# *A First Step Toward Support for MPI Partitioned Communication on SYCL-Programmed FPGAs*


*Workshop on Heterogeneous High-Performance Reconfigurable Computing*

***Steffen Christgau***, *Marius Knaust, Thomas Steinke*

*Zuse Institute Berlin, Germany*

# Programming FPGAs using SYCL

ZIB
ZUSE INSTITUTE BERLIN

- Single Source C++-based programming, requiring compiler support → Intel oneAPI

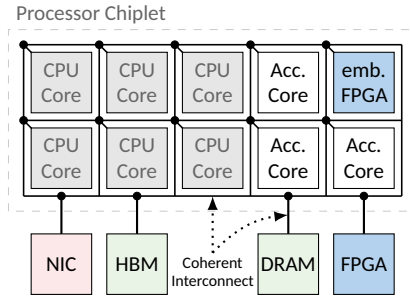
```
sycl::queue queue { fpga_device_selector() };
float *buf = sycl::malloc_shared<float>(N, queue); /* USM allocation */

queue.single_task([=]() {
  /* Kernel/Device Code */
  #pragma unroll 64
  for (int i = 0; i < N; i++) {
    buf[i] = compute(i);
    communicate(buf);
  }
});
```

# Inter-Node Communication with FPGAs

- MPI too complex for full FPGA implementation
- HPC-specialized/MPI-aware NICs in nodes
- Intra-node coherent interconnects (→ CXL)
  (cf. European Processor Initiative)

Appropriate subset → **Partitioned Communication**



# MPI Partitioned Communication

- Since MPI 4.0 (June 2021) for better thread scalability (OpenMP) on many-cores
- *"Contributions to a buffer from multiple actors"* → CPU/GPU threads, **FPGA pipelines/CUs**

**Partitioned Communication**

Sender

```
MPI_Psend_Init(...)
MPI_Start(request)

  Let this be device code!
  buf[f(i)] = compute(i)
  MPI_Pready(i, request)

MPI_Wait(request)
```

Note: No Dependency!

Receiver

```
MPI_Precv_Init(...)
MPI_Start(request)

  Let this be device code!
  while(!f)
     MPI_Parrived(request, i, &f)
  process(buf[f(i)])

MPI_Wait(request)
```

# *Using Partitioned Communication on FPGAs*

ZIB
ZUSE INSTITUTE BERLIN

- No implementation of MPI Partitioned Communication (PC) available for FPGAs
- Current focus of MPI Hybrid Working Group: GPUs

## Our contribution

Demonstrate feasibility of MPI PC for SYCL-programmed FPGAs

- Aim: Lightweight and general SYCL + MPI solution
- Identify shortcomings in the interaction between MPI and SYCL

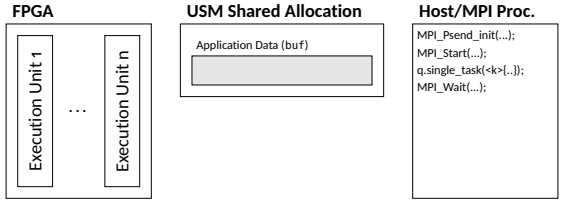
Note: Similar work done by MPI Hybrid WG for GPUs, but independently developed

# *Design Space: Data Access*

Challenge: Accessible device data for transfers → device-only allocation not usable

Possibile Solutions:

1. **Double Buffering:** Dedicated Buffers for Computation and Communication
   - Copy partition data to/from host-accessible buffer
   - Doubles memory usage, copy logic → overhead → not favorable
2. **USM Shared Allocation:** Device- and host-accessible memory
   - Data transfers transparent to application
   - Future: CXL → coherent access by CPU, NIC, FPGA



# *Design Space: Signalling*

Challenges:

1. Inform MPI about ready partition (`MPI_Pready` was called on FPGA)
2. Inform FPGA about arrival of partition (return true in `MPI_Parrived`)

Possible Solutions

1. **Host Pipes**: pass (partition, request ID) tuple from/to host
   - Typed interface
   - blocking or non-blocking behavior
   - Host pipe flavor **not available**/implemented for FPGAs

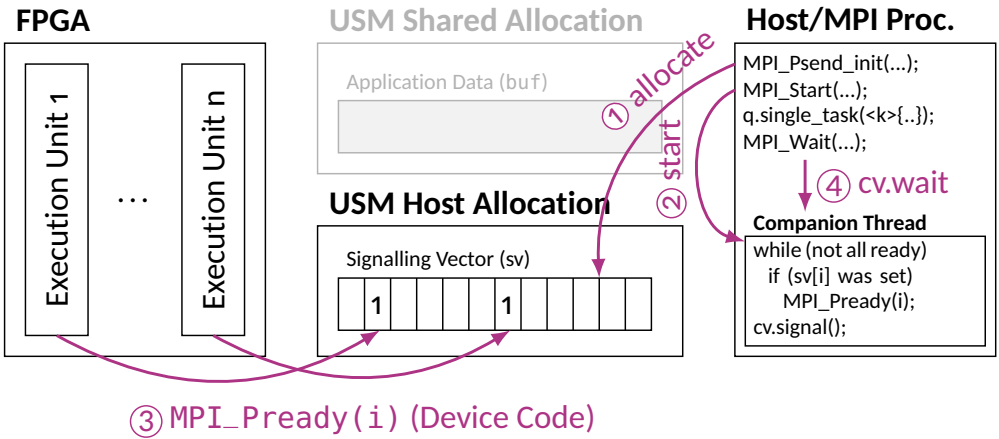
   Favorite!

2. **Polling**:
   - Write tuple to shared data structure + poll for changes
   - Options: ring buffer, queue, per-request **bit vector**

Host side: enable concurrent message transfers → **additional thread** for progress

# Signalling in Prototype Implementation

## Sender Side



# *Design Space: Request Passing*

ZIB
ZUSE INSTITUTE BERLIN

Challenges:

1. **Opaque MPI request** object (Pointer, Integer) → internal information not FPGA-accessible
2. Request-specific data required → e.g. pointer to Signalling Vector

Possible Solutions

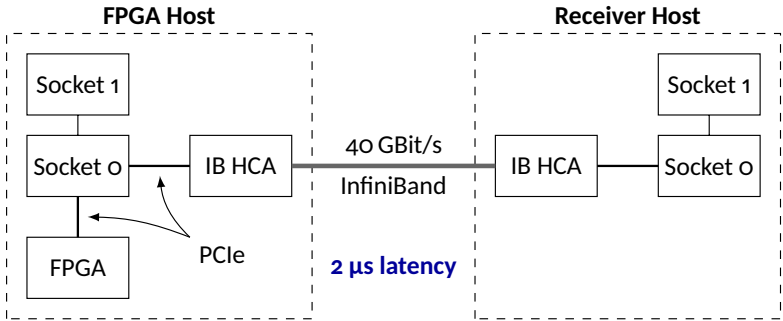
- New MPI API call(s) to create **device-specific requests** (containing required data)
  - Currently no SYCL binding
    - Generic return type (`void*`) needed → implies possibly costly pointer dereferencing
    - For SYCL-based implementation `sycl::queue` object required, not transparently accessible
  - Wish for SYCL binding, like `mpi::sycl::get_device_request(queue, mpi_request)`
- **Exposure** of internal information → bad from software perspective

# Prototype Implementation Details

- Lightweight C++ header-only on top of MPICH (should work with Open MPI as well)
- Limited to sender side only
- Platform: Intel PAC D5005 Card, Stratix 10 FPGA
- 6358 Logic Cells (1 %), 24 Block RAM Units (< 1 %), 340 MHz frequency
- Work-around for request passing issue
  - Wrapper around native `MPI_Request` + overloaded MPI methods
  - Request captured by C++ Kernel Lambda → device-accessible

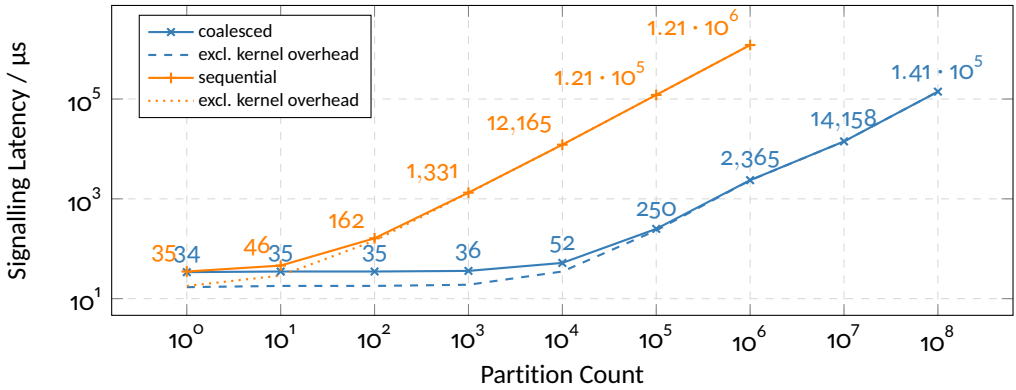
# Test-Bed for Experimental Evaluation

- Two nodes, dual-socket nodes (Intel Xeon 6246 (12C), 3.3 GHz)
- Back-to-back 40 GBit/s InfiniBand Connection, Mellanox Connect X-5 MT27800
- FPGA on single host only



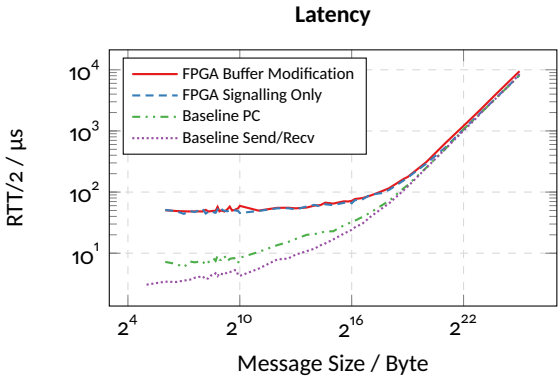
# Evaluation: Signalling Latency

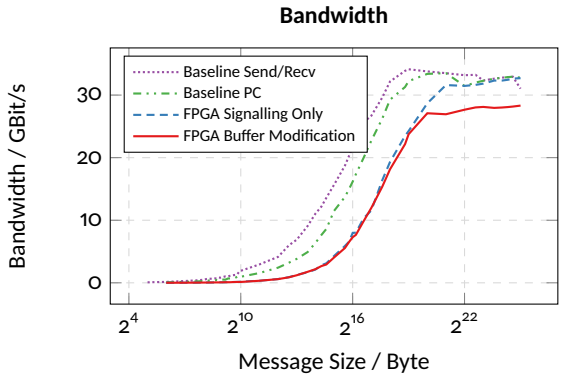
- Kernel marks partitions as ready (unrolled/coalesced and sequential version)
- No MPI communication, no buffer transfers/modifications
- Measure time from kernel launch to completion (**launch overhead: 18 µs**)



# Communication Latency and Bandwidth

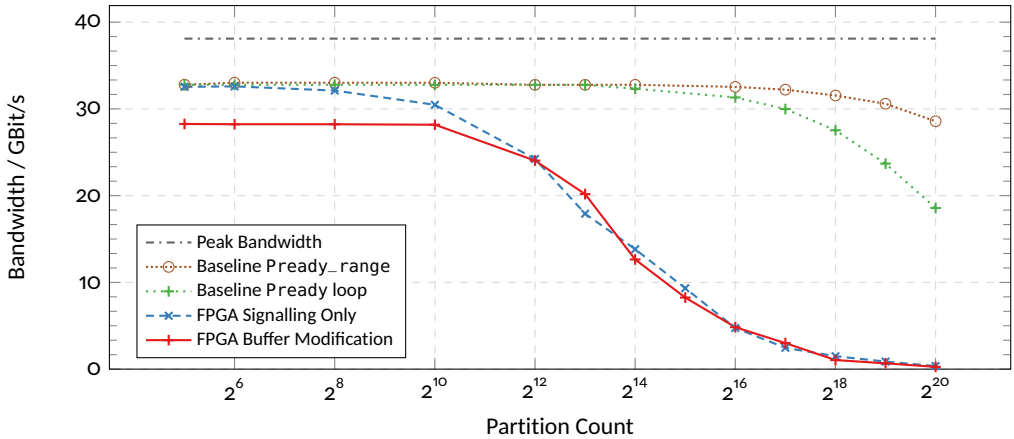
- Classic MPI **pingpong benchmark**, reporting 1/2 Round-Trip Time (RTT), 32 partitions
- Communication between FPGA (“Ping” sender) and receiving host (CPU-only process)



# Partition Count Scaling

- Pingpong with fixed message size = 32 MByte



# *Summary*

- MPI Partitioned Communication for FPGAs **is feasible** today
- **Interoperability challenges** exist, but solvable
- Kernel startup and signalling time limit communication performance

Thank you for your attention.

Questions?