13th International Workshop on OpenCL and SYCL

IWOCL 2025

Evaluating oneAPI I/O Pipes in a Case Study of Scaling a SYCL Jacobi Solver to multiple FPGAs

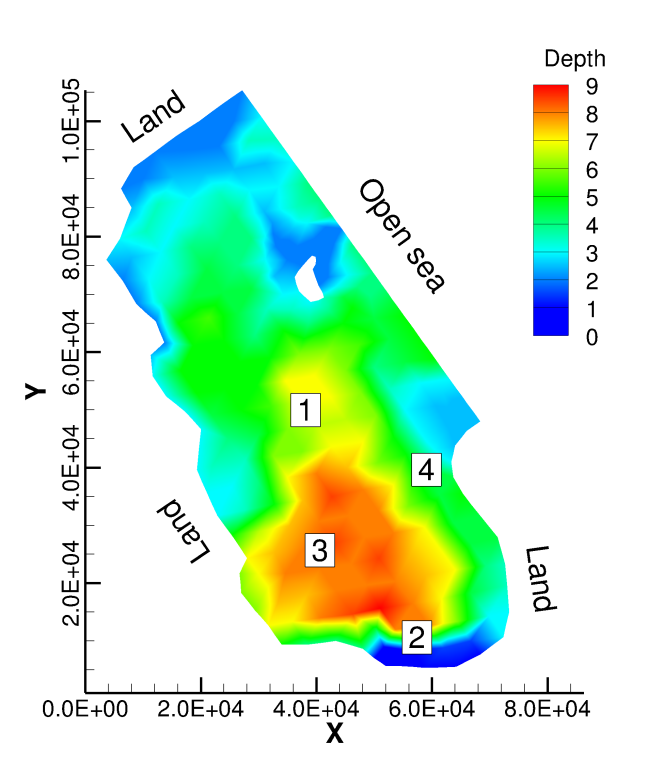
Christoph Alt, Paderborn Center for Parallel Computing, Paderborn University

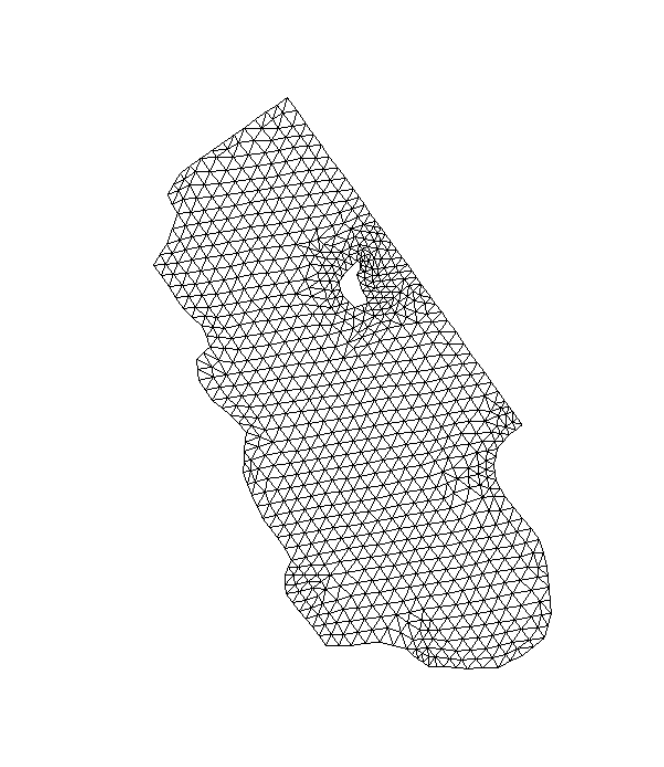
Tobias Kenter, Christian Plessl; Paderborn Center for Parallel Computing, Paderborn University

April 7-11, 2025 | Heidelberg, Germany | iwocl.org

Motivation

Shallow water applications (SWE) on FPGAs





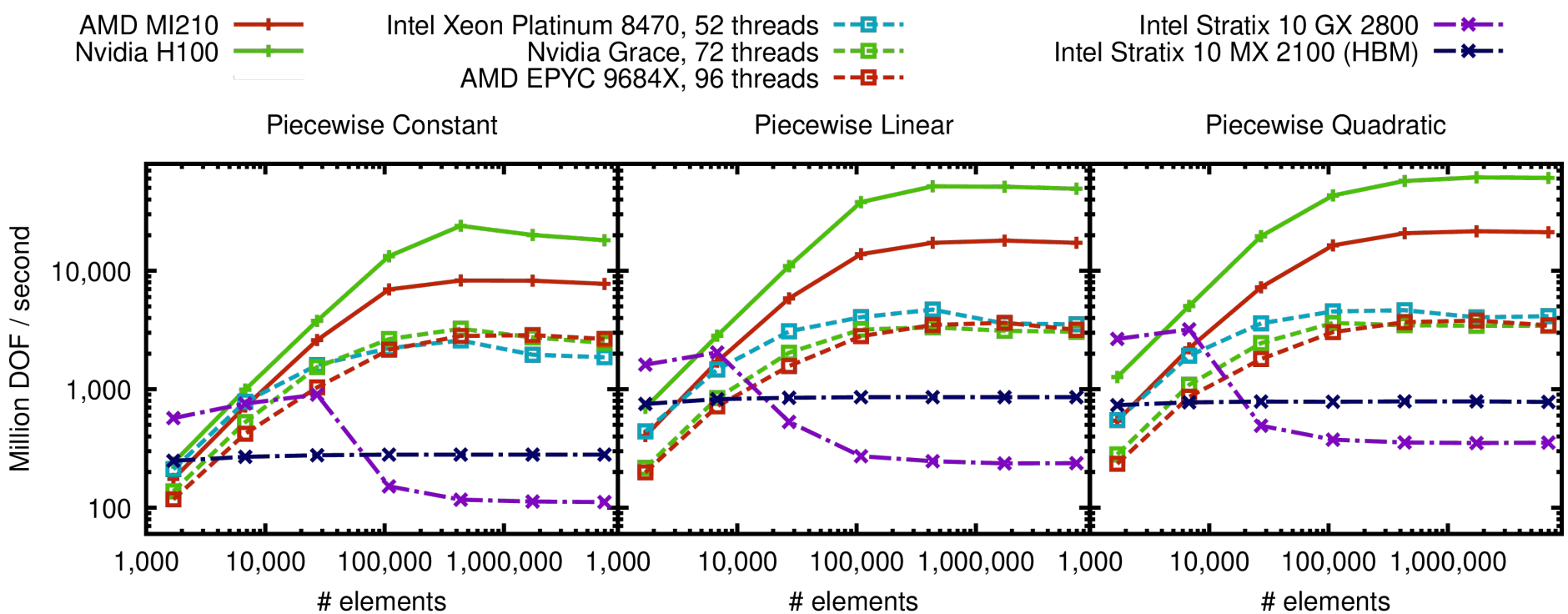
Bahamas (Bight of Abaco): Bathymetry, boundary type, and observation stations (left), original triangular mesh with 1696 elements (right).

First FPGA Design[1]

- Single FPGA
- OpenCL
- fast, but limited to smaller grid size per device

1. Kenter, T., Shambhu, A., Faghih-Naini, S., Aizinger, V., 2021. Algorithm-hardware co-design of a discontinuous Galerkin shallow-water model for a dataflow architecture on FPGA, in: Proceedings of the Platform for Advanced Scientific Computing Conference, ACM, Geneva Switzerland. pp. 1–11. URL: https://dl.acm.org/doi/10.1145/3468267.3470617, doi:10.1145/3468267.3470617.

Portable SYCL implementation on CPUs, GPUs and FPGAs[1]



Array-specific Caches on FPGAs

1. Büttner et. al., Enabling performance portability for shallow water equations on cpus, gpus, and fpgas with sycl. In Proceedings of the Platform for Advanced Scientific Computing Conference, PASC ’24, Association for Computing Machinery.

C.Alt christoph.alt@uni-paderborn.de

Paderborn Center for Parallel Computing

Goals of this work

- Preparation of multi-FPGA support for SYCL-based SWE implementation
- Previous OpenCL multi-FPGA Designs[1]
- Explore oneAPI functionality for direct FPGA-to-FPGA communication

=> Simple 2D multi-FPGA Jacobi Solver

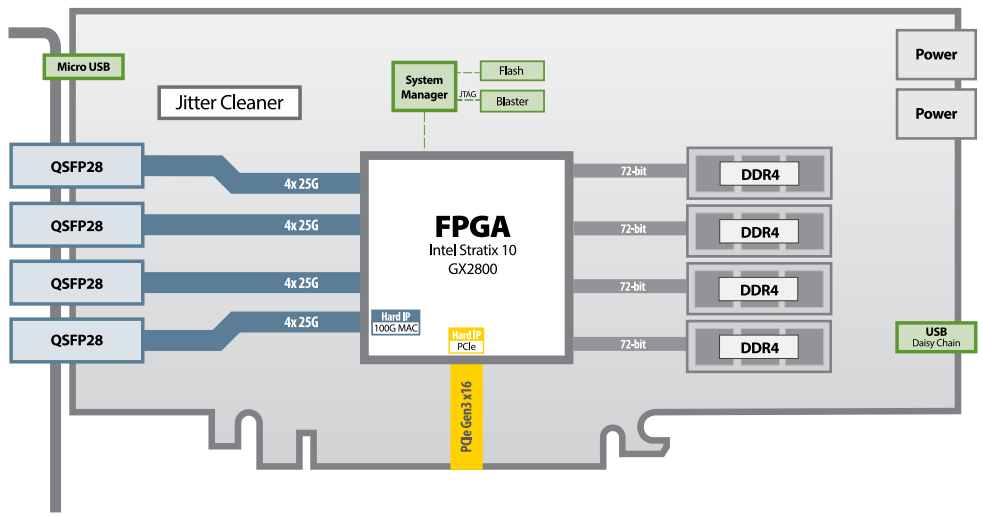
```
for (int i = 0; i < x_dim; i++){
  for(int j = 0; j < y_dim; j++){
    dst[i, j] = (src[i + 1, j] + src[i - 1, j] + src[i, j - 1] + src[i, j + 1]) * 0.25;
  }
}
```

1. Faj et. al. Scalable multi-FPGA design of a discontinuous galerkin shallow-water model on unstructured meshes. In Proc. Platform for Advanced Scientific Computing Conf. (PASC), pages 1–12, 2023

Background

Used FPGA: Bittware 520N

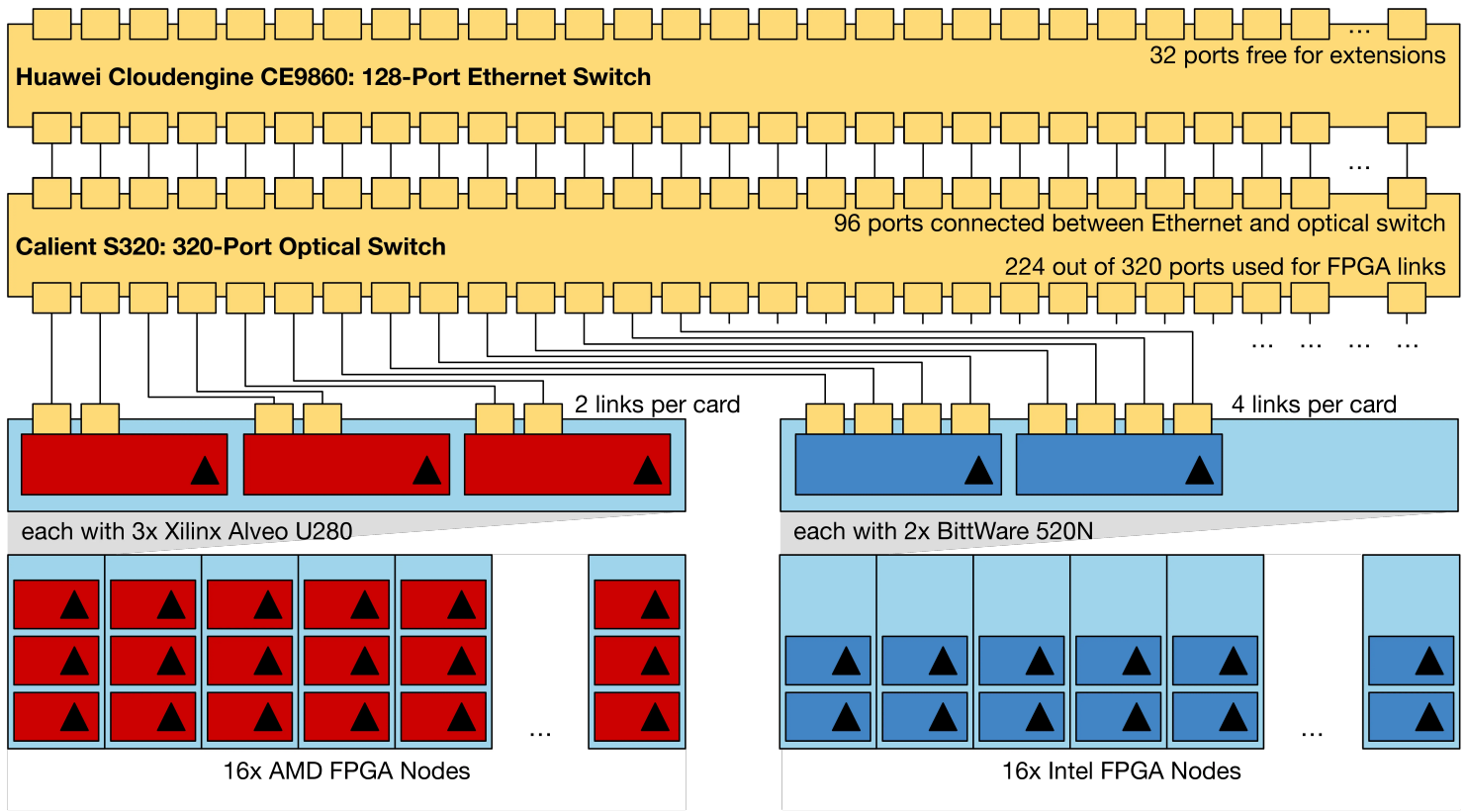
- Intel Stratix 10 GX 2800
- 32 GB DDR Memory
- Resources:
  - 5,760 DSP blocks (1 single precision FMA/cycle)
  - 11,721 M20K RAM blocks (20Kb each)
  - 933,120 logic blocks
- 4 QSFP+ ports



Block Diagram of the Bittware 520N[1]

1. Bittware https://www.bittware.com/products/520n/

Noctua 2 - FPGA Partition



Schematic view of FPGA partition and FPGA-to-FPGA interconnect[1]

1. Bauer et. al. Noctua 2 supercomputer. Journal of large-scale research facilities (JLSRF), 9(1), 2024.

oneAPI I/O Pipes - Usage[1]

```
struct kernel_output_ch0 {
    static constexpr unsigned id = 1;                                    ①
};

struct kernel_input_ch1 {
    static constexpr unsigned id = 2;                                    ②
};

constexpr size_t channel_width = 256 / 8;                                ③
template <typename T>
struct channel_data {
    T data[channel_width/sizeof(T)];
};

using sycl::ext::intel;
template <typename T, size_t channel_depth=512>
using read_pipe = kernel_readable_io_pipe<kernel_input_ch1, channel_data<T>, channel_depth>;

template <typename T, size_t channel_depth=512>
using write_pipe = kernel_writeable_io_pipe<kernel_output_ch0, channel_data<T>, channel_depth>;
```

① Writes on QSFP port 0

② Reads from QSFP port 1

③ Ports are 256 bit wide

1. Intel® oneAPI DPC++/C++ Compiler Handbook for FPGAs

oneAPI I/O Pipes - Usage[1]

```
struct kernel_output_ch0 {
    static constexpr unsigned id = 1;                                   ①
};

struct kernel_input_ch1 {
    static constexpr unsigned id = 2;                                   ②
};

constexpr size_t channel_width = 256 / 8;                               ③
template <typename T>
struct channel_data {
    T data[channel_width/sizeof(T)];
};

using sycl::ext::intel;
template <typename T, size_t channel_depth=512>
using read_pipe = kernel_readable_io_pipe<kernel_input_ch1, channel_data<T>, channel_depth>;

template <typename T, size_t channel_depth=512>
using write_pipe = kernel_writeable_io_pipe<kernel_output_ch0, channel_data<T>, channel_depth>;
```

① Writes on QSFP port 0

② Reads from QSFP port 1

③ Ports are 256 bit wide

1. Intel® oneAPI DPC++/C++ Compiler Handbook for FPGAs

oneAPI I/O Pipes - Usage[1]

```
struct kernel_output_ch0 {
    static constexpr unsigned id = 1;                                    ①
};

struct kernel_input_ch1 {
    static constexpr unsigned id = 2;                                    ②
};

constexpr size_t channel_width = 256 / 8;                                ③
template <typename T>
struct channel_data {
    T data[channel_width/sizeof(T)];
};

using sycl::ext::intel;
template <typename T, size_t channel_depth=512>
using read_pipe = kernel_readable_io_pipe<kernel_input_ch1, channel_data<T>, channel_depth>;

template <typename T, size_t channel_depth=512>
using write_pipe = kernel_writeable_io_pipe<kernel_output_ch0, channel_data<T>, channel_depth>;
```

① Writes on QSFP port 0

② Reads from QSFP port 1

③ Ports are 256 bit wide

1. Intel® oneAPI DPC++/C++ Compiler Handbook for FPGAs

oneAPI I/O Pipes - Usage[1]

```
struct kernel_output_ch0 {
    static constexpr unsigned id = 1;                                    ①
};

struct kernel_input_ch1 {
    static constexpr unsigned id = 2;                                    ②
};

constexpr size_t channel_width = 256 / 8;                                ③
template <typename T>
struct channel_data {
    T data[channel_width/sizeof(T)];
};

using sycl::ext::intel;
template <typename T, size_t channel_depth=512>
using read_pipe = kernel_readable_io_pipe<kernel_input_ch1, channel_data<T>, channel_depth>;

template <typename T, size_t channel_depth=512>
using write_pipe = kernel_writeable_io_pipe<kernel_output_ch0, channel_data<T>, channel_depth>;
```

① Writes on QSFP port 0

② Reads from QSFP port 1

③ Ports are 256 bit wide

1. Intel® oneAPI DPC++/C++ Compiler Handbook for FPGAs

oneAPI I/O Pipes - Usage[1]

```
struct kernel_output_ch0 {
    static constexpr unsigned id = 1;                                    ①
};

struct kernel_input_ch1 {
    static constexpr unsigned id = 2;                                    ②
};

constexpr size_t channel_width = 256 / 8;                                ③
template <typename T>
struct channel_data {
    T data[channel_width/sizeof(T)];
};

using sycl::ext::intel;
template <typename T, size_t channel_depth=512>
using read_pipe = kernel_readable_io_pipe<kernel_input_ch1, channel_data<T>, channel_depth>;

template <typename T, size_t channel_depth=512>
using write_pipe = kernel_writeable_io_pipe<kernel_output_ch0, channel_data<T>, channel_depth>;
```

① Writes on QSFP port 0

② Reads from QSFP port 1

③ Ports are 256 bit wide

1. Intel® oneAPI DPC++/C++ Compiler Handbook for FPGAs

oneAPI I/O Pipes - Usage[1]

```
struct kernel_output_ch0 {
    static constexpr unsigned id = 1;                                    ①
};

struct kernel_input_ch1 {
    static constexpr unsigned id = 2;                                    ②
};

constexpr size_t channel_width = 256 / 8;                                ③
template <typename T>
struct channel_data {
    T data[channel_width/sizeof(T)];
};

using sycl::ext::intel;
template <typename T, size_t channel_depth=512>
using read_pipe = kernel_readable_io_pipe<kernel_input_ch1, channel_data<T>, channel_depth>;

template <typename T, size_t channel_depth=512>
using write_pipe = kernel_writeable_io_pipe<kernel_output_ch0, channel_data<T>, channel_depth>;
```

① Writes on QSFP port 0

② Reads from QSFP port 1

③ Ports are 256 bit wide

1. Intel® oneAPI DPC++/C++ Compiler Handbook for FPGAs

oneAPI I/O Pipes - Usage

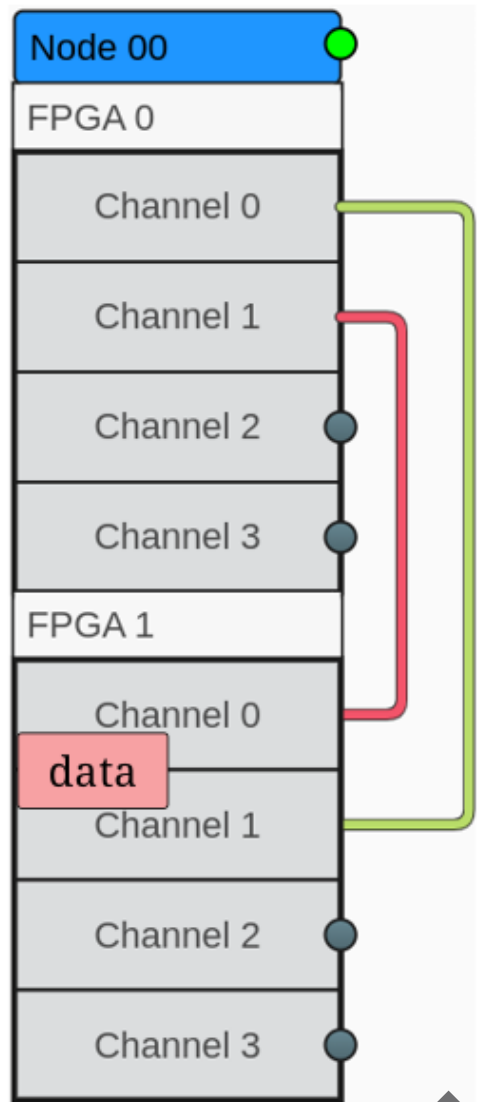
```
1
2 channel_data<float> data = read_pipe<float>::read();                ①
3
4 for (int i = 0; i < sending_size; i++){
5   data.data[i] += 1;
6 }
7
8 write_pipe<float>::write(data);
```

① Read on channel 1

oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);
```

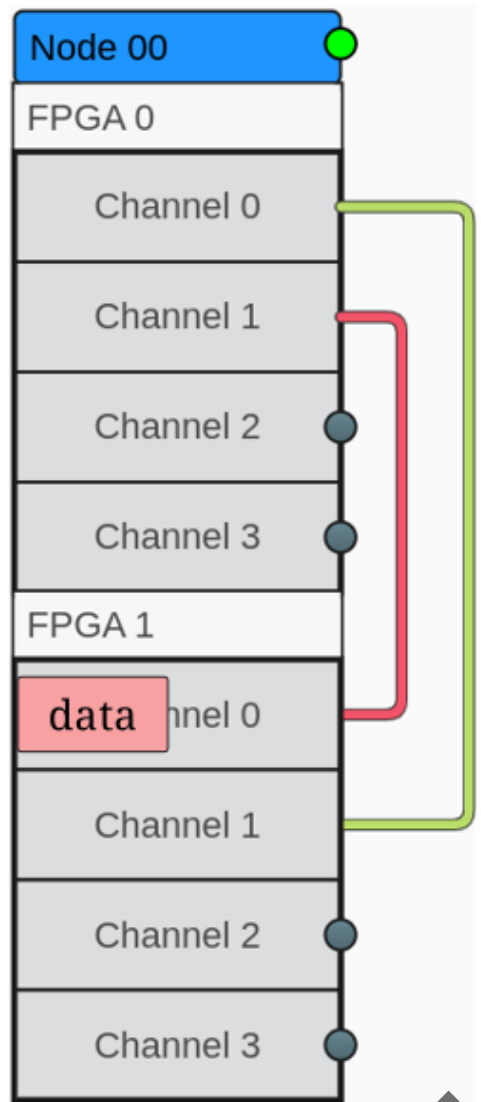
① Read on channel 1



oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);
```

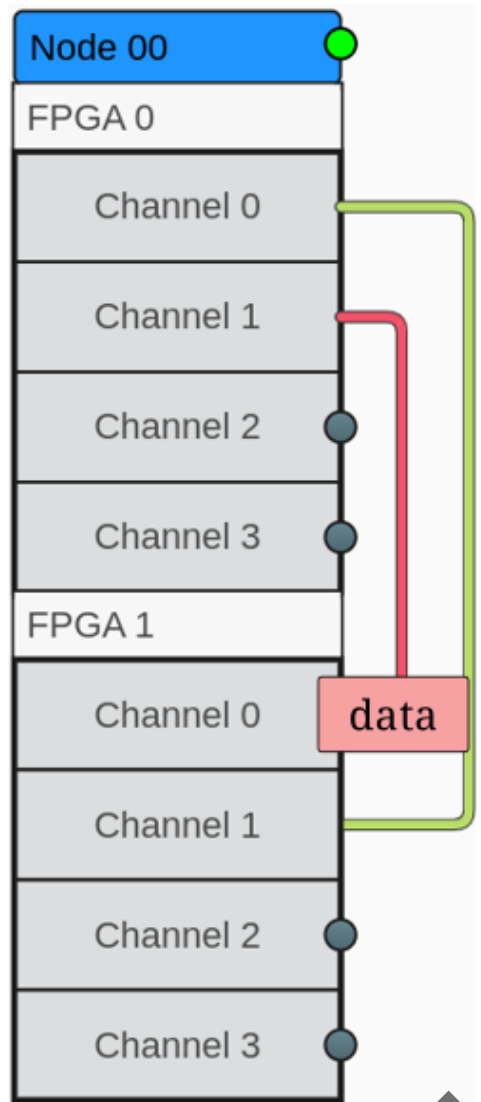
① Read on channel 1



oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);
```

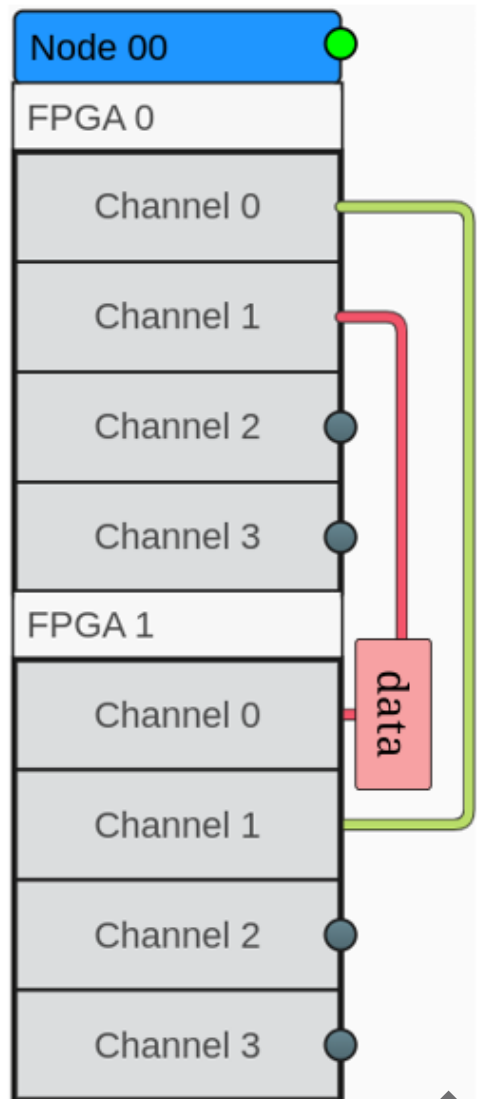
① Read on channel 1



oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);
```

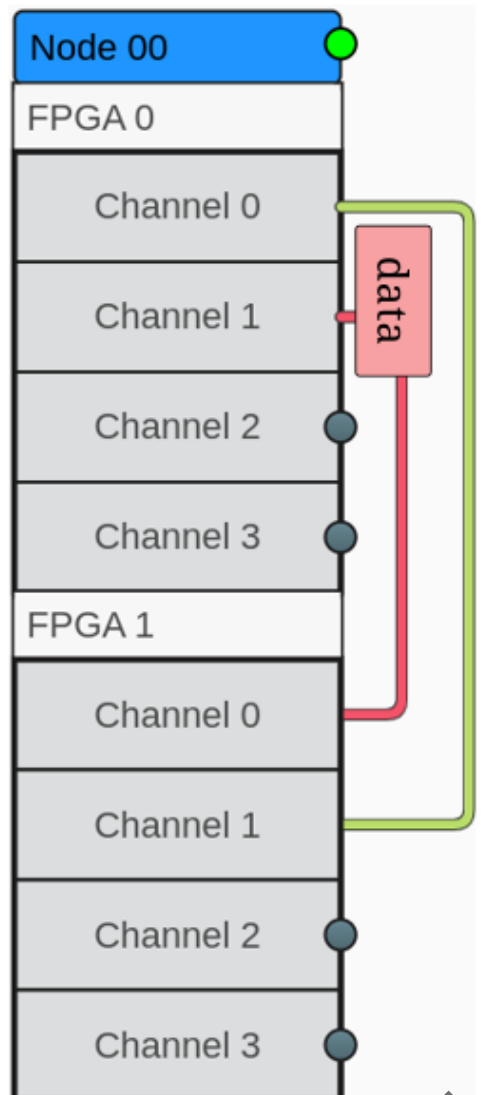
① Read on channel 1



oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);
```

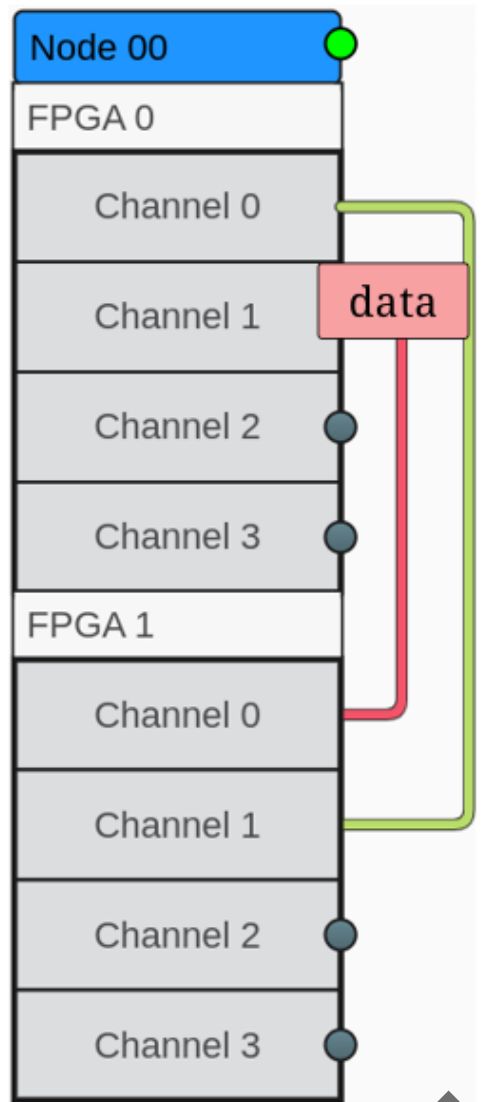
① Read on channel 1



oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);
```

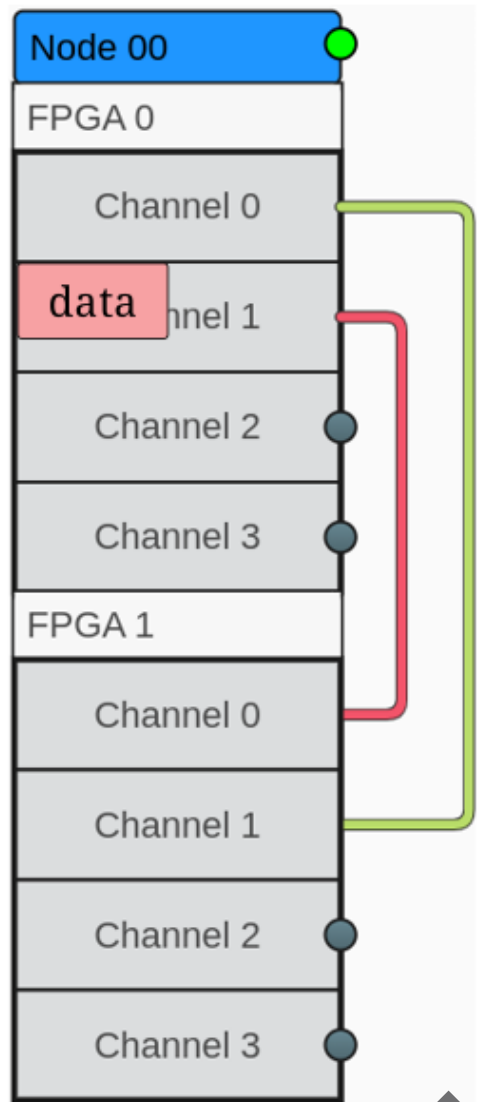
① Read on channel 1



oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);
```

① Read on channel 1

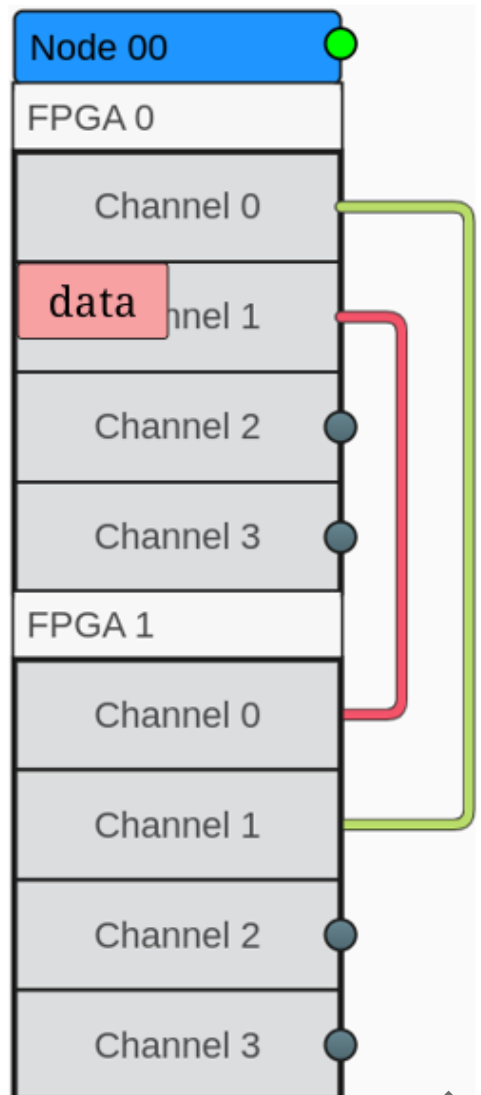


oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);                                     ②
```

① Read on channel 1

② Write on channel 0



oneAPI I/O Pipes - Usage

```
1
2 channel_data<float> data = read_pipe<float>::read();                ①
3
4 for (int i = 0; i < sending_size; i++){
5   data.data[i] += 1;
6 }
7
8 write_pipe<float>::write(data);                                     ②
```

① Read on channel 1

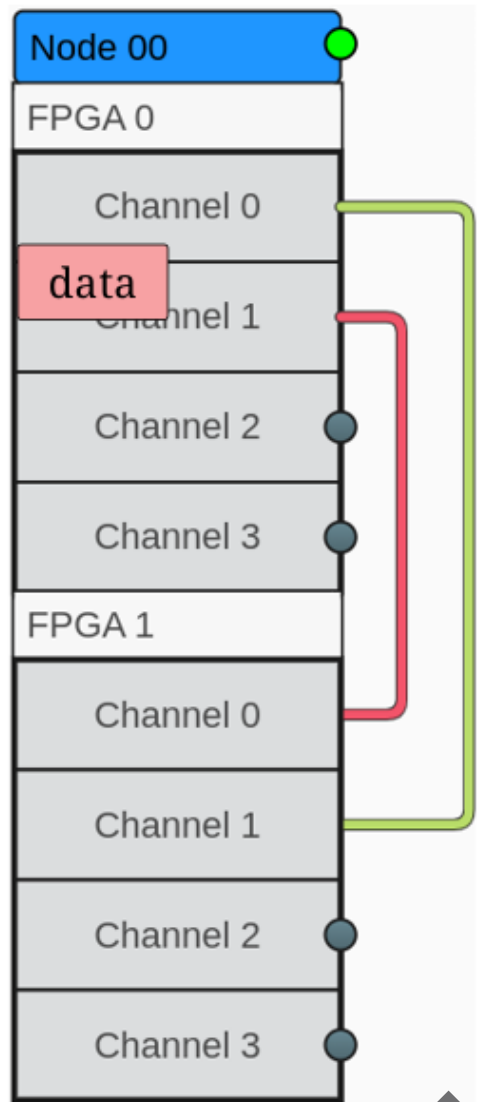
② Write on channel 0

oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);                                     ②
```

① Read on channel 1

② Write on channel 0

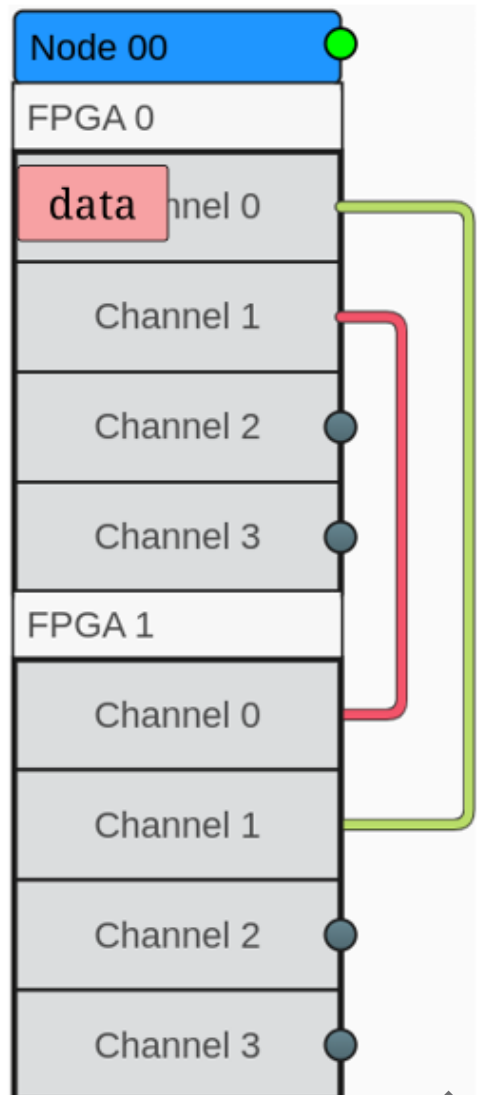


oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);                                     ②
```

① Read on channel 1

② Write on channel 0

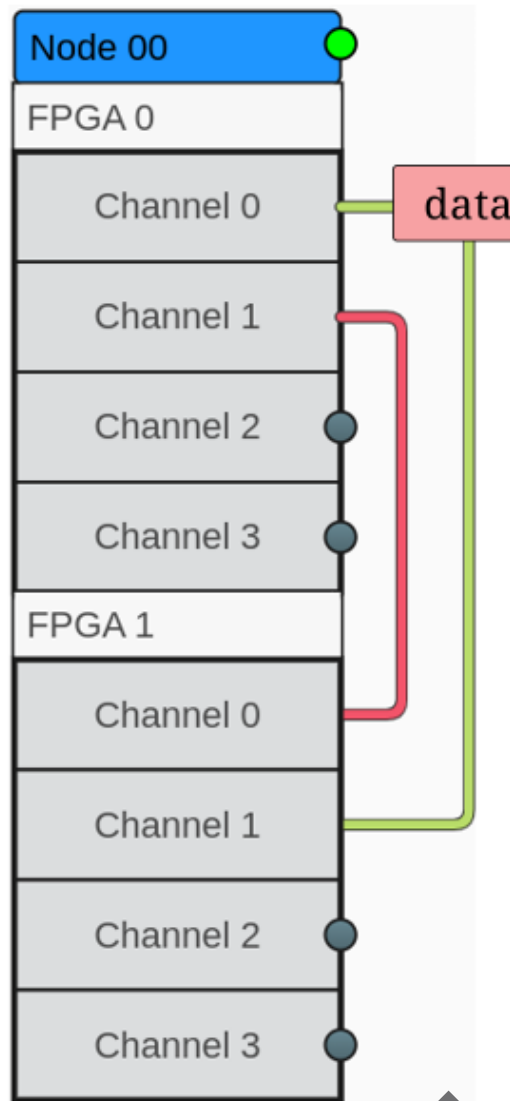


oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);                                     ②
```

① Read on channel 1

② Write on channel 0

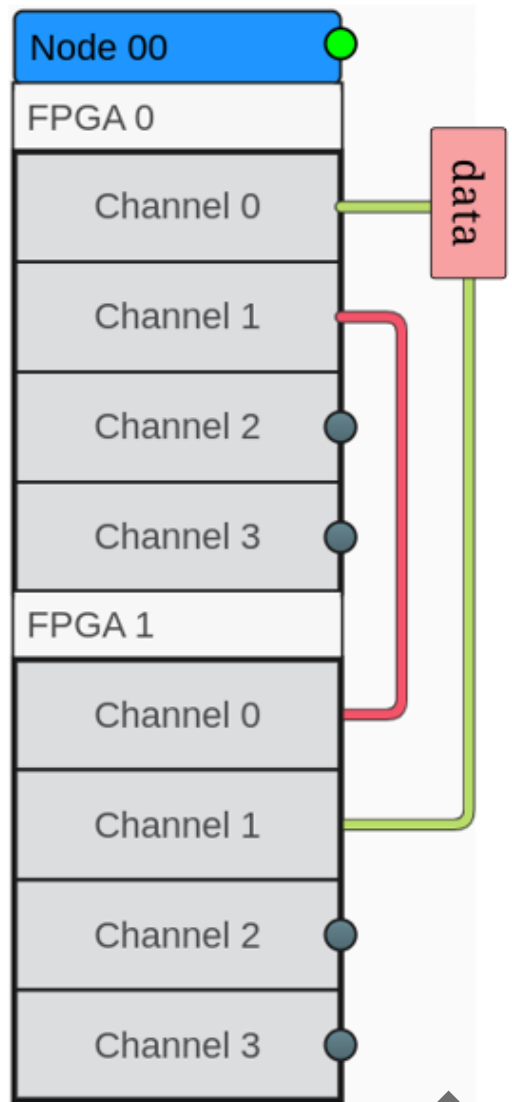


oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);                                     ②
```

① Read on channel 1

② Write on channel 0

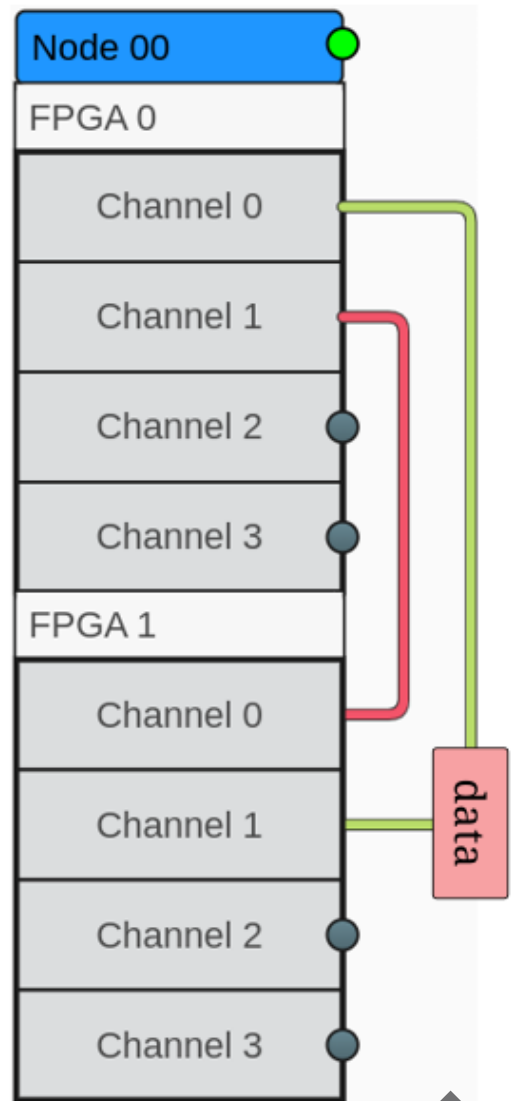


oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);                                     ②
```

① Read on channel 1

② Write on channel 0

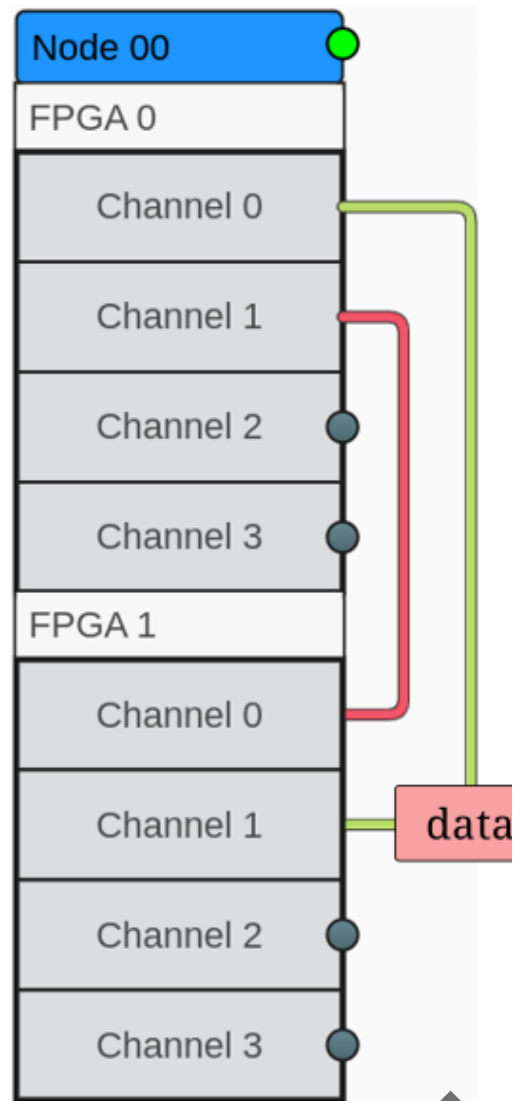


oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);                                     ②
```

① Read on channel 1

② Write on channel 0

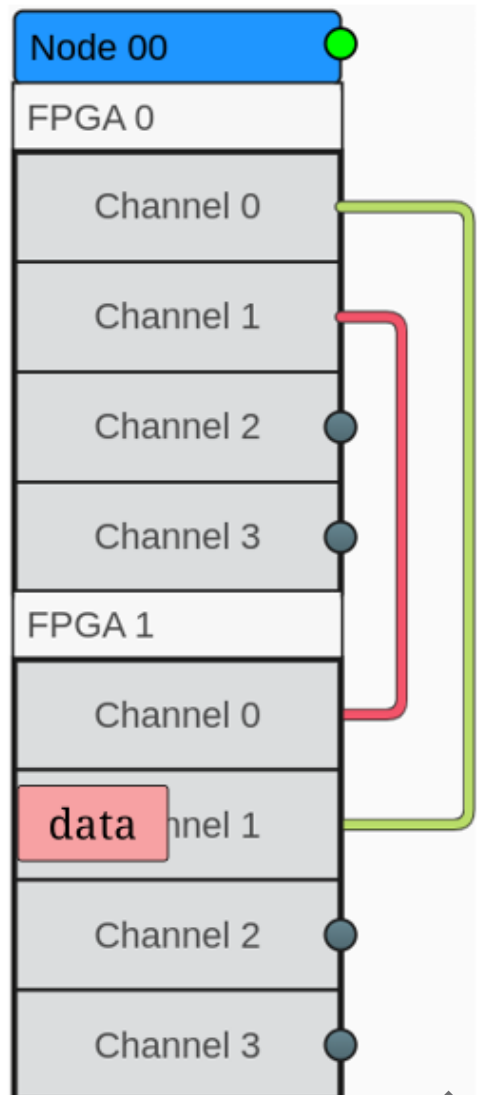


oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);                                     ②
```

① Read on channel 1

② Write on channel 0

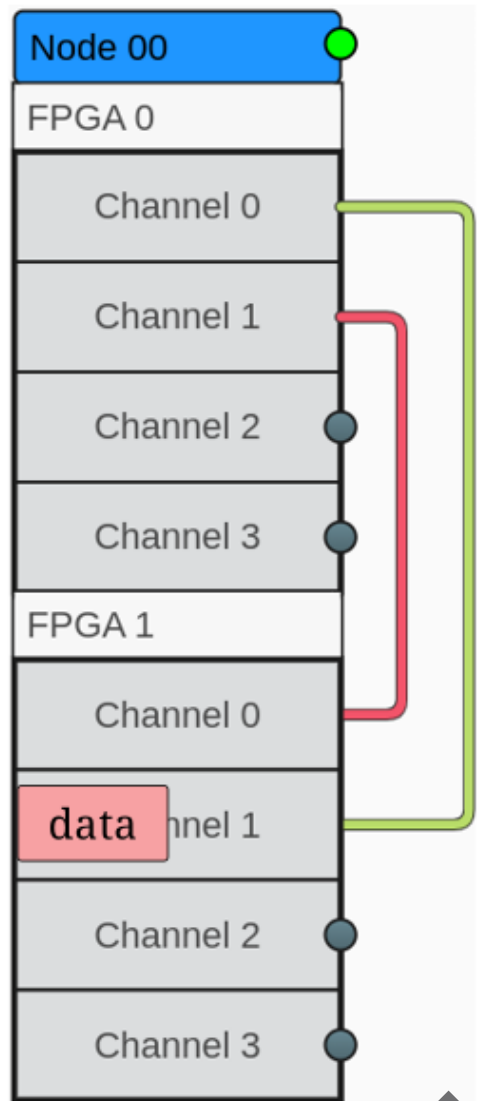


oneAPI I/O Pipes - Usage

```
1
2  channel_data<float> data = read_pipe<float>::read();                ①
3
4  for (int i = 0; i < sending_size; i++){
5    data.data[i] += 1;
6  }
7
8  write_pipe<float>::write(data);                                     ②
```

① Read on channel 1

② Write on channel 0



Channel Mapping Problem

- since oneAPI version 2023.2
- If only 1 read/write in kernel they are mapped to channel 0
- If more reads/writes the mapping gets unpredictable

<projectname>.bc.xml

```
<INTERFACE port="iord_bl_io_acl_c_kernel_input_ch1_pipe_channel"
           ... chan_id="kernel_input_ch0" />
<INTERFACE port="iowr_bl_io_acl_c_kernel_output_ch0_pipe_channel"
           ... chan_id="kernel_output_ch0" />
```

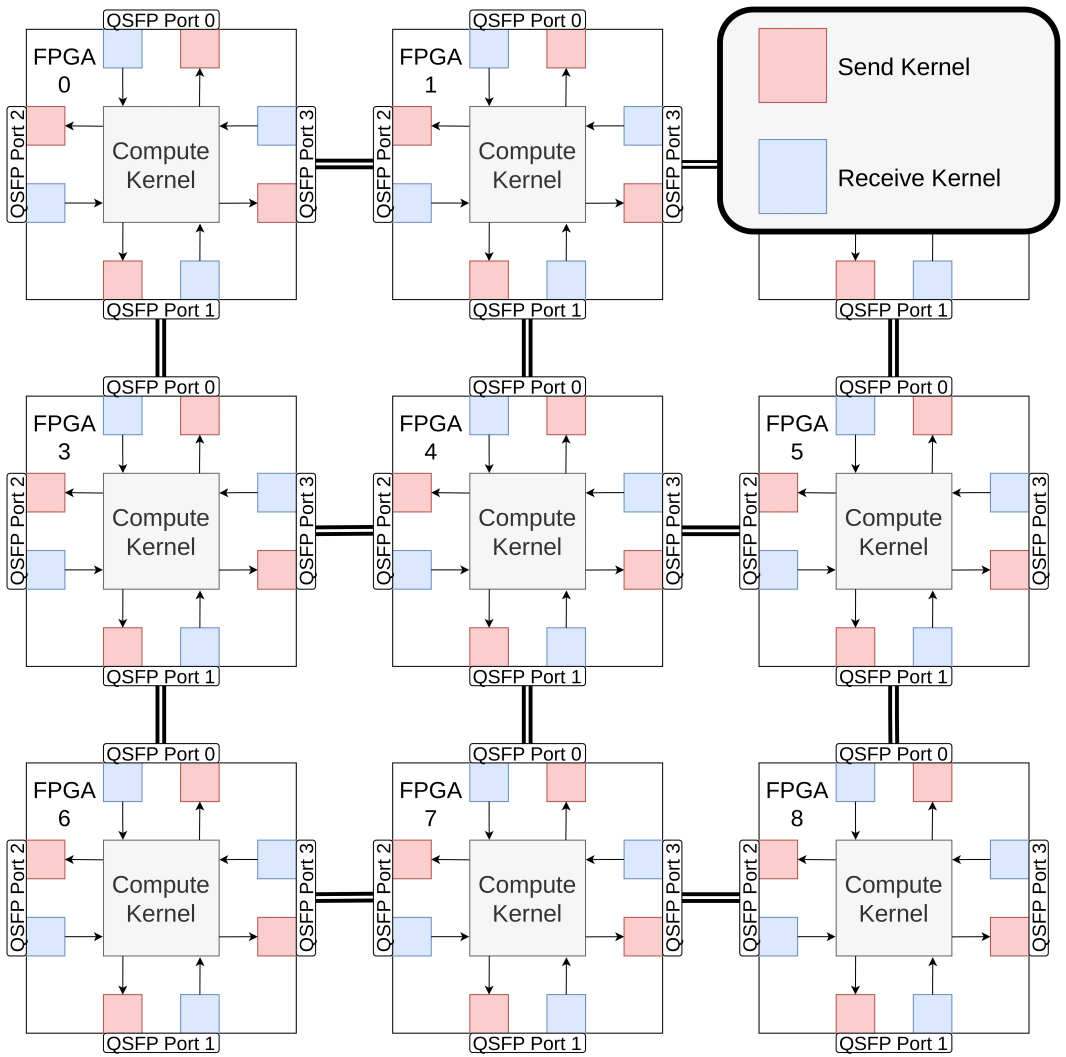
- Fix on Noctua 2:
  - `system_integrator` is called after `<projectname>.bc.xml` is written
  - wrapper for `system_integrator`:
    - fixes `<projectname>.bc.xml`
    - and proceeds with regular `system_integrator`

Implementation

Design

- MPI-ranks launch kernels
- Every FPGA has (up to) 4 neighbours
- Compute kernel + 4 sending kernels + 4 receiving kernels per device
- I/O kernels interact with I/O pipes
- Compute kernel and I/O Kernels communicate with local pipes
- Compute loop is unrolled 16 times

Topology

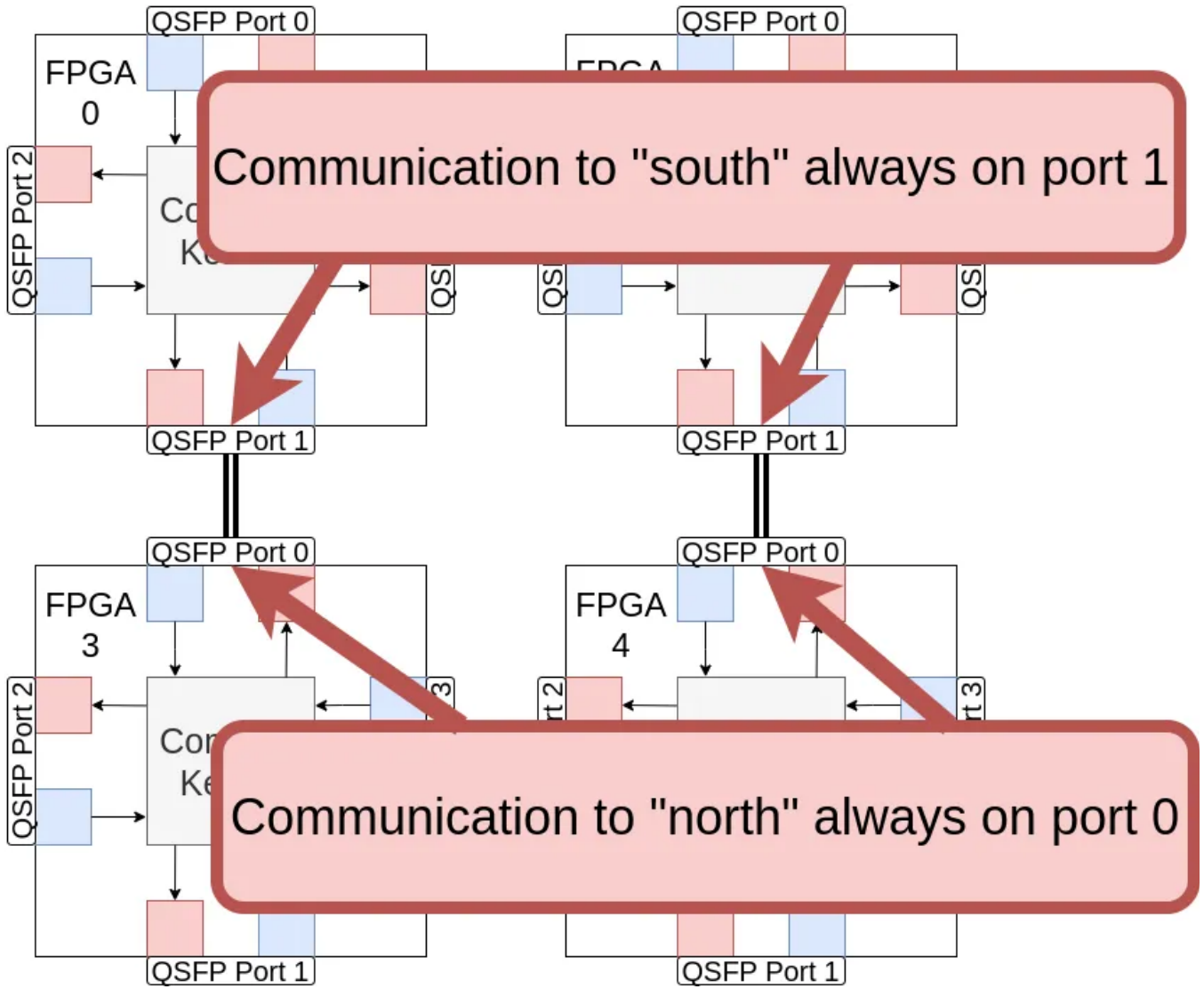


2D Torus topology in a 3 × 3 layout, the direct connections between the FPGAs is realized via an Optical Circuit Switch. On every FPGA for every QSFP+ port an extra I/O kernel for sending and one for receiving data is launched.

C.Alt christoph.alt@uni-paderborn.de

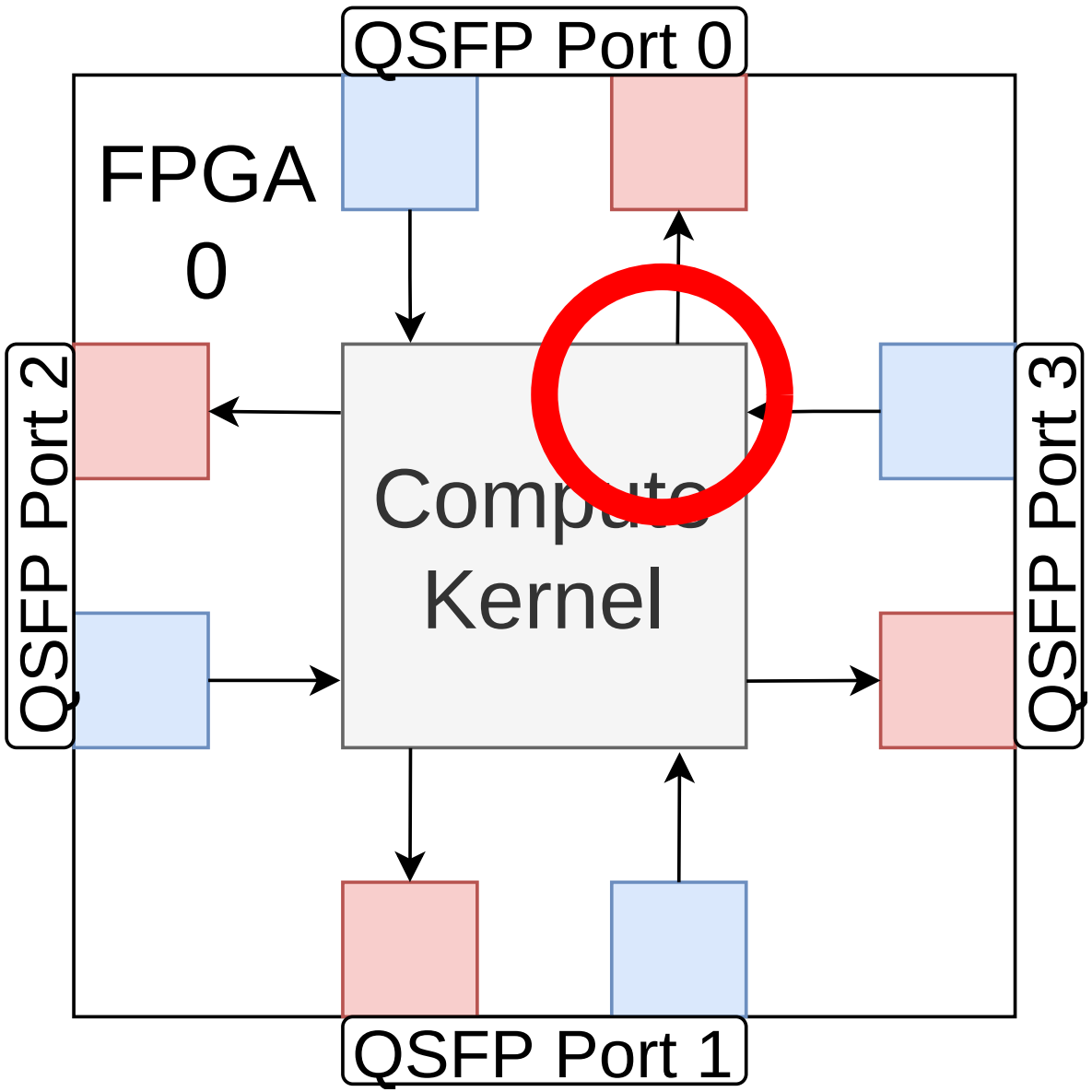
Paderborn Center for Parallel Computing

Topology



2D Torus topology in a 3 × 3 layout, the direct connections between the FPGAs is realized via an Optical Circuit Switch. On every FPGA for every QSFP+ port an extra I/O kernel for sending and one for receiving data is launched.

Sending data to I/O kernel



Sending data to I/O kernel

```
index_t sending_size = 8; // 256 bit / 8  / sizeof(float)
index_t comm_dims[4] = {y_dim, y_dim, x_dim, x_dim};
index_t x_sends = (x_dim + sending_size - 1)/sending_size;
index_t y_sends = (y_dim + sending_size - 1)/sending_size;
index_t send_limits[4] = {x_sends, x_sends, y_sends, y_sends};

using SendPipes = fpga_tools::PipeArray<class SendPipe, float, PIPE_DEPTH, 4>;

index_t total_sends = std::max(x_sends, y_sends) * sending_size;

for (index_t i = 0; i < total_sends; i++) {
  // channel_id => 0=north, 1=south, 2=west, 3=east
  fpga_tools::UnrolledLoop<4>([&](auto channel_id) {                    ①
    if (has[channel_id] && // check for neighbor
     i < send_limits[channel_id] * sending_size) {
       float v = (i < comm_dims[channel_id])
               ? send_buffer[i] : NAN;
       SendPipeAt<channel_id>::write(value);                            ②
    }
  });
}
```

① Using `UnrolledLoop` from [oneAPI Samples Repository]

② Using `PipeArray` from [oneAPI Samples Repository]

Sending data to I/O kernel

```
index_t sending_size = 8; // 256 bit / 8  / sizeof(float)
index_t comm_dims[4] = {y_dim, y_dim, x_dim, x_dim};
index_t x_sends = (x_dim + sending_size - 1)/sending_size;
index_t y_sends = (y_dim + sending_size - 1)/sending_size;
index_t send_limits[4] = {x_sends, x_sends, y_sends, y_sends};

using SendPipes = fpga_tools::PipeArray<class SendPipe, float, PIPE_DEPTH, 4>;

index_t total_sends = std::max(x_sends, y_sends) * sending_size;

for (index_t i = 0; i < total_sends; i++) {
  // channel_id => 0=north, 1=south, 2=west, 3=east
  fpga_tools::UnrolledLoop<4>([&](auto channel_id) {                    ①
    if (has[channel_id] && // check for neighbor
     i < send_limits[channel_id] * sending_size) {
       float v = (i < comm_dims[channel_id])
               ? send_buffer[i] : NAN;
       SendPipeAt<channel_id>::write(value);                            ②
    }
  });
}
```

① Using `UnrolledLoop` from [oneAPI Samples Repository]

② Using `PipeArray` from [oneAPI Samples Repository]

Sending data to I/O kernel

```
index_t sending_size = 8; // 256 bit / 8  / sizeof(float)
index_t comm_dims[4] = {y_dim, y_dim, x_dim, x_dim};
index_t x_sends = (x_dim + sending_size - 1)/sending_size;
index_t y_sends = (y_dim + sending_size - 1)/sending_size;
index_t send_limits[4] = {x_sends, x_sends, y_sends, y_sends};

using SendPipes = fpga_tools::PipeArray<class SendPipe, float, PIPE_DEPTH, 4>;

index_t total_sends = std::max(x_sends, y_sends) * sending_size;

for (index_t i = 0; i < total_sends; i++) {
  // channel_id => 0=north, 1=south, 2=west, 3=east
  fpga_tools::UnrolledLoop<4>([&](auto channel_id) {                    ①
    if (has[channel_id] && // check for neighbor
     i < send_limits[channel_id] * sending_size) {
       float v = (i < comm_dims[channel_id])
               ? send_buffer[i] : NAN;
       SendPipeAt<channel_id>::write(value);                            ②
    }
  });
}
```

① Using `UnrolledLoop` from [oneAPI Samples Repository]

② Using `PipeArray` from [oneAPI Samples Repository]

Sending data to I/O kernel

```
index_t sending_size = 8; // 256 bit / 8  / sizeof(float)
index_t comm_dims[4] = {y_dim, y_dim, x_dim, x_dim};
index_t x_sends = (x_dim + sending_size - 1)/sending_size;
index_t y_sends = (y_dim + sending_size - 1)/sending_size;
index_t send_limits[4] = {x_sends, x_sends, y_sends, y_sends};

using SendPipes = fpga_tools::PipeArray<class SendPipe, float, PIPE_DEPTH, 4>;

index_t total_sends = std::max(x_sends, y_sends) * sending_size;

for (index_t i = 0; i < total_sends; i++) {
  // channel_id => 0=north, 1=south, 2=west, 3=east
  fpga_tools::UnrolledLoop<4>([&](auto channel_id) {                    ①
    if (has[channel_id] && // check for neighbor
     i < send_limits[channel_id] * sending_size) {
       float v = (i < comm_dims[channel_id])
               ? send_buffer[i] : NAN;
       SendPipeAt<channel_id>::write(value);                            ②
    }
  });
}
```

① Using `UnrolledLoop` from [oneAPI Samples Repository]

② Using `PipeArray` from [oneAPI Samples Repository]

Sending data to I/O kernel

```
index_t sending_size = 8; // 256 bit / 8  / sizeof(float)
index_t comm_dims[4] = {y_dim, y_dim, x_dim, x_dim};
index_t x_sends = (x_dim + sending_size - 1)/sending_size;
index_t y_sends = (y_dim + sending_size - 1)/sending_size;
index_t send_limits[4] = {x_sends, x_sends, y_sends, y_sends};

using SendPipes = fpga_tools::PipeArray<class SendPipe, float, PIPE_DEPTH, 4>;

index_t total_sends = std::max(x_sends, y_sends) * sending_size;

for (index_t i = 0; i < total_sends; i++) {
  // channel_id => 0=north, 1=south, 2=west, 3=east
  fpga_tools::UnrolledLoop<4>([&](auto channel_id) {                    ①
    if (has[channel_id] && // check for neighbor
     i < send_limits[channel_id] * sending_size) {
       float v = (i < comm_dims[channel_id])
               ? send_buffer[i] : NAN;
       SendPipeAt<channel_id>::write(value);                            ②
    }
  });
}
```

① Using `UnrolledLoop` from [oneAPI Samples Repository]

② Using `PipeArray` from [oneAPI Samples Repository]

Sending data to I/O kernel

```
index_t sending_size = 8; // 256 bit / 8  / sizeof(float)
index_t comm_dims[4] = {y_dim, y_dim, x_dim, x_dim};
index_t x_sends = (x_dim + sending_size - 1)/sending_size;
index_t y_sends = (y_dim + sending_size - 1)/sending_size;
index_t send_limits[4] = {x_sends, x_sends, y_sends, y_sends};

using SendPipes = fpga_tools::PipeArray<class SendPipe, float, PIPE_DEPTH, 4>;

index_t total_sends = std::max(x_sends, y_sends) * sending_size;

for (index_t i = 0; i < total_sends; i++) {
  // channel_id => 0=north, 1=south, 2=west, 3=east
  fpga_tools::UnrolledLoop<4>([&](auto channel_id) {                    ①
    if (has[channel_id] && // check for neighbor
     i < send_limits[channel_id] * sending_size) {
       float v = (i < comm_dims[channel_id])
               ? send_buffer[i] : NAN;
       SendPipeAt<channel_id>::write(value);                            ②
    }
  });
}
```

① Using `UnrolledLoop` from [oneAPI Samples Repository]

② Using `PipeArray` from [oneAPI Samples Repository]

Sending data to I/O kernel

```
index_t sending_size = 8; // 256 bit / 8  / sizeof(float)
index_t comm_dims[4] = {y_dim, y_dim, x_dim, x_dim};
index_t x_sends = (x_dim + sending_size - 1)/sending_size;
index_t y_sends = (y_dim + sending_size - 1)/sending_size;
index_t send_limits[4] = {x_sends, x_sends, y_sends, y_sends};

using SendPipes = fpga_tools::PipeArray<class SendPipe, float, PIPE_DEPTH, 4>;

index_t total_sends = std::max(x_sends, y_sends) * sending_size;

for (index_t i = 0; i < total_sends; i++) {
  // channel_id => 0=north, 1=south, 2=west, 3=east
  fpga_tools::UnrolledLoop<4>([&](auto channel_id) {                    ①
    if (has[channel_id] && // check for neighbor
     i < send_limits[channel_id] * sending_size) {
       float v = (i < comm_dims[channel_id])
               ? send_buffer[i] : NAN;
       SendPipeAt<channel_id>::write(value);                            ②
    }
  });
}
```

① Using `UnrolledLoop` from [oneAPI Samples Repository]

② Using `PipeArray` from [oneAPI Samples Repository]

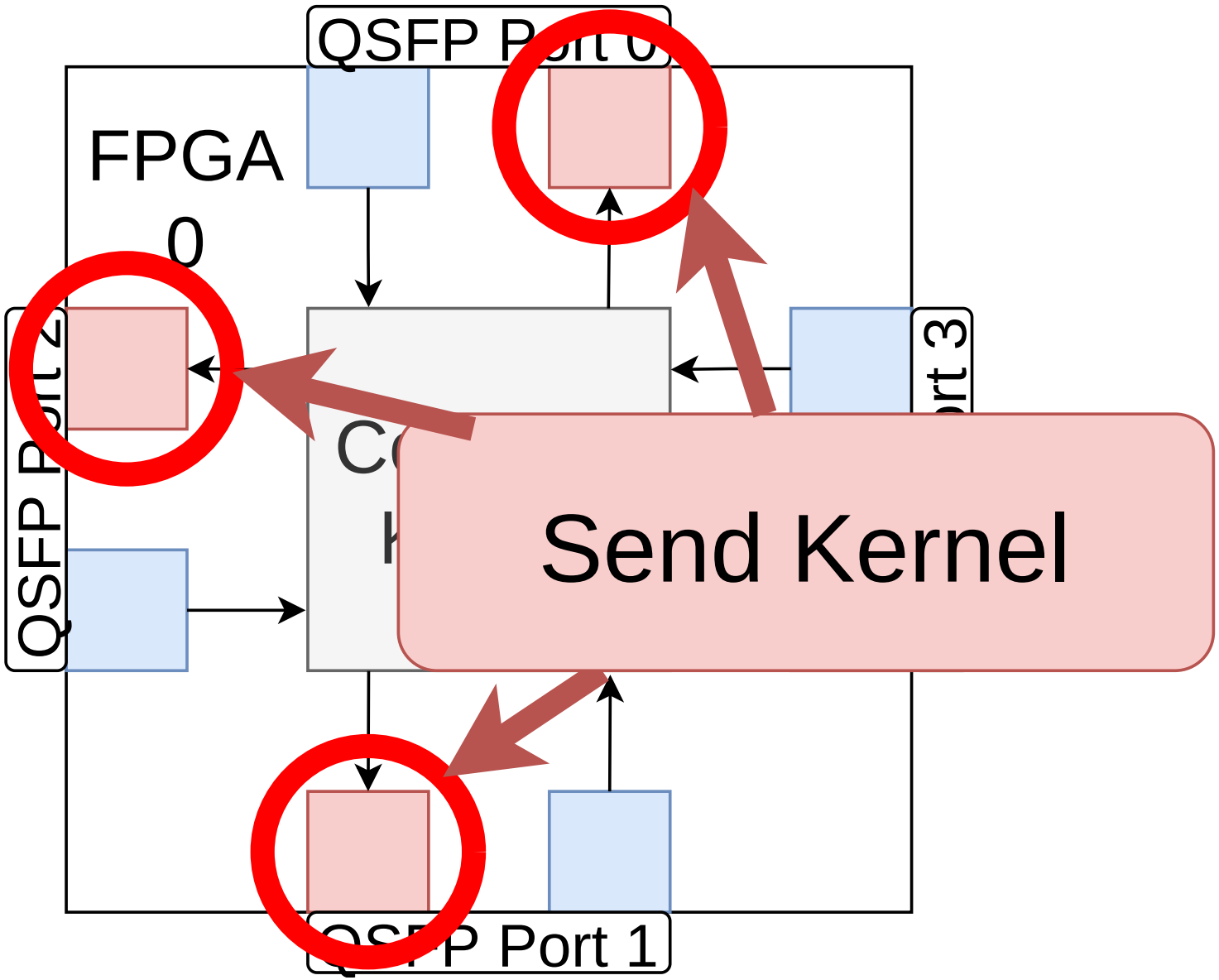
Sending data to I/O kernel

```
index_t sending_size = 8; // 256 bit / 8  / sizeof(float)
index_t comm_dims[4] = {y_dim, y_dim, x_dim, x_dim};
index_t x_sends = (x_dim + sending_size - 1)/sending_size;
index_t y_sends = (y_dim + sending_size - 1)/sending_size;
index_t send_limits[4] = {x_sends, x_sends, y_sends, y_sends};

using SendPipes = fpga_tools::PipeArray<class SendPipe, float, PIPE_DEPTH, 4>;

index_t total_sends = std::max(x_sends, y_sends) * sending_size;

for (index_t i = 0; i < total_sends; i++) {
  // channel_id => 0=north, 1=south, 2=west, 3=east
  fpga_tools::UnrolledLoop<4>([&](auto channel_id) {                    ①
    if (has[channel_id] && // check for neighbor
     i < send_limits[channel_id] * sending_size) {
       float v = (i < comm_dims[channel_id])
               ? send_buffer[i] : NAN;
       SendPipeAt<channel_id>::write(value);                            ②
    }
  });
}
```

① Using `UnrolledLoop` from [oneAPI Samples Repository]

② Using `PipeArray` from [oneAPI Samples Repository]

I/O Kernel for Sending



I/O Kernel for Sending

```
1  template <unsigned channel_id, typename real_t>
2  void submit_send_kernel(sycl::queue& q, bool send_on,
3          index_t time_steps, index_t sends) {
4   if (send_on) {
5    q.submit([&](sycl::handler& h) {
6    h.single_task<class_name>([=]() {
7     for (index_t t = 0; t < time_steps; t++) {
8       for (index_t s = 0; s < sends; s++) {
9        channel_data<real_t> sd;
10       for (index_t i = 0; i < sending_size; i++) {
11        sd.data[i] =
12          SendPipes::PipeAt<channel_id>::read();                    ①
13       }
14       write_channel<channel_id, real_t>::write(sd);
15      }
16    }
17   }); }); } }
```

① Using `PipeArray` from [oneAPI Samples Repository]

I/O Kernel for Sending

```
1  template <unsigned channel_id, typename real_t>
2  void submit_send_kernel(sycl::queue& q, bool send_on,
3          index_t time_steps, index_t sends) {
4   if (send_on) {
5    q.submit([&](sycl::handler& h) {
6    h.single_task<class_name>([=]() {
7     for (index_t t = 0; t < time_steps; t++) {
8       for (index_t s = 0; s < sends; s++) {
9        channel_data<real_t> sd;
10       for (index_t i = 0; i < sending_size; i++) {
11        sd.data[i] =
12          SendPipes::PipeAt<channel_id>::read();                    ①
13       }
14       write_channel<channel_id, real_t>::write(sd);
15      }
16     }
17   }); }); } }
```

① Using `PipeArray` from [oneAPI Samples Repository]

I/O Kernel for Sending

```
1  template <unsigned channel_id, typename real_t>
2  void submit_send_kernel(sycl::queue& q, bool send_on,
3          index_t time_steps, index_t sends) {
4   if (send_on) {
5    q.submit([&](sycl::handler& h) {
6    h.single_task<class_name>([=]() {
7     for (index_t t = 0; t < time_steps; t++) {
8       for (index_t s = 0; s < sends; s++) {
9        channel_data<real_t> sd;
10       for (index_t i = 0; i < sending_size; i++) {
11        sd.data[i] =
12          SendPipes::PipeAt<channel_id>::read();                    ①
13       }
14       write_channel<channel_id, real_t>::write(sd);
15      }
16    }
17  }); }); } }
```

① Using `PipeArray` from [oneAPI Samples Repository]

I/O Kernel for Sending

```
1  template <unsigned channel_id, typename real_t>
2  void submit_send_kernel(sycl::queue& q, bool send_on,
3          index_t time_steps, index_t sends) {
4   if (send_on) {
5    q.submit([&](sycl::handler& h) {
6    h.single_task<class_name>([=]() {
7     for (index_t t = 0; t < time_steps; t++) {
8       for (index_t s = 0; s < sends; s++) {
9        channel_data<real_t> sd;
10       for (index_t i = 0; i < sending_size; i++) {
11        sd.data[i] =
12          SendPipes::PipeAt<channel_id>::read();                    ①
13       }
14       write_channel<channel_id, real_t>::write(sd);
15      }
16    }
17   }); }); } }
```

① Using `PipeArray` from [oneAPI Samples Repository]

I/O Kernel for Sending

```
1  template <unsigned channel_id, typename real_t>
2  void submit_send_kernel(sycl::queue& q, bool send_on,
3          index_t time_steps, index_t sends) {
4   if (send_on) {
5    q.submit([&](sycl::handler& h) {
6    h.single_task<class_name>([=]() {
7     for (index_t t = 0; t < time_steps; t++) {
8       for (index_t s = 0; s < sends; s++) {
9        channel_data<real_t> sd;
10       for (index_t i = 0; i < sending_size; i++) {
11        sd.data[i] =
12          SendPipes::PipeAt<channel_id>::read();                    ①
13       }
14       write_channel<channel_id, real_t>::write(sd);
15      }
16    }
17   }); }); } }
```

① Using `PipeArray` from [oneAPI Samples Repository]

I/O Kernel for Sending

```
1  template <unsigned channel_id, typename real_t>
2  void submit_send_kernel(sycl::queue& q, bool send_on,
3          index_t time_steps, index_t sends) {
4   if (send_on) {
5    q.submit([&](sycl::handler& h) {
6    h.single_task<class_name>([=]() {
7     for (index_t t = 0; t < time_steps; t++) {
8       for (index_t s = 0; s < sends; s++) {
9        channel_data<real_t> sd;
10       for (index_t i = 0; i < sending_size; i++) {
11        sd.data[i] =
12          SendPipes::PipeAt<channel_id>::read();                    ①
13       }
14       write_channel<channel_id, real_t>::write(sd);
15      }
16    }
17   }); }); } }
```

① Using `PipeArray` from [oneAPI Samples Repository]

I/O Kernel for Sending

```
1  template <unsigned channel_id, typename real_t>
2  void submit_send_kernel(sycl::queue& q, bool send_on,
3          index_t time_steps, index_t sends) {
4   if (send_on) {
5    q.submit([&](sycl::handler& h) {
6    h.single_task<class_name>([=]() {
7     for (index_t t = 0; t < time_steps; t++) {
8       for (index_t s = 0; s < sends; s++) {
9        channel_data<real_t> sd;
10       for (index_t i = 0; i < sending_size; i++) {
11        sd.data[i] =
12          SendPipes::PipeAt<channel_id>::read();                    ①
13       }
14       write_channel<channel_id, real_t>::write(sd);
15      }
16    }
17   }); }); } }
```

① Using `PipeArray` from [oneAPI Samples Repository]

Results

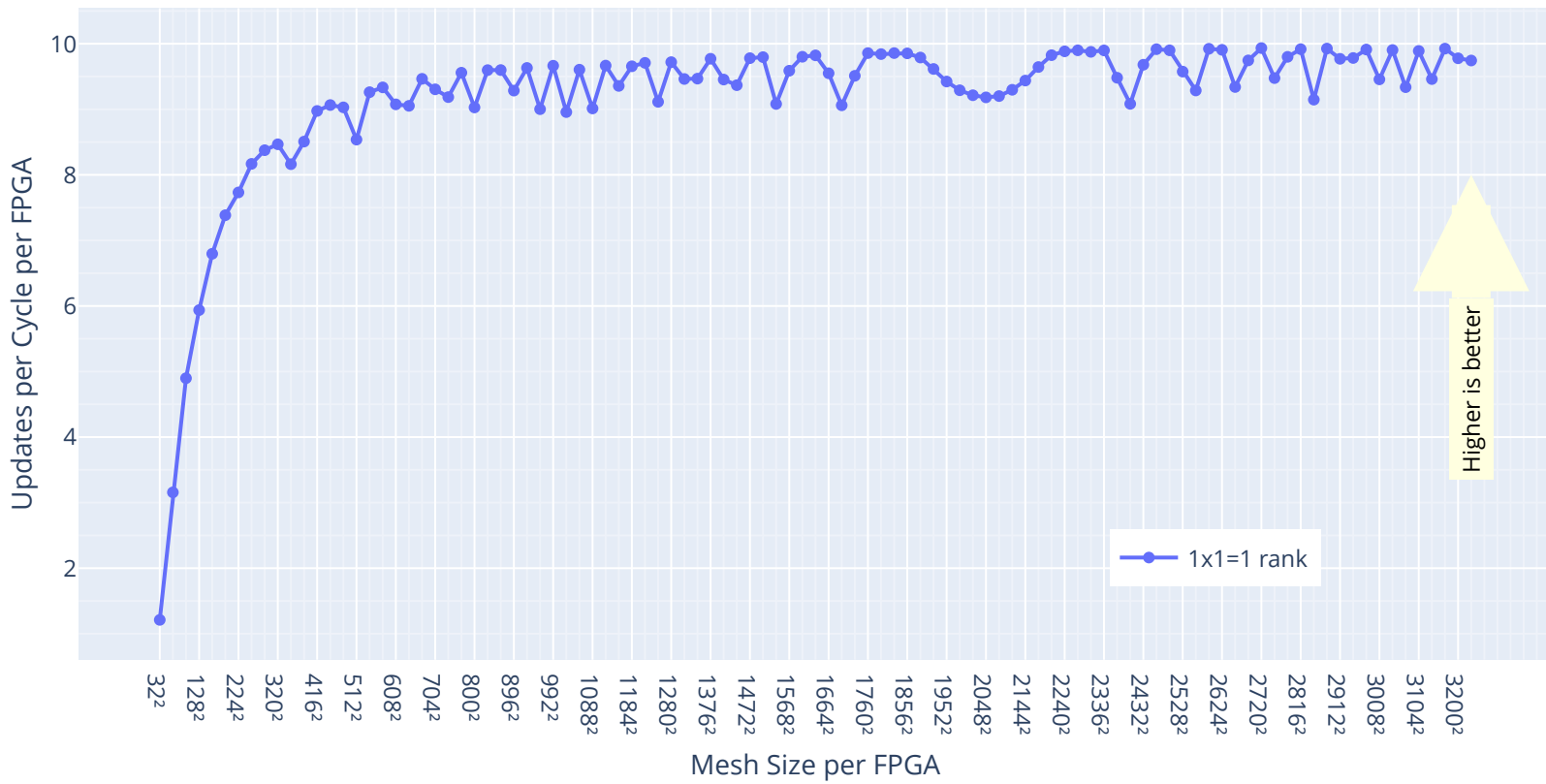
Resource usage

- Intel OneAPI version 2024.2.1
- Quartus Version 20.4.0
- Single-precision floating-point numbers

Clock (Mhz)	Logic Blocks	DSP	RAM
318.75	222,215 (24 %)	65 (1%)	3,427 (29%)

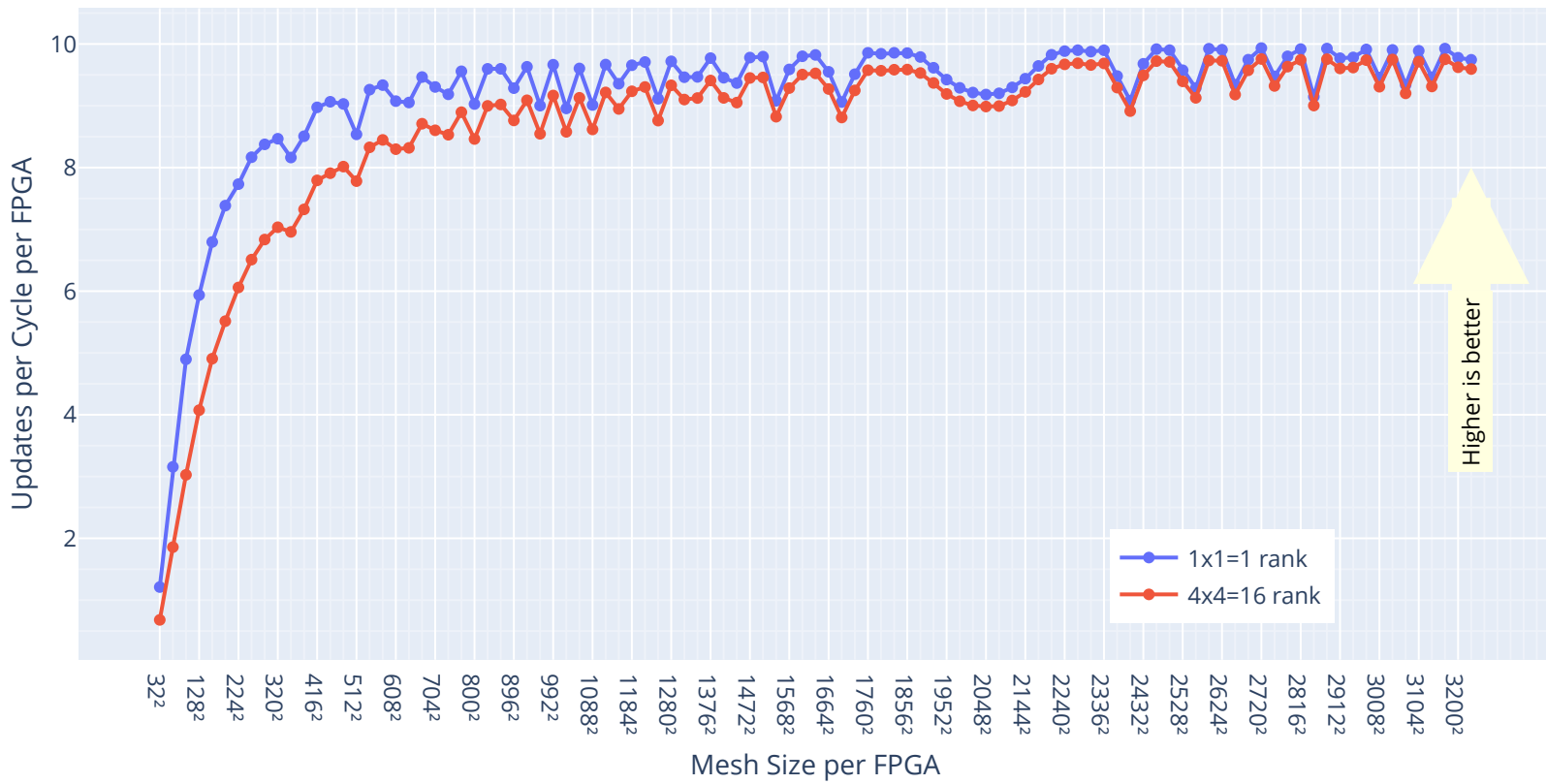
- RAM usage is mainly accounted to send/receive buffers

Single FPGA Performance



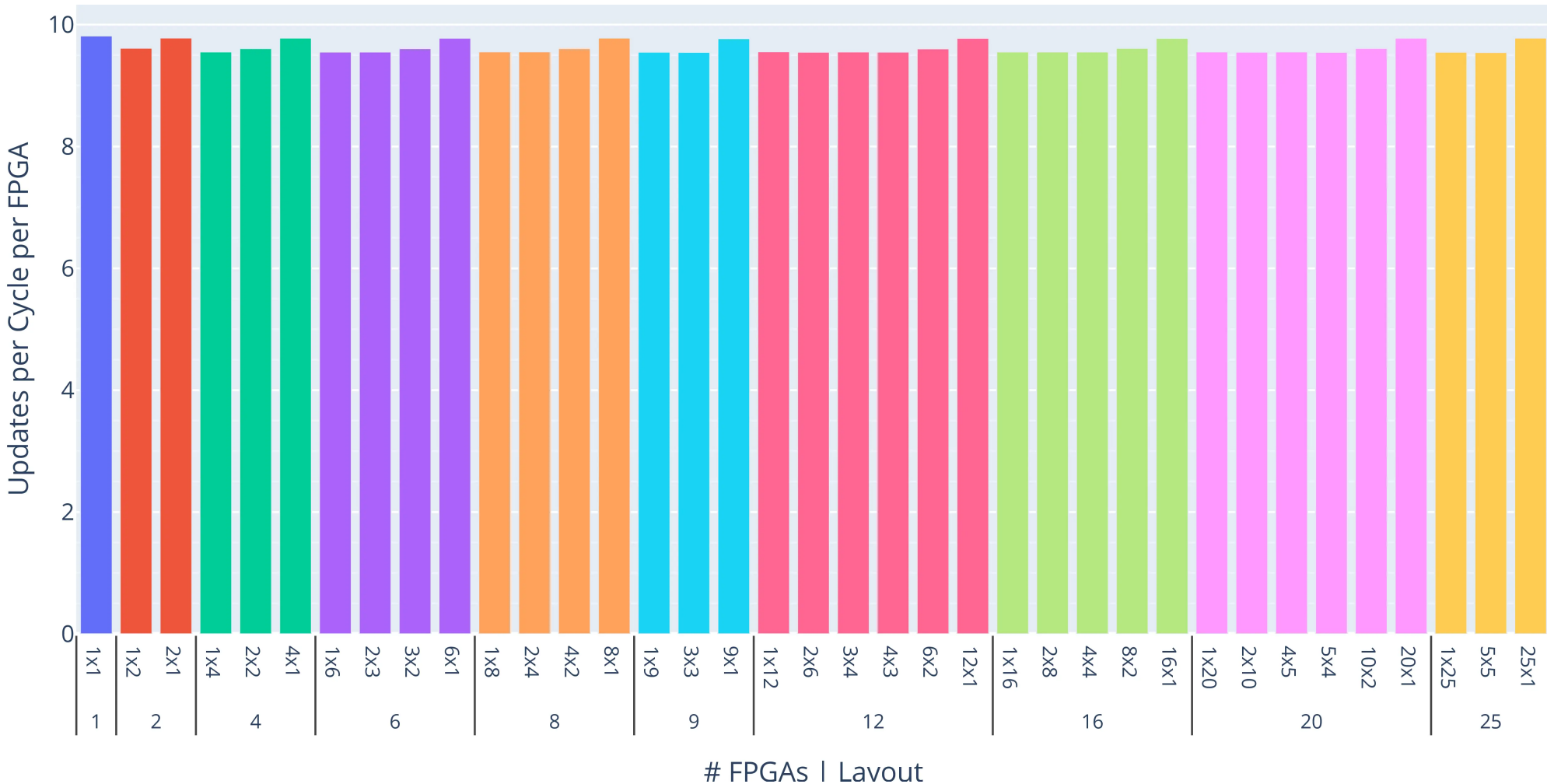
Increasing local mesh sizes for 1 FPGA. Starting by 32 × 32 the local mesh size was increased by 32 elements in each dimension

Multi FPGA Performance



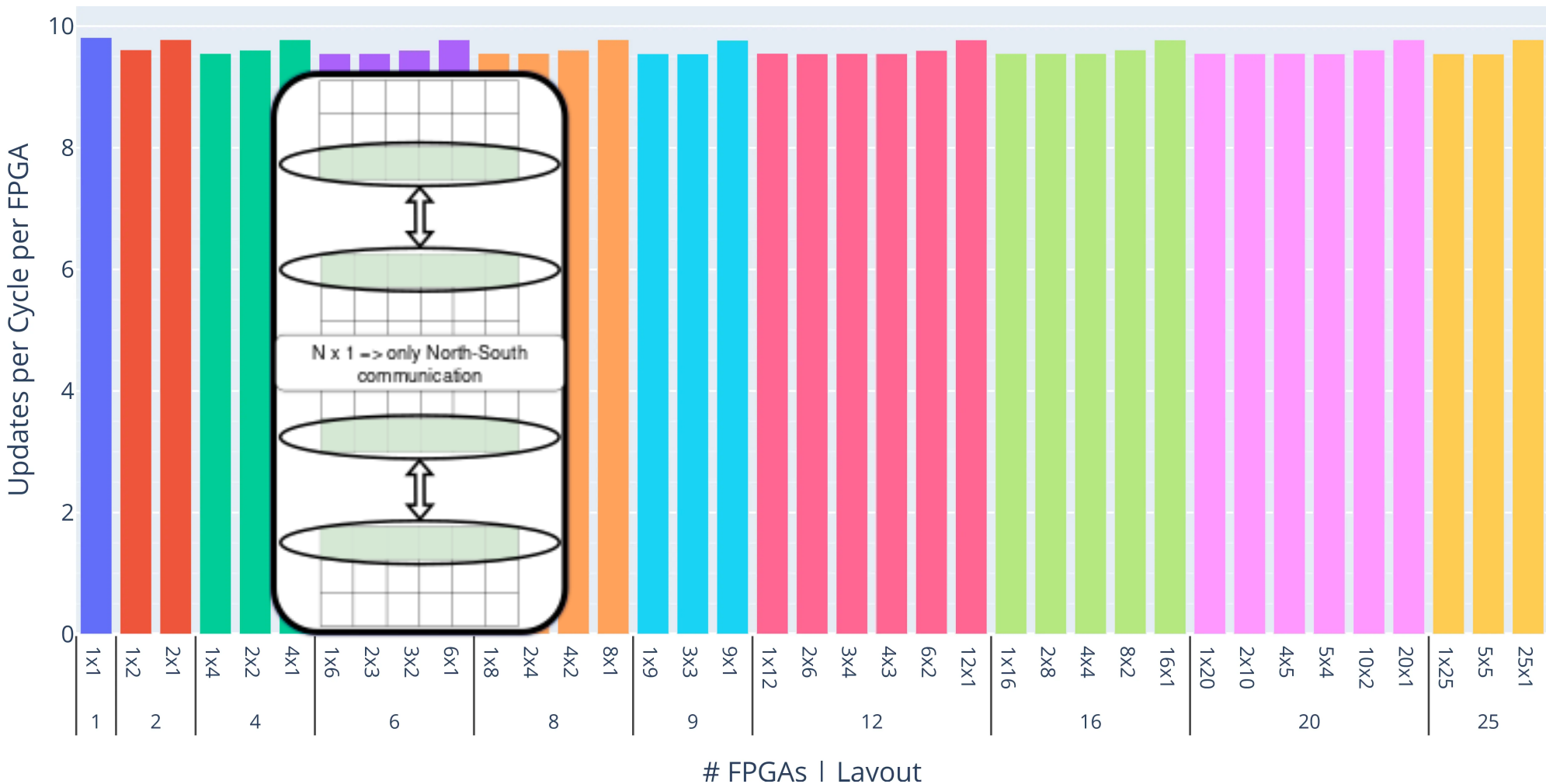
Increasing local mesh sizes for 1 FPGA and 16 FPGAs. Starting by 32 × 32 the local mesh size was increased by 32 elements in each dimension

Weak scaling



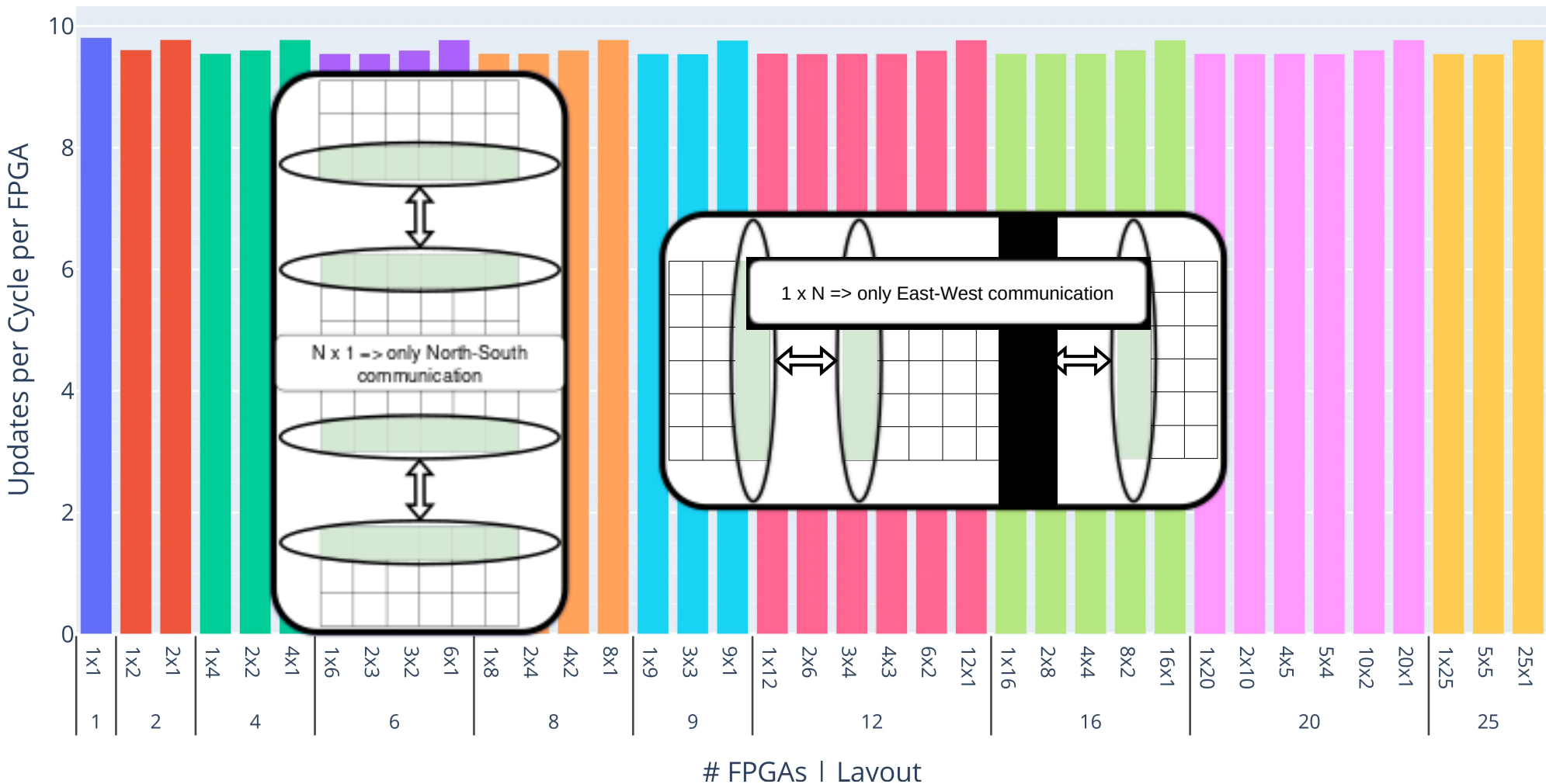
Updates per cycle per FPGA in a weak scaling experiment with a mesh size of 2000 × 2000 per FPGA. For each rank count every rectangular layout was tested.

Weak scaling



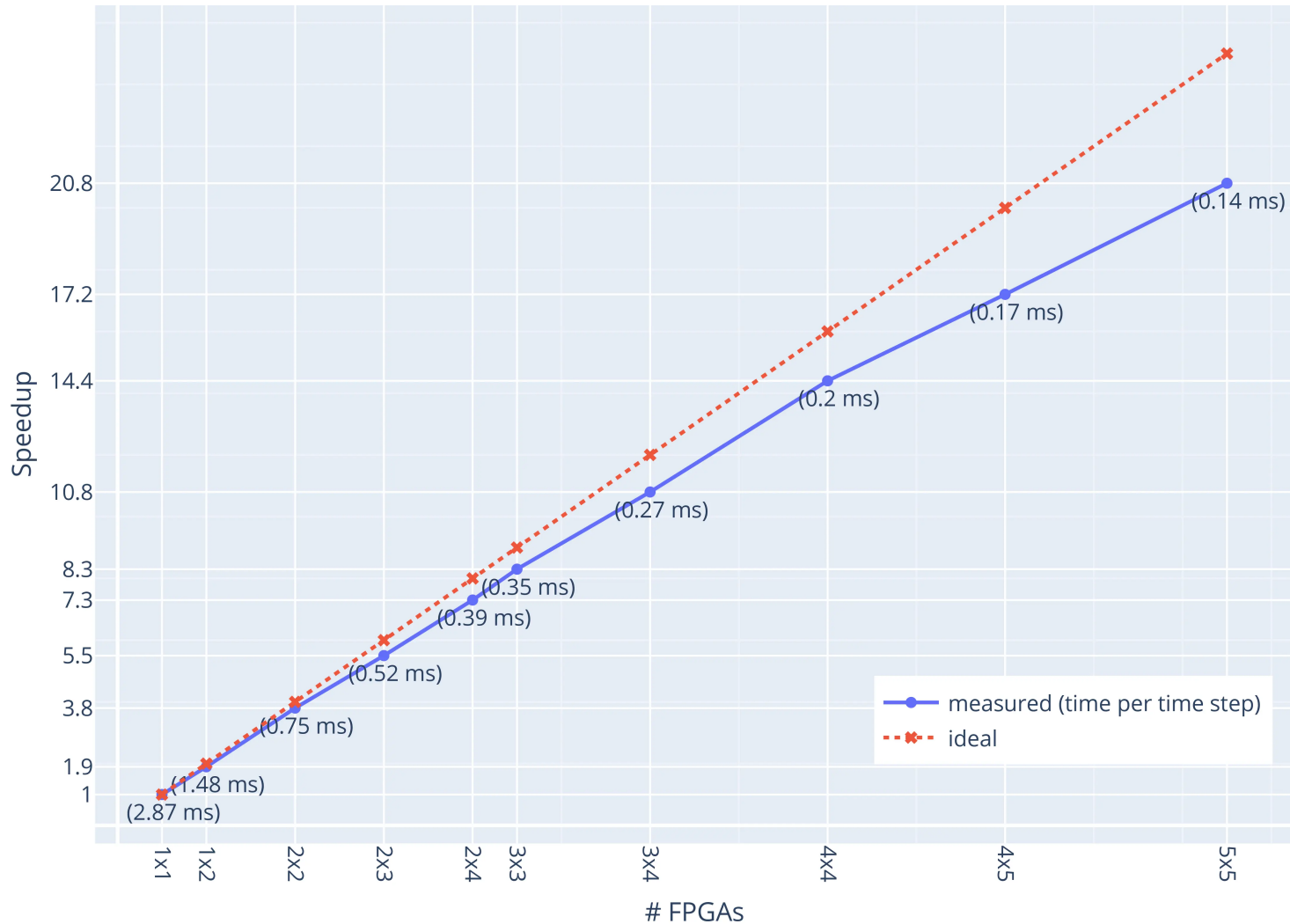
Updates per cycle per FPGA in a weak scaling experiment with a mesh size of 2000 × 2000 per FPGA. For each rank count every rectangular layout was tested.

Weak scaling



Updates per cycle per FPGA in a weak scaling experiment with a mesh size of 2000 × 2000 per FPGA. For each rank count every rectangular layout was tested.

Strong scaling

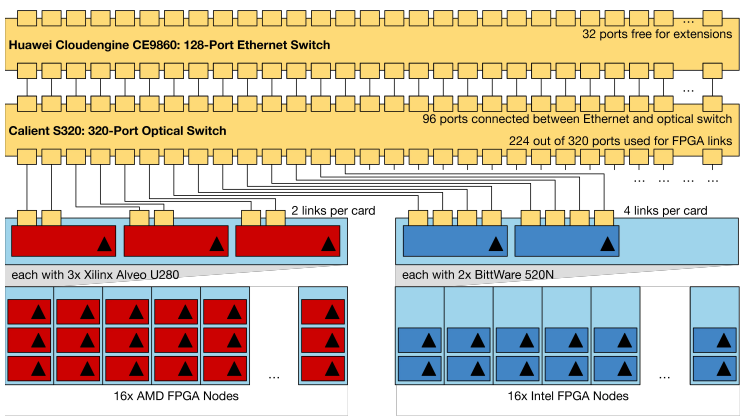


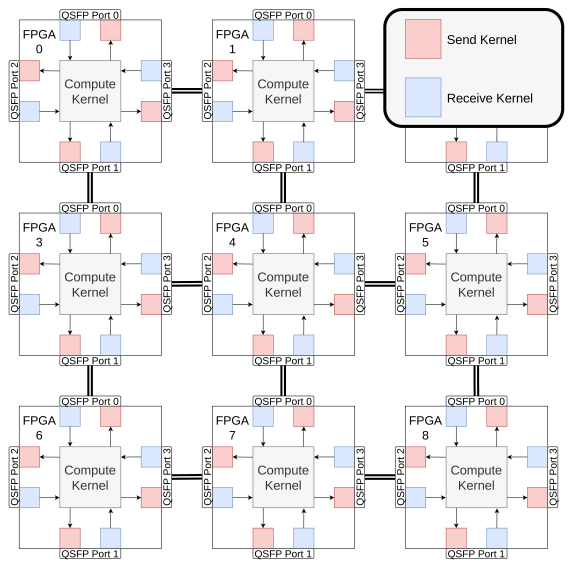
Speedup for a strong scaling experiment on a global mesh size of 3000 × 3000. The blue continuous lines are the measured speedup and the red dotted lines are the ideal speedup. The numbers are the time per complete time step including communication.

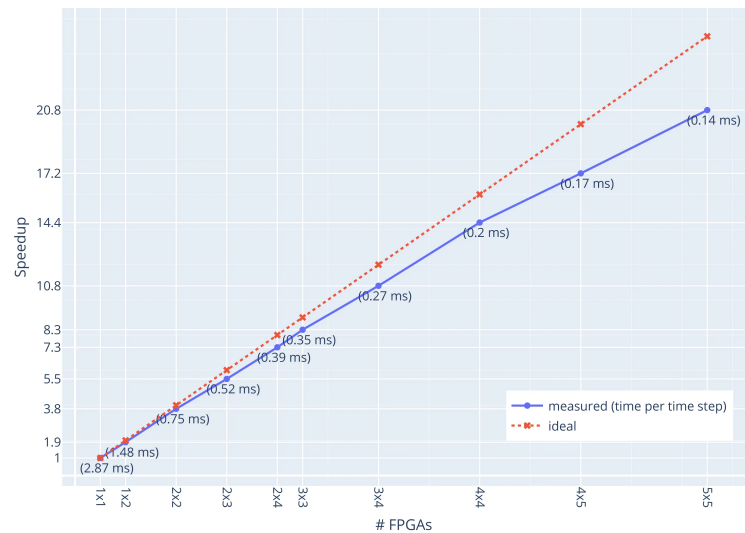
Conclusion & Outlook

- oneAPI I/O pipes can be used for direct FPGA-to-FPGA communication (with fix on Noctua 2)
- Programming abstraction with C++ templates allow compact formulation
- Good scaling behaviour to 25 FPGAs
- Future Work:
  - Integration into SYCL shallow water codebase (more information on our Poster: *Performance-Portable Implementation of the Shallow Water Equations on CPUs, GPUs, and FPGAs*)

Questions?







Thank you for the attention

C.Alt christoph.alt@uni-paderborn.de

Paderborn Center for Parallel Computing