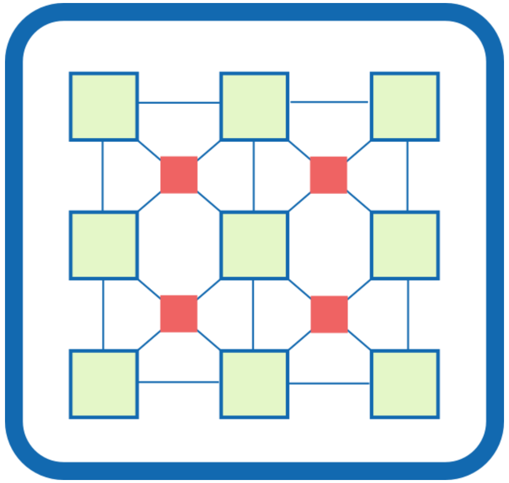
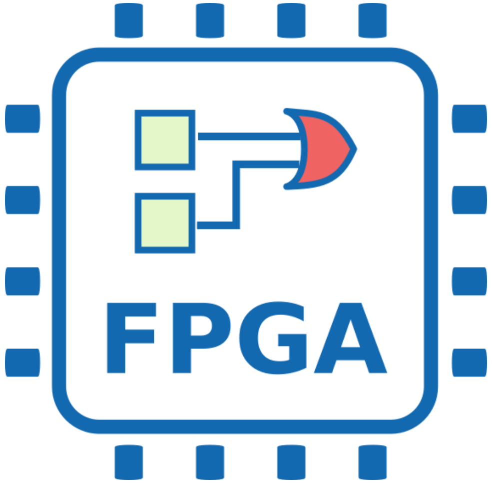
TIZIANO DE MATTEIS, JOHANNES DE FINE LICHT, JAKUB BERÁNEK, TORSTEN HOEFLER

Streaming Message Interface: High-Performance Distributed Memory Programming on Reconfigurable Hardware





Reconfigurable Hardware is a viable option to overcome architectural von-Neumann bottleneck

Modern high performance **FPGAs** and **High-Level Synthesis** (HLS) tools are attractive for HPC

Distributed Memory Programming on **Reconfigurable Hardware** is needed to scale to multi-node

Communication is typically handled either by going through the host machine or by streaming across fixed device-to-device connections

We propose Streaming Messages:

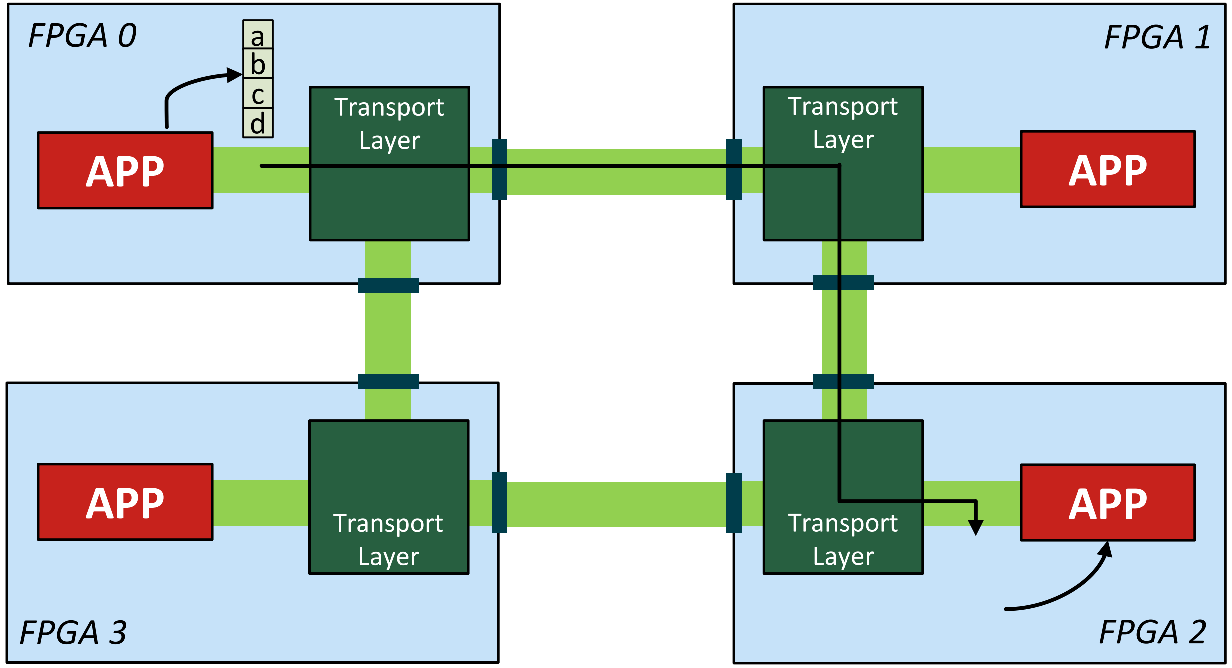
- a distributed memory programming model for FPGAs that unifies message passing and hardware programming with HLS
- SMI, an HLS communication interface specification for programming streaming messages

github.com/spcl/smi

Existing communication models: Message Passing

With Message Passing, *ranks* use local buffers to send and receive information

```
#pragma pipeline
for (int i = 0; i < N; i++)
        buffer[i] = compute(data[i]);
SendMessage(buffer, N, my_rank + 2);
```



Flexible: End-points are specified dynamically

Bad match for HLS programming model:

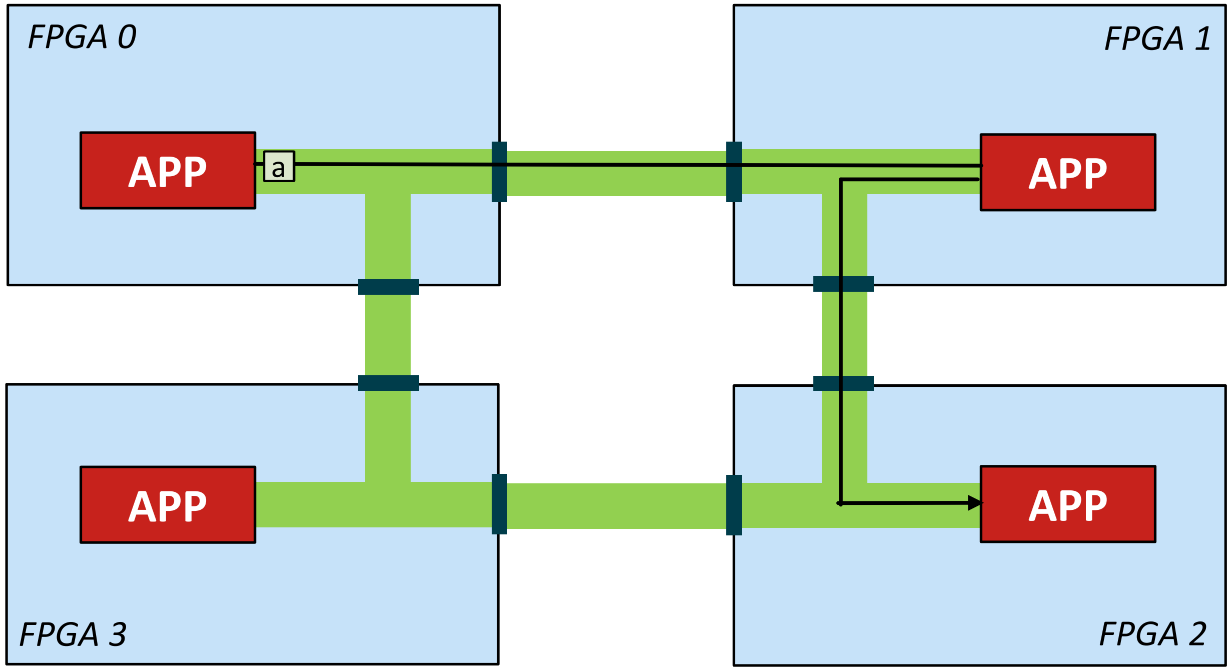
- relies on bulk transfers
- (potentially dynamically sized) buffers required to store messages

Manuel Saldaña et al. *"MPI As a Programming Model for High-Performance Reconfigurable Computers"*. ACM Transactions on Reconfigurable Technology System, 2010
Nariman Eskandari et al. *"A Modular Heterogeneous Stack for Deploying FPGAs and CPUs in the Data Center"*. In FPGA, 2019

Existing communication models: Streaming

Data is streamed across inter-FPGA connections in a pipelined fashion

```
// Channel fixed in the architecture
#pragma pipeline
for (int i = 0; i < N; i++)
   stream.Push(compute(data[i]));
```



Communication model **fits** the HLS programming model

Inflexible, the user must:

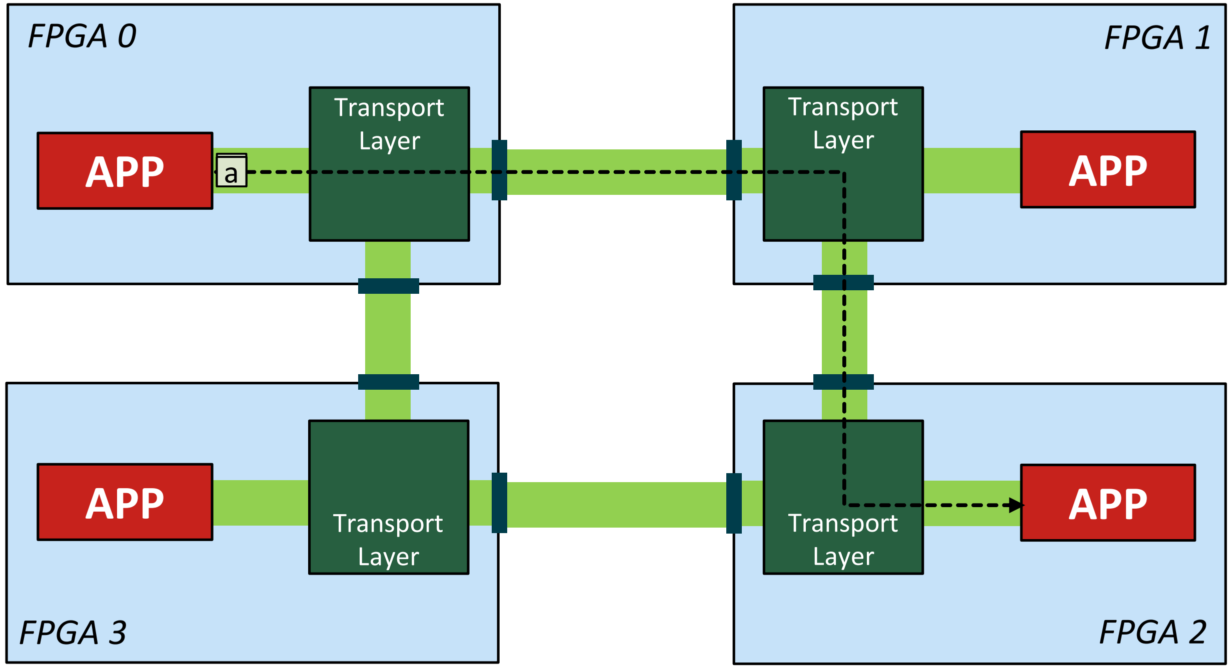
- construct the exact path between end-points
- handle all the forwarding logic

Rom Dimond et al. *"Accelerating largescale HPC Applications using FPGAs"*. IEEE Symposium on Computer Arithmetic, 2011
Kentaro Sano et al. *"4. Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth"*. IEEE Transactions on Parallel and Distributed Systems, 2014

Our proposal: Streaming Messages

Traditional, buffered messages are replaced with pipeline-friendly *transient channels*

```
Channel channel(N, my_rank + 2, 0); // Dynamic target
#pragma pipeline
for (int i = 0; i < N; i++)
   channel.Push(compute(data[i]));
```



Combines the best of both worlds:

- Channels are transiently established, as ranks are specified dynamically
- Data is pushed to the channel during processing in a **pipelined** fashion

Key facts:

- Each channel is identified by a ***port***, used to implements an hardware streaming interface
- All channels can operate in parallel
- Ranks can be programmed **either in a *SPMD* or *MPMD* fashion**

Streaming Message Interface

A communication interface for HLS programs that exposes primitives for both point-to-point and collective communications

Point-to-Point channels are unidirectional FIFO queues used to send a message between two endpoints:

```
void Rank0(const int N, /* ...args... */) {
  SMI_Channel chs = SMI_Open_send_channel(  // Send to
         N, SMI_INT, 1, 0, SMI_COMM_WORLD); // rank 1

  #pragma pipeline // Pipelined loop
  for (int i = 0; i < N; i++) {
    int data = /* create or load interesting data */;
    SMI_Push(&chs, &data);
} }
```

```
void Rank1(const int N, /* ...args... */) {
  SMI_Channel chr = SMI_Open_recv_channel(// Receive from
         N, SMI_INT, 0, 0, SMI_COMM_WORLD); // from rank 0
  #pragma pipeline // Pipelined loop
  for (int i = 0; i < N; i++) {
    int data;
    SMI_Pop(&chr, &data);
    // ...do something useful with data...
} }
```

Streaming Message Interface

A communication interface for HLS programs that exposes primitives for both point-to-point and collective communications

Point-to-Point channels are unidirectional FIFO queues used to send a message between two endpoints:

```
void Rank0(const int N, /* ...args... */) {
  SMI_Channel chs1 = SMI_Open_send_channel(N, SMI_INT, 1, 0, SMI_COMM_WORLD);    // Send to rank 1
  SMI_Channel chs2 = SMI_Open_send_channel(N, SMI_FLOAT, 2, 1, SMI_COMM_WORLD);  // Send to rank 2
  #pragma pipeline // Pipelined loop
  for (int i = 0; i < N; i++) {
    int i_data = /* create or load interesting data */;
    float f data = /* create or load interesting data */;
    SMI_Push(&chs, &i_data);
    SMI_Push(&chs2, &f_data);
} }
```

Data elements are sent in order

Calls can be pipelined in single clock cycle

Communication is programmed in **the same way** data is normally streamed between intra-FPGA modules

Streaming Message Interface

Collective channels are used to implement collective communications. SMI defines `Bcast`, `Reduce`, `Scatter` and `Gather`

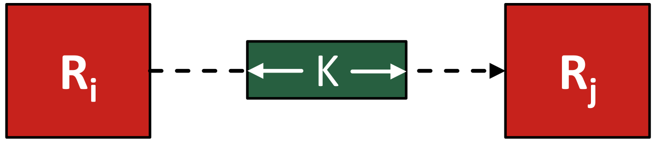
```
void App(int N, int root, SMI_Comm comm, /* ... */) {
  SMI_BChannel chan = SMI_Open_bcast_channel(
          N, SMI_FLOAT, 0, root, comm);
  int my_rank = SMI_Comm_rank(comm);
  #pragma pipeline // Pipelined loop
  for (int i = 0; i < N; i++) {
    int data;
    if (my_rank == root)
      data = /* create or load interesting data */;
    SMI_Bcast(&chan, &data);
    // ...do something useful with data...
} }
```

- If the caller is the **root**, it will *push* data towards other ranks
- otherwise it will *pop* data elements from network

SMI allows multiple collective communications of the same type to execute in **parallel**

Buffering and Communication mode

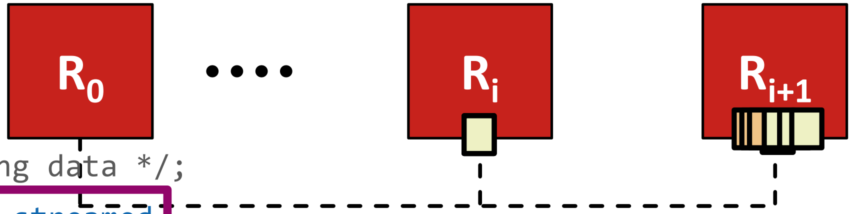
SMI channels are characterized by an **asynchronicity degree K ≥ 0**:
the sender can run ahead of the receiver by up to K elements



Point-to-Point Communication modes: **Eager** (if N ≤ K) and **Rendez-vous** (otherwise)

Collectives: we can not rely on flow control alone. Example: `Gather`

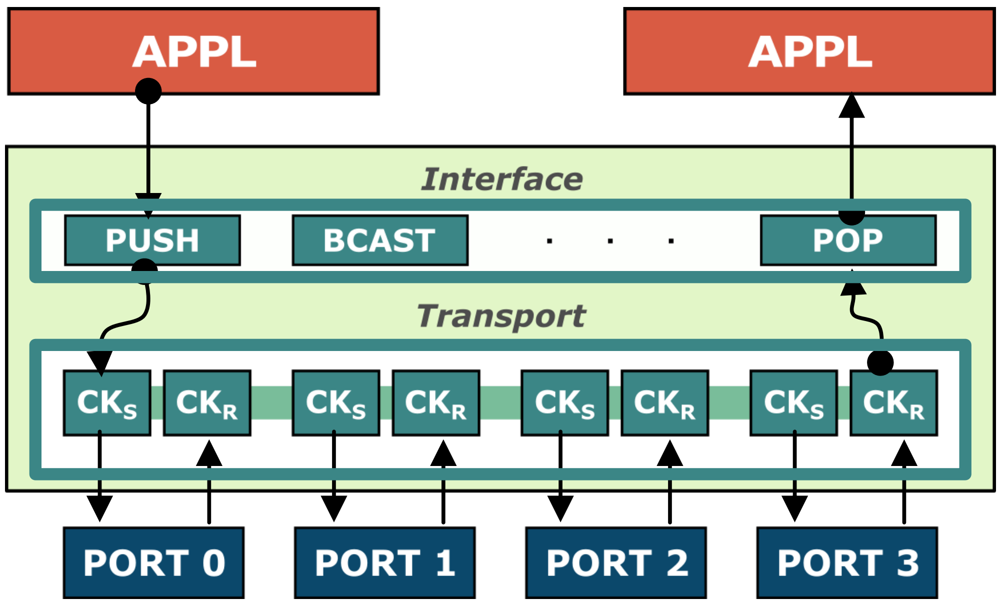
```
SMI_GatherChannel chan = SMI_Open_gather_channel(
            N, SMI_FLOAT, 0, root, comm);
#pragma pipeline // Pipelined loop
for (int i = 0; i < N; i++) {
    int data;
    if (my_rank != root)
      data = /* create or load interesting data */;
    SMI_Gather(&chan, &data); // Data is streamed
    if (my_rank == root) // ...do something useful with data...
}
```



To ensure correctness, the implementations need to synchronize ranks, depending on the used collective
For Gather, the root communicates to each rank when it is ready to receive

Reference Implementation

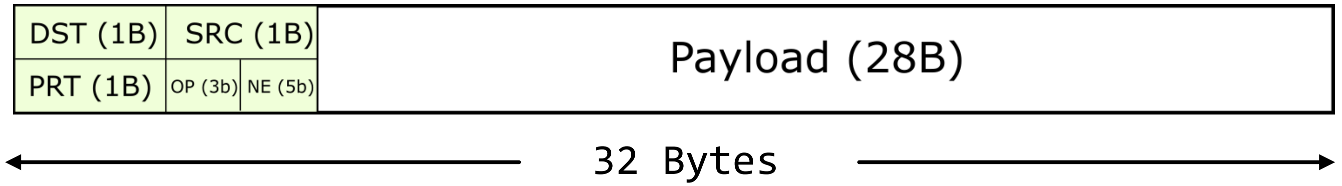
We implemented a proof-of-concept HLS-based implementation (targeting Intel FPGA)



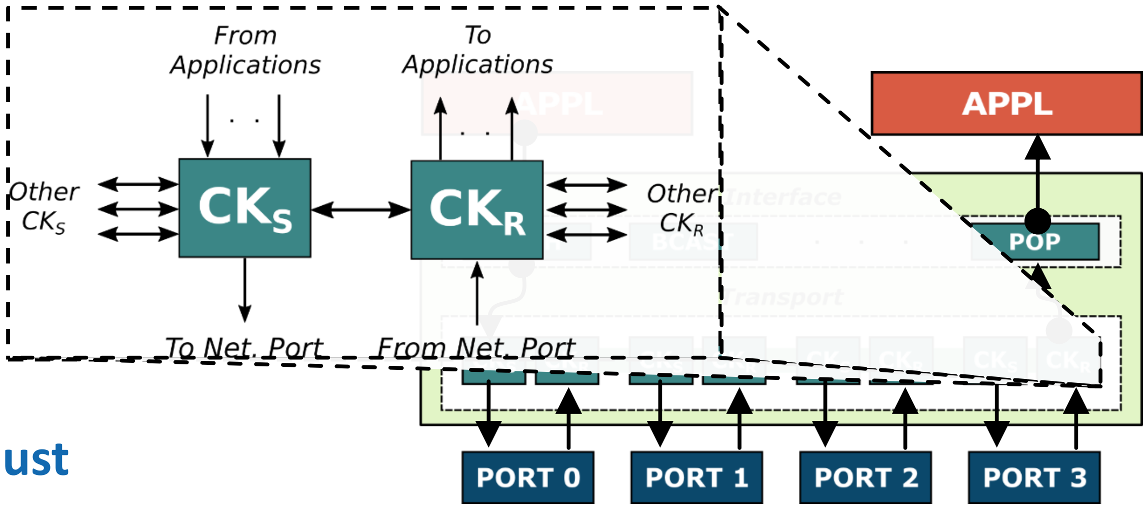
SMI implementation organized in **two main components**

`Port` numbers declared in **`Open_channel`** primitives are used to lay down the hardware

Messages packaged in **network packets**, forwarded using packet switching on dedicated intra-FPGA connections



Reference implementation

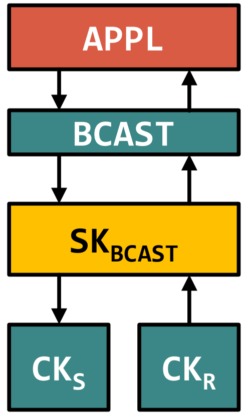


Each FPGA net. connection is managed by a pair of **Communication Kernels (CK)**

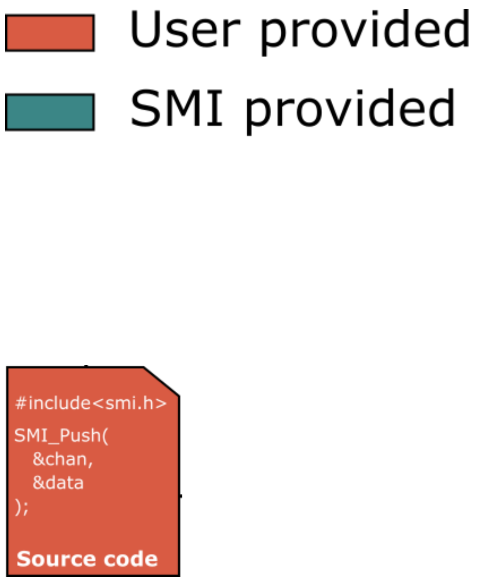
Each CK has a dynamically loaded routing table that is used to forward data accordingly

If the network topology or number of rank change, we just need to rebuild the routing tables, not the entire bitstream

Collectives are implemented using **Support Kernels**:



Development Workflow

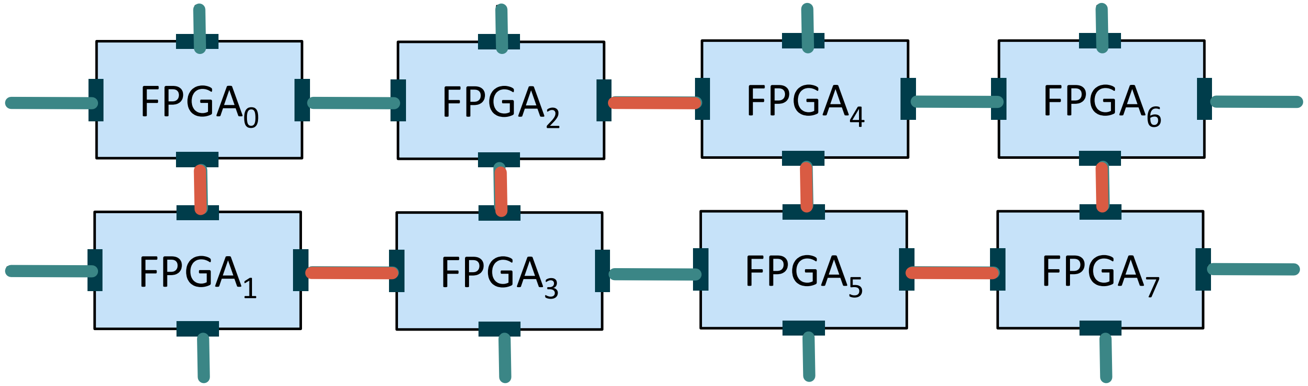


1. The **Code Generator** parses the user devices code and creates the SMI communication logic
2. The generated and user codes are synthesized. **<u>For SPMD program, only one instance of the bitstream is generated</u>**
3. A **Routes Generator** creates the routing tables (user can change the routes w/o recompiling the bitstream)
4. The user host program takes routing table and bitstream, and uses generated host header to start all SMI components

Evaluation

Testbed: 8 Nallatech 520N boards (Stratix 10), each with 4x 40Gbit/s QSFP, host attached using PCI-E 8x

The FPGAs are organized in 4 host nodes, interconnected with an Intel Omni-Path 100Gbit/s network



Evaluation over different topologies **simply by changing the topology file**

```
FPGA0:port0 – FPGA1:port2
FPGA0:port1 – FPGA2:port4
FPGA0:port2 – FPGA1:port0
FPGA0:port4 – FPGA6:port1
…
```

2D-Torus.json

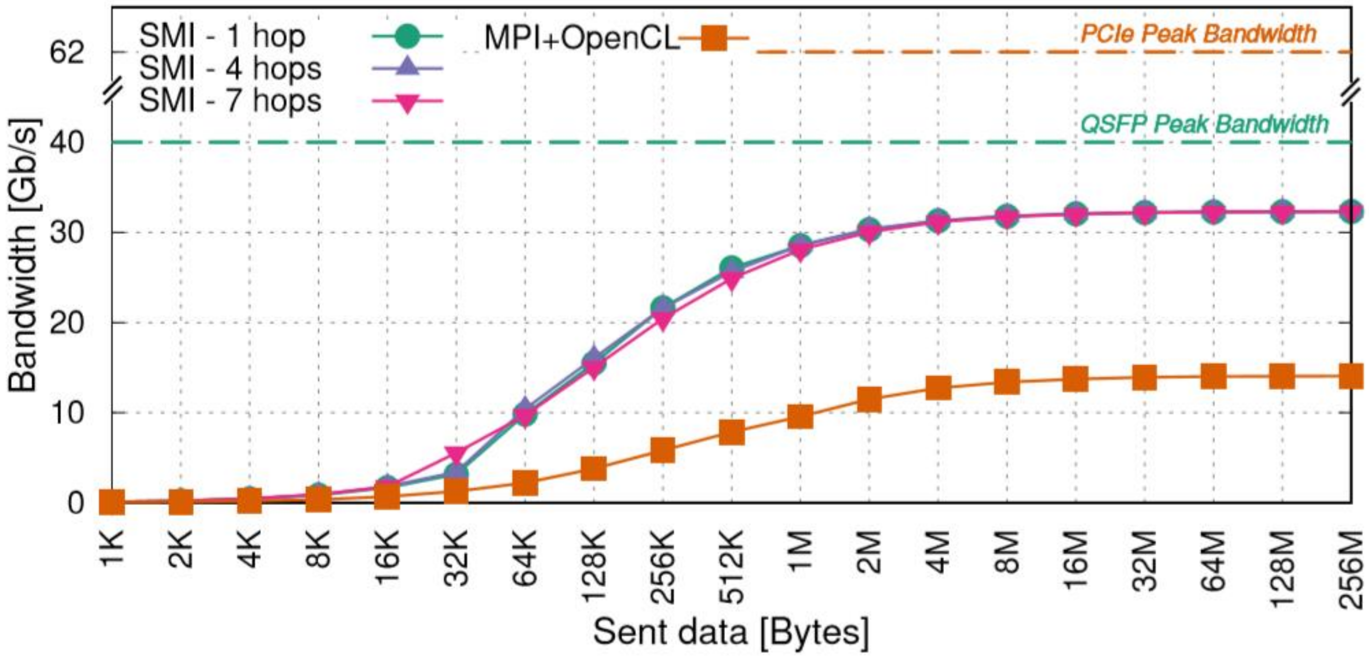
```
FPGA0:port2 – FPGA1:port0
FPGA1:port1 – FPGA3:port4
FPGA3:port0 – FPGA2:port2
FPGA2:port1 – FPGA4:port4
…
```

Bus.json

We wish to thank the Paderborn Center for Parallel Computing (PC²) for granting access, support, maintenance, and upgrades on their Noctua multi-FPGAs system.

Microbenchmarks

Resource Utilization

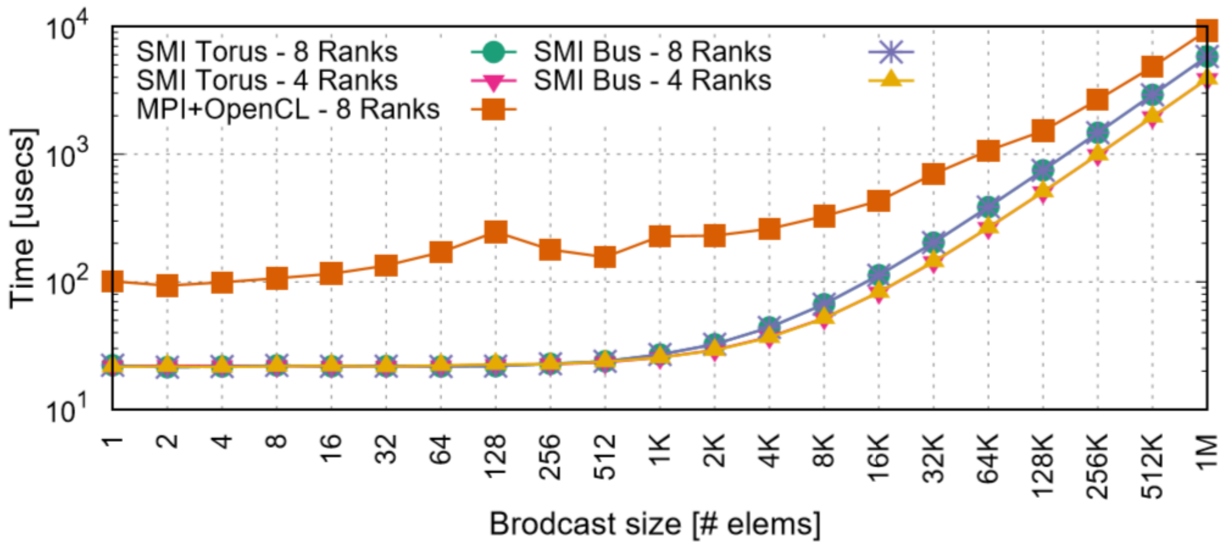


Latency (usec) – P2P

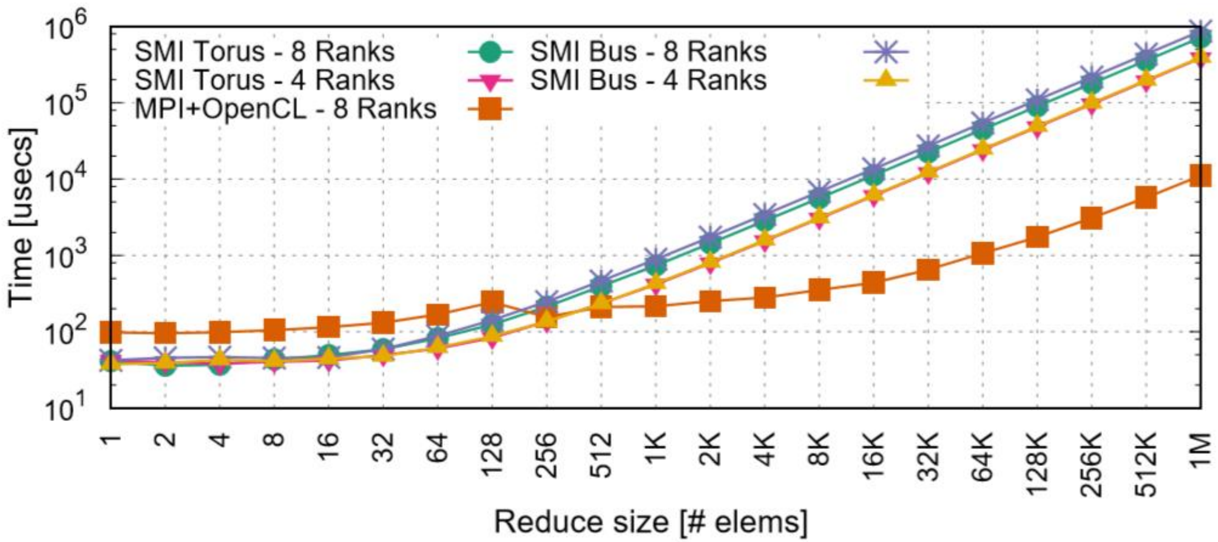
MPI+OpenCL	SMI-1	SMI-4	SMI-7
36.61	0.801	2.896	5.103

Microbenchmarks

Resource Utilization

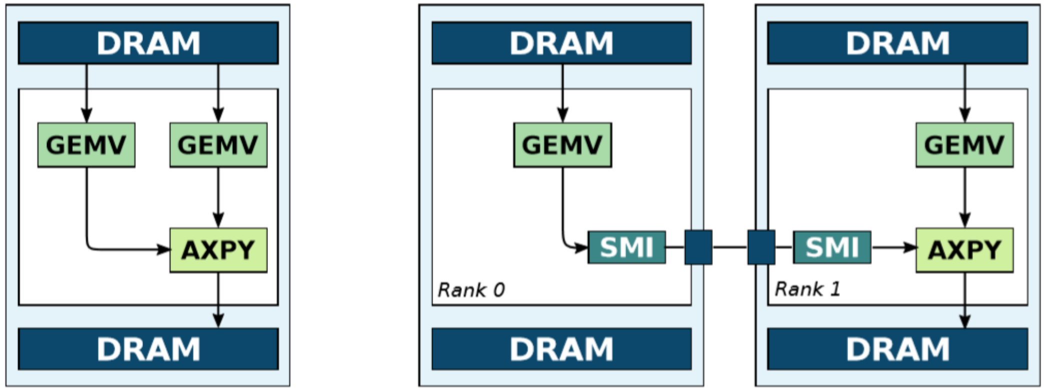


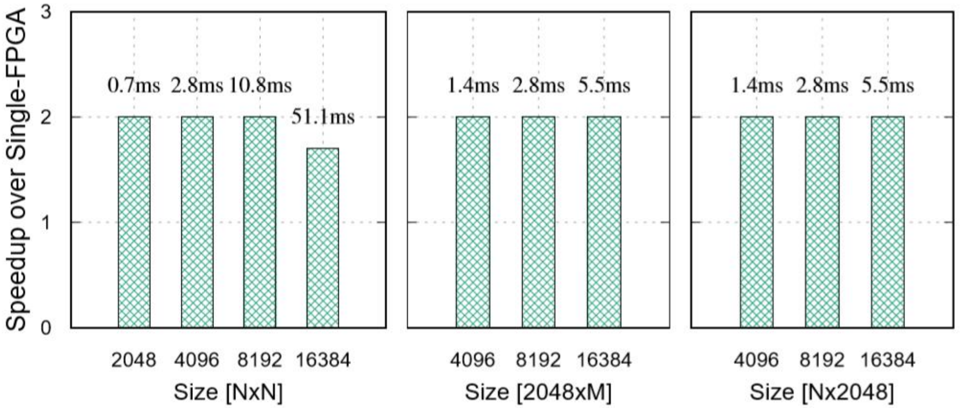
Reduce



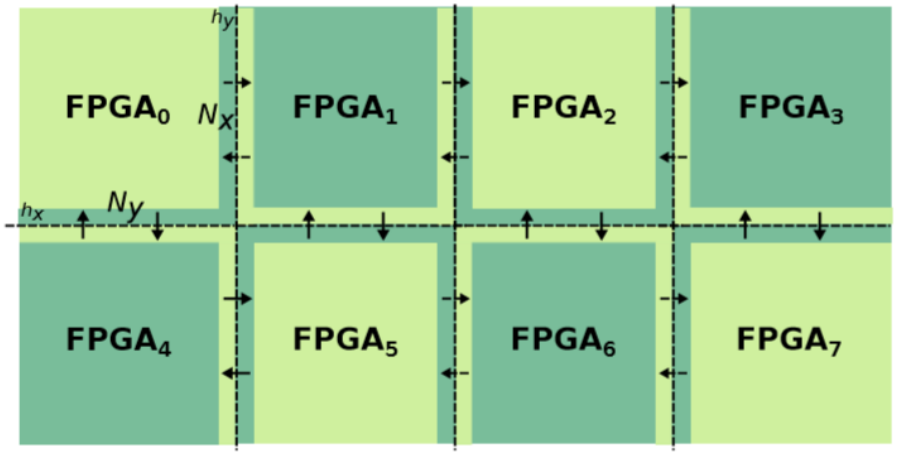
Applications

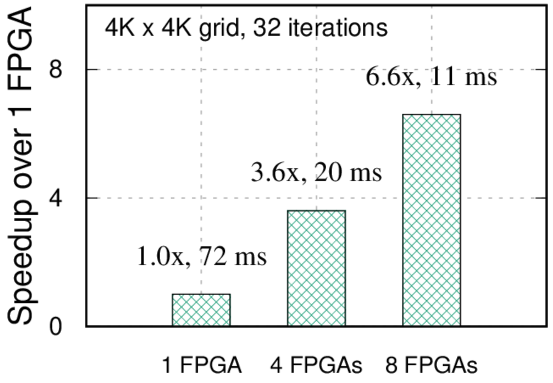
GESUMMV: MPMD program over two ranks

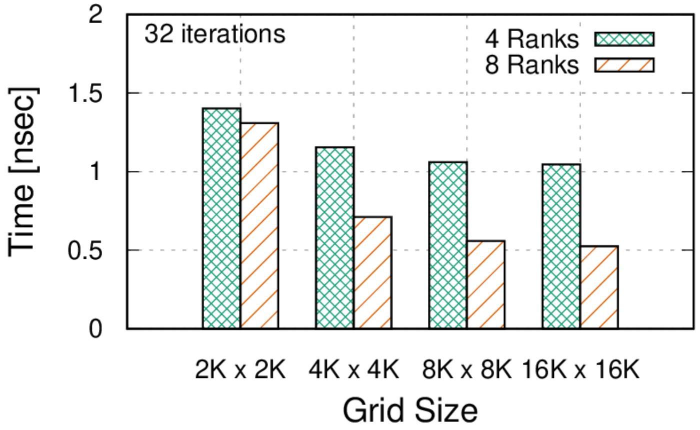




SPMD: spatially tiled 2D Jacobi stencil (**same** bitstream for all the ranks)





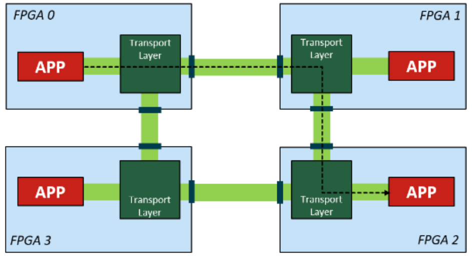


Summary

Our proposal: Streaming Messages

Traditional, buffered messages are replaced with pipeline-friendly *transient channels*

```
Channel channel(N, my_rank + 2, 0); // Dynamic target
#pragma pipeline
for (int i = 0; i < N; i++)
   channel.Push(compute(data[i]));
```



Combines the best of both worlds:
- Channels are transiently established, as ranks are specified dynamically
- Data is pushed to the channel during processing in a **pipelined** fashion

Key facts:
- Each channel is identified by a ***port***, used to implements an hardware streaming interface
- All channels can operate in parallel
- Ranks can be programmed **either in a *SPMD* or *MPMD* fashion**

Streaming Message Interface

A communication interface for HLS programs that exposes primitives for both point-to-point and collective communications.

Point-to-Point channels are unidirectional FIFO queues used to send a message between two endpoints:

```
void Rank0(const int N, /* ...args... */) {
  SMI_Channel chs = SMI_Open_send_channel(  // Send to
       N, SMI_INT, 1, 0, SMI_COMM_WORLD); // rank 1

  #pragma ii 1 // Pipelined loop
  for (int i = 0; i < N; i++) {
    int data = /* create or load interesting data */;
    SMI_Push(&chs, &data);
} }
```

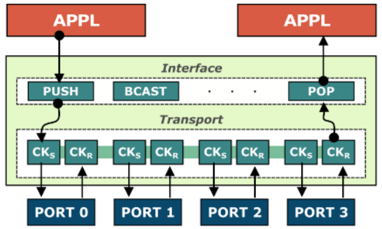
```
void Rank1(const int N, /* ...args... */) {
  SMI_Channel chr = SMI_Open_recv_channel(// Receive from
       N, SMI_INT, 0, 0, SMI_COMM_WORLD); // from rank 0
  #pragma ii 1 // Pipelined loop
  for (int i = 0; i < N; i++) {
    int data;
    SMI_Pop(&chr, &data);
    // ...do something useful with data...
} }
```

- Data elements are sent in order
- Calls can be pipelined in single clock cycle

Communication is programmed in the same way data is normally streamed between intra-PGA modules

Reference Implementation

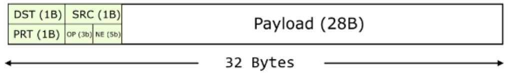
We implemented a proof-of-concept HLS-based implementation (targeting Intel FPGA)



SMI implementation organized in **two main components**

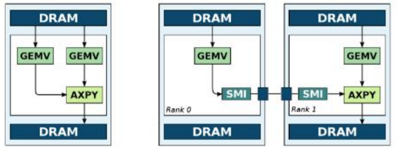
Port numbers declared in **Open_channel** primitives are used to lay down the hardware

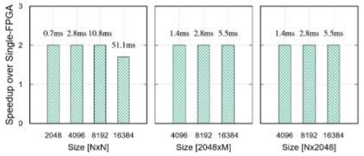
Messages packaged in **network packets**, forwarded using packet switching on dedicated intra-FPGA connections



Applications

GESUMMV: MPMD program over two ranks





SPMD: spatially tiled 2D stencil (**same** bitstream for all the ranks)