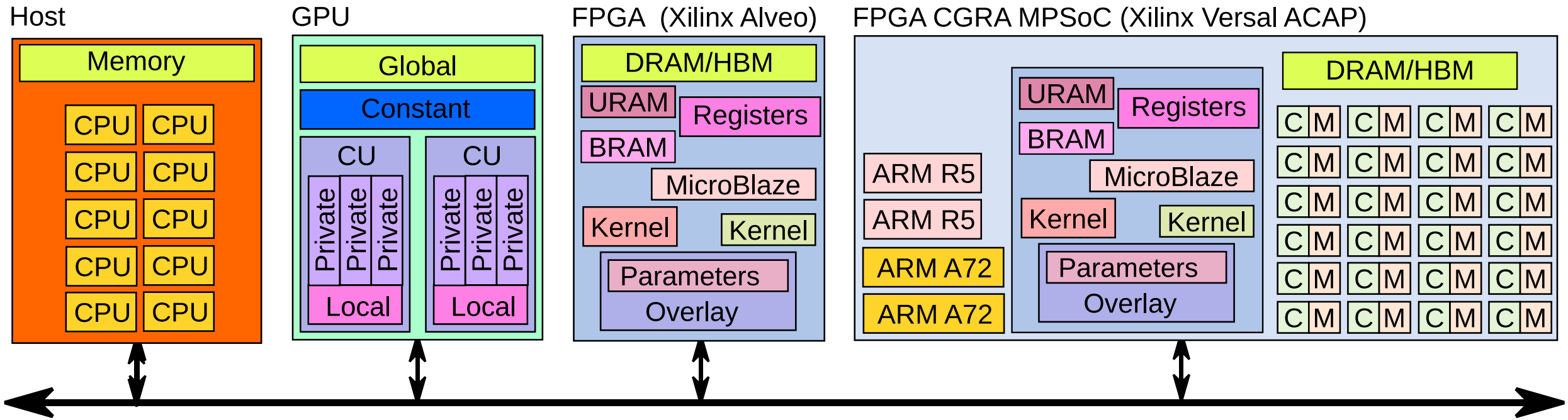
XILINX

Porting SYCL with oneAPI DPC++ to Xilinx FPGA & Versal ACAP CGRA

Luc Forget, Gauthier Harnisch, Ronan Keryell (rkeryell@xilinx.com), Ralph Wittig
Xilinx Research Labs (San José, California)
2021/06/22
oneAPI Developer Summit at ISC 2021

SYCL

Programing a *full* modern/future system…



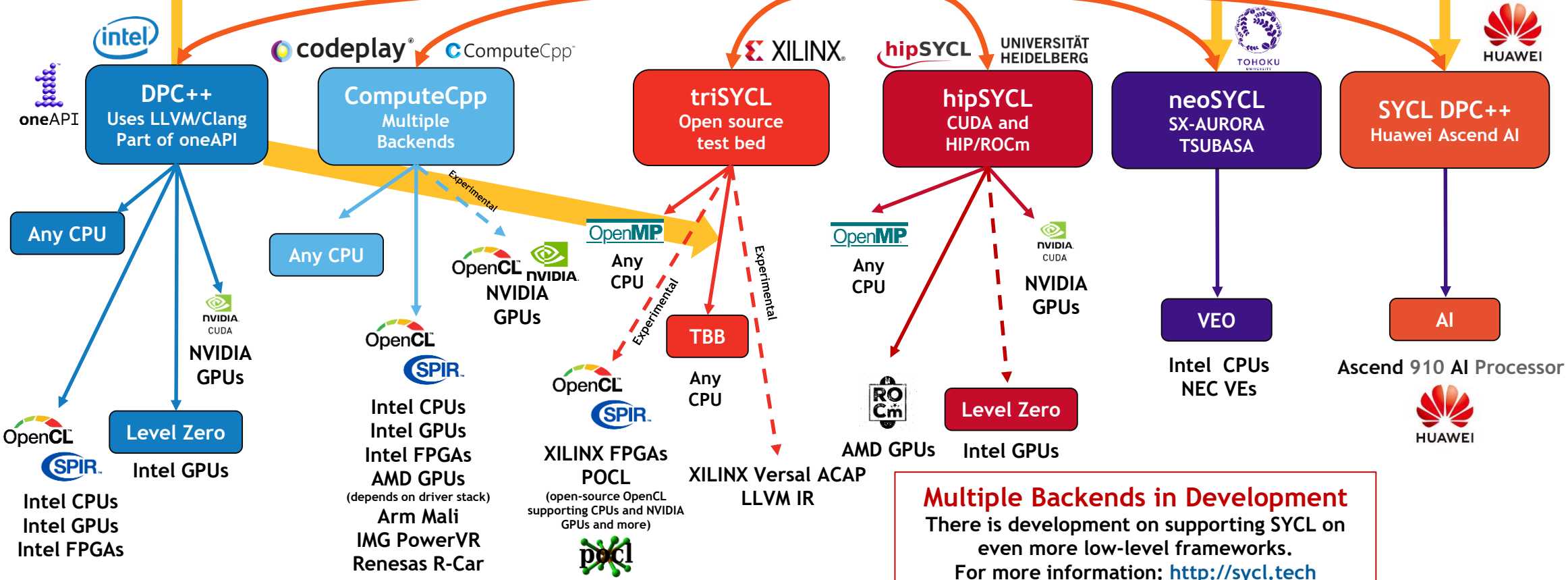
- Add your own accelerator to this picture...
- Scale this at the data-center/HPC level too…
- Need a programming model…
- Tim Mattson's law (Intel): no new language! ☺

SYCL ecosystem is growing

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies

SYCL Source Code

SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute



Multiple Backends in Development
There is development on supporting SYCL on even more low-level frameworks.
For more information: http://sycl.tech

- https://www.khronos.org/blog/sycl-2020-what-do-you-need-to-know

+ Celerity: über-SYCL on MPI+SYCL

#include <C++>

SYCL 2020 ≡ heterogeneous simplicity with modern C++

```
#include <sycl/sycl.hpp>
#include <iostream>
using namespace sycl;
constexpr int N = 32;

int main () {
  buffer<int> buf { N };
  queue {}.submit([&](auto &h) {
     accessor a { buf, h, write_only, no_init };
     h.parallel_for(N, [=](auto i) { a[i] = i; });
  });
  for (host_accessor a { buf }; auto e : a)
     std::cout << e << std::end;
}
```

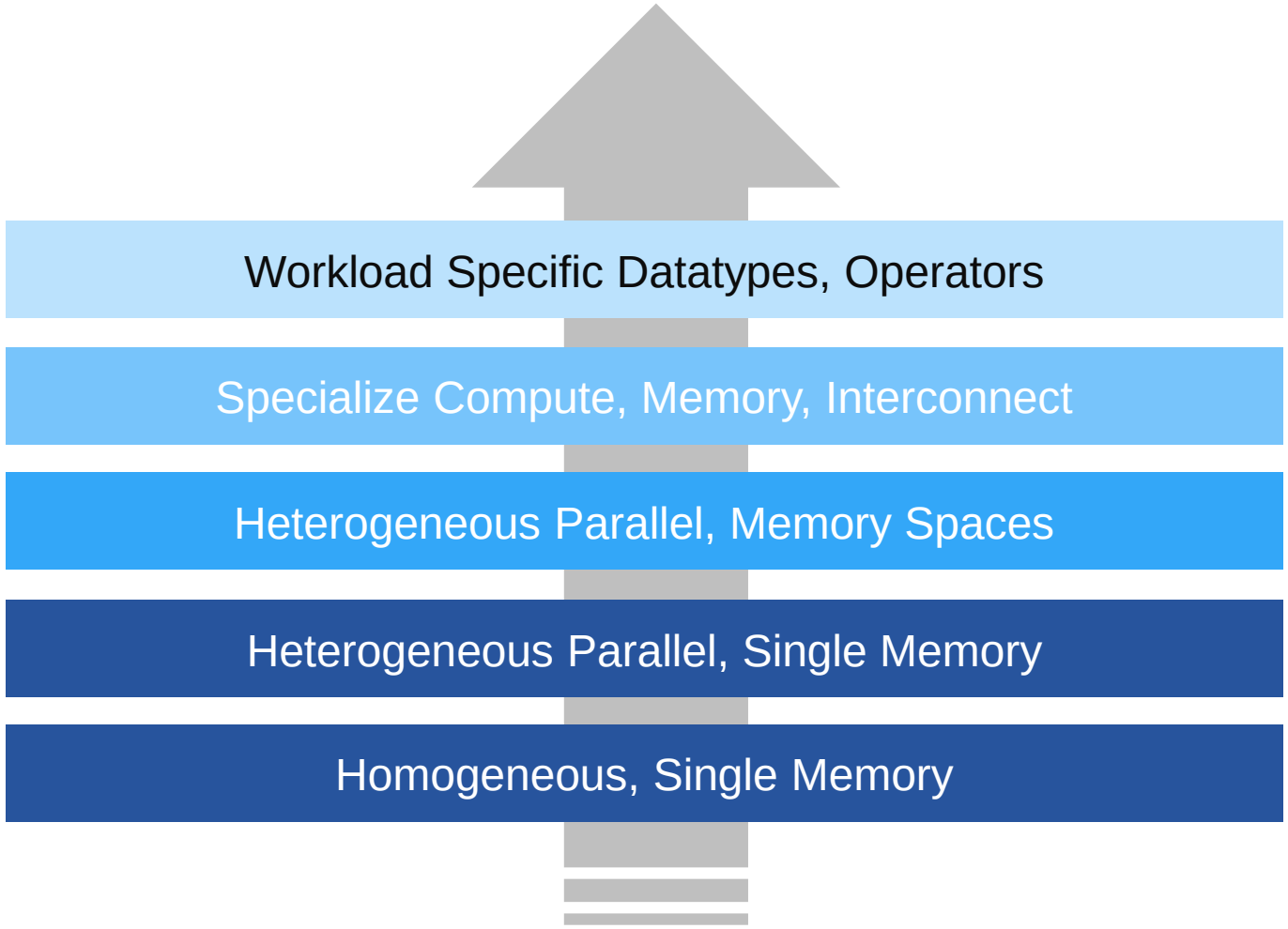
- Abstract storage

- Code executed on device (“kernel”)
- “Single-source”
  - Seamless integration in host code
  - Type-safety

- Accessor
  - Express access intention
  - Implicit data flow graph
  - Automatic data transfers across devices
  - Overlap computation & communication

#include <C++>

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



Domain Lib

Architecture Lib

C++SYCL

C++Parallel Lib

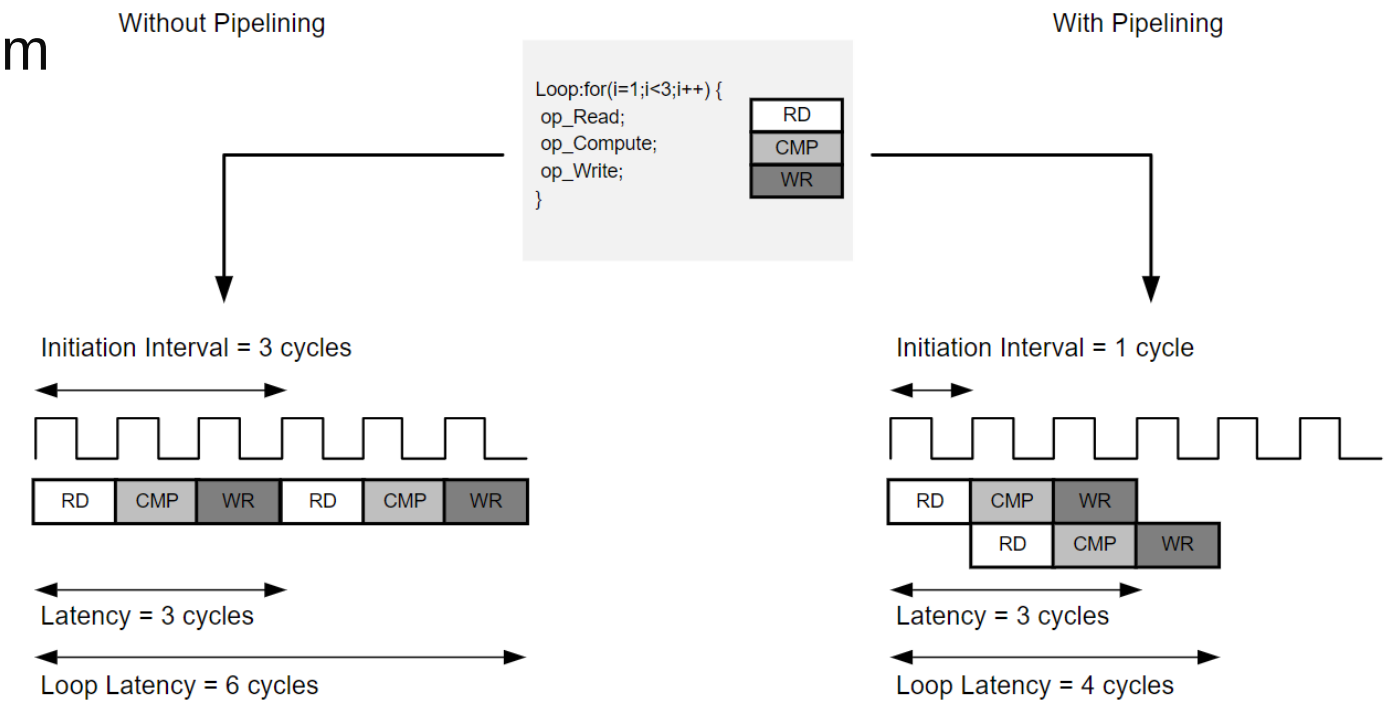
C++20

XILINX

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



- 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

Partitioning memories

- Remember bank conflicts on Cray in the 70's HPC? ☺
- In FPGA world, even memory is configurable!
- Example of 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

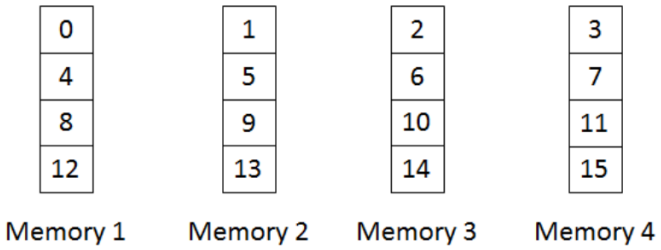


Figure 7-1: **Physical Layout of Buffer After Cyclic Partitioning**

- 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

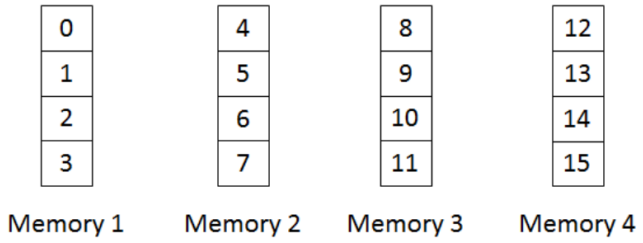


Figure 7-2: **Physical Layout of Buffer After Block Partitioning**

- Complete Partitioning
  - Extreme distribution
  - Extreme bandwidth
  - Low latency

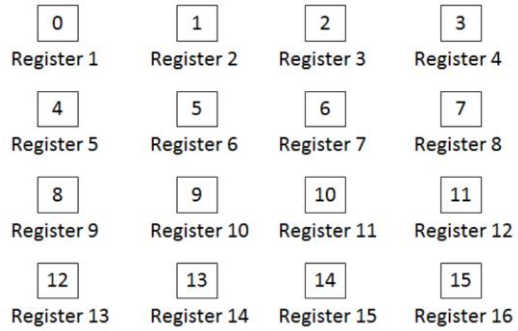


Figure 7-3: **Physical Layout of Buffer After Complete Partitioning**

Array partitioning & pipelining in triSYCL for Xilinx FPGA

```
cgh.single_task<krnl_sobel>([=] {
    auto gX = xilinx::partition_array<char, 9,
            xilinx::partition::complete<1>>({-1, 0, 1, -2, 0, 2, -1, 0, 1});
    auto gY = xilinx::partition_array<char, 9,
            xilinx::partition::complete<1>>({1, 2, 1, 0, 0, 0, -1, -2, -1});
    for (size_t x = 1; x < width - 1; ++x) {
      for (size_t y = 1; y < height - 1; ++y) {
        auto magX = 0;
        auto magY = 0;
        xilinx::pipeline([&] {
          for(size_t k = 0; k < 3; ++k) {
            for(size_t l = 0; l < 3; ++l) {
              auto gI = k * 3 + l;
              auto pIndex =  (x + k - 1) + (y + l - 1) * width;
              magX += gX[gI] * pixel_rb[pIndex];
              magY += gY[gI] * pixel_rb[pIndex];
            }
          }
        });
        pixel_wb[x + y * width] = cl::sycl::min((int)(cl::sycl::abs(magX)
                                    + cl::sycl::abs(magY)), 0xFF);
      }
    }
  });
```

https://github.com/triSYCL/sycl/blob/sycl/unified/next/sycl/test/on-device/xocc/edge_detection/edge_detection.cpp

#include <C++>

Specify DDR bank mapping with accessor properties

```
#include <sycl/sycl.hpp>
int main() {
  sycl::buffer<cl::sycl::cl_int> Buffer { 4 };
  sycl::queue Queue;

  Queue.submit([&](auto &cgh) {
    sycl::accessor Accessor { Buffer, cgh, sycl::write_only, { sycl::vendor::xilinx::ddr_bank<3> }};
    cgh.parallel_for<class SmallerTestb>(Buffer.get_size(), [=](int WIid) { Accessor[WIid] = WIid; });
  });
}
```

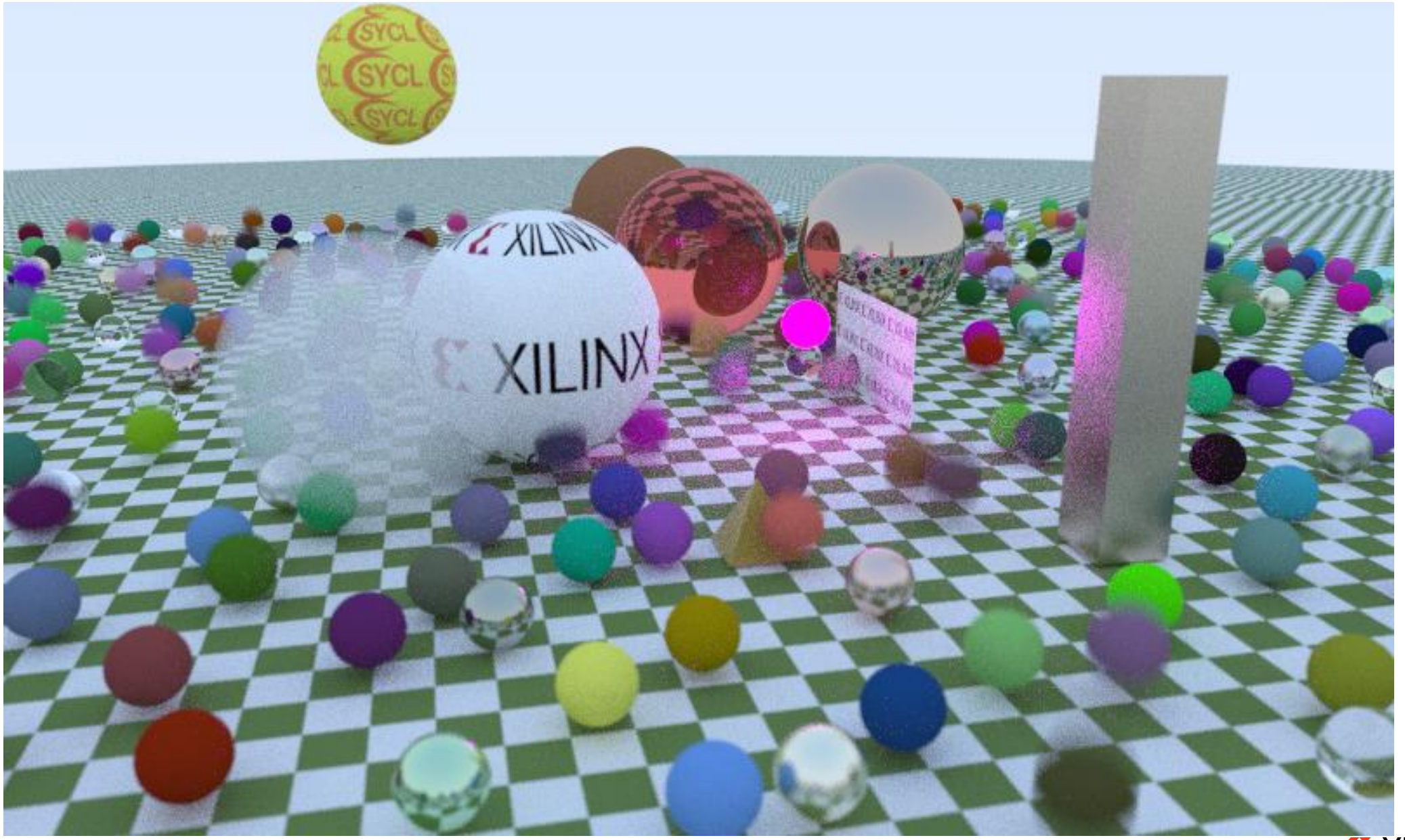
Specify Vitis options with kernel property decorators

```
#include <sycl/sycl.hpp>
#include <sycl/vendor/xilinx/fpga.hpp>
using namespace sycl::vendor::xilinx::literal;

int main() {
  sycl::buffer<int> Buffer { 4 };
  sycl::queue Queue;

  Queue.submit([&](sycl::handler &cgh) {
    auto Accessor = Buffer.get_access<sycl::access_mode::write>(cgh);
    cgh.single_task<class FirstKernel>(sycl::vendor::xilinx::kernel_param("--optimize 2"_cstr,
                                                                          [=] { Accessor[0] = 0; }));
  });
  Queue.submit([&](sycl::handler &cgh) {
    auto Accessor = Buffer.get_access<sycl::access_mode::write>(cgh);
    cgh.single_task<class SecondKernel>
            (sycl::vendor::xilinx::kernel_param("--kernel_frequency 300"_cstr,
                                                [=] { Accessor[1] = 1; }));
  });
}
```

Enable new application domains on FPGA: path tracing!



- https://github.com/triSYCL/path_tracer

Replace old dynamic polymorphism with C++17 std::variant

- Dynamic polymorphism usually not handled by accelerators: ≈ function pointers ☹
  - https://raytracing.github.io/books/RayTracingInOneWeekend.html#surfacenormalsandmultipleobjects/anabstractionforhittableobjects
- HLS does some trivial devirtualization when there is only 1 class in use…
- C++17 `std::variant` allows other way to handle multiple dispatch

```
struct hittable {
  virtual bool hit(const ray& r, double t_min, double t_max, hit_record& rec)
  const = 0;
};
struct sphere : hittable {
  virtual bool hit(const ray& r, double t_min, double t_max, hit_record& rec)
  const override { ... };
};
struct rectangle : hittable {
  virtual bool hit(const ray& r, double t_min, double t_max, hit_record& rec)
  const override { ... };
};

color ray_color(const ray& r, const hittable& world) {
  hit_record rec;
  if (world.hit(r, 0, infinity, rec)) {
     return 0.5 * (rec.normal + color(1,1,1));
  }
  vec3 unit_direction = unit_vector(r.direction());
  auto t = 0.5*(unit_direction.y() + 1.0);
  return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

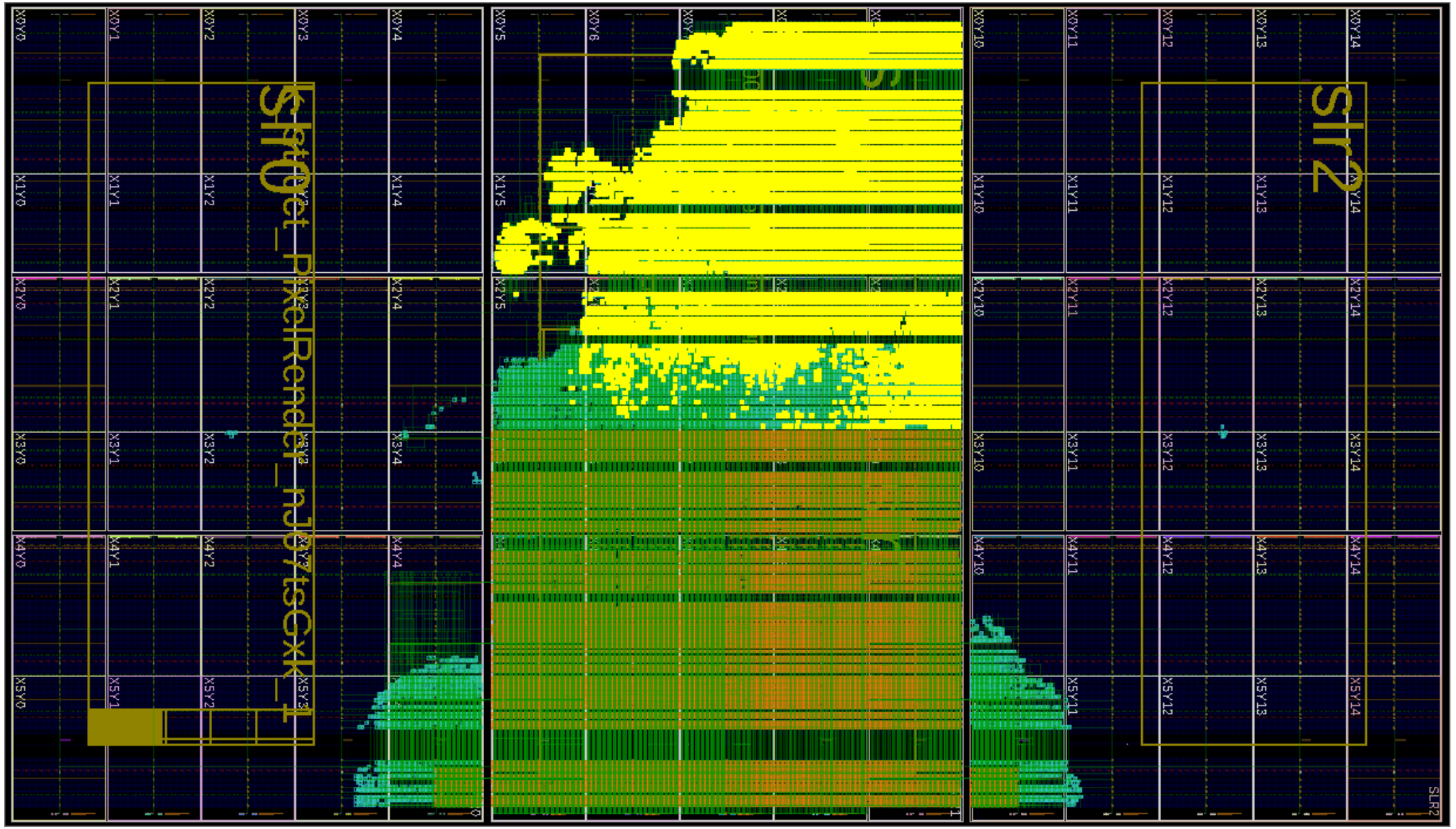
```
// "Sum type" or "union type" from functional languages
using hittable_t = std::variant<sphere, rectangle>;

struct sphere {
  bool hit(const ray& r, double t_min, double t_max, hit_record& rec)
  const { ... };
};
struct rectangle {
  bool hit(const ray& r, double t_min, double t_max, hit_record& rec)
  const { ... };
};

color ray_color(const ray& r, const hittable_t& world) {
  hit_record rec;
  if (std::visit([&](auto&& arg) { arg.hit(r, 0, infinity, rec); }, world)) {
     return 0.5 * (rec.normal + color(1,1,1));
  }
  vec3 unit_direction = unit_vector(r.direction());
  auto t = 0.5*(unit_direction.y() + 1.0);
  return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

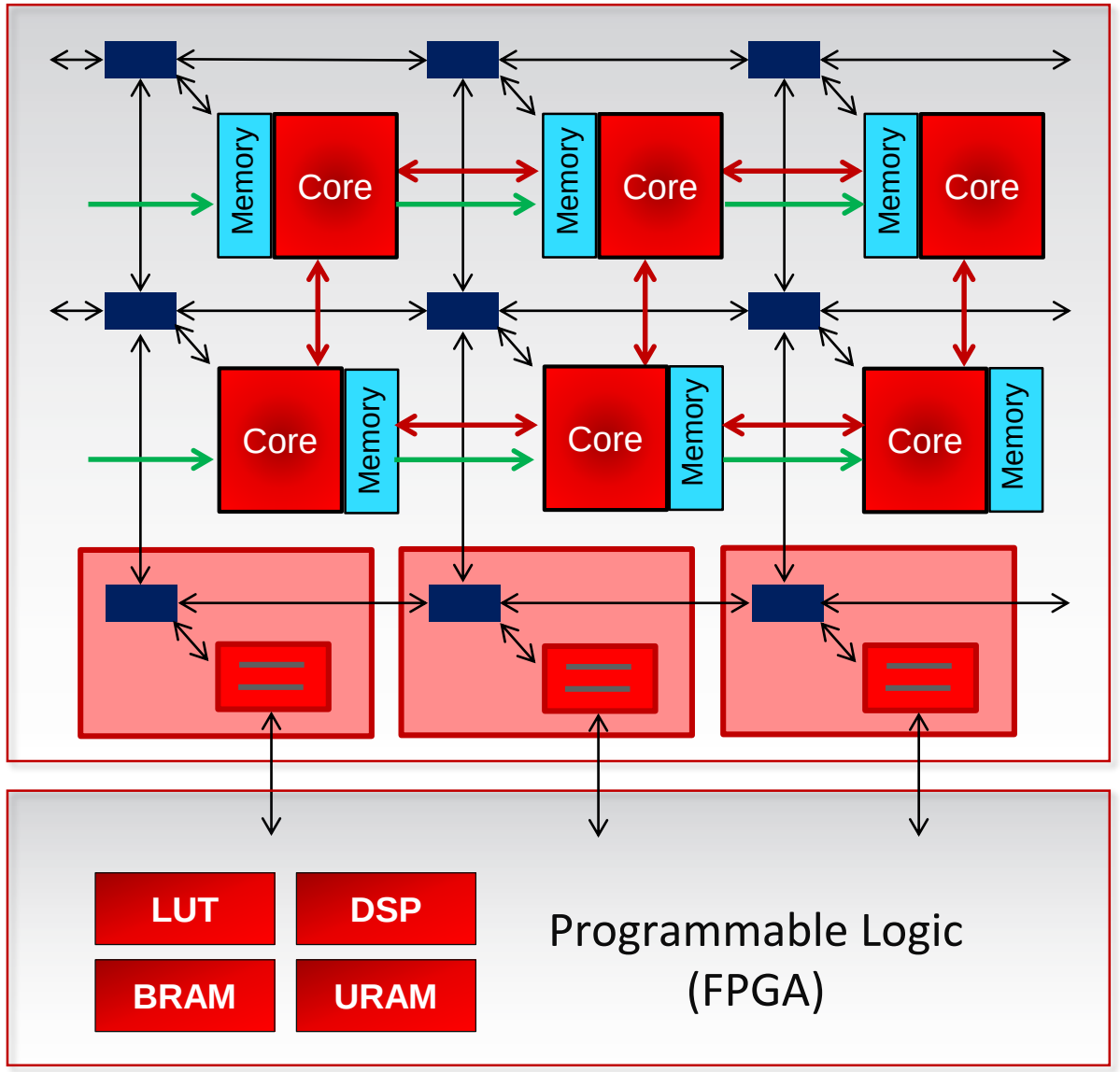
- An FPGA can dispatch a `std::visit` in O(1)! ☺

Layout of path_tracer on Xilinx Alveo U200 FPGA PCIe card



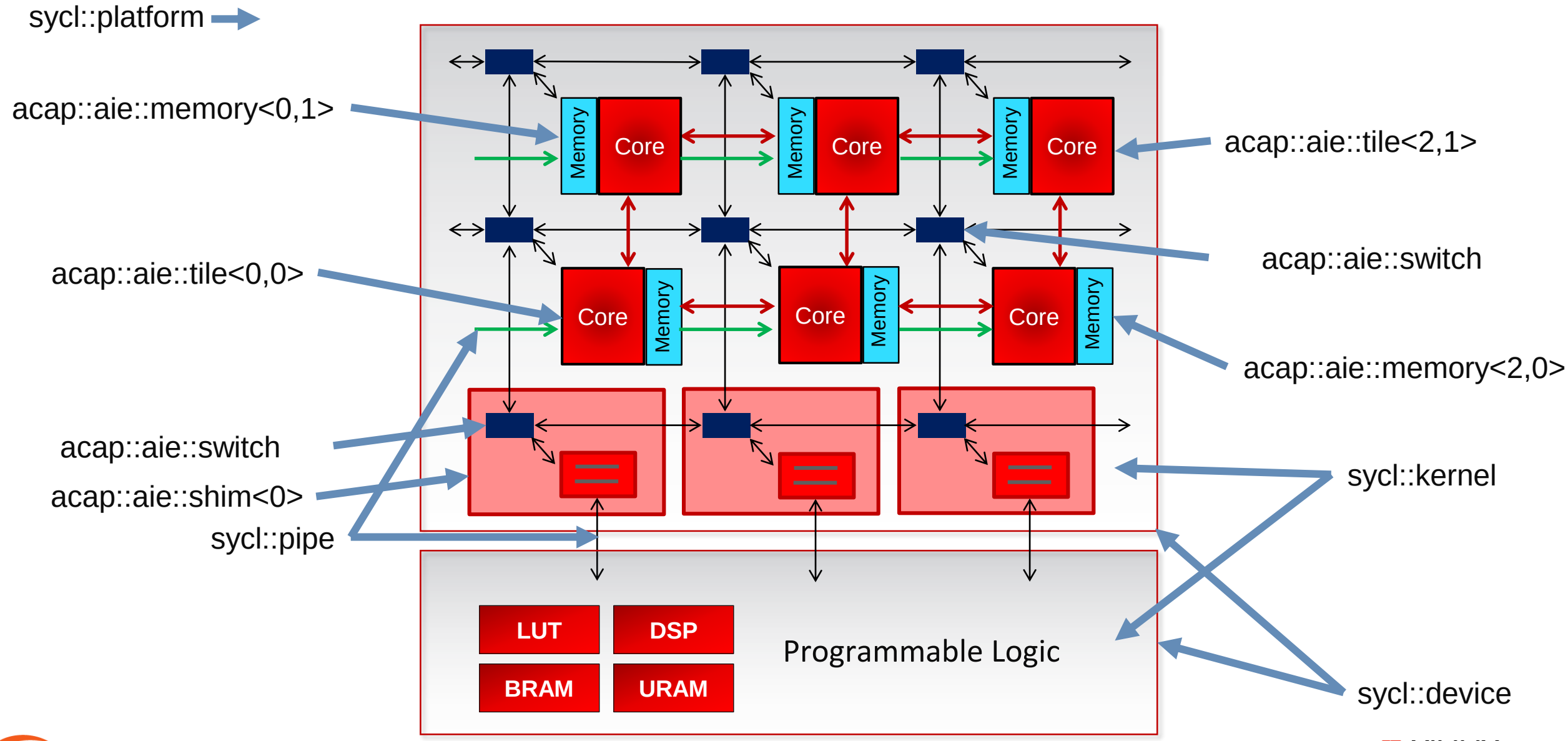
Xilinx ACAP SYCL extensions

CGRA inside Xilinx Versal ACAP VC1902 (37B tr, 7nm)



- Scalar Engines:
  - ARM dual-core Cortex-A72 (Host)
  - ARM dual-core Cortex-R5
- Programmable Logic (FPGA)
- Coarse-Grain Reconfigurable Array
  - 400 AI Engine (AIE) cores (tiles)
  - Each AIE tile contains:
    - 32-bit scalar RISC processor
    - 512-bit VLIW SIMD vector processor
    - 32KiB RAM
    - Each tile can access neighbor's memory
      - 128KiB shared
    - 16KiB program memory
- NoC, Fast I/O…

ACAP++: SYCL abstractions templated by 2D coordinates



#include <C++>

SPMD Mandelbrot

- Start simple: no (neighborhood) communication ☺

```
#include "triSYCL/vendor/Xilinx/graphics.hpp"
#include <complex>
#include <cstdint>
#include <sycl/sycl.hpp>

using namespace sycl::vendor::xilinx;
auto constexpr image_size = 229;

int main(int argc, char* argv[]) {
  acap::aie::device< acap::aie::layout::vc1902> aie;
  graphics::application a;

  // Open a graphic view of a AIE array
  a.start(argc, argv, aie.x_size, aie.y_size, image_size, image_size, 1)
      .image_grid()
      .get_palette()
      .set(graphics::palette::rainbow, 100, 2, 0);
```

```
  // Short-cut to launch same kernel on AIE tiles
  aie.uniform_run([&](auto th) {
    // The local pixel tile inside the complex plane
    std::uint8_t plane[image_size][image_size];
    // Computation rectangle in the complex plane
    auto constexpr x0 = -2.1, y0 = -1.2, x1 = 0.6, y1 = 1.2;
    auto constexpr D = 100; // Divergence norm
    // Size of an image tile
    auto constexpr xs = (x1 - x0) / th.x_size() / image_size;
    auto constexpr ys = (y1 - y0) / th.y_size() / image_size;
    while (!a.is_done()) {
      for (int j = 0; j < image_size; ++j)
        for (int k, i = 0; i < image_size; ++i) {
          std::complex c { x0 + xs * (th.x() * image_size + i),
                           y0 + ys * (th.y() * image_size + j) };
          std::complex z { 0.0 };
          for (k = 0; norm(z = z * z + c) < D && k <= 255; k++)
            ;
          plane[j][i] = k;
        }
      a.update_tile_data_image(th.x(), th.y(), &plane[0][0], 0, 255);
    }
  });
}
```

Design pattern 1

#include <C++>

XILINX

Zoom in: each tile as a sub-device + spatial iterating functions

```
#include <iostream>
#include <sycl/sycl.hpp>

using namespace sycl::vendor::xilinx;

int main() {
  // Define an AIE CGRA with all the tiles of a VC1902
  acap::aie::device<acap::aie::layout::vc1902> d;
  // 1 buffer per tile
  sycl::buffer<int> b[d.x_size][d.y_size];
  // Initialize on the host each buffer with 3 sequential values
  d.for_each_tile_index([&](int x, int y) {
    b[x][y] = { 3 };
    sycl::host_accessor a { b[x][y] };
    std::iota(a.begin(), a.end(), (d.x_size * y + x) * a.size());
  });
```

Design pattern 2

```
  //  Submit some work on each tile
  d.for_each_tile_index([&](int x, int y) {
    d.tile(x, y).submit([&](auto& cgh) {
      acap::aie::accessor a { b[x][y], cgh };
      cgh.single_task([=] {
        for (auto& e : a)
          e += 42;
      });
    });
  });
  // Wait for the end of each tile execution
  d.for_each_tile([](auto& t) { t.wait(); });
  // Check the result
  d.for_each_tile_index([&](int x, int y) {
    for (sycl::host_accessor a { b[x][y] };
          auto&& [i, e] : ranges::views::enumerate(a))
      if (e != (d.x_size * y + x) * a.size() + i + 42)
        throw "Bad computation";
  });
}
```


#include <C++>

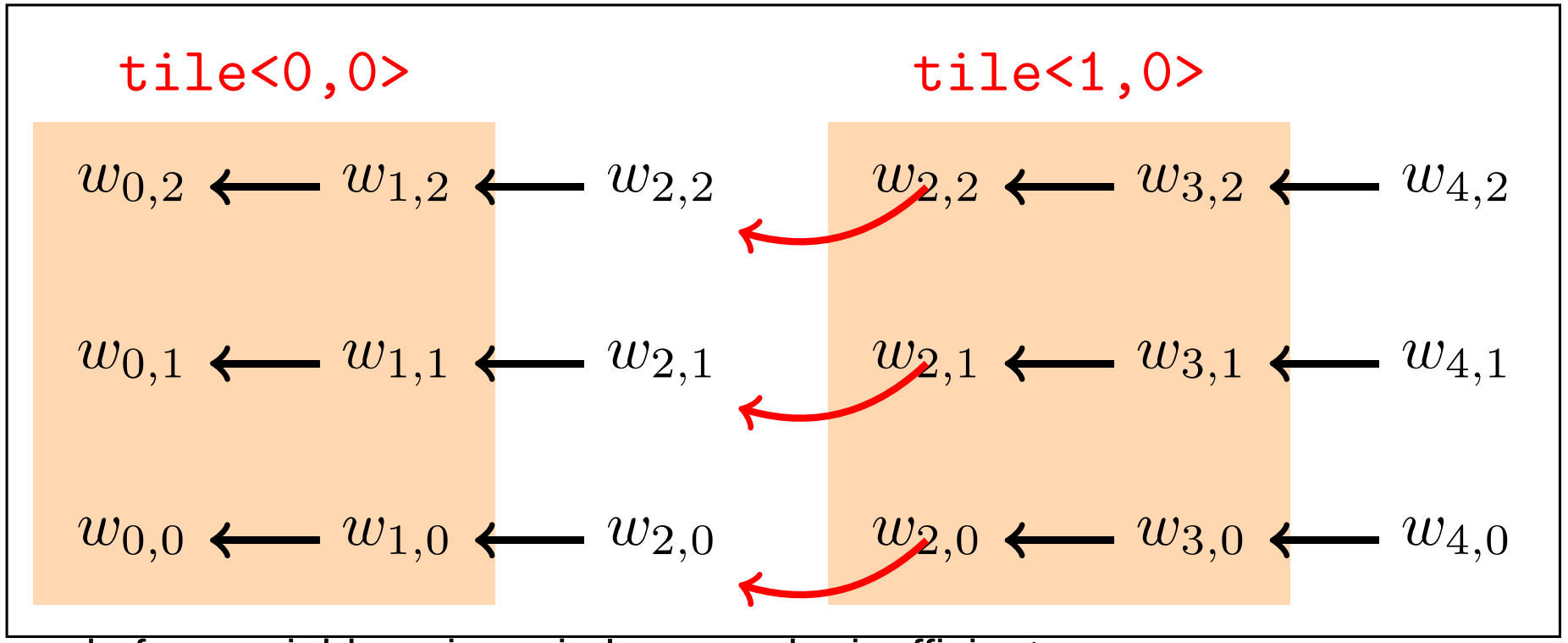
Interacting with neighbors

```
/// Compute a time-step of wave propagation partial differential equation
  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;
      }
  }
```

Design pattern 3

Use overlaping/ghost/halo variables

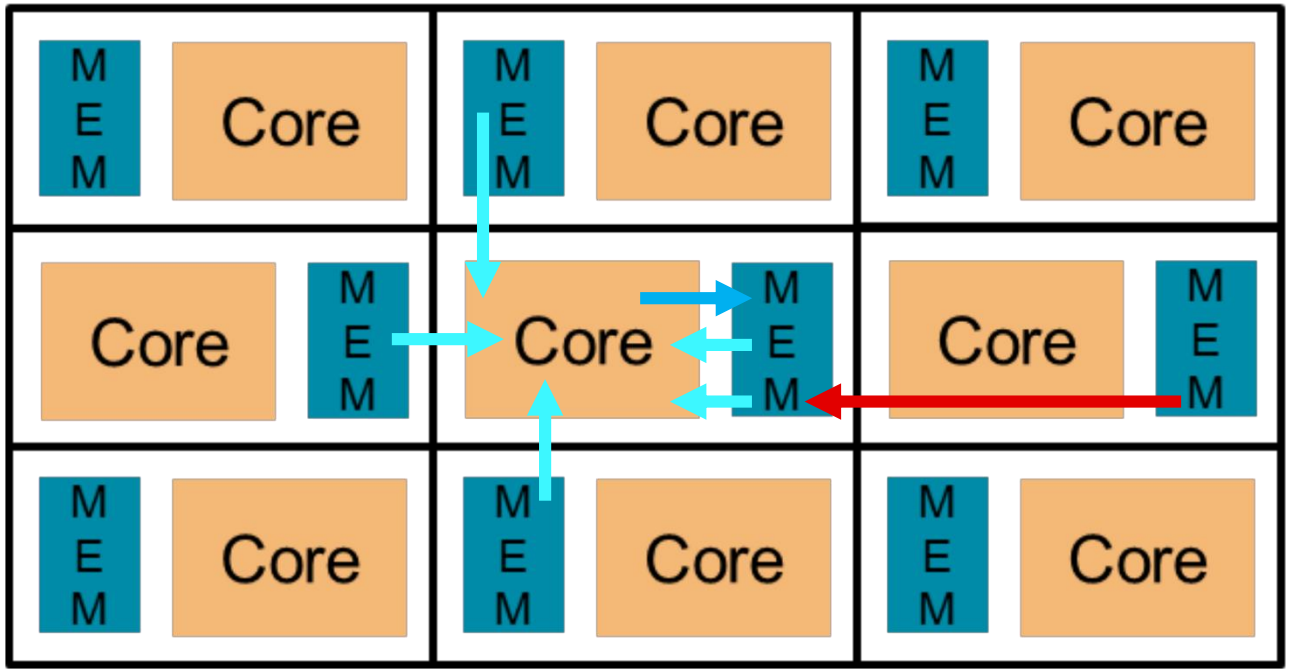
$$u_{i,j} = f(w_{i+1,j}, w_{i,j})$$



- 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

2D memory module/tile subtleties: checkerboard…

- Typical 1-neighbor stencil code with 4 neighbors
- AIE can actually access only 3 neighbor memories…



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];
    }
  }
```

- TODO
  - Make a nice generic C++ library to handle generic stencil code
  - Possible because SYCL is single-source

Play with asynchronous DMA on AIE tiles (code)

Design pattern 4

```
template <typename AIE, int X, int Y>
struct right_neighbor : acap::aie::tile<AIE, X, Y> {
  using t = acap::aie::tile<AIE, X, Y>;
  void run() {
    unsigned int src[data_size];
    unsigned int dst[data_size];
    std::iota(std::begin(src), std::end(src), 0);
    for (int i = 0; i < local_transfers; ++i) {
      if constexpr (!t::is_east_column())
        // There is a neighbor on the right: send some data
        t::tx_dma(0).send(src);
      if constexpr (!t::is_west_column())
        // There is a neighbor on the left: receive some data
        t::rx_dma(0).receive(dst);
      if constexpr (!t::is_east_column())
        // There is a neighbor on the right: wait for the end of transmission
        t::tx_dma(0).wait();
      if constexpr (!t::is_west_column())
        // There is a neighbor on the left: wait for the end of reception
        t::rx_dma(0).wait();
    }
    if constexpr (!t::is_west_column())
      // Once it is received, we can check the result
      BOOST_CHECK(ranges::equal(src, dst));
  }
};

auto measure_bandwidth = [](auto transfered_bytes, const auto& some_work) {
  auto starting_point = clk::now();
  some_work();
  // Get the duration in seconds as a double
  std::chrono::duration<double> duration = clk::now() - starting_point;
  std::cout << " time: " << duration.count()
            << " s, bandwidth: " << transfered_bytes / duration.count()
            << " B/s" << std::endl;
};
```

```
int test_main(int argc, char* argv[]) {
  try {
    using d_t = acap::aie::device<layout::vc1902>;
    d_t d;
    // Configure the AIE NoC connections
    d.for_each_tile_index([&](auto x, auto y) {
      // When it is possible, connect each tile to its right neighbor
      if (d_t::geo::is_x_y_valid(x + 1, y)) {
        d.tile(x, y).connect(d_t::csp::dma_0, d_t::cmp::east_0);
        d.tile(x + 1, y).connect(d_t::csp::west_0, d_t::cmp::dma_0);
      }
    });

    // Dump the configuration for these slides ☺
    d.display("async_transfer.tex");

    std::cout << "Start right neighbor with AIE device (" << d_t::geo::x_size
              << ',' << d_t::geo::y_size << ')' << std::endl;

    // Right now, the communications are globally costly so keep them
    // globally constant
    local_transfers =
       total_data_communication / (d_t::geo::x_max - 1) / d_t::geo::y_size;
    // Only x_max since the last column has no right neigbors to talk to
    auto transmitted_bytes = sizeof(std::int32_t) * data_size *
                             local_transfers * (d_t::geo::x_max - 1) *
                             d_t::geo::y_size;

    measure_bandwidth(transmitted_bytes, [&] { d.run<right_neighbor>(); });
  } catch (sycl::exception& e) {
    // Display the string message of any SYCL exception
    std::cerr << e.what() << std::endl;
    // Rethrow to make clear something bad happened
    throw;
  }
  return 0;
}
```

Implementation

Multi-level implementation/emulation for codesign & debug

Different types of implementations

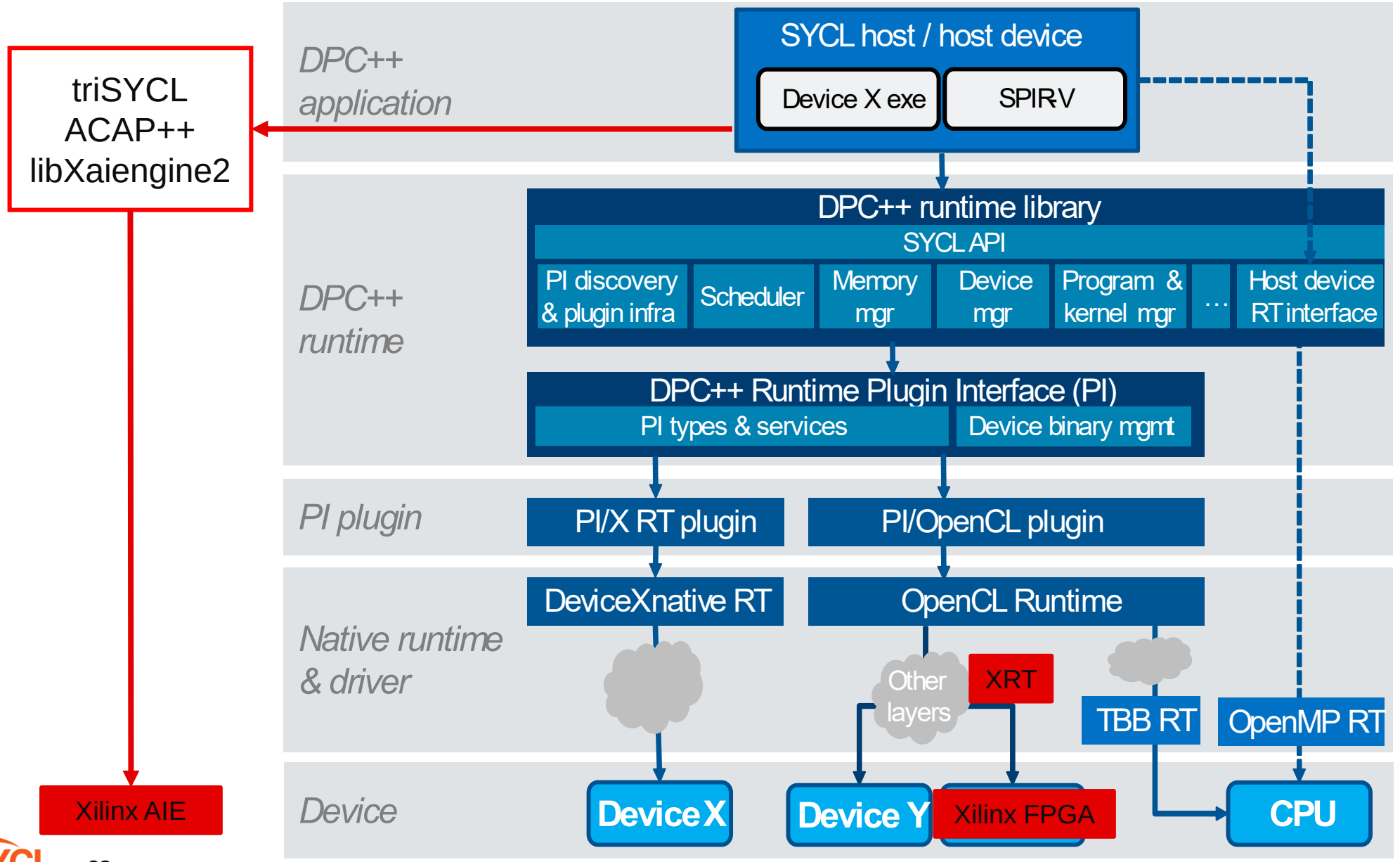
- Full SYCL compiler & runtime implementation
  - Run on real hardware or hardware simulator
- Pure SYCL C++ implementation
  - No specific compiler required!
  - Run on (laptop) host CPU at full C++ speed, standard debugging, thread-sanitizer of hardware features…
    - 1 thread per host… thread, 1 thread per AIE tile, 1 thread per GPU work-item, 1 thread per FPGA work-item
  - Easy code instrumentation for statistics by adapting SYCL C++ classes
  - Use normal debugger
    - Gdb is scriptable in Python to expose new features ☺
  - Can experiment with Xilinx devices from year 2030 ☺
- Mix-and-match
  - Run some parts of the hardware remotely or in simulators
  - Allow kernels on host CPU while using memory-mapped real hardware (DMA, AXI streams, NoC…)
  - Distribute execution across datacenter (Celerity SYCL for MPI+SYCL)

Interaction with other tools: a complementary tool…

- SYCL
  - Just cares about simple single-source programming of host+devices
  - No special a priori magic (no auto-parallelization, no polyhedral model, no auto-data-flow…)
  - In pure C++ we trust
- Leverage any magic from other tools in the backend
  - Benefit from HLS magic for FPGA device part! ☺
  - Benefit from AIE optimizations and AIE API on AIE device part ☺
  - Benefit from GPU optimizations on GPU device part
- Interoperability mode of SYCL to use explicitly other programming model if needed
  - Simplify OpenCL host & XRT host-side programming
  - Avoid yet-another-my-wrapper NIH syndrome or N+1 problem
- Use C++ libraries to expose hardware features in a portable way
  - Enable cross-device emulation for free

Xilinx branch of oneAPI DPC++ SYCL interoperability backstage

Adapted from https://github.com/intel/llvm/blob/sycl/sycl/doc/CompilerAndRuntimeDesign.md



https://github.com/intel/llvm/blob/sycl/sycl/doc/PluginInterface.md

#include <C++>

Inclusive heterogeneous computing…

No transistors left behind!

Execution on Intel CPU + Xilinx FPGA (OpenCL) + Nvidia GPU (CUDA)

```
#include <iostream>
#include <sycl/sycl.hpp>
int main() {
 sycl::buffer<int> v { 10 };

 auto run = [&](auto sel, auto work) {
   sycl::queue { sel }.submit([&](auto& h) {
     auto a = sycl::accessor { v, h };
     h.parallel_for(a.get_count(), [=](auto i) { work(i, a); });
   });
 };
 run(sycl::host_selector {}, [](auto i, auto a) { a[i] = i; }); // CPU
 run(sycl::accelerator_selector {}, [](auto i, auto a) { a[i] = 2*a[i]; }); // FPGA
 run(sycl::gpu_selector {}, [](auto i, auto a) { a[i] = a[i] + 3; }); // GPU

 sycl::host_accessor acc { v };
 for (int i = 0; i != v.get_count(); ++i)
   std::cout << acc[i] << ", ";
 std::cout << std::endl;
}
```

- No `template` or `typename` or `class` or… ☺
- No extension or attribute or… ☺
- Generic & type-safe
- No explicit data motion or boiler-plate code
- Different accelerators/vendors in same program!

```
clang++ -std=c++20 -fsycl -fsycl-targets=nvptx64-nvidia-cuda-sycldevice,fpga64_sw_emu FPGA_GPU_CPU.cpp -o FPGA_GPU_CPU
```


#include <C++>

Resources

- Open-source fusion triSYCL + Intel to Xilinx FPGA
  - https://github.com/triSYCL/sycl
- Intel SYCL open-source to be up-streamed, aka oneAPI DPC++
  - https://github.com/intel/llvm
- Open-source triSYCL (mainly Xilinx)
  - https://github.com/triSYCL/triSYCL
- Fusion triSYCL + Intel SYCL to Xilinx Versal ACAP AIE CGRA: ACAP++
  - To be open-sourced
    - https://gitenterprise.xilinx.com/rkeryell/acappp (fork of triSYCL runtime)
    - https://gitenterprise.xilinx.com/rkeryell/sycl (fork of oneAPI DPC++)

Conclusion

- Only open-standard and open-source matter
  - Very good spirit among SYCL WG and implementors
  - oneAPI DPC++ is a strong SYCL implementation in the process to be up-streamed into Clang/LLVM
- SYCL 2020 ≡ pure modern C++ DSL
  - Extensible with abstractions for any device: FPGA, ACAP++, PiM/IMC…
  - Target emulation, debug & co-design on CPU for free
  - 3nm devices deserve 3nm C++! ☺
- Inclusive heterogeneous computing
  - Adaptable interoperability with any device from any vendor in same application
    - Heterogeneous programming CPU+GPU+CGRA+FPGA+DSP+Vector+AI+PiM/IMC…
  - No transistor left behind! ☺
  - Can even combine various SYCL implementations (no size fits all)

Thank You