Applying Models of Computation to OpenCL Pipes for FPGA Computing

Nachiket Kapre + Hiren Patel
nachiket@uwaterloo.ca
Outline

- Models of Computation and Parallelism
  - OpenCL code samples
- Synchronous Dataflow (SDF)
- Bulk Synchronous Parallel (BSP)
Models of Computation (MoC)

▶ A MoC is a way to think + organize + analyze your computational problem
Models of Computation (MoC)

- A MoC is a way to think + organize + analyze your computational problem
- (1) Provide semantics of concurrent execution of computational components (actors), and
Models of Computation (MoC)

- A MoC is a way to think + organize + analyze your computational problem
- (1) Provide semantics of concurrent execution of computational components (actors), and
- (2) Define possible communication interactions between the compute components
Models of Computation (MoC)

- A MoC is a way to think + organize + analyze your computational problem
- (1) Provide semantics of concurrent execution of computational components (actors), and
- (2) Define possible communication interactions between the compute components
- Consciously sacrifice expressive freedom for guarantees
Models of Computation (MoC)

- A MoC is a way to think + organize + analyze your computational problem
- (1) Provide semantics of concurrent execution of computational components (actors), and
- (2) Define possible communication interactions between the compute components
- Consciously sacrifice expressive freedom for guarantees
  - e.g. SDF provides bounds on FIFO sizes + guaranteed schedule
Models of Computation (MoC)

- A MoC is a way to think + organize + analyze your computational problem
- (1) Provide semantics of concurrent execution of computational components (actors), and
- (2) Define possible communication interactions between the compute components
- Consciously sacrifice expressive freedom for guarantees
  - e.g. SDF provides bounds on FIFO sizes + guaranteed schedule
- Inspiration → Edward Lee’s Ptolemy project at Berkeley, Axel Jantsch’s models work
Models of Computation (MoC)

- A MoC is a way to think + organize + analyze your computational problem
- (1) Provide semantics of concurrent execution of computational components (actors), and
- (2) Define possible communication interactions between the compute components
- Consciously sacrifice expressive freedom for guarantees
  - e.g. SDF provides bounds on FIFO sizes + guaranteed schedule
- Inspiration → Edward Lee’s Ptolemy project at Berkeley, Axel Jantsch’s models work
- In this proposal, OpenCL compute model + MoC Communication Schemes
Models of Computation Taxonomy

- **CSP** Communicating Seq Proc

![Diagram of CSP]

[Link to Ptolemy website]
Models of Computation Taxonomy

- **CSP** Communicating Seq Proc
  - Threads communicate via explicit *rendezvous*

---

CSP

http://ptolemy.eecs.berkeley.edu
Models of Computation Taxonomy

- **CSP** Communicating Seq Proc
  - Threads communicate via explicit *rendezvous*
- **KPN** Kahn Process Networks

[CSP](#) [KPN](#)
Models of Computation Taxonomy

- **CSP** Communicating Seq Proc
  - Threads communicate via explicit *rendezvous*

- **KPN** Kahn Process Networks
  - Allows relaxed communication

[CSP](#) → [KPN](#)

[CSP](#) → Communicating Seq Proc

[KPN](#) → Kahn Process Networks

http://ptolemy.eecs.berkeley.edu
Models of Computation Taxonomy

- **CSP** Communicating Seq Proc
  - Threads communicate via explicit *rendezvous*

- **KPN** Kahn Process Networks
  - Allows relaxed communication
  - Requires *unbounded* FIFOs

[Diagram showing CSP and KPN with nested relationships]
Models of Computation Taxonomy

- **CSP** Communicating Seq Proc
  - Threads communicate via explicit *rendezvous*

- **KPN** Kahn Process Networks
  - Allows relaxed communication
  - Requires *unbounded* FIFOs

- **DDF** Dynamic Dataflow

http://ptolemy.eecs.berkeley.edu
Models of Computation Taxonomy

- **CSP** Communicating Seq Proc
  - Threads communicate via explicit *rendezvous*

- **KPN** Kahn Process Networks
  - Allows relaxed communication
  - Requires *unbounded* FIFOs

- **DDF** Dynamic Dataflow
  - Indivisible “rd+fire+wr”

---

```
http://ptolemy.eecs.berkeley.edu
```
Models of Computation Taxonomy

- **CSP** Communicating Seq Proc
  - Threads communicate via explicit *rendezvous*

- **KPN** Kahn Process Networks
  - Allows relaxed communication
  - Requires *unbounded* FIFOs

- **DDF** Dynamic Dataflow
  - Indivisible “rd+fire+wr”
  - Reduces context-switching
Models of Computation Taxonomy

- **CSP** Communicating Seq Proc
  - Threads communicate via explicit **rendezvous**

- **KPN** Kahn Process Networks
  - Allows relaxed communication
  - Requires *unbounded* FIFOs

- **DDF** Dynamic Dataflow
  - Indivisible “rd+fire+wr”
  - Reduces context-switching

- **SDF** Synchronous Dataflow

http://ptolemy.eecs.berkeley.edu
Models of Computation Taxonomy

- **CSP** Communicating Seq Proc
  - Threads communicate via explicit *rendezvous*

- **KPN** Kahn Process Networks
  - Allows relaxed communication
  - Requires *unbounded* FIFOs

- **DDF** Dynamic Dataflow
  - Indivisible “rd+fire+wr”
  - Reduces context-switching

- **SDF** Synchronous Dataflow
  - Deterministic firing rates

http://ptolemy.eecs.berkeley.edu
Models of Computation Taxonomy

- **CSP** Communicating Seq Proc
  - Threads communicate via explicit *rendezvous*

- **KPN** Kahn Process Networks
  - Allows relaxed communication
  - Requires *unbounded* FIFOs

- **DDF** Dynamic Dataflow
  - Indivisible “rd+fire+wr”
  - Reduces context-switching

- **SDF** Synchronous Dataflow
  - Deterministic firing rates
  - *Bounded* FIFOs + *Guaranteed* schedule

[http://ptolemy.eecs.berkeley.edu](http://ptolemy.eecs.berkeley.edu)
Models of Computation Taxonomy

- **CSP** Communicating Seq Proc
  - Threads communicate via explicit *rendezvous*
- **KPN** Kahn Process Networks
  - Allows relaxed communication
  - Requires *unbounded* FIFOs
- **DDF** Dynamic Dataflow
  - Indivisible “rd+fire+wr”
  - Reduces context-switching
- **SDF** Synchronous Dataflow
  - Deterministic firing rates
  - *Bounded* FIFOs + *Guaranteed* schedule

▶ http://ptolemy.eecs.berkeley.edu
Quick Intro to OpenCL Pipes

- Pipes provide a **disciplined** way to share data between kernels + allow overlapped multi-kernel operation
- Buffering of data between the producer-consumer pair possible
Pipes on FPGAs a *match made in heaven*

- FPGAs have abundant on-chip wiring structures
Pipes on FPGAs *a match made in heaven*

- FPGAs have abundant on-chip wiring structures
  - 80–90% of FPGA silicon is devoted to wiring
Pipes on FPGAs a *match made in heaven*

- FPGAs have abundant on-chip wiring structures
  - 80–90% of FPGA silicon is devoted to wiring
- Unlike other architectures, point-to-point comms possible
Pipes on FPGAs a match made in heaven

- FPGAs have abundant on-chip wiring structures
  - 80–90% of FPGA silicon is devoted to wiring
- Unlike other architectures, point-to-point comms possible
  - No DMA controllers, Cache coherency
Pipes on FPGAs *a match made in heaven*

- FPGAs have abundant on-chip wiring structures
  - 80–90% of FPGA silicon is devoted to wiring
- Unlike other architectures, point-to-point comms possible
  - No DMA controllers, Cache coherency
- Pipes are a natural way to exploit FPGA wiring
Pipes on FPGAs *a match made in heaven*

- FPGAs have abundant on-chip wiring structures
  - 80–90% of FPGA silicon is devoted to wiring
- Unlike other architectures, point-to-point comms possible
  - No DMA controllers, Cache coherency
- Pipes are a natural way to exploit FPGA wiring
- On-chip BRAMs can be configured as FIFOs
OpenCL code sketches (CSP)

```c
// aoc --board de5a_net_i2 csp.cl -o csp.aoco -c --report

__kernel void csp_kernel0(__global int* x, __write_only pipe int c0) {
    int i = get_local_id(0);
    int done = -1, temp = 0;
    temp = x[i] * x[i]; // dummy compute
    while (done != 0) {
        done = write_pipe(c0, &temp);
    }
}

__kernel void csp_kernel1(__global int* y, __read_only pipe int c0) {
    int i = get_local_id(0);
    int done = -1, temp = 0;
    while (done != 0) {
        done = read_pipe(c0, &temp);
    }
    y[i] = temp;
}
```
OpenCL code sketches (KPN)

```c
// aoc --board de5a_net_i2 kpn.cl -o kpn.aoco -c --report
#define INF 16

__kernel void kpn_kernel0(__global int* x,
   __write_only pipe int __attribute__((depth(INF))) c0)
{
    int i = get_local_id(0);
    int done = -1, temp = 0;
    temp = x[i]*x[i]; // dummy compute
    done = write_pipe(c0, &temp);
    if(done!=0){printf("Unbounded FIFO cannot be full");}
}

__kernel void kpn_kernel1(__global int* y,
   __read_only pipe int __attribute__((depth(INF))) c0)
{
    int i = get_local_id(0);
    int done = -1, temp = 0;
    while(done!=0) {
        // cannot read empty pipe
        done = read_pipe(c0, &temp);
    }
    y[i] = temp;
}
```
OpenCL code sketches (DDF)

// aoc --board de5a_net_i2 ddf.cl -o ddf.aoco -c --report
#define INF 16
int get_pipe_num_packets(__read_only pipe int x) {return 0;}
__kernel void ddf_kernel0(__global int* x,
    __write_only pipe int __attribute__((depth(INF))) c0)
{
    int i=get_local_id(0);
    int done=-1, temp=0;
    while(done!=0) {
        temp = x[i]*x[i]; // dummy compute
        done = write_pipe(c0, &temp); // done=0 is guaranteed
    }
}
__kernel void ddf_kernel1(__global int* y,
    __read_only pipe int __attribute__((depth(INF))) c0)
{
    int i=get_local_id(0);
    int done=-1, temp=0;
    while(done!=0 && get_pipe_num_packets(c0)>0) {
        done = read_pipe(c0,&temp); // done=0 is guaranteed
    }
    y[i]=temp;
}
Limitations of OpenCL Pipes

- Xilinx and Intel/Altera support the OpenCL pipes spec in different ways
Limitations of OpenCL Pipes

- Xilinx and Intel/Altera support the OpenCL pipes spec in different ways
  - e.g. `get_pipe_num_packets()`
Limitations of OpenCL Pipes

- Xilinx and Intel/Altera support the OpenCL pipes spec in different ways
  - e.g. `get_pipe_num_packets()`
- Liberal use of vendor-specific extensions (portable?)
Limitations of OpenCL Pipes

- Xilinx and Intel/Altera support the OpenCL pipes spec in different ways
  - e.g. `get_pipe_num_packets()`
- Liberal use of vendor-specific extensions (portable?)
- Not necessarily using the right approach for FPGA-friendly communication
Limitations of OpenCL Pipes

- Xilinx and Intel/Altera support the OpenCL pipes spec in different ways
  - e.g. \texttt{get\_pipe\_num\_packets()}
- Liberal use of vendor-specific extensions (portable?)
- Not necessarily using the right approach for FPGA-friendly communication
- Feedback loops or cycles not supported? Initial value problem.
SDF Model for OpenCL Pipes

- Synchronous Dataflow model ideal for **streaming** computation

![Diagram showing OpenCL Kernel 0, OpenCL Pipe c0, and OpenCL Kernel 1 with data flow and memory operations.]
SDF Model for OpenCL Pipes

- Synchronous Dataflow model ideal for **streaming** computation
- Constraint: Production and consumption rates must be known at compile time → not data-dependent

![Diagram showing OpenCL Kernel 0 and Kernel 1 connected through pipes c0 and c1 with data flow of 2wr, 1rd, 2rd, 1wr]
SDF Model for OpenCL Pipes

- Synchronous Dataflow model ideal for **streaming** computation
- Constraint: Production and consumption rates must be known at compile time → not data-dependent
- Outcome: Compiler can analyze exact FIFO size + schedule order

![Diagram showing the flow between OpenCL Kernel 0, OpenCL Pipe c0, OpenCL Kernel 1, and OpenCL Pipe c1 with operations 2wr, 1rd, 2rd, 1wr.]
SDF Model for OpenCL Pipes

- Synchronous Dataflow model ideal for **streaming** computation
- Constraint: Production and consumption rates must be known at compile time → not data-dependent
- Outcome: Compiler can analyze exact FIFO size + schedule order
  - e.g. Firing sequence: Kernel 0, Kernel 1, Kernel 1
__kernel void sdf_kernel0(__read_only pipe int __attribute__((sdf)) c1, __write_only pipe int __attribute__((sdf)) c0)
{
    int i=get_local_id(0);
    int temp1=0, temp2=0, result1=0, result2=0;
    // no need to check FIFO full/empty
    read_pipe(c1, &temp1);
    read_pipe(c1, &temp2);
    result1 = temp1*temp2;  // dummy compute
    result2 = temp2/temp1;  // dummy compute
    write_pipe(c0, &result1);
    write_pipe(c0, &result2);
}

__kernel void sdf_kernel1(__write_only pipe int __attribute__((sdf)) c1, __read_only pipe int __attribute__((sdf)) c0)
{
    int i=get_local_id(0);
    int temp=0, result=0;
    // no need to check FIFO full/empty
    read_pipe(c0,&temp);
    result=temp/10;  // dummy compute
    write_pipe(c1,&result);
}

Implication of SDF in OpenCL→FPGA mapping

- For FPGA mapping, schedule is an area-time tradeoff

  - Multiple reads or writes to a Pipe should affect Initiation Interval of circuit
  - Consider FIFO port bandwidth constraint during HLS scheduling
Implication of SDF in OpenCL → FPGA mapping

- For FPGA mapping, schedule is an area-time tradeoff
  - Schedule dictates how many times and in what order, the OpenCL kernels will evaluate

- Multiple reads or writes to a Pipe should affect Initiation Interval of circuit

- Consider FIFO port bandwidth constraint during HLS scheduling
Implication of SDF in OpenCL→FPGA mapping

- For FPGA mapping, schedule is an area-time tradeoff
  - Schedule dictates how many times and in what order, the OpenCL kernels will evaluate
  - Implied II (Initiation Interval) constraint on each connected OpenCL kernel
Implication of SDF in OpenCL → FPGA mapping

- For FPGA mapping, schedule is an area-time tradeoff
  - Schedule dictates how many times and in what order, the OpenCL kernels will evaluate
  - Implied II (Initiation Interval) constraint on each connected OpenCL kernel
    - If cannot be met, scale the schedule
Implication of SDF in OpenCL→FPGA mapping

- For FPGA mapping, schedule is an area-time tradeoff
  - Schedule dictates how many times and in what order, the OpenCL kernels will evaluate
  - Implied \( \mathbf{II} \) (Initiation Interval) constraint on each connected OpenCL kernel
    - If cannot be met, scale the schedule
    - If met easily, get greedy and try to use more of the FPGA
Implication of SDF in OpenCL $\rightarrow$ FPGA mapping

- For FPGA mapping, schedule is an area-time tradeoff
  - Schedule dictates how many times and in what order, the OpenCL kernels will evaluate
  - Implied II (Initiation Interval) constraint on each connected OpenCL kernel
    - If cannot be met, scale the schedule
    - If met easily, get greedy and try to use more of the FPGA
  - Multiple reads or writes to a Pipe should affect Initiation Interval of circuit
Implication of SDF in OpenCL → FPGA mapping

- For FPGA mapping, schedule is an area-time tradeoff
  - Schedule dictates how many times and in what order, the OpenCL kernels will evaluate
  - Implied II (Initiation Interval) constraint on each connected OpenCL kernel
    - If cannot be met, scale the schedule
    - If met easily, get greedy and try to use more of the FPGA
  - Multiple reads or writes to a Pipe should affect Initiation Interval of circuit
    - Consider FIFO port bandwidth constraint during HLS scheduling
Example FPGA Mapping Options

- e.g. Firing sequence: Kernel 0, Kernel 1, Kernel 1

![Diagram showing FPGA mapping options with OpenCL kernels and pipes.]
Example FPGA Mapping Options

- e.g. Firing sequence: Kernel 0, Kernel 1, Kernel 1
- e.g. Kernel 0 II: $x$, Kernel 1 II: $\frac{x}{2}$ → can save area by using higher II constraint on kernel 0
Final Outcomes of SDF + OpenCL Pipes

- SDF disallows work-item variant code → no data-dependent conditional access to pipe from different work-items
Final Outcomes of SDF + OpenCL Pipes

- SDF disallows work-item variant code → no data-dependent conditional access to pipe from different work-items
- SDF allows multiple reads/write from same work-item
Final Outcomes of SDF + OpenCL Pipes

- SDF disallows work-item variant code → no data-dependent conditional access to pipe from different work-items
- SDF allows multiple reads/write from same work-item
- Compiler determines depth attribute on pipes + area allocated to each kernel (subject to II minimization)
BSP Model for OpenCL Pipes

- Bulk Synchronous Parallel model ideal for **irregular** computation
BSP Model for OpenCL Pipes

- Bulk Synchronous Parallel model ideal for **irregular** computation
- Constraint: Message src-dest pairs must be supplied to the pipe for routing
BSP Model for OpenCL Pipes

- Bulk Synchronous Parallel model ideal for **irregular** computation
- Constraint: Message src-dest pairs must be supplied to the pipe for routing
- Outcome: Compiler inserts a NoC or a multi-ported RAM to enable exchange
__kernel void bsp_kernel0(__global int* x,
   __global int* dest,
   __write_only pipe int __attribute__((bsp)) c)
{
    int i=get_local_id(0);
    write_bsp_pipe(c, x[i], dest[i]);

    barrier(CLK_BSP_MEM_FENCE);
}

__kernel void bsp_kernel1(__global int* y,
   __read_only pipe int __attribute__((bsp)) c)
{
    int i=get_local_id(0);

    barrier(CLK_BSP_MEM_FENCE);

    int temp=0;
    read_bsp_pipe(c,&temp);
    y[i]=temp;
}
Message Routing between threads

Routing Network

Kernel 0

Kernel 1
For FPGA mapping, the threads in a kernel **must** be parallelized for simultaneous dispatch of messages.
For FPGA mapping, the threads in a kernel **must** be parallelized for simultaneous dispatch of messages

- Parallelism within kernel achieved through replication of compute units and/or unrolling of threads
Implication of BSP in OpenCL→FPGA mapping

- For FPGA mapping, the threads in a kernel **must** be parallelized for simultaneous dispatch of messages
  - Parallelism within kernel achieved through replication of compute units and/or unrolling of threads
  - Transport of pipe messages need a network-on-chip
For FPGA mapping, the threads in a kernel must be parallelized for simultaneous dispatch of messages
- Parallelism within kernel achieved through replication of compute units and/or unrolling of threads
- Transport of pipe messages need a network-on-chip
- Also need a new synchronization barrier to ensure that the pipe/NoC has routed all messages
Implication of BSP in OpenCL $\rightarrow$ FPGA mapping

- For FPGA mapping, the threads in a kernel **must** be parallelized for simultaneous dispatch of messages
  - Parallelism within kernel achieved through replication of compute units and/or unrolling of threads
  - Transport of pipe messages need a network-on-chip
  - Also need a new synchronization barrier to ensure that the pipe/NoC has routed all messages
  - Potential implications on storage costs at destination
Implication of BSP in OpenCL→FPGA mapping

- For FPGA mapping, the threads in a kernel **must** be parallelized for simultaneous dispatch of messages
  - Parallelism within kernel achieved through replication of compute units and/or unrolling of threads
  - Transport of pipe messages need a network-on-chip
  - Also need a new synchronization barrier to ensure that the pipe/NoC has routed all messages
  - Potential implications on storage costs at destination
- Depending on bottleneck, optimize either logic or the network
BSP allows work-items to talk to each other in arbitrary manner. We must tag each pipe operation with extra metadata $\langle \text{src}, \text{dest} \rangle$. 

Final Outcomes of BSP + OpenCL Pipes
Final Outcomes of BSP + OpenCL Pipes

- BSP allows work-items to talk to each other in arbitrary manner. We must tag each pipe operation with extra metadata \( \langle \text{src}, \text{dest} \rangle \)
- BSP requires a new form of synchronization \( \rightarrow \) probably analogous to `commit_pipe`
Final Outcomes of BSP + OpenCL Pipes

- BSP allows work-items to talk to each other in arbitrary manner. We must tag each pipe operation with extra metadata $\langle \text{src}, \text{dest} \rangle$.
- BSP requires a new form of synchronization $\rightarrow$ probably analogous to \texttt{commit\_pipe}.
- BSP message-passing can be implemented using an FPGA NoC.
Wrapup: Vision for Pipes on FPGAs

- Add compute model semantics to pipes
Wrapup: Vision for Pipes on FPGAs

- Add compute model semantics to pipes
  - Vendor-specific extension?
Wrapup: Vision for Pipes on FPGAs

- Add compute model semantics to pipes
  - Vendor-specific extension?
  - Violate OpenCL spec? → multiple writes/reads per workitem, work-item can talk to any work-item
Add compute model semantics to pipes
  ▶ Vendor-specific extension?
  ▶ Violate OpenCL spec? → multiple writes/reads per workitem, work-item can talk to any work-item
  ▶ Clarify spec? → ordering of events on pipes?

Feedback and Fanout in Pipes
Support Pipes with FPGA NoCs
→ packet-switched communication

TODO: Someone please make an OpenCL lexer for Pygments + LaTeX
Wrapup: Vision for Pipes on FPGAs

- Add compute model semantics to pipes
  - Vendor-specific extension?
  - Violate OpenCL spec? → multiple writes/reads per workitem, work-item can talk to any work-item
  - Clarify spec? → ordering of events on pipes?
- Feedback and Fanout in Pipes
Wrapup: Vision for Pipes on FPGAs

- Add compute model semantics to pipes
  - Vendor-specific extension?
  - Violate OpenCL spec? → multiple writes/reads per workitem, work-item can talk to any work-item
  - Clarify spec? → ordering of events on pipes?

- Feedback and Fanout in Pipes

- Support Pipes with FPGA NoCs → packet-switched communication

TODO: Someone please make an OpenCL lexer for Pygments + LaTeX
Wrapup: Vision for Pipes on FPGAs

- Add compute model semantics to pipes
  - Vendor-specific extension?
  - Violate OpenCL spec? → multiple writes/reads per work-item, work-item can talk to any work-item
  - Clarify spec? → ordering of events on pipes?

- Feedback and Fanout in Pipes
- Support Pipes with FPGA NoCs → packet-switched communication
- TODO: Someone please make an OpenCL lexer for Pygments + LaTeX