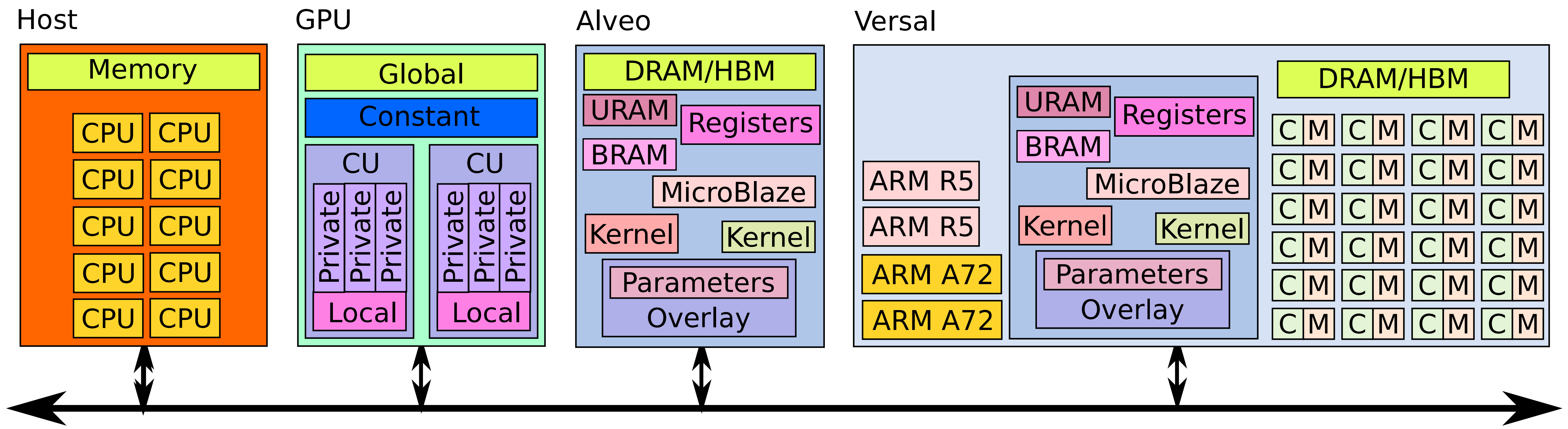
SYCL

A Single-Source C++ Standard for Heterogeneous Computing

Ronan Keryell
Xilinx Research Labs (San José, California) & Khronos SYCL specification editor
2019/11/17 SC19/H2RC keynote

Programing a *full* modern/future system…



> Add your own accelerator to this picture...

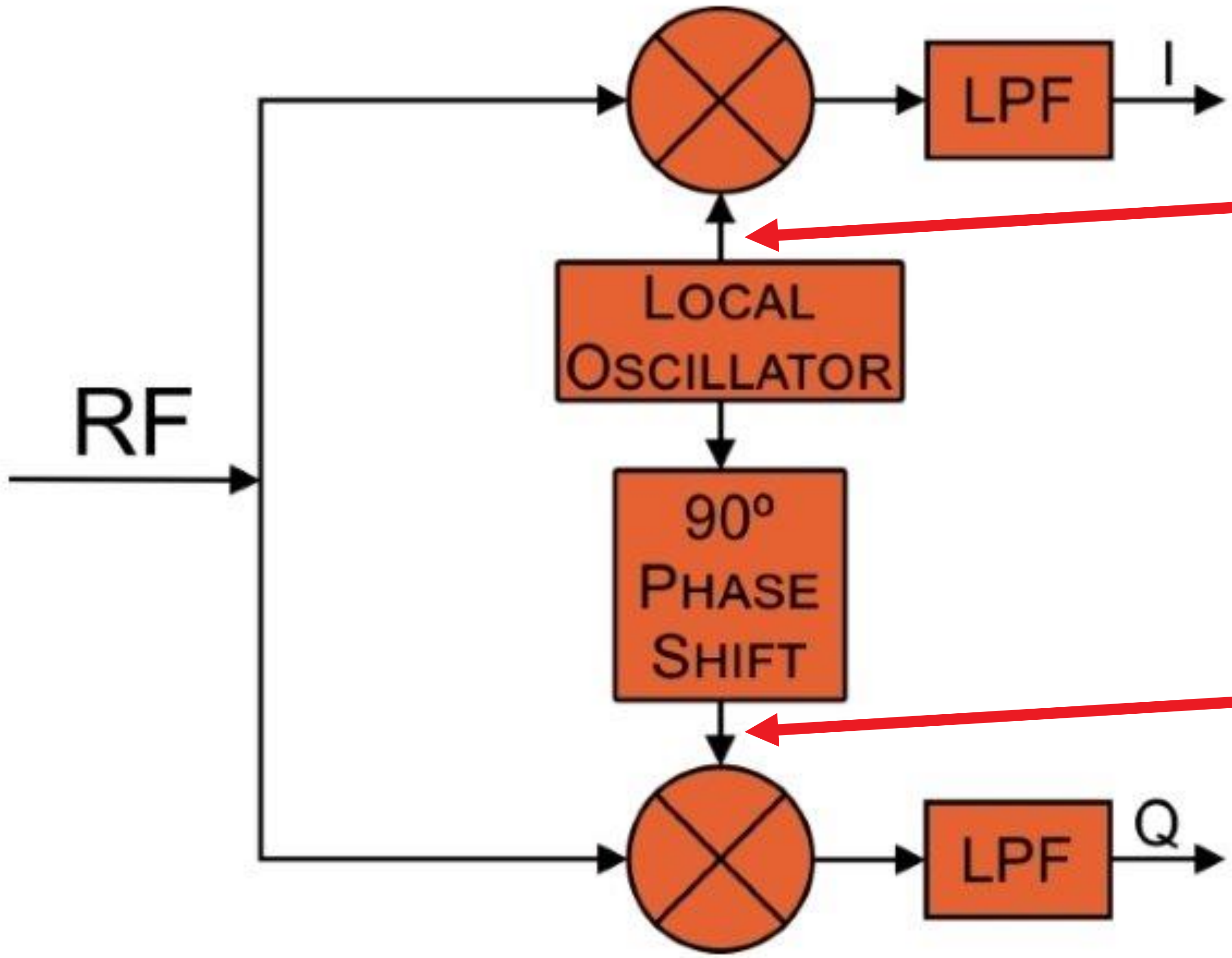
> Scale this at the data-center/HPC level too…

Example: radio receiver

RUNS CUSTOM BLEND OF ANDROID AND iOS
SIDE-FACING CAMERA
SIMULATES ALTERNATIVE SPEED OF LIGHT (DEFAULT: 100 MILES PER HOUR) AND ADJUSTS CLOCK AS PHONE ACCELERATES
CLEAR SCREEN
REALISTIC CASE
WIRELESS
ACCELEROMETER DETECTS WHEN PHONE IS IN FREE FALL AND MAKES IT SCREAM
FLIGHTAWARE PARTNERSHIP: MAKES AIRPLANE NOISE WHEN FLIGHTS PASS OVERHEAD
WHEN EXPOSED TO LIGHT, PHONE SAYS "HI!"
INTRODUCING
THE XKCD PHONE
YOUR MOBILE WORLD JUST WENT DIGITAL®

> https://xkcd.com/1363

Simple example: generate sequence in quadrature



In_phase

quadrature

> https://www.allaboutcircuits.com/textbook/radio-frequency-analysis-design/radio-frequency-demodulation/understanding-quadrature-demodulation

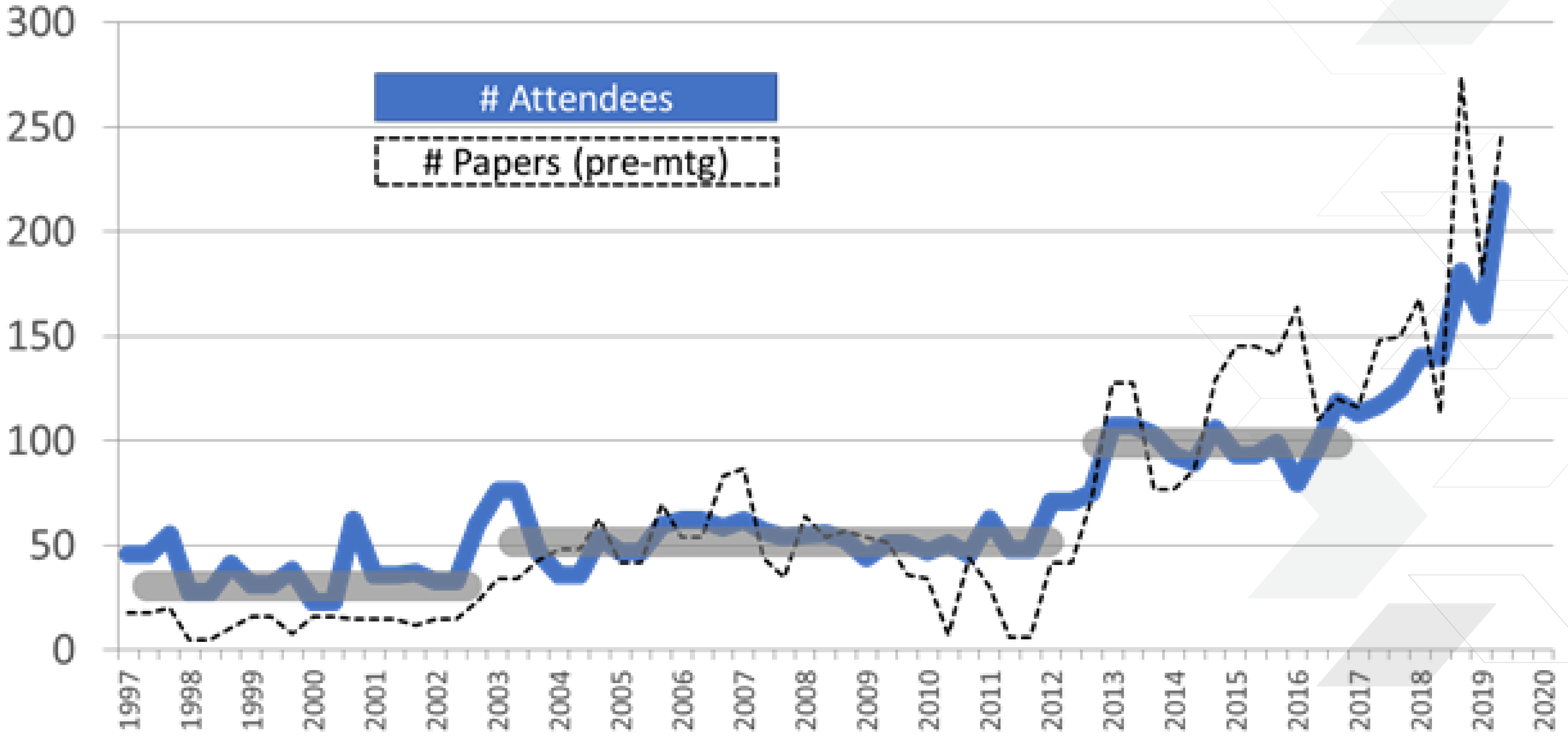
Generate sequence in quadrature: the Python 3.8 way

```python
#! /usr/bin/env python3
import itertools
import sys

samples = 8
if samples % 4 != 0 :
  sys.exit("sample numbers need to be a multiple of 4")

table = [ t/(samples - 1) for t in range(samples) ]

def tabulate(x):
  return table[x]

(t, t2) = itertools.tee(itertools.cycle(range(samples)), 2)
in_phase = map(tabulate, t)
quadrature = map(tabulate, itertools.islice(t2, int(samples/4), None))
output = itertools.zip_longest(in_phase, quadrature)

for v in output:
  print ("{:f} {:f}".format(v[0], v[1]))
```

More efficiency? Remember C++?

- **C++ used to be almost a dead language…**
- **C++ reboot since C++11 with 1 version every 3 years**
  - Very different from old 1998 C++…
    - But people often just remember *only* this dark time ☹
  - …But still compatible!
- **C & C++ run the world infrastructure**
- **C & C++ compatible with all other programming languages**
- **2-line description of C++ by Bjarne Stroustrup is *still* valid**
  - Direct mapping to hardware
  - Zero-overhead abstraction

ISO C++ committee proposal papers & attendance



XILINX.

Generate sequence in quadrature: compare the C++20 way

```python
#! /usr/bin/env python3
import itertools
import sys

samples = 8
if samples % 4 != 0 :
  sys.exit("sample numbers need to be a multiple of 4")

table = [ t/(samples - 1) for t in range(samples) ]

def tabulate(x):
  return table[x]

(t, t2) = itertools.tee(itertools.cycle(range(samples)), 2)
in_phase = map(tabulate, t)
quadrature = map(tabulate,
                 itertools.islice(t2, int(samples/4), None))
output = itertools.zip_longest(in_phase, quadrature)

for v in output:
  print ("{:f} {:f}".format(v[0], v[1]))
```

```cpp
#include <iostream>
#include <range/v3/all.hpp>
using namespace ranges;

auto constexpr samples = 8;
static_assert(samples % 4 == 0,
              "sample numbers need to be a multiple of 4");
auto constexpr generate_table() {
  std::array<float, samples> a;
  for (int i = 0; i < samples; ++i)
    a[i] = i/(samples - 1.f);
  return a;
}
auto constinit table = generate_table();
auto tabulate = [](auto v) { return table[v]; };

int main() {
  auto r = views::ints | views::take(samples) | views::cycle;
  auto in_phase = r | views::transform(tabulate);
  auto quadrature = r | views::drop(samples/4)
                      | views::transform(tabulate);
  auto output = views::zip(in_phase, quadrature);

  for (auto e : output)
    std::cout << e.first << ' ' << e.second << std::endl;
}
```

https://godbolt.org/z/VUv73N

Comparison between Python 3.8 & C++20

> **Python3**
>> Interpreted
>> Easy to write incrementally without recompiling everytime
>> Error detected at runtime
>> 188 ns/pair value on my Linux Intel Xeon E-2176M 2.7GHz laptop

> **C++20**
>> Compiled
>> Slow to compile when developing
>> Error detected at compile time
– Strong type-safety
>> Explicit resource control
>> 2.408 ns/pair value (x78 speedup) on my Linux Intel Xeon E-2176M 2.7GHz laptop

> **C++20: Python productivity + C++ efficiency**

Modern Python/C++20/Old C++

> **Modern Python (3.8)**

```
v = [ 1,2,3,5,7 ]
for e in v:
  print(e)
```

> **Modern C++ (C++20)**

```
std::vector v { 1,2,3,5,7 };
for (auto e : v)
  std::cout << e << std::endl;
```

> **Old C++ (C++98/C++03)**

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(5);
v.push_back(7);
for (std::vector<int>::iterator i =
       v.begin(); i != v.end(); ++i)
  std::cout << *i << std::endl;
```

C++20 down to assembly code specialization

```
#include <cassert>
#include <type_traits>
constexpr auto sum = [](auto arg, auto... args) {
  if (std::is_constant_evaluated()) {
    return (arg + ... + args);
  } else {
    ([&](auto value) {
      asm("leal (%1, %2), %0"
        : "=r" (arg)
        : "r"  (arg),
          "r"  (value));
    }(args), ...);
    return arg;
  }
};
static_assert(6 == sum(1, 2, 3));
int main(int argc, char *[]) {
  assert(10 == sum(argc, 2, 3, 4));
}
```

- Remember inline assembly???
- Now with compile-time specialization without code change!!!
- https://godbolt.org/z/CJsKZT
- C is just *le passé*… ☺

- Inspired by Kris Jusiak @krisjusiak Oct 24 2019
  - https://twitter.com/krisjusiak/status/1187387616980635649
  - C++20: Folding over inline-assembly in constexpr

But wait…

There is NO heterogeneous computing in C++ yet !

SYCL: C++ standard for heterogenous computing (I)

- **Queues to direct computational kernels on some devices operating on data buffers**
  - Simple useful abstractions similar to other C++ API (OpenCL, CUDA, C++AMP, HCC, HIP…)

- **System-wide modern pure C++ class-based DSEL**
  - Pure: 0 extension
  - “Single-source”: host & kernel code in same language in same translation unit
  - Very powerful to programmatically move code between device & host
  - Host fall-back with device emulation: easily develop and debug applications on the host without a device

- **Khronos Group standard to deliver and experiment ahead for ISO C++**
  - ISO C++ is an elite democracy: only what is working can be accepted nowadays…

SYCL: C++ standard for heterogenous computing (II)

- **SYCL is quite higher-level than OpenCL**
  - OpenCL was inspired by the lower-level CUDA Driver API (non single-source)
    - Most of the programmers use the higher-level CUDA Runtime API (single-source C++)

- **“Improved CUDA”**
  - Open-standard from Khronos Group as answer to CUDA
    - Vendor-independent
    - Pure C++ → no specific compiler to start with on CPU

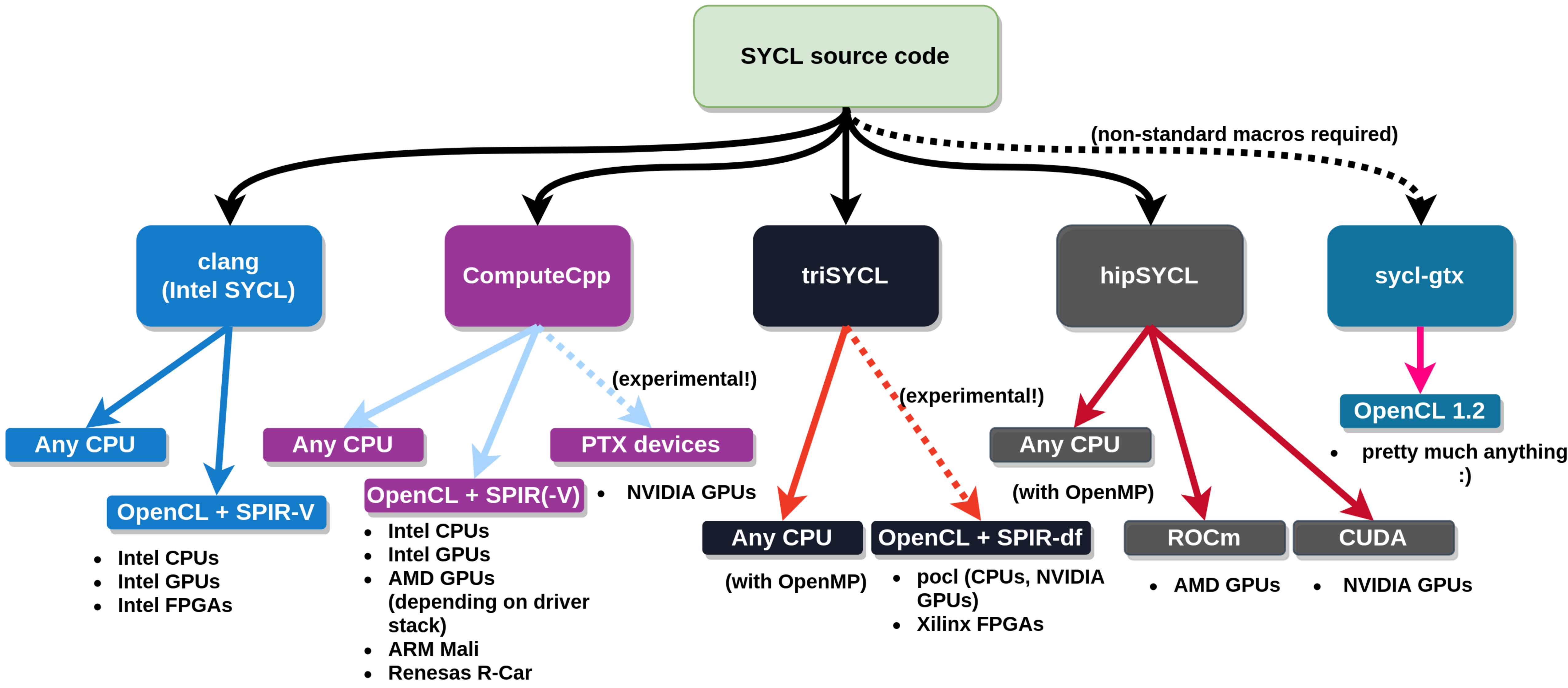
Matrix addition with multiple devices in SYCL 1.2.1

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl::sycl;
// Size of the matrices
constexpr size_t N = 2000;
constexpr size_t M = 3000;
int main() {
 { // By sticking all the SYCL work in a {} block, we ensure
  // all SYCL tasks must complete before exiting the block

  // The queues direct work to the devices
  queue q_r5dl { vendor::xilinx::arm_R5_dual_lockstep {} };
  queue q_fpga { vendor::xilinx::fpga_selector {} };
  queue q_gpu { gpu_selector {} };
  // Create some 2D buffers of float for our matrices
  buffer<double, 2> a{{ N, M }};
  buffer<double, 2> b{{ N, M }};
  buffer<double, 2> c{{ N, M }};
  // Launch a first asynchronous kernel to initialize a
  q_r5dl.submit([&](auto &cgh) {
    // The kernel write a, so get a write accessor on it
    auto A = a.get_access<access::mode::write>(cgh);

    // Enqueue parallel kernel on a N*M 2D iteration space
    cgh.parallel_for<class init_a>({ N, M },
                           [=] (auto index) {
                             A[index] = index[0]*2 + index[1];
                           });
   });
```

```
 // Launch an asynchronous kernel to initialize b
 q_fpga.submit([&](auto &cgh) {
   // The kernel write b, so get a write accessor on it
   auto B = b.get_access<access::mode::write>(cgh);
   // Enqueue a parallel kernel on a N*M 2D iteration space
   cgh.parallel_for<class init_b>({ N, M },
                       [=] (auto index) {
                         B[index] = index[0]*2014 + index[1]*42;
                       });
  });
 // Launch asynchronous kernel to compute matrix addition c = a + b
 q_gpu.submit([&](auto &cgh) {
   // In the kernel a and b are read, but c is written
   auto A = a.get_access<access::mode::read>(cgh);
   auto B = b.get_access<access::mode::read>(cgh);
   auto C = c.get_access<access::mode::write>(cgh);

   // Enqueue a parallel kernel on a N*M 2D iteration space
   cgh.parallel_for<class matrix_add>({ N, M },
                             [=] (auto index) {
                               C[index] = A[index] + B[index]
                             });
  });
 auto C = c.get_access<access::mode::read>();
 std::cout << std::endl << "Result:" << std::endl;
 for (size_t i = 0; i < N; i++)
  for (size_t j = 0; j < M; j++)
   // Compare the result to the analytic value
   if (C[i][j] != i*(2 + 2014) + j*(1 + 42)) {
    std::cout << "Wrong value " << C[i][j] << " on element "
              << i << ' ' << j << std::endl;
    exit(-1);
   }
}
std::cout << "Good computation!" << std::endl;
return 0;
}
```

XILINX

Current ecosystem



Aksel Alpay 2019/11/15 https://raw.githubusercontent.com/illuhad/hipSYCL/master/doc/img/sycl-targets.png

SYCL (short) story

> SYCL started as Higher-Level Model for OpenCL around 2012 with Codeplay & Qualcomm
  - Followed CLU from Intel
> https://github.com/triSYCL/triSYCL
  - Started in 2014 at AMD to understand/validate the specification & experiment/research/…
  - Now mainly backed by Xilinx, with various other contributions (RedHat, Intel…)
> ComputeCpp proprietary implementation by Codeplay, with Community edition for free
  - Only implementation passing the Conformance Test Suite (2018)
  - Good focus on ADAS & Safety-Critical
> First real usable specification: SYCL 1.2.1 2017/12
> Intel SYCL open-source implementation to be upstreamed to Clang/LLVM 2019/01/11
> Open-source + open standard → bootstrap the ecosystem
  - Open-source specification & Conformance Test Suite since IWOCL/DHPCC++ 2019/05/13
> SYCL independent WG from OpenCL since 2019/09/20!
  - OpenCL WG too GPU-centric (mostly graphics vendors…)
> Old SYCL 1.2.1 still based on OpenCL
  - Already targeting PTX CUDA, Vulkan (Android), accelerators, etc. with extensions today
> Future SYCL 2020…
  - Generic back-end (including OpenCL & SPIR-V)
  - Simplifications
  - New features (Unified Shared Memory, FPGA pipes, mixing implementations…)
> Aurora supercomputer @ ANL, 2021: Cray Shasta, Intel Xeon, Intel Xe GPU, SYCL (aka OneAPI DPC++)

Active SYCL committee members

> AMD

> Argone National Lab/University Chicago (Kokkos on top of SYCL)

> Codeplay (ComputeCpp & sycl-gtx)

> Heidelberg Universität (hipSYCL)

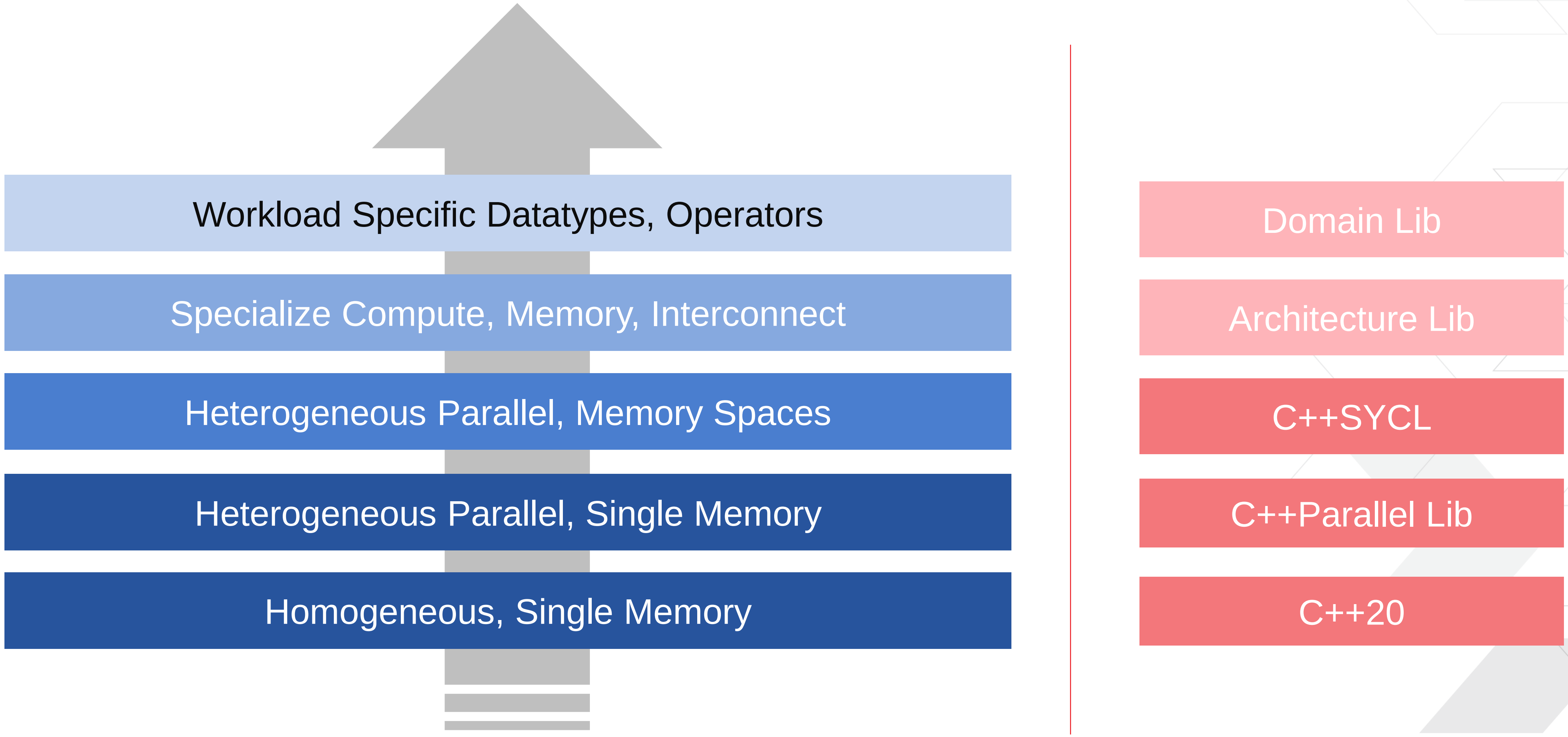
> Intel (Upstream SYCL)

> StreamHPC

> Xilinx (triSYCL)

> Please tell your friends! ☺

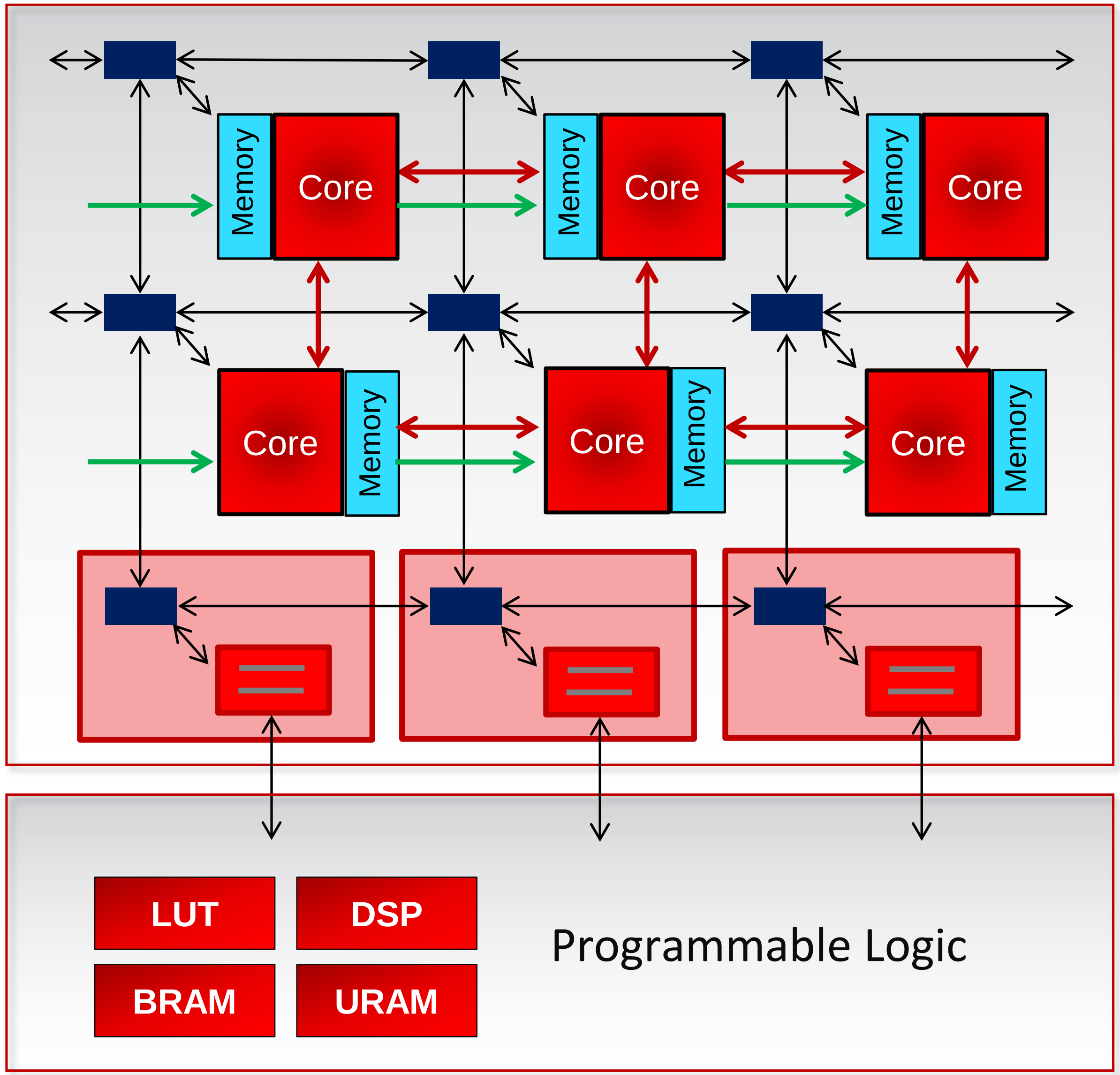
Refinement levels with plain C++ libraries


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 Committee evolution

- **Deliver specification with a train model like ISO C++**
  - Specifications on a regular basis
- **Evolve towards a more open organization like ISO C++**
- **Public extensions to SYCL https://github.com/KhronosGroup/SYCL-Shared**
  - Other extensions per implementation
    - https://github.com/codeplaysoftware/standards-proposals
    - https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions
    - https://github.com/illuhad/hipSYCL/blob/master/doc/extensions.md
    - https://github.com/triSYCL/triSYCL/issues?q=is%3Aissue+label%3Aextension
  - Get feedback from implementers and users
  - Can become part of the standard if SYCL members agree
  - Some features can be optional to allow maximum performance per platform
    - C++ allows feature testing & adaptability through metaprogramming

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



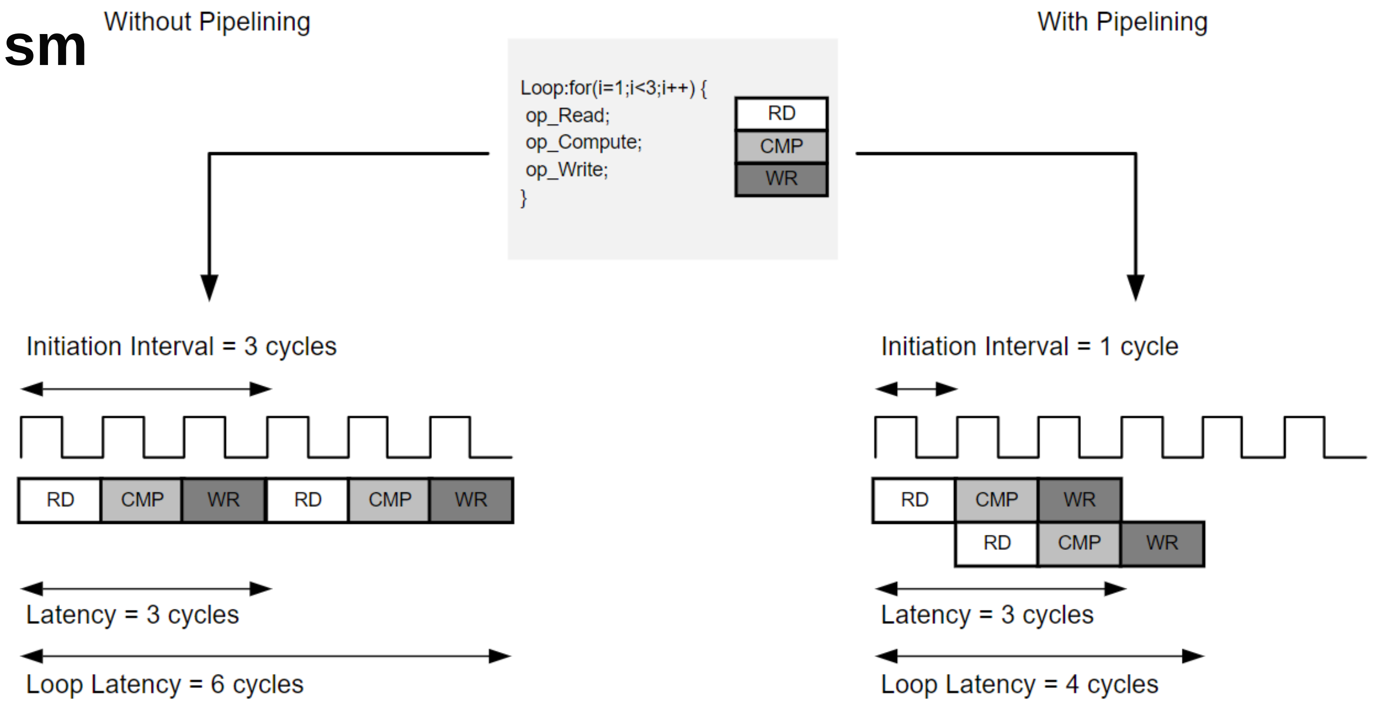
- **Scalar Engines:**
  - ARM dual-core Cortex-A72 (Host)
  - ARM dual-core Cortex-R5
- **Programmable Logic (FPGA)**
- **Coarse-Grain Reconfigurable Array**
  - 400 AI Engine (AIE) cores (tiles)
  - Each AIE Tile contains:
    - 32-bit Scalar RISC processor
    - 512-bit VLIW SIMD vector processor
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

But wait…

There is nothing for FPGA or AIE in SYCL!!! ☹

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

Decorating code for 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 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

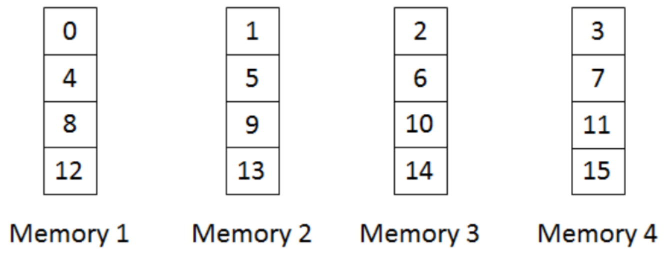


Figure 7-1: **Physical Layout of Buffer After Cyclic Partitioning**

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

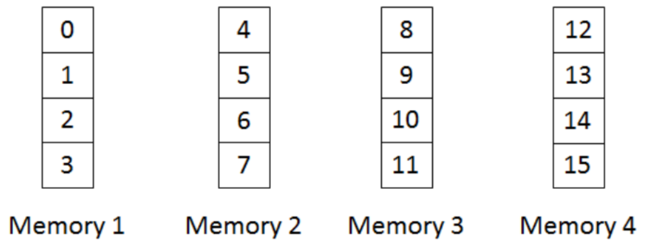


Figure 7-2: **Physical Layout of Buffer After Block Partitioning**

Complete Partitioning

– Extreme distribution

– Extreme bandwidth

– Low latency

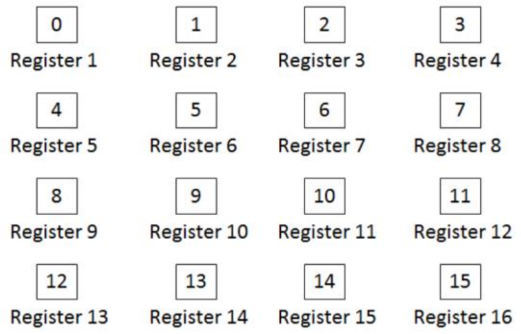


Figure 7-3: **Physical Layout of Buffer After Complete Partitioning**

partition_array class in triSYCL use case

```
// 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;
  …
});
```

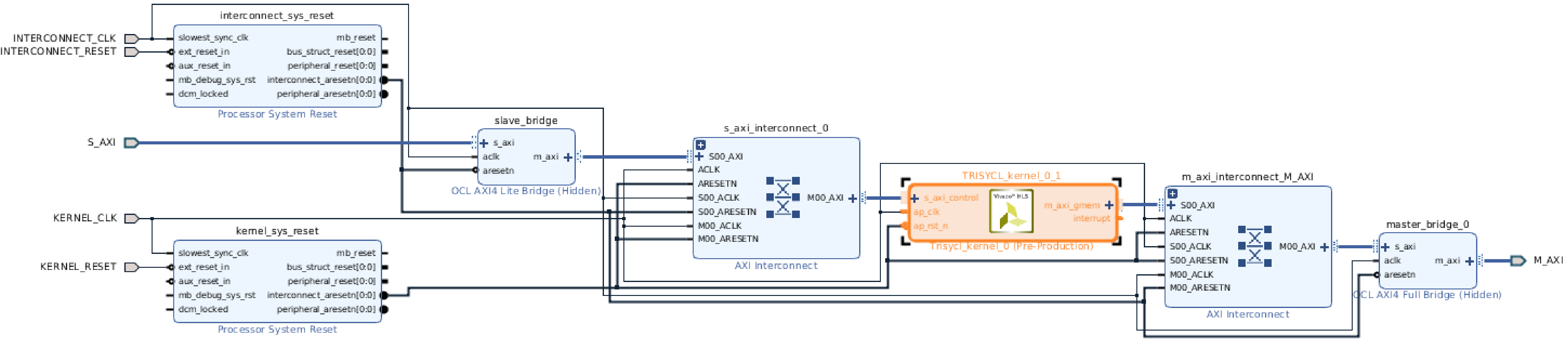
```
int A[MAX_DIM * MAX_DIM];
int B[MAX_DIM * MAX_DIM];
int C[MAX_DIM * MAX_DIM];
//Cyclic Partition for A as matrix multiplication needs row-wise parallel access
#pragma HLS ARRAY_PARTITION variable=A dim=1 cyclic factor=64
//Block Partition for B as matrix multiplication needs column-wise parallel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

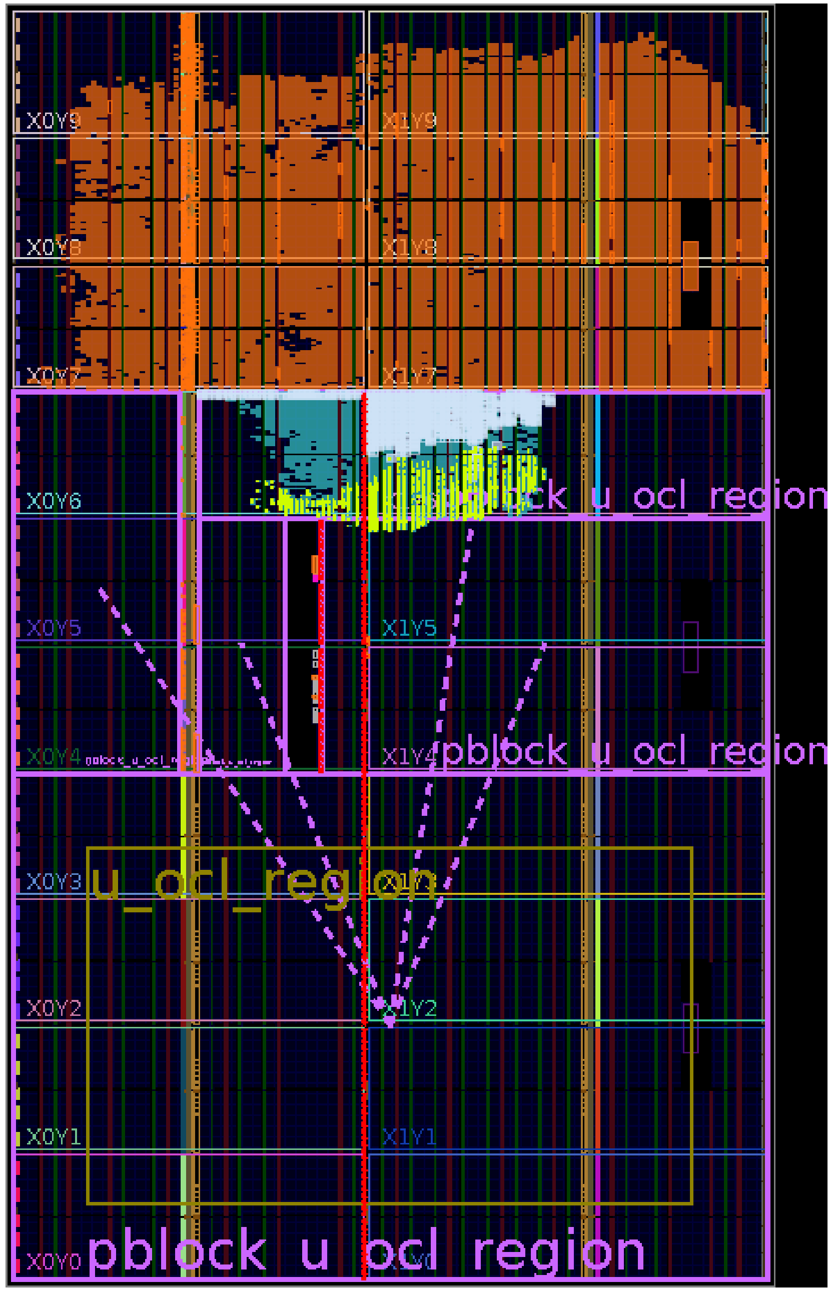
Xilinx OpenCL C

After Xilinx SDx xocc ingestion…



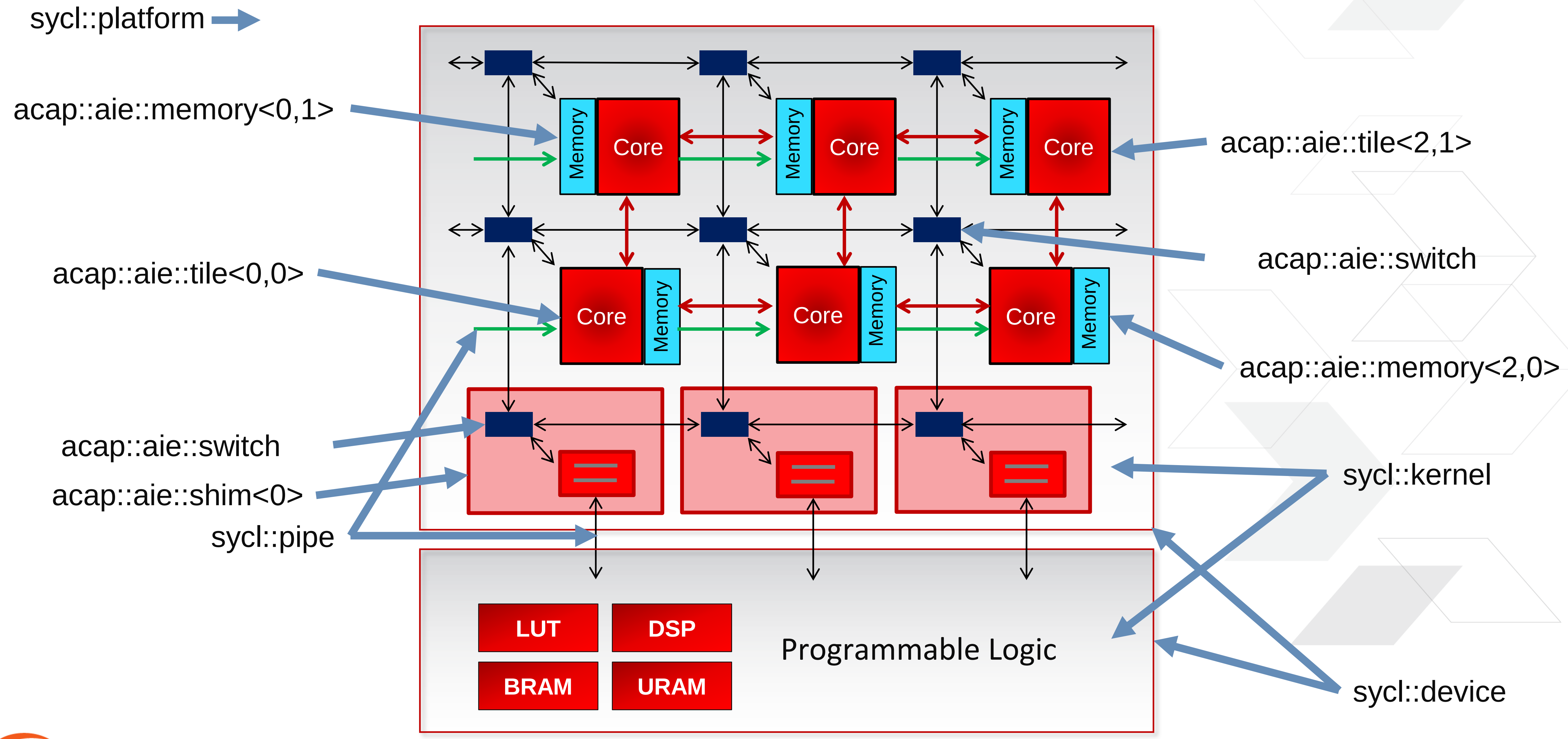
XILINX

After Xilinx SDx xocc ingestion… FPGA layout!



XILINX

ACAP++: SYCL abstractions templated by 2D coordinates



XILINX.

Typical ACAP design patterns

All the tiles do the same

Hello World in C++ ACAP

```cpp
#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>();
}
```

XILINX.

All the tiles do the same

with their own memory

Mandlebrot set computation (cryptocurrency POW too…)

> Start simple: no neighborhood communication ☺

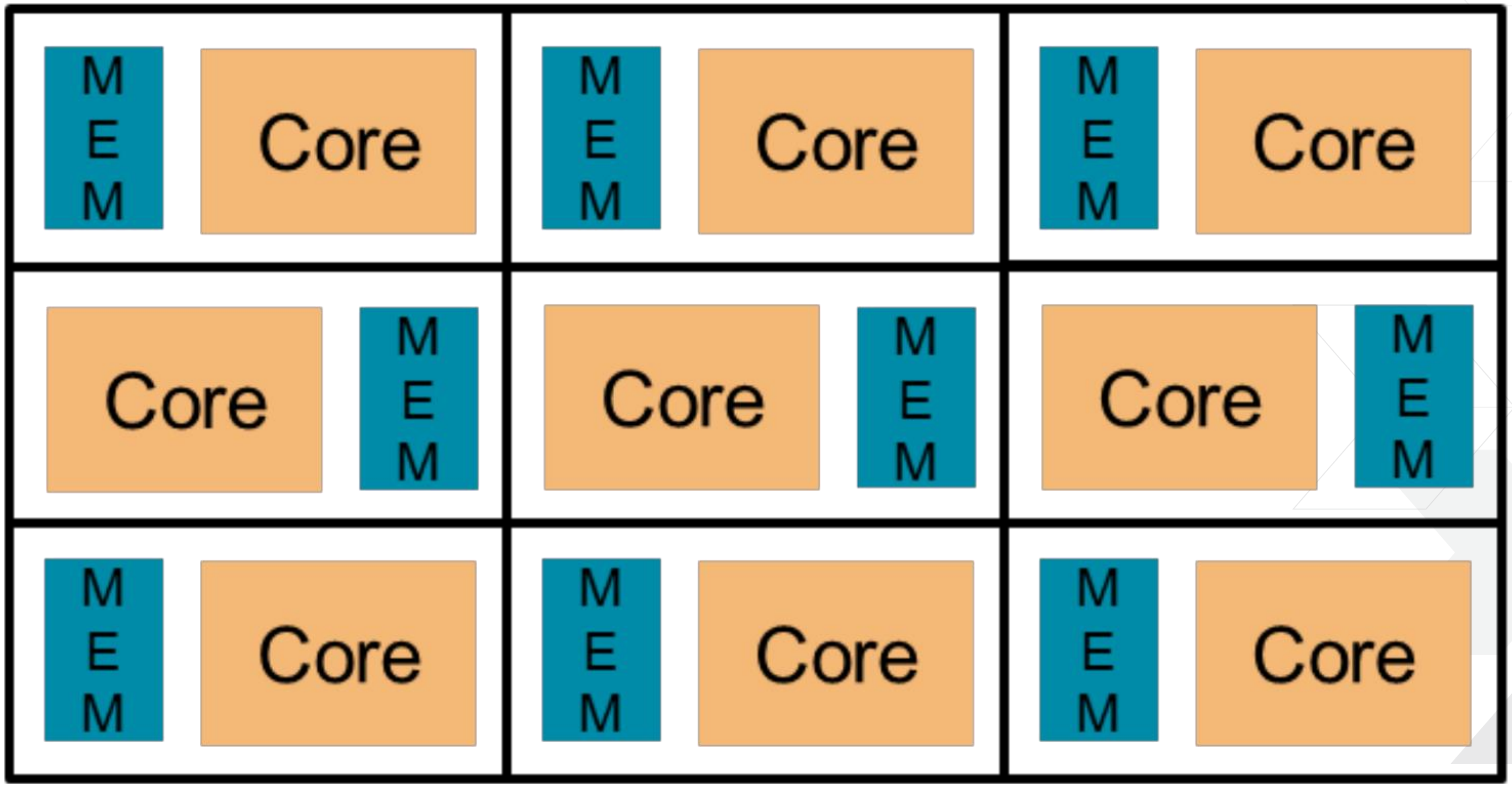
```
using namespace sycl::vendor::xilinx;

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();
}
```

Communicating tiles

through memories

2D memory modules: shared by 4 neighbors



Stencil computation: 2D shallow water wave propagation

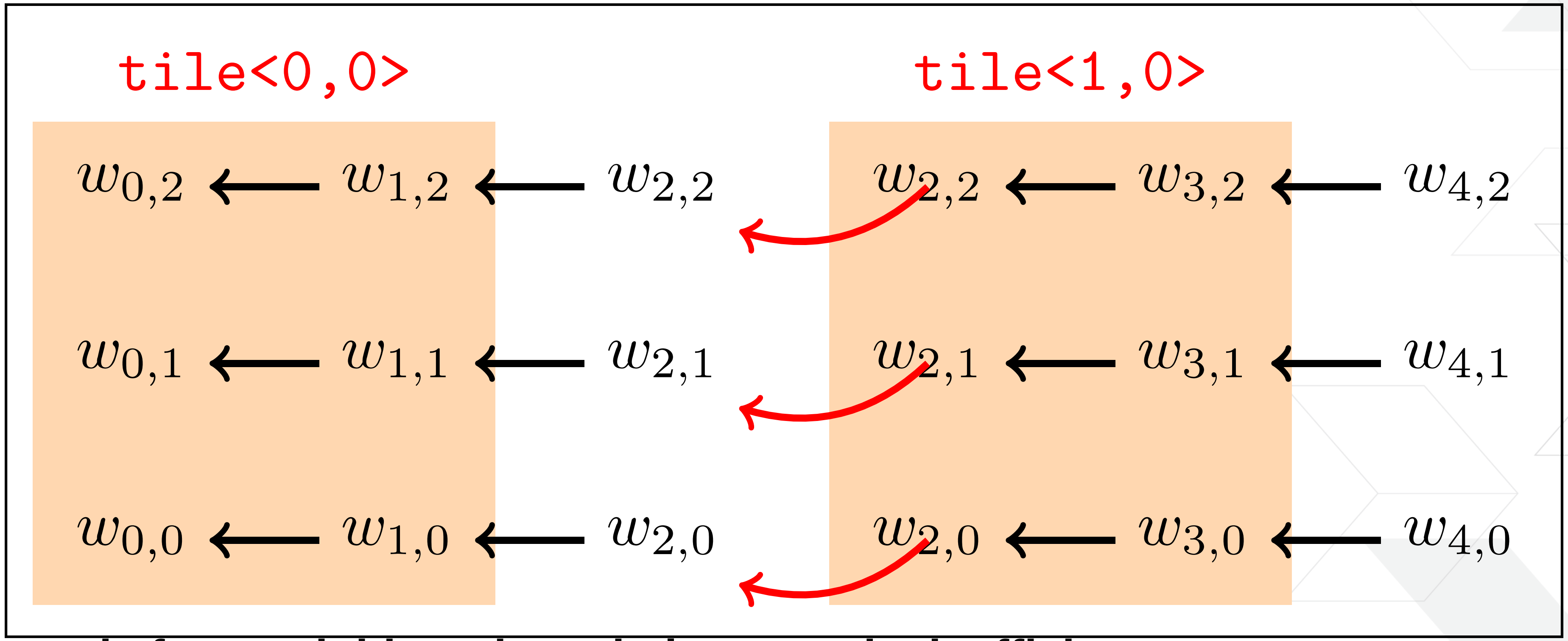
- **2D tiling of data & computations**
- **Typical PDE application from HPC, usually using MPI & OpenMP**
- **Use simple 1-order differentiation schema** $\frac{df}{dx} \approx \frac{f_{x+1}-f_x}{dx}$
- **Only communication with 4 direct neighbors**
  - Impossible to do on GPU without global memory transfer
    - 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

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

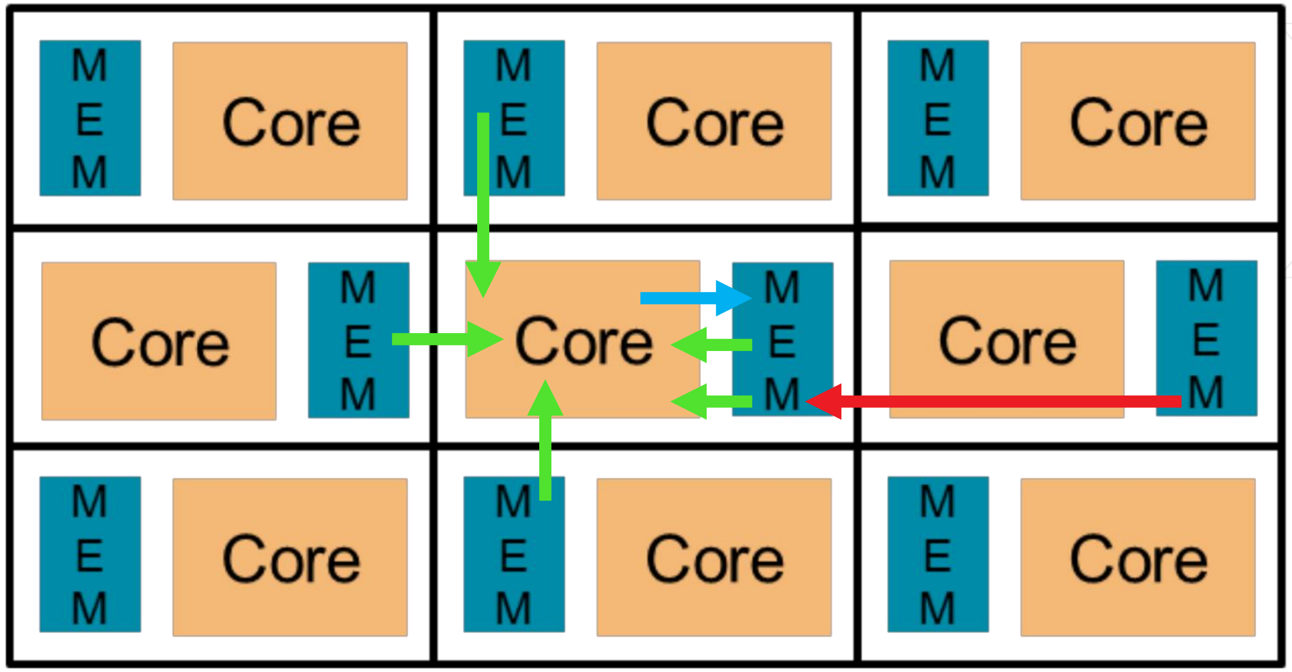


tile<0,0>
tile<1,0>
$w_{0,2}$ $w_{1,2}$ $w_{2,2}$ $w_{2,2}$ $w_{3,2}$ $w_{4,2}$
$w_{0,1}$ $w_{1,1}$ $w_{2,1}$ $w_{2,1}$ $w_{3,1}$ $w_{4,1}$
$w_{0,0}$ $w_{1,0}$ $w_{2,0}$ $w_{2,0}$ $w_{3,0}$ $w_{4,0}$

- **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**

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];
  }
}
```

Neighbor memory access… beware of race conditions

```
/** Default program. This will be tile(0,0) only.

    The tile(0,0) uses memory module on the right */
template <typename AIE, int X, int Y>
struct program : acap::me::tile<AIE, X, Y> {
[…]
    auto &m = t::mem_right();
    // Unprotected access in respect to tile(1,0)
    m.v[0] = 42;
[…]
};
```

```
/** Specialize for the tile(1,0).

    The tile(1,0) uses memory module on the left */
template <typename AIE>
struct program<AIE, 1, 0> : acap::me::tile<AIE, 1, 0> {
[…]
    auto &m = t::mem_left();
    // Unprotected access in respect to tile(0,0)
    m.v[0] = 314;
[…]
}
```

> **Helgrind, Clang/GCC ThreadSanitizer… can catch this in CPU mode! ☺**

```
WARNING: ThreadSanitizer: data race (pid=14351)
  Write of size 4 at 0x7ffeb2762be4 by thread T2:
    #0 program<cl::sycl::vendor::xilinx::acap::aie::array<cl::sycl::vendor::xilinx::acap::aie::layout::size<2, 1>, program,
memory>, 1, 0>::run() acap/race_condition_uninitialized.cpp:48 (race_condition_uninitialized+0xfd7c)
 Previous write of size 4 at 0x7ffeb2762be4 by thread T1:
    #0 program<cl::sycl::vendor::xilinx::acap::aie::array<cl::sycl::vendor::xilinx::acap::aie::layout::size<2, 1>, program,
memory>, 0, 0>::run() acap/race_condition_uninitialized.cpp:29 (race_condition_uninitialized+0xfb48)
Tile (1,0) receives from left neighbour: 1
Tile (1,0) read overflow: -6.25985e+18
Tile (1,0) read uninitialized: 0
Total size of the tiles: 32 bytes.
ThreadSanitizer: reported 1 warnings
```

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…

Some other SYCL extensions/proposals

- Back-end generalization (Codeplay)
- On-chip memory (Codeplay)
- Built-in kernels easier to use (Codeplay)
- Feature-based & optionality
  - Can target simpler hardware
- OneAPI from Intel with DataParallelC++ based on SYCL
  - Simpler porting from CUDA, skipping accessors
    - Unified Shared Memory
    - Synchronous queue
  - Lot of FPGA extensions from Intel
- Combining different implementations (Xilinx)
- …

Celerity: High-level C++ for Accelerator Clusters

> https://celerity.github.io

> **Scale-out SYCL concepts at the data-center level**

>> Based on C++ + SYCL + MPI

```
queue.submit([=](celerity::handler& cgh) {
    auto r_input = input_buf.get_access<cl::sycl::access::mode::read>(cgh, celerity::access::neighborhood<2>(1, 1));
    auto dw_edge = edge_buf.get_access<cl::sycl::access::mode::discard_write>(cgh, celerity::access::one_to_one<2>());
    cgh.parallel_for<class MyEdgeDetectionKernel>(cl::sycl::range<2>(img_height - 1, img_width - 1), cl::sycl::id<2>(1, 1),
                                                                          [=](cl::sycl::item<2> item) {
          int sum = r_input[{item[0] + 1, item[1]}] + r_input[{item[0] - 1, item[1]}]
                  + r_input[{item[0], item[1] + 1}] + r_input[{item[0], item[1] - 1}];
          dw_edge[item] = 255 - std::max(0, sum - (4 * r_input[item]));
        }
    );
});
queue.with_master_access([=](celerity::handler& cgh) {
    auto out = edge_buf.get_access<cl::sycl::access::mode::read>(cgh, edge_buf.get_range());
    cgh.run([=]() {
        stbi_write_png("result.png", img_width, img_height, 1, out.get_pointer(), 0);
    });
});
```

> **Other C++ HPC frameworks like HPX or Kokkos provide similar concepts on top of SYCL but with a non-SYCL syntax**

Simplify porting code from CUDA

- **SYCL is higher-level than CUDA to ease programmer life ☺**
  - Buffers for abstract storage, accessors to express dependencies, compute/communication overlap...
- **But when porting CUDA code: dealing with raw device pointers, explicit kernel launches without any dependencies... ☹**
  - Feedback from porting Eigen & TensorFlow to SYCL...
  - → Paradox: cumbersome to fit in standard SYCL model... ☹
- https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions/OrderedQueue **Ordered queue, in-order execution, like CUDA default stream**
- https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/USM **Unified Shared Memory like in OpenMP 5, various level of sharing according to hardware**

```
sycl::ordered_queue q;
auto dev = q.get_device();
auto ctxt = q.get_context();
float* a = static_cast<float*>(sycl::malloc_shared(10*sizeof(float), dev, ctxt));
float* b = static_cast<float*>(sycl::malloc_shared(10*sizeof(float), dev, ctxt));
float* c = static_cast<float*>(sycl::malloc_shared(10*sizeof(float), dev, ctxt));
q.parallel_for<class vec_add>(range<1> {10}, [=](id<1> i) { c[i] = a[i] + b[i]; });
```

Documentation & resources

- **https://www.khronos.org/sycl**
  - https://www.khronos.org/files/sycl/sycl-121-reference-card.pdf 8 pages of SYCL &
- **SYCL-related community resource web site http://sycl.tech**
- **https://tech.io/playgrounds/48226/introduction-to-sycl/introduction-to-sycl-2**
  - playground on “Introduction to SYCL” with on-line code execution
- **Code samples from Codeplay**
  - https://github.com/codeplaysoftware/computecpp-sdk/tree/master/samples
- **Parallel Research Kernels implemented with different frameworks, including SYCL**
  - Useful to compare or translate between different programming models https://github.com/ParRes/Kernels
- **Intel open-source to be up-streamed**
  - https://github.com/intel/llvm/blob/sycl/sycl/doc/GetStartedWithSYCLCompiler.md
- **Open-source (mainly Xilinx)**
  - https://github.com/triSYCL/triSYCL
- **Open-source fusion triSYCL+Intel to Xilinx FPGA**
  - https://github.com/triSYCL/sycl

Conclusion

- **7nm hardware needs 7nm software → C++20 !!!**
  - Python productivity and C++ hardware efficiency ☺
- **SYCL: DSEL for system-wide heterogeneous programming: CGRA+FPGA+CPU+GPU+…**
  - Focus on the **whole** system: SYstem-wide Compute Language
  - Single-source → cross-optimization & metaprogramming the whole infrastructure
    - Co-design, emulation & debugability out-of-the box !
  - C++ device libraries (ACAP++…) allow exposing different levels of details
- **Open-source & open standard**
  - No user lock-in!
  - Possible dream today with open-source Clang/LLVM and SYCL up-streamed! ☺
    - Software today too complex for a single company
  - Please participate in the standards to have a real impact! ☺
  - Several implementations with various back-ends
    - Interoperability with other ecosystems (OpenCL, CUDA, Vulkan, proprietary...)
    - MLIR “single-source” accelerator dialect
- **Enable adaptable higher-level libraries (Kokkos/HPX/Raja/Legion/PGAS/…)**