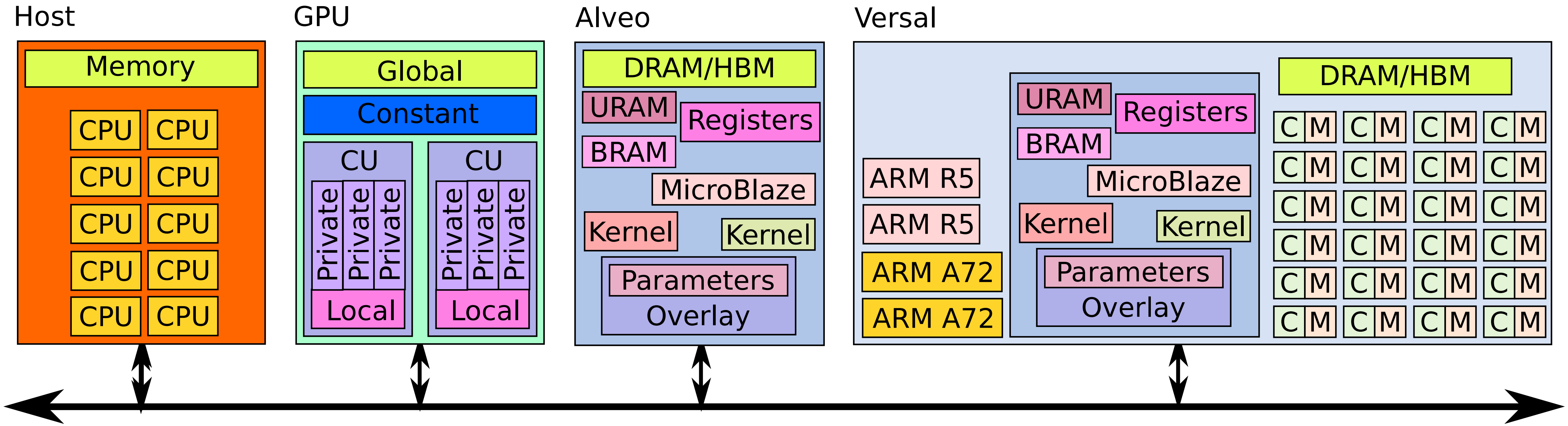
triSYCL

Some extensions for Xilinx FPGA & Versal ACAP

Ronan Keryell
Xilinx Research Labs (San José, California) & Khronos SYCL specification editor

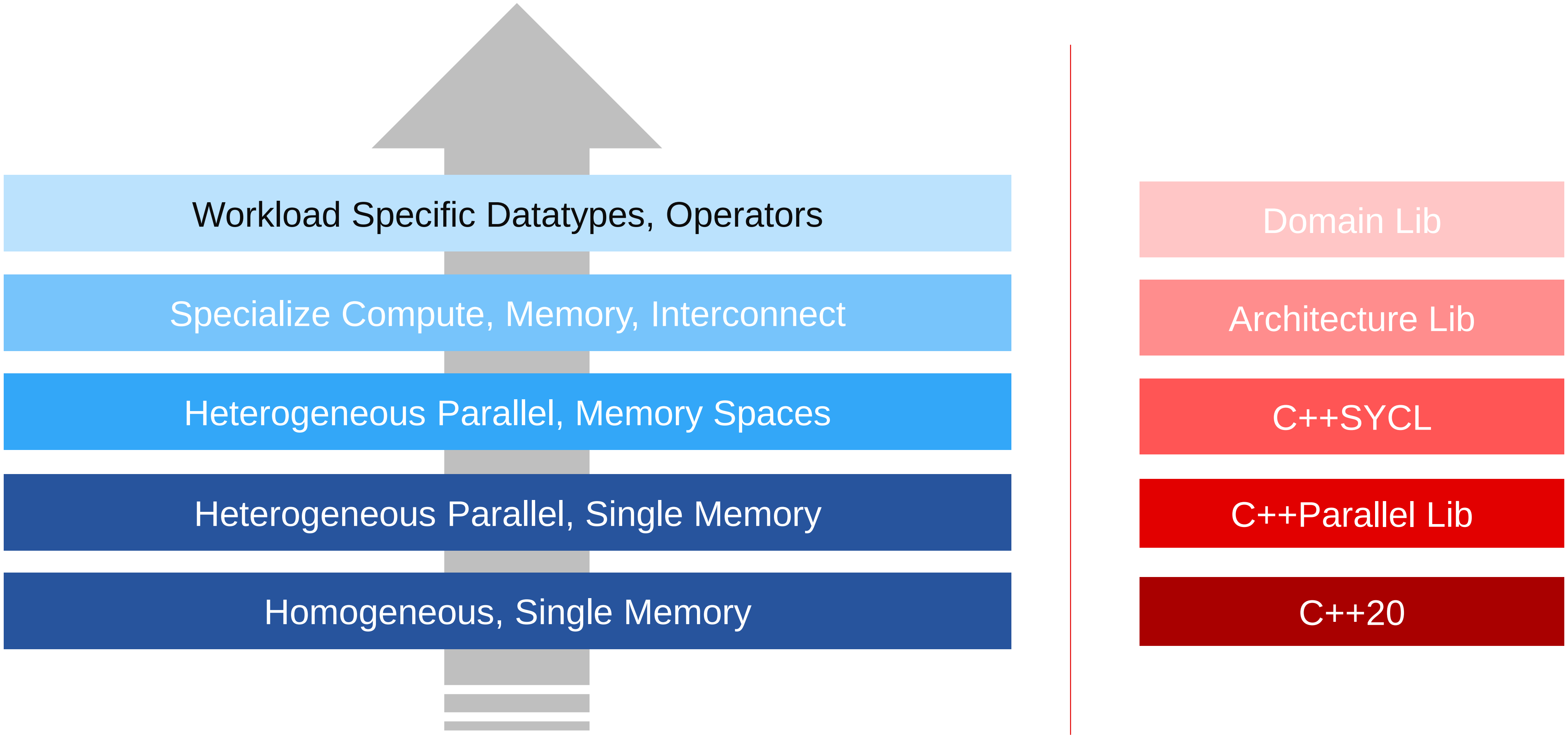
2020/04/27

Programing a *full* modern/future system…



- Add your own accelerator to this picture...
- Scale this at the data-center/HPC level too…
- Need a programming model…
- Tim Mattson’s law (Intel): no new language! ☺
  - → C++
    - → SYCL

SYCL ≡ plain C++ → refinement levels with plain C++ libraries



Workload Specific Datatypes, Operators

Specialize Compute, Memory, Interconnect

Heterogeneous Parallel, Memory Spaces

Heterogeneous Parallel, Single Memory

Homogeneous, Single Memory

Domain Lib

Architecture Lib

C++SYCL

C++Parallel Lib

C++20

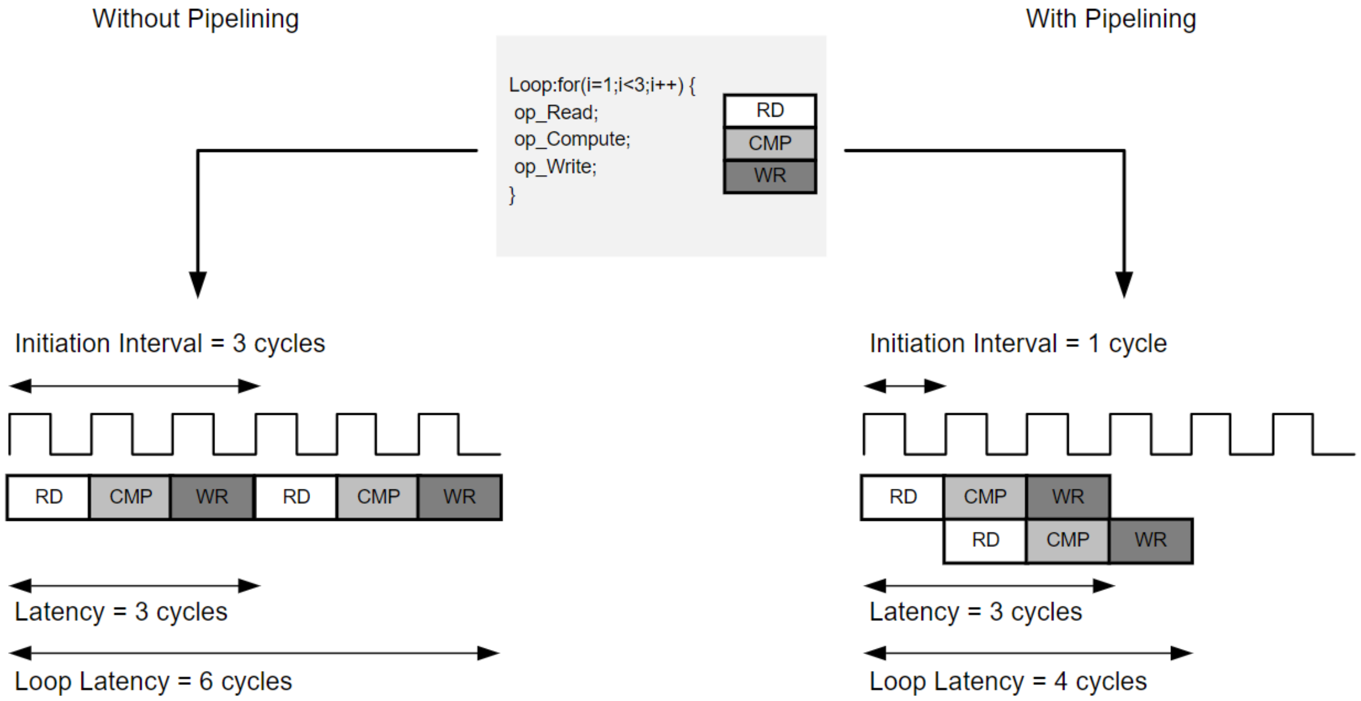
- SYCL is a C++ DSL… → Add more DSL directly in C++!

Some extensions in triSYCL

Xilinx FPGA extensions

Pipelining loops on FPGA

- Loop instructions sequentially executed by default
  - Loop iteration starts only after last operation from previous iteration
  - Sequential pessimism → idle hardware and loss of performance ☹
- → Use loop pipelining for more parallelism



```
Loop:for(i=1;i<3;i++) {
 op_Read;
 op_Compute;
 op_Write;
}
```

- Efficiency measure in hardware realm: Initiation Interval (II)
  - Clock cycles between the starting times of consecutive loop iterations
  - II can be 1 if no dependency and short operations

Decorating code for Xilinx FPGA pipelining in triSYCL

- Use native C++ construct instead of alien #pragma or attribute (vendor OpenCL or HLS C++…)

```
template<typename T, typename U>
void compute(T (&buffer_in)[BLOCK_SIZE], U (&buffer_out)[BLOCK_SIZE]) {
  for(int i = 0; i < NUM_ROWS; ++i) {
    for (int j = 0; j < WORD_PER_ROW; ++j) {
      vendor::xilinx::pipeline([&] {
        int inTmp = buffer_in[WORD_PER_ROW*i+j];
        int outTmp = inTmp * ALPHA;
        buffer_out[WORD_PER_ROW*i+j] = outTmp;
      });
    }
  }
}
```

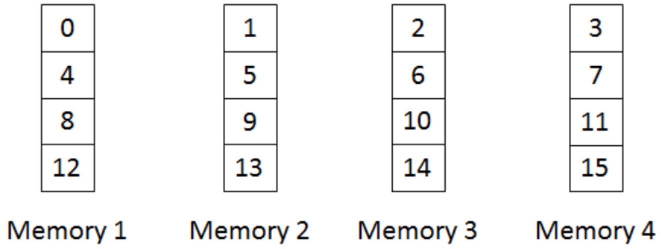
```
/** Execute loops in a pipelined manner

    A loop with pipeline mode processes a new input every clock
    cycle. This allows the operations of different iterations of the
    loop to be executed in a concurrent manner to reduce latency.

    \param[in] f is a function with an innermost loop to be executed
in a
    pipeline way.
*/
auto pipeline = [] (auto functor) noexcept {
  /* SSDM instruction is inserted before the argument functor
     to guide xocc to do pipeline. */
  _ssdm_op_SpecPipeline(1, 1, 0, 0, "");
  functor();
};
```

```
#ifdef TRISYCL_DEVICE
extern "C" {
  /// SSDM Intrinsics: dataflow operation
  void _ssdm_op_SpecDataflowPipeline(...) __attribute__ ((nothrow, noinline, weak));
  /// SSDM Intrinsics: pipeline operation
  void _ssdm_op_SpecPipeline(...) __attribute__ ((nothrow, noinline, weak));
  /// SSDM Intrinsics: array partition operation
  void _ssdm_SpecArrayPartition(...) __attribute__ ((nothrow, noinline, weak));
}
#else
/* If not on device, just ignore the intrinsics as defining them as
   empty variadic macros replaced by an empty do-while to avoid some
   warning when compiling (and used in an if branch */
#define _ssdm_op_SpecDataflowPipeline(...) do { } while (0)
#define _ssdm_op_SpecPipeline(...) do { } while (0)
#define _ssdm_SpecArrayPartition(...) do { } while (0)
#endif
```

- Compatible with metaprogramming
- No need for specific parser/tool-chain!
  - Just use lambda + intrinsics! ☺
  - Good opportunity to standardize the syntax with other vendors! ☺

Partitioning memories

- Remember bank conflicts on Cray in the 70’s? ☺
- In FPGA world, even memory is configurable!
- Example of 1D array with 16 elements…
- Cyclic Partitioning
  - Each array element distributed to physical memory banks in order and cyclically
  - Banks accessed in parallel → improved bandwidth
  - Reduce latency for pipelined sequential accesses

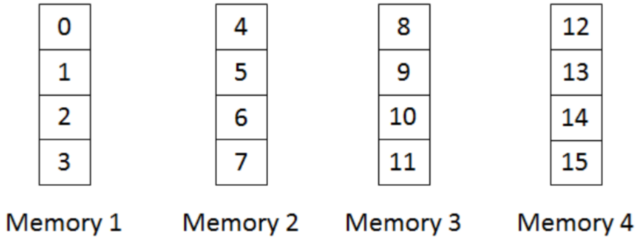


Memory 1	Memory 2	Memory 3	Memory 4
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 7-1: **Physical Layout of Buffer After Cyclic Partitioning**

Block Partitioning

- Each array element distributed to physical memory banks by block and in order
- Banks accessed in parallel → improved bandwidth
- Reduce latency for pipelined accesses with some distribution

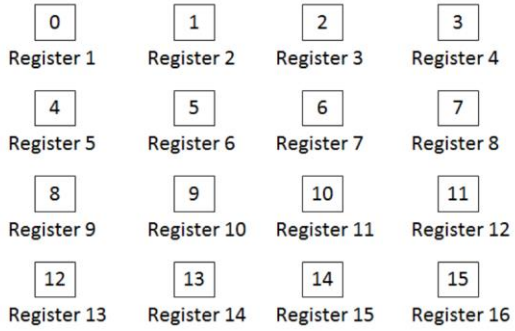


Memory 1	Memory 2	Memory 3	Memory 4
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Figure 7-2: **Physical Layout of Buffer After Block Partitioning**

Complete Partitioning

- Extreme distribution
- Extreme bandwidth
- Low latency



0 Register 1	1 Register 2	2 Register 3	3 Register 4
4 Register 5	5 Register 6	6 Register 7	7 Register 8
8 Register 9	9 Register 10	10 Register 11	11 Register 12
12 Register 13	13 Register 14	14 Register 15	15 Register 16

Figure 7-3: **Physical Layout of Buffer After Complete Partitioning**

partition_array class in triSYCL for Xilinx FPGA

```
// A typical FPGA-style pipelined kernel
  cgh.single_task<class mat_mult>([=] {
  // Cyclic Partition for A as matrix multiplication needs
  // row-wise parallel access
  xilinx::partition_array<Type, BLOCK_SIZE,
             xilinx::partition::cyclic<MAX_DIM>> A;
  // Block Partition for B as matrix multiplication needs
  //column-wise parallel access
  xilinx::partition_array<Type, BLOCK_SIZE,
             xilinx::partition::block<MAX_DIM>> B;
  xilinx::partition_array<Type, BLOCK_SIZE> C;
  …
});
```

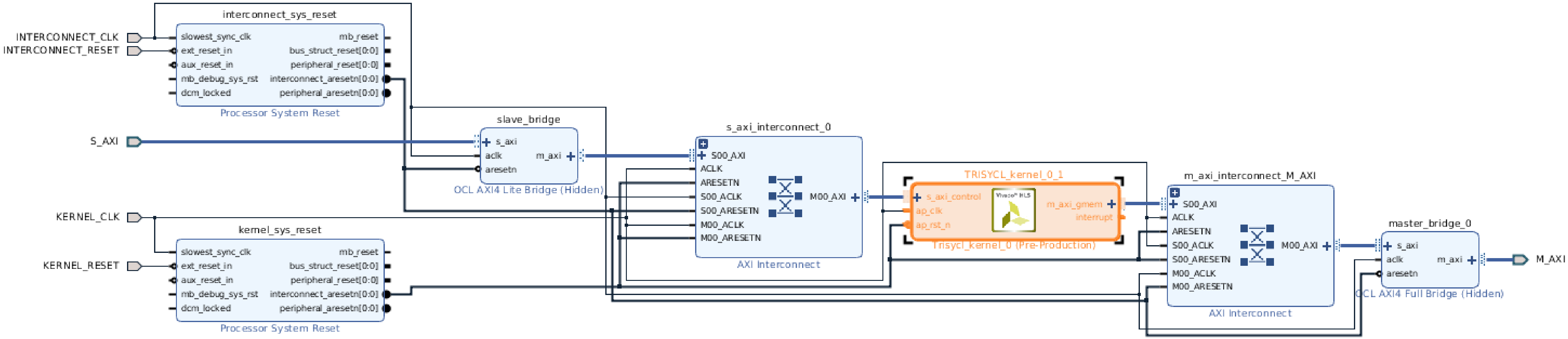
```
int A[MAX_DIM * MAX_DIM];
int B[MAX_DIM * MAX_DIM];
int C[MAX_DIM * MAX_DIM];
//Cyclic Partition for A as matrix multiplication needs row-wise parallel access
#pragma HLS ARRAY_PARTITION variable=A dim=1 cyclic factor=64
//Block Partition for B as matrix multiplication needs column-wise parallel access
#pragma HLS ARRAY_PARTITION variable=B dim=1 block factor=64
 …
```

Xilinx HLS C++

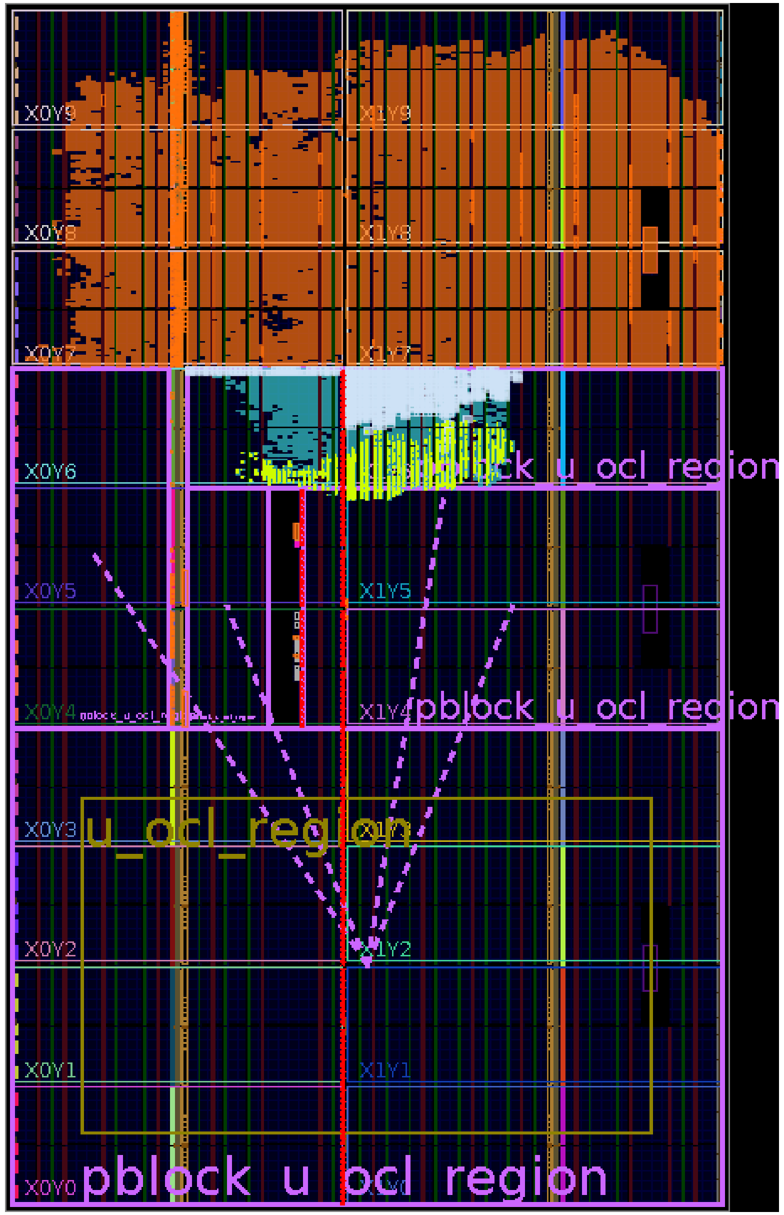
```
//Cyclic Partition for A as matrix multiplication needs row-wise parallel access
int A[MAX_DIM * MAX_DIM]
__attribute__((xcl_array_partition(cyclic, MAX_DIM, 1)));
//Block Partition for B as matrix multiplication needs column-wise parallel access
int B[MAX_DIM * MAX_DIM]
__attribute__((xcl_array_partition(block, MAX_DIM, 1)));
int C[MAX_DIM * MAX_DIM];
…
```

Xilinx OpenCL C

After Xilinx `xocc` ingestion…



After Xilinx `xocc` ingestion… FPGA layout!



Dataflow programming with Intel SYCL extension: pipes

- Extend FPGA abstractions (Intel Altera channels, Xilinx AXI-streams) with modern C++ flavor
  - Simplify SYCL [deprecated] ~~2.2~~ proposal with simpler static-only syntax

```
// Pipes are types: static-compile time kernel-pipe connection! No performance compromise!!!
using my_pipe = sycl::pipe<class some_pipe, int>;

sycl::queue q { sycl::fpga_selector_v };

// Producer kernel on FPGA
q.single_task([] { for(;;) { my_pipe::write(produce()); });

// Consumer kernel on FPGA
q.single_task([] { for(;;) { process(my_pipe::read()); });


// Hardware definition provided by the device/board vendor for exposition only outside of user code
namespace my_platform {
  using ethernet_packet = ...;
  using ethernet_read_pipe = sycl::kernel_readable_io_pipe<ethernet_pipe_id<0>, ethernet_packet>;
  using ethernet_write_pipe = sycl::kernel_writeable_io_pipe<ethernet_pipe_id<1>, ethernet_packet>;
}

using nic_pipe = pipe<class sniffing, my_platform::ethernet_packet, 1000>;

// DPI with a sniffing kernel on FPGA
q.single_task([] {
  for(;;) {
    auto packet = my_platform::ethernet_read_pipe::read();
    if (filter(packet))
      nic_pipe::write(packet);
  }
});

// On host, since not inside a kernel
for(;;) analyze(nic_pipe::read());
```

data-flow
for dummies !
Deluxe compiler edition

- Implemented in triSYCL on CPU with `boost::fibers::buffered_channel`

- C++20 `std::range` brings similar lazy dataflow programming style…

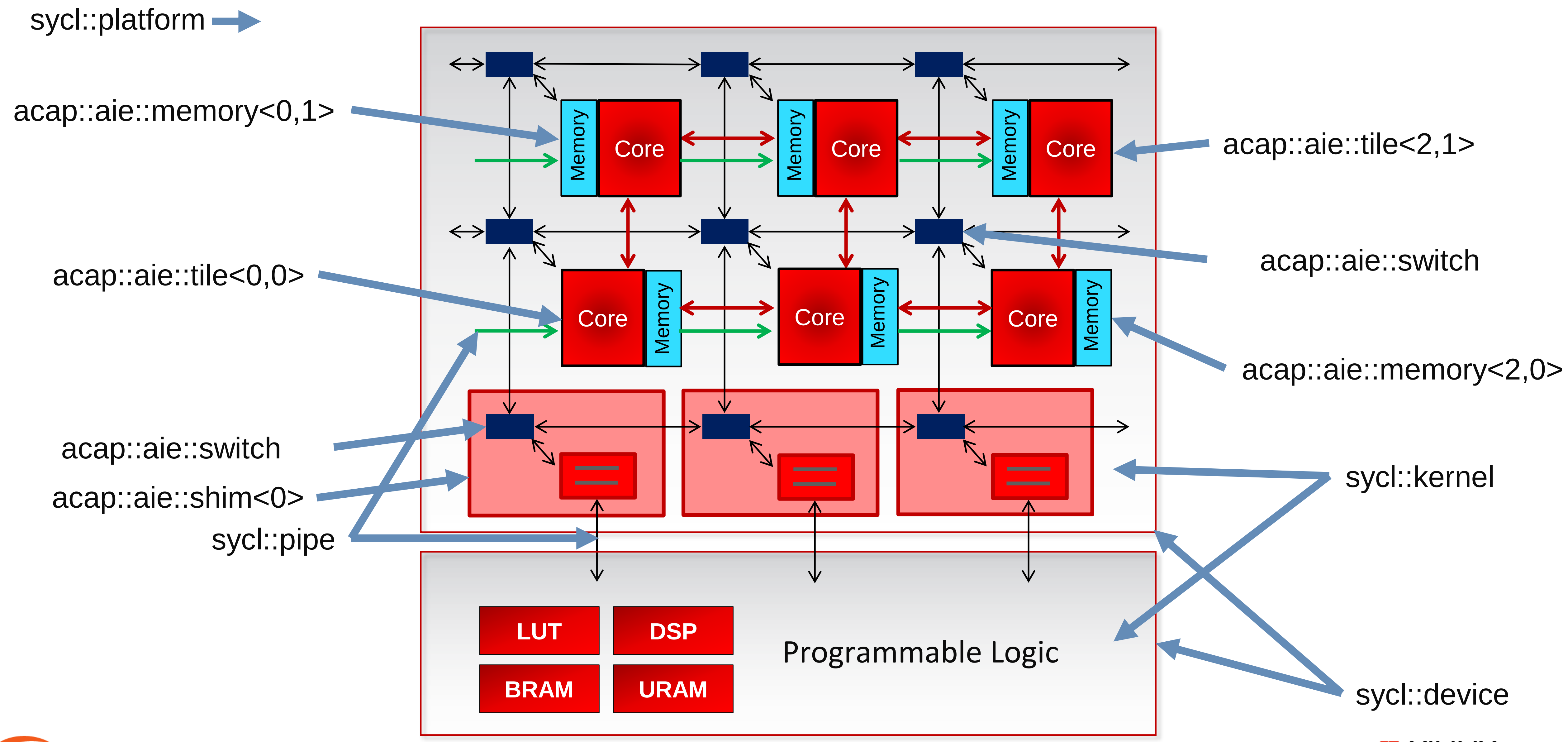
Xilinx ACAP extensions

Versal Adaptive Compute Acceleration Platform (ACAP)

- Coarse Grain Reconfigurable Array (CGRA)
    - Adaptable Engine (AIE)

13

ACAP++: SYCL abstractions templated by 2D coordinates



Hello World in C++ ACAP

```
#include <iostream>
#include <SYCL/sycl.hpp>
using namespace sycl::vendor::xilinx;
/** A small AI Engine program

    The definition of a tile program has to start this way

    \param AIE is an implementation-defined type

    \param X is the horizontal coordinate

    \param Y is the vertical coordinate */
template <typename AIE, int X, int Y>
struct prog : acap::aie::tile<AIE, X, Y> {
  /// The run member function is defined as the tile program
  void run() {
    std::cout << "Hello, I am the AIE tile (" << X << ',' << Y  << ")"
              << std::endl;
  }
};
int main() {
  // Define AIE CGRA with all the tiles of a VC1902 and run up to completion prog on all the tiles
  acap::aie::device<acap::aie::layout::vc1902> {}.run<prog>();
}
```

Graphics library for debugging

- Graphical display is extremely useful to find troubles
- Small Gnome gtkmm-based tool to display distributed arrays on tiles
- API based on `std::mdspan` ISO C++ proposal https://wg21.link/P0009

Mandlebrot set computation (cryptocurrency POW too…)

- Start simple: no neighborhood communication ☺

```
using namespace sycl::vendor::xilinx;

static auto constexpr image_size = 229;
graphics::application a;

// All the memory modules are the same
template <typename AIE, int X, int Y>
struct pixel_tile : acap::aie::memory<AIE, X, Y> {
  // The local pixel tile inside the complex plane
  std::uint8_t plane[image_size][image_size];
};

// All the tiles run the same Mandelbrot program
template <typename AIE, int X, int Y>
struct mandelbrot : acap::aie::tile<AIE, X, Y> {
  using t = acap::aie::tile<AIE, X, Y>;
  // Computation rectangle in the complex plane
  static auto constexpr x0 = -2.1, y0 = -1.2, x1 = 0.6, y1 = 1.2;
  static auto constexpr D = 100; // Divergence norm
  // Size of an image tile
  static auto constexpr xs = (x1 - x0)/t::geo::x_size/image_size;
  static auto constexpr ys = (y1 - y0)/t::geo::y_size/image_size;
```

```
  void run() {
    // Access to its own memory
    auto& m = t::mem();
    while (!a.is_done()) {
      for (int i = 0; i < image_size; ++i)
        for (int k, j = 0; j < image_size; ++j) {
          std::complex c { x0 + xs*(X*image_size + i),
                           y0 + ys*(Y*image_size + j) };
          std::complex z { 0.0 };
          for (k = 0; k <= 255; k++) {
            z = z*z + c;
            if (norm(z) > D)
              break;
          }
          m.plane[j][i] = k;
        }
      a.update_tile_data_image(t::x, t::y, &m.plane[0][0], 0, 255);
    }
  }
};

int main(int argc, char *argv[]) {
  acap::aie::device<acap::aie::layout::size<2,3>> aie;
  // Open a graphic view of a AIE array
  a.start(argc, argv, decltype(aie)::geo::x_size, decltype(aie)::geo::y_size,
          image_size, image_size, 1);
  a.image_grid().palette().set(graphics::palette::rainbow, 100, 2, 0);

  // Launch the AI Engine program
  aie.run<mandelbrot, pixel_tile>();
  // Wait for the graphics to stop
  a.wait();
}
```

Stencil computation: 2D shallow water wave propagation

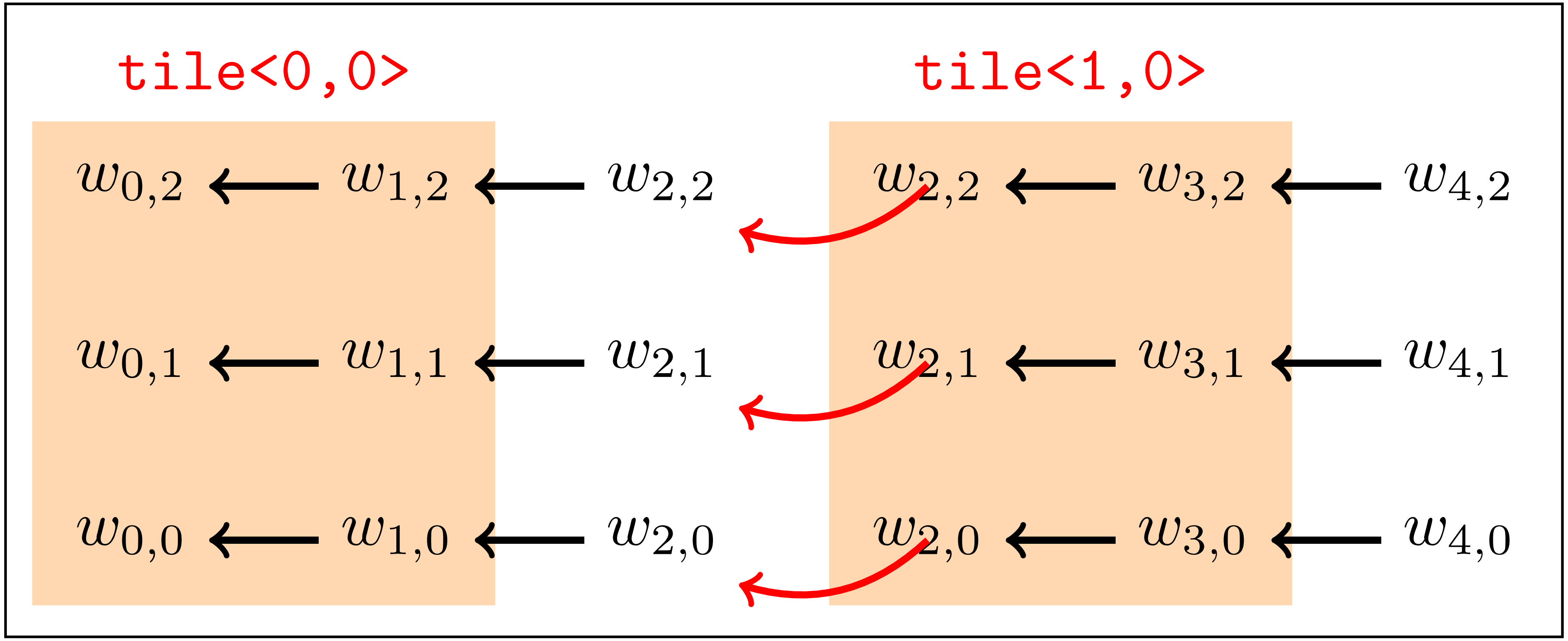
- 2D tiling of data & computations
- Typical PDE (Partial Differential Equation) application from HPC, usually using MPI & OpenMP
- Demo with simple 1-order differentiation schema $\frac{df}{dx} \approx \frac{f_{x+1} - f_x}{dx}$
- Only communication with 4 direct neighbors
  - Impossible to do on GPU without global memory transfer
    - High latency, high energy ☹
  - Should be perfect for AIE!
    - Low latency, low energy ☺

Wave propagation sequential reference code (mdspan!)

```
/// Compute a time-step of wave propagation
 void compute() {
  for (int j = 0; j < size_y; ++j)
   for (int i = 0; i < size_x - 1; ++i) {
    // dw/dx
    auto up = w(j,i + 1) - w(j,i);
    // Integrate horizontal speed
    u(j,i) += up*alpha;
   }
  for (int j = 0; j < size_y - 1; ++j)
   for (int i = 0; i < size_x; ++i) {
    // dw/dy
    auto vp = w(j + 1,i) - w(j,i);
    // Integrate vertical speed
    v(j,i) += vp*alpha;
   }
  for (int j = 1; j < size_y; ++j)
   for (int i = 1; i < size_x; ++i) {
    // div speed
    auto wp = (u(j,i) - u(j,i - 1)) + (v(j,i) - v(j - 1,i));
    wp *= side(j,i)*(depth(j,i) + w(j,i));
    // Integrate depth
    w(j,i) += wp;
    // Add some dissipation for the damping
    w(j,i) *= damping;
   }
 }
```

Use overlaping/ghost/halo variables

$$u_{i,j} = f(w_{i+1,j}, w_{i,j})$$



tile<0,0>
tile<1,0>
$w_{0,2}$ ← $w_{1,2}$ ← $w_{2,2}$
$w_{2,2}$ ← $w_{3,2}$ ← $w_{4,2}$
$w_{0,1}$ ← $w_{1,1}$ ← $w_{2,1}$
$w_{2,1}$ ← $w_{3,1}$ ← $w_{4,1}$
$w_{0,0}$ ← $w_{1,0}$ ← $w_{2,0}$
$w_{2,0}$ ← $w_{3,0}$ ← $w_{4,0}$

- Reading code from neighbors in main loop can be inefficient
- Increase size of local storage with required neighbor data: ghost/overlap/halo/…
- Prefetch missing data before computation (DMA…)
- No change to main computation ☺
- Leverage ISO C++ `mdspan` proposal for stitching & graphics rendering

Moving lines and columns around…

```
void compute() {
  auto& m = t::mem();

  for (int j = 0; j < image_size; ++j)
   for (int i = 0; i < image_size - 1; ++i) {
    // dw/dx
    auto up = m.w[j][i + 1] - m.w[j][i];
    // Integrate horizontal speed
    m.u[j][i] += up*alpha;
   }

  for (int j = 0; j < image_size - 1; ++j)
   for (int i = 0; i < image_size; ++i) {
    // dw/dy
    auto vp = m.w[j + 1][i] - m.w[j][i];
    // Integrate vertical speed
    m.v[j][i] += vp*alpha;
   }

  t::barrier();
[...]

  if constexpr (t::is_memory_module_up()) {
   auto& above = t::mem_up();
   for (int i = 0; i < image_size; ++i)
    above.w[0][i] = m.w[image_size - 1][i];
  }
```

```
t::barrier();

// Transfer last line of w to next memory module on the right
if constexpr (Y & 1) {
 if constexpr (t::is_memory_module_right()) {
  auto& right = t::mem_right();
  for (int j = 0; j < image_size; ++j)
   right.w[j][0] = m.w[j][image_size - 1];
 }
}
if constexpr (!(Y & 1)) {
 if constexpr (t::is_memory_module_left()) {
  auto& left = t::mem_left();
  for (int j = 0; j < image_size; ++j)
   m.w[j][0] = left.w[j][image_size - 1];
 }
}
```

Debug: the killer application for single-source SYCL!

- It is pure plain single-source modern C++… ☺
  - At least when running on CPU in emulation mode
  - Can debug the full real application, not only host or device code independently!

- Use normal debugger
  - 1 thread per host… thread
  - 1 thread per AIE tile
  - 1 thread per GPU work-item
  - 1 thread per FPGA work-item
  - Gdb is scriptable in Python ☺
- Use normal memory checker (Valgrind, Clang/GCC UBSan, AddressSanitizer…)
  - Valgrind scriptable from Gdb which is scriptable in Python which is…
- Use normal thread checker (Helgrind, ThreadSanitizer…)
  - Can detect memory tile lock issues & race conditions! ☺
- Possible to call graphics code inside AIE code too
- Possible to write co-simulation code inside AIE code too
- Can compare with global sequential execution at the same time
  - Do not require separate application, huge test vectors…

Conclusion

- SYCL ≡ C++ DSL providing foundational modern single-source heterogeneous computing
- Focus on the whole system: SYstem-wide Compute Language
  - Single-source → address the full application & enable generic libraries
  - Heterogeneous programming CPU+GPU+CGRA+FPGA+DSP+…
- triSYCL used as workbench to develop DSL extensions on top of SYCL
  - Spatial computing
    - Xilinx FPGA
    - Versal ACAP CGRA
  - Explicit programming model for bare metal efficiency
    - Abstractions for DMA & multi-level memory-hierarchy / overlay architectures
- Pure modern C++ DSL
  - Target emulation, debug & co-design on CPU for free! ☺

Thank You