amongst others. For FPGA-based computation, we are interested in MoCs that are amenable

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
IWOCL '17, Toronto, Canada
© 2017 Copyright held by the owner/author(s). 978-1-4503-5214-7/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3078155.3078163

to FPGA implementation such as dataflow MoCs that promote streaming applications. In this paper, we highlight the limitations of current FPGA OpenCL compilers from Xilinx and Intel, and present a proposal for improvements guided through the use of models of computation. We expect the tool vendors to adopt some of the ideas presented here to deliver better outcomes for FPGA developers using OpenCL Pipes in their designs.

The key contributions of this paper are enumerated below:

- We show how to apply the Synchronous Dataflow [5] (SDF) MoC to compute buffering bounds, and get guidance on scheduling of kernels on the FPGA fabric.
- We also describe a strategy for using the Bulk Synchronous Parallel [7] (BSP) MoC for irregular message-passing between compute kernels on the FPGA.
- We also sketch a straw man implementation proposal for integrating these models in an OpenCL kernel from both the programmer and compiler development perspectives.

2 IDEA

Both Xilinx SDAccel and Intel FPGA SDKs support OpenCL 2.0 Pipe implementation to varying degrees. These implementations support a subset of the complete OpenCL 2.0 Pipe specification, and impose further constraints on how and when they can be used. In both cases, the implementations allow producer-consumer style relationships between two OpenCL kernels. It is also possible to connect an OpenCL Pipe to an IO interface on an FPGA board with suitable setup. The advantages of using OpenCL pipes in a FPGA context are (1) localization of data transfers within the chip over FPGA interconnect and FIFO buffers instead of relying on external global DRAM for staging of intermediate results, and (2) improved performance through concurrent evaluation of multiple OpenCL kernels on the FPGA.

However, despite these advantages, the OpenCL API support is a mixture of restrictions and permitted behavior that may scare prospective programmers. The API allows non-blocking calls to the pipe which could be potentially confusing to a programmer if they are used without care. We can implement blocking calls as well by checking the return values of the non-blocking calls. With blocking calls alone, and a buffer depth of 1, an OpenCL program could implement CSP [1] (Communicating Sequential Processes) model of computation with rendezvous. While this is simple to understand and implement, but is unlikely to take full advantage of parallel resources. With non-blocking calls, the producer and consumer are decoupled exposing additional parallelism under the Kahn Process Networks model of computation. However, this model assumes unbounded FIFOs that cannot be realized with finite hardware. The FPGA OpenCL compilers impose additional (arbitrary) restrictions on the use of Pipes that may violate the semantics of CSP and Kahn models of computation, further confusing a potential developer. We need additional guidance, better models of computation, that are more amenable to FPGA realization. Thus, with the right discipline, we can improve programmability (expressive freedom), performance, as well as storage costs for realizing pipes on an FPGA.

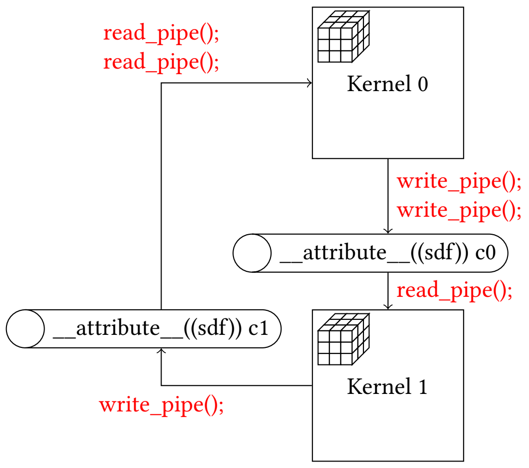

Figure 1: Visual representation of a Synchronous Dataflow Pipe. The presence of this attribute will indicate to the OpenCL compiler how to determine work-item behavior, compute a schedule for the OpenCL kernels and determine resource use for desired throughput. Each kernel is an NDrange kernel (workgroup shown as a 3D cube) with a static ordering of work-items.

2.1 Synchronous Dataflow (SDF) in OpenCL

A programmer constructs an SDF model [5] by specifying a set of computation components (called actors), and interconnecting them using FIFO-like pipes. Every actor must be specified with production and consumption rates. These rates specify the number of data tokens that are produced, and consumed with one execution of the actor. Note that the SDF MoC [5] is a special type of dataflow MoC where the order in which the actors execute is pre-computed and fixed. SDF offers several benefits that include the ability to statically compute schedules, and buffer sizes for the FIFO pipes. One can map an SDF actor to an OpenCL kernel in a manner shown in Figure 1 and accompanying code Listing 1. Here, we have two kernels setup in a feedback arrangement through two pipes. Each work-item in the first kernel writes twice as many tokens as a work-item in the second kernel. Multiple call sites is unsupported by the FPGA OpenCL compiler, and feedback pipes are poorly supported. SDF models allow both of these.

We show the code for this SDF example in Listing 1. As we can see, `sdf_kernel0` reads two tokens from pipe `c1` and writes two tokens to pipe `c0`. Then `sdf_kernel1` consumes one token from pipe `c0` and writes one token to pipe `c1`. The programmer must simply annotate the pipes with `__attribute__ ((sdf))` and rely on the compiler to size buffer depths for `c0` and `c1` while also guaranteeing a valid schedule. In an FPGA design, this may involve design decisions such as unrolling, vectorizing, or replicating `sdf_kernel1` by a factor of two, if resources are available, to match the production rate of `sdf_kernel0`. Alternatively, one could also enable resource sharing for `sdf_kernel0` to halve its Initiation Interval to save area and match the consumption rate of `sdf_kernel1`. Neither of these decisions are exposed to the programmer and would become the responsibility of the compiler. Additionally, the compiler may use a Network-on-Chip to transport the data streams between the kernels if wiring resources are insufficient. While not recommended, a

```
__global sdf_kernel0(
  __attribute__ ((sdf)) pipe int c1,
  __attribute__ ((sdf)) pipe int c0)
{
  int i=get_local_id(0);

  int x;
  read_pipe(c1, &x);
  write_pipe(c0, f(x)); // some func f
  read_pipe(c1, &x);
  write_pipe(c0, f(x)); // some func f
}
__global sdf_kernel1(__global int *x,
  __attribute__ ((sdf)) pipe int c0,
  __attribute__ ((sdf)) pipe int c1)
{
  int i=get_local_id(0);
  int y;
  read_pipe(c0, &y);
  write_pipe(c1, g(y)); // some func g
}
```

Listing 1: OpenCL code example showing the use of an Synchronous Dataflow Pipe with a new ((sdf)) attribute on the Pipe. No other syntactical changes are needed to the code. We expect the compiler to use this knowledge to enforce work-item execution, scheduling, and buffer sizing.

compiler may also be directed to offload excess pipe state to off-chip DRAM storage if insufficient on-chip resources are available.

With SDF semantics, the programmer need not worry about synchronization between each producer and consumer. We can freely use non-blocking pipe calls and discard the return values, because the compiler can guarantee availability of input data on a read, and presence of empty space on a write. All these benefits come at the expense of the restriction that work-items may not diverge in data-dependent manner.

We discuss specific limitations of the existing FPGA compilers for SDF operation below:

- The Intel OpenCL compiler [2, 3] will warn a programmer if there is *work-item-variant* code in the OpenCL kernel that may causes non-deterministic ordering of data written to the OpenCL channel[1]. **With SDF pipes, we modify the compiler to make this warning become an error condition**. Effectively, we prevent data-dependent divergence across the different work-items.
- The Intel OpenCL compiler also disallows multiple call sites to the same pipe, or loop unrolling with pipes **This should be supported by enforcing dependence constraints on the OpenCL pipe write port and relaxing the Initiation Interval of the kernel by 1**. Multi-rate behavior is natively supported by the SDF model.
- Feedback pipes are poorly supported by the Intel compilers, and are discouraged. **With a suitable way to describe initialization tokens, pipes that loopback to the same kernel should suffer no performance penalties.**

[1]Intel provides better support for vendor-specific channels extension, than OpenCL 2.0 pipes.

- The Intel OpenCL compiler attempts to perform buffer sizing based on scheduling information of the read and write operations. This is done in a simplistic fashion for a single execution iteration of the kernels, and a global scheduling view of the interacting kernels is needed to compute exact bounds. The Xilinx SDAccel compiler [8, 9] requires buffer depth to be explicitly specified by the programmer. **With SDF scheduling, since the rates of production and consumption are known statically, we can determine the buffer sizes needed within a scheduling interval exactly**.

2.2 Bulk-Synchronous Parallelism (BSP) in OpenCL

Bulk Synchronous Parallel [7] (BSP) model of computation allows a programmer to express irregular parallelism with simpler, coarse-grained synchronization barriers. The BSP MoC assumes multiple processing elements connected via a network with message passing between pairs of processing elements, and a mechanism to synchronize across all subsets of the processing elements. The execution of actors in a BSP MoC proceeds in a sequence of computation and communication supersteps. Computation supersteps perform operations, and communication supersteps transfer data between processing elements and perform basic communications between them. Instead of synchronizing for each producer-consumer relationship, a single global barrier is used to drive computation progress forward. This decouples the producer and consumer kernels, and allows a greater degree of parallelism to be processed under the right conditions.

In Figure 2, we show a high-level view of kernels interacting through a Bulk-Synchronous pipe. The different work-items in the producer kernel deposit their tokens onto the pipe in any order. This relaxes the sequential ordering restriction that the FPGA OpenCL compilers impose on kernels interacting with pipes. It is important to note that we now need to know the index information of the destination location (work-item, or absolute address) for each write access to the pipe. Before the consumer kernel is allowed to read the pipe values, a barrier must be inserted to ensure consistent

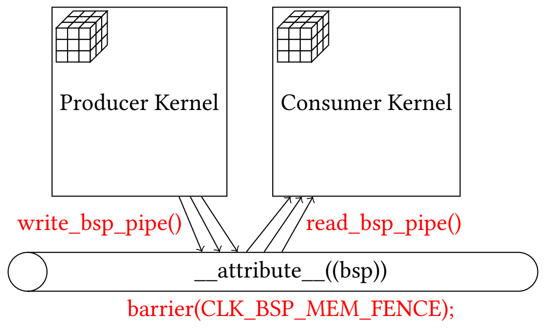

Figure 2: Visual representation of a Bulk Synchronous Pipe with asynchronous, arbitrarily ordered updates from multiple kernels. An explicit global synchronization `CLK_BSP_MEM_FENCE` is needed along with new APIs for `write_bsp_pipe` and `read_bsp_pipe`.

access to tokens. It is also possible to implement a pull-based version where the index information is supplied by the reader of the pipe. While this may seem restrictive, global barriers are cheap to implement on FPGAs (AND-reduce trees) and encourage aggressive parallel operation within the producer and consumer kernels by eliminating per-message synchronization needs. It is possible to unify producer and consumer kernels if required as long as a barrier separation between read and write operations is preserved. If a ping-pong buffer is used in the pipe implementation, the read and write operations can proceed simultaneously.

In Listing 2, we show an example of how a BSP-enabled Pipe would be programmed in OpenCL. Here, the programmer needs a new API for reading and writing into the pipe as additional metadata for destination location is required. For a pull-based model (not shown), this indexing information may be supplied in the read call to the pipe. In either case, the index information must be supplied by the programmer. Additionally, all read operations to this pipe must be protected with a new barrier `CLK_BSP_MEM_FENCE`. This ensures consistent access to the pipe data. With a shadow (ping-pong) buffer, it is possible to relax this constraint (not shown). Beyond this, the OpenCL compiler is free to parallelize the producer and consumer kernels through loop unrolling, vectorization, or compute unit replication. The pipe itself may be implemented using distributed on-chip memories to store the communication state. The compiler must also insert a network-on-chip to support to enable packetized communication between the work-items. It is functionally possible to support a BSP model using on-chip global memories. However, if the number of kernels that are interconnected with this pipe is greater than the number of ports on the RAM, we will not be able to scale to larger kernel counts.

```
__global bsp_producer_kernel(__global int *x,
  __global int *dest,
  __attribute__ ((bsp)) pipe int c)
{
  int i=get_local_id(0);

  write_bsp_pipe(c, x[i], dest[i]);
  barrier(CLK_BSP_MEM_FENCE);
}
__global bsp_consumer_kernel(
  __attribute__ ((bsp)) pipe int c)
{
  int i=get_local_id(0);

  barrier(CLK_BSP_MEM_FENCE);
  int y;
  read_bsp_pipe(c, &y);
}
```

.

Listing 2: OpenCL code example showing the use of a Bulk Synchronous Pipe with a new ((bsp)) attribute on the Pipe, two new functions `write_bsp_pipe` + `read_bsp_pipe`, and a new bulk synchronization function `barrier(CLK_BSP_MEM_FENCE)`

We discuss specific limitations of the existing FPGA compilers for BSP operation below:

- Both the Intel OpenCL compiler [2, 3], and the Xilinx SDAccel OpenCL compiler [8, 9] enforces a work-item order in both the producer and consumer of the OpenCL pipe to ensure consistent operation. **In the BSP model, we must allow any producer work-item to send a value to any consumer work-item.** This also relaxes the constraint of forcing a sequential evaluation order across work-items. To enable this feature, we must define a new attribute *__attribute__((bsp))* that allows the producer work-item to tag a consumer work-item along with the data being sent to the consumer kernel. This allows us to use a single pipe declaration to capture all producer-consumer relationships between different work-items.
- Under the dataflow model, each producer and consumer kernels synchronize implicitly with blocking calls to the pipes. **To support BSP global synchronization, we need to introduce a new fence operation** *CLK_BSP_MEM_FENCE* **to ensure that the values generated by the producer kernel are safe to consume.** This fence is crucial to support arbitrary interleaving of work-items when scheduling the work-group on the device.

3 CONCLUSIONS AND FUTURE WORK

OpenCL 2.0 Pipes are well-suited for exploiting spatial parallelism in modern FPGA applications. While both Xilinx and Intel OpenCL compilers recognize pipes, the support is preliminary with many restrictions. In this paper, we sketch a proposal for integrating two models of computation with OpenCL Pipes to deliver better outcomes for an FPGA developer. These models deliver resource guarantees (FIFO depths), throughput optimization (area-time tradeoffs), and performance results (faster solutions). In particular, we take a closer look at SDF (synchronous dataflow), and BSP (bulk-synchronous parallel) models and discuss their applicability to the FPGA OpenCL platform. Our proposal includes new programmer attribute annotations to the pipe declarations for SDF and BSP models, new APIs for reading and writing BSP pipes, and a new BSP barrier. We also identify compiler transformations that the FPGA vendor compiler need to support for implementing these models. Going forward, we can integrate other models of computation as appropriate within the OpenCL FPGA environment to support a broader set of possible applications.

REFERENCES

[1] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. DOI: https://doi.org/10.1145/359576.359585
[2] Intel. 2016. Intel FPGA SDK for OpenCL Best Practices Guide. https://www.altera.com/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf. (December 2016).
[3] Intel. 2016. Intel FPGA SDK for OpenCL Programming Guide. https://www.altera.com/literature/hb/opencl-sdk/aocl_programming_guide.pdf. (October 2016).
[4] Khronos. 2015. The OpenCL Specification. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf. (July 2015).
[5] Edward A Lee and David G Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.
[6] Claudius Ptolemaeus (Ed.). 2014. *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org. http://ptolemy.org/books/Systems
[7] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.
[8] Xilinx. 2017. UG1023: SDAccel Environment User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug1023-sdaccel-user-guide.pdf. (March 2017).
[9] Xilinx. 2017. UG1207: SDAccel Environment Optimization Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug1207-sdaccel-optimization-guide.pdf. (March 2017).