XILINX

triSYCL brings C++20 to Xilinx FPGA & CGRA with Vitis

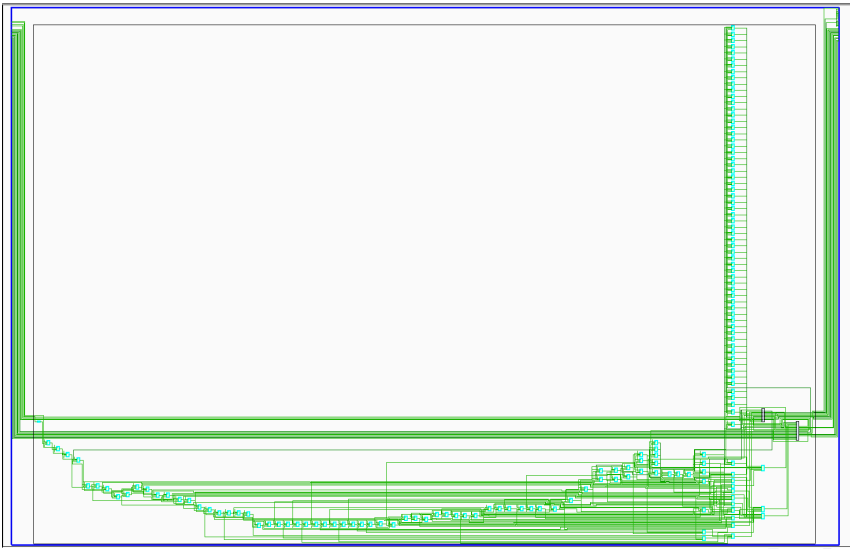
Ronan Keryell (rkeryell@xilinx.com)
Xilinx Research Labs, San José, California & SYCL specification editor
2021/12/07 @ Khronos SYCL Webinar

Outline

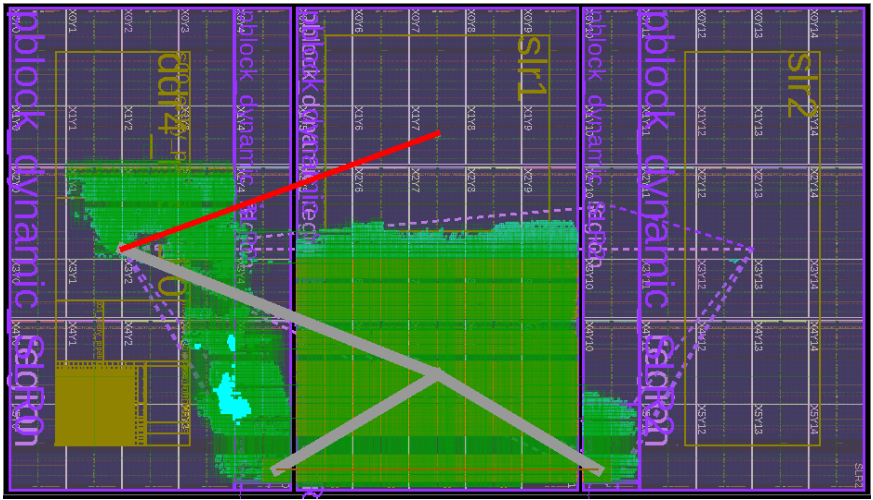
Compute the universal answer on FPGA

```
// The universal answer from an heterogeneous world
#include <sycl/sycl.hpp>
#include <iostream>
#include "../utilities/device_selectors.hpp"
int main() {
  // Allocate 1 int of 1D abstract memory
  sycl::buffer<int> answer { 1 };
  // Create a queue on Xilinx FPGA
  sycl::queue q { selector_defines::CompiledForDeviceSelector {} };
  std::cout << "Queue Device: "
            << q.get_device().get_info<sycl::info::device::name>() << std::endl;
  std::cout << "Queue Device Vendor: "
            << q.get_device().get_info<sycl::info::device::vendor>() << std::endl;
  // Submit a kernel on the FPGA
  q.submit([&] (sycl::handler &cgh) {
      // Get a write-only access to the buffer
      sycl::accessor a { answer, cgh, sycl::write_only };
      // The computation on the accelerator
      cgh.single_task<class forty_two>([=] { a[0] = 42; });
    });
  // Verify the result
  sycl::host_accessor ans { answer, sycl::read_only };
  std::cout << "The universal answer to the question is " << ans[0] << std::endl;
```

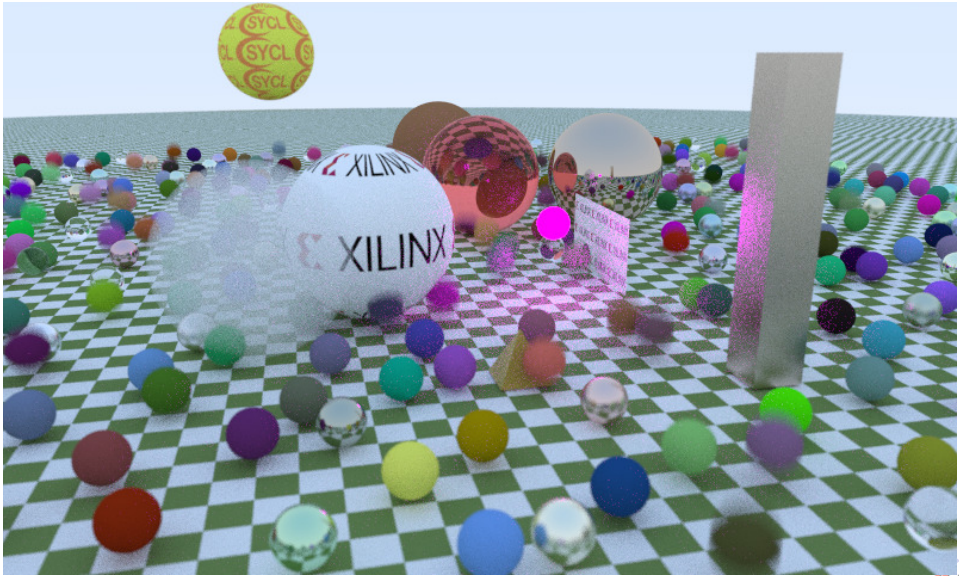
Schematics on Xilinx Alveo U200 FPGA PCIe card



Layout on Xilinx Alveo U200 FPGA PCIe card



Enable new application domains on FPGA: path tracing!



Path tracing to push the limits of SYCL, HLS & XRT

- Started as a joke inside the Khronos committee
  - An FPGA is adaptable and can do anything, right? ☺
- Experiment direct brute force implementation
  - Used as a big pipe-cleaner application outside of usual FPGA ML & vision applications
  - Overcome SYCL limitations inside kernels
    - Different C++ coding style (no function pointers, no dynamic polymorphism...)
    - Improve our SYCL workflow
  - Triggered some Vitis HLS bugs
    - Math libraries with OpenCL, LLVM pass ordering issue,...
    - Solved only in Vitis 2021.1 or 2021.2 or...
  - Triggered some Xilinx XRT bugs
    - OpenCL host API bugs, unimplemented features...
    - Fixes pushed upstream thanks to open-source!
- Use only a small part of FPGA and no optimization for now
  - Generated hardware match the written C++ code
  - Is it possible to have competitive path tracer implementation? ☺

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

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

- Dynamic polymorphism usually not handled by accelerators ≈ function pointers

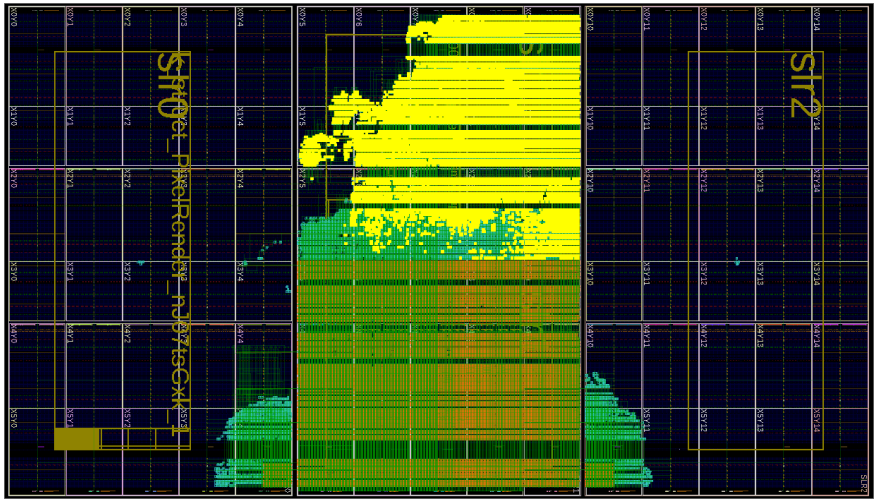
  https://raytracing.github.io/books/RayTracingInOneWeekend.html#surfacenormalsandmultipleobjects/anabstractionforhittableobjects

- Vitis HLS does some trivial devirtualization when there is only 1 class in use
- But C++17 std::variant allows other way to handle multiple dispatch
  - An FPGA can dispatch a std::visit in $\mathcal{O}(1)$ ☺

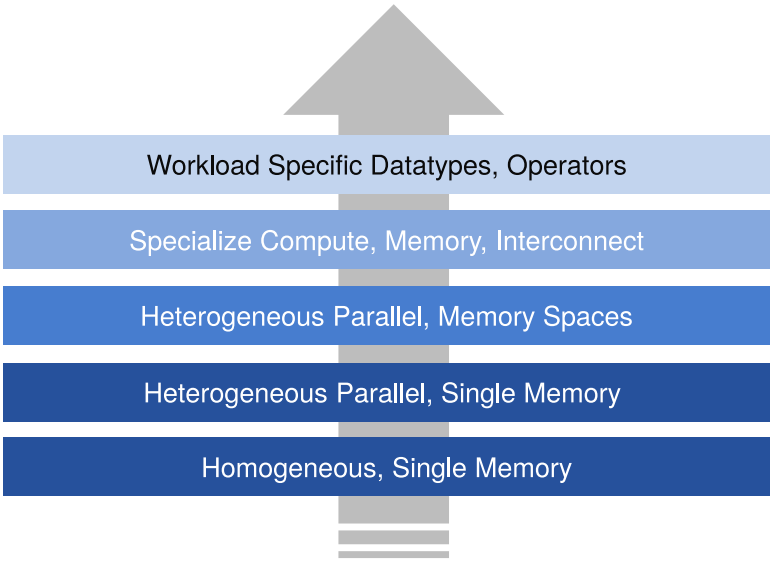
```
struct hittable {
  virtual bool hit(const ray& r, double t_min, double t_max,
                   hit_record& rec) const = 0;
};
struct sphere : hittable {
  virtual bool hit(const ray& r, double t_min, double t_max,
                   hit_record& rec) const override { ... };
};
struct rectangle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

Layout of path tracer on Xilinx Alveo U200 FPGA PCIe card



SYCL can provide refinement levels with C++ abstractions



Domain Lib

Architecture Lib

C++SYCL

C++Parallel Lib

C++20

Handling several external memory banks

- An FPGA can have various external memories & banks: DDR, HBM, QDR SRAM...

```
#include <sycl/sycl.hpp>
int main() {
  sycl::buffer<int> buf { 4 };
  sycl::queue {}.submit([&](auto &cgh) {
    sycl::accessor a { buf, cgh, sycl::write_only,
                       { sycl::ext::xilinx::ddr_bank<3> }};
    cgh.parallel_for(buf.size(), [=](int i) { a[i] = i; });
  });
}
```

Arbitrary precision arithmetic

- Surf on Clang/LLVM implementation of ISO WG14 C23
  - `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2763.pdf` *Adding a Fundamental Type for N-bit integers*
  - `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2775.pdf` *Literal suffixes for bit-precise integer*

```
_BitInt(3) a, b, c;
c = a + b;
```

- Directly translated into minimalist FPGA hardware with 3-bit storage and operators!!!
- Wrapped into C++ `ap_int<N>` for type safety and usual C++ goodies

Build pure C++ user libraries for fancy arithmetic

github.com/yuguen/hint/blob/ExtIntImpl/include/backend/extint_impl.ipp
gitlab.inria.fr/lforget/marto

```
constexpr unsigned int WE = 7; // 7 bit exponents
constexpr unsigned int WF = 14; // 14 bit mantissa
constexpr unsigned int Width = WE + WF + 1; // 22 bit floating-point
using IEEE_t = IEEENumber<WE, WF, hint::ExtIntWrapper>;
void print_custom_ieee(IEEE_t const op) {
  std::cout << hint::to_string(op.getSign()) << " " <<
      hint::to_string(op.getExponent()) << " " <<
      hint::to_string(op.getFractionnalPart()) << std::endl;
  if constexpr(WE <= 11 && WF <= 52) {
    auto frac = static_cast<uint64_t>(op.getFractionnalPart().unravel());
    auto exp = static_cast<int16_t>(op.getExponent().unravel());
    auto sign = static_cast<bool>(op.getSign().unravel());
    bool isNormal = (exp != 0);
    if (isNormal) frac |= (uint64_t{1} << WF);
    double val = frac;
    exp -= IEEEDim<WE, WF>::BIAS;
    if (!isNormal) exp++;
    val = ldexp(val, exp - WF);
    if (sign) val *= -1.0;
    std::cout << "value: " << val << std::endl;
  }
}
IEEE_t operator+(IEEE_t const op0, IEEE_t const op1) {
  return ieee_add_sub_impl(op0, op1);
}
int main(int argc, char** argv) {
  if (argc != 3) {
    std::cerr << "Usage : ieee_adder op0_repr op1_repr" << std::endl;
    return -1;
  }
  IEEE_t a {{static_cast<unsigned _ExtInt(Width)>(std::atoi(argv[1]))}};
  IEEE_t b {{static_cast<unsigned _ExtInt(Width)>(std::atoi(argv[2]))}};
  std::cout << "a: ";
  print_custom_ieee(a);
  std::cout << "b: ";
  print_custom_ieee(b);
  sycl::queue queue;
  sycl::range<1> dim{1};
  sycl::buffer<IEEE_t, 1> in0(dim), in1(dim), out(dim);
  {
    auto ain0 = in0.get_access<sycl::access::mode::discard_write>();
    auto ain1 = in1.get_access<sycl::access::mode::discard_write>();
    ain0[0] = a;
    ain1[0] = b;
  }
  queue.submit([&](sycl::handler& cgh){
    auto ain0 = in0.get_access<sycl::access::mode::read>(cgh);
    auto ain1 = in1.get_access<sycl::access::mode::read>(cgh);
    auto aout = out.get_access<sycl::access::mode::discard_write>(cgh);
    cgh.single_task([=]{ aout[0] = ain0[0] + ain1[0]; });
  });
  {
    auto aout = out.get_access<sycl::access::mode::read>();
    cout << "a + b:";
    print_custom_ieee(aout[0]);
  }
  return 0;
}
```

Arbitrary Vitis options on kernels

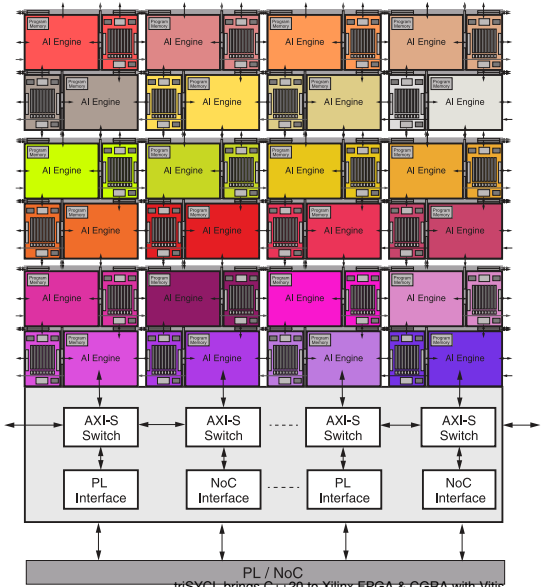
- Traditional FPGA flow requires a lot of configuration files ☹
- How to specify kernel-specific options in a single-source world?
- Use UDL (user-defined literals) to decorate the kernels! ☺

```
q.submit([&](sycl::handler &cgh) {
  sycl::accessor a {buf, cgh, sycl::write_only, sycl::no_init};
  cgh.single_task("--kernel_frequency 400 --optimize 2"_vitis_option([=] {
    for (int i = 0; i != size; ++i)
      a[i] = i;
    }));
});
```

- Allows metaprogramming Vitis options from SYCL
- Just ignored when not targeting Xilinx FPGA

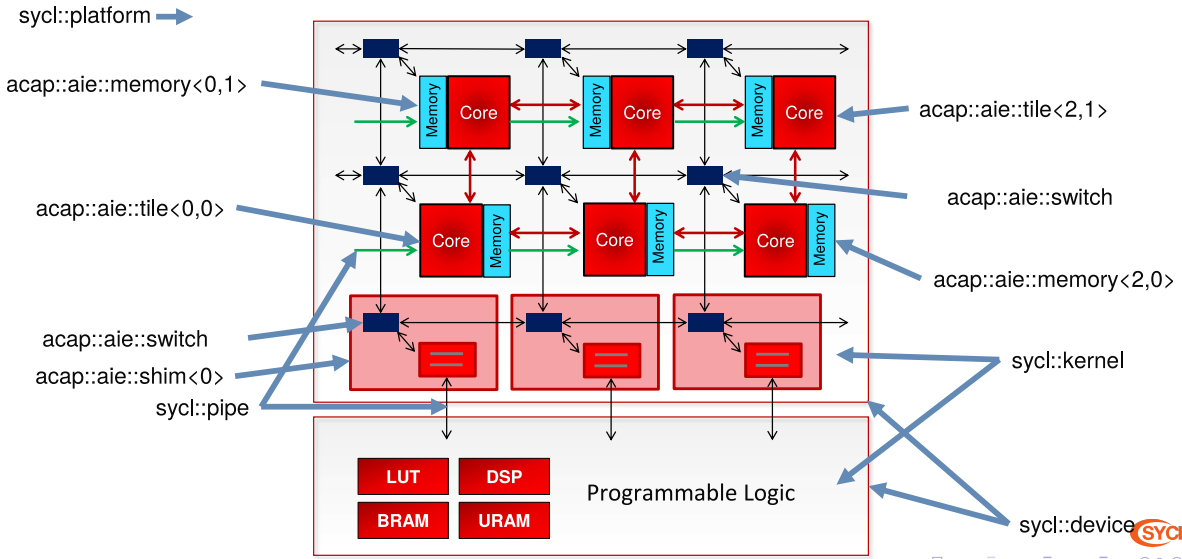
Outline

Recap of Versal ACAP's CGRA: AI Engine Array



- 400 AI Engine (AIE) cores in a tile array
- Each AIE tile:
  - 32-bit scalar + 512-bit SIMD VLIW processor
  - 32 KiB RAM (total of 12.5 MiB of L1 distributed across the CGRA)
  - Each tile has direct access to its neighbors memory (128 KiB shared)
  - 16 KiB program memory
- The AIE Tiles have an alternating checkerboard pattern which has some impact on memory accesses from a Tiles perspective
- Possible to transfer memory between AIE Tiles and externally to the rest of the system

Xilinx C++ ACAP: templated 2D SYCL abstractions



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.get_count());
  });
```

```
  //  Submit some work on each tile, which is SYCL sub-device
  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.get_count() + i + 42)
        throw "Bad computation";
  });
}
```

Provide also more collaborative spatial programming model

```
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 north = m.w[j][i + 1] - m.w[j][i];
      // Integrate horizontal speed
      m.u[j][i] += north*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();

  // Transfer first column of u to next memory module to the West
  if constexpr (Y & 1) {
    if constexpr (t::is_memory_module_east()) {
      auto& east = t::mem_east();
      for (int j = 0; j < image_size; ++j)
        m.u[j][image_size - 1] = east.u[j][0];
    }
  }
  if constexpr (!(Y & 1)) {
    if constexpr (t::is_memory_module_west()) {
      auto& west = t::mem_west();
      for (int j = 0; j < image_size; ++j)
        west.u[j][image_size - 1] = m.u[j][0];
    }
  }

  if constexpr (t::is_memory_module_south()) {
    auto& below = t::mem_south();
    for (int i = 0; i < image_size; ++i)
      below.v[image_size - 1][i] = m.v[0][i];
  }

  t::barrier();

  for (int j = 1; j < image_size; ++j)
    for (int i = 1; i < image_size; ++i) {
      // div speed
      auto wp  = (m.u[j][i] - m.u[j][i - 1])
               + (m.v[j][i] - m.v[j - 1][i]);
      wp *= m.side[j][i]*(m.depth[j][i] + m.w[j][i]);
      // Integrate depth
      m.w[j][i] += wp;
      // Add some dissipation for the damping
      m.w[j][i] *= damping;
    }

[...]

}
```

Outline

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" across device...
    - 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 (similar to Celerity SYCL for MPI+SYCL)

Implementation

- Resycle Intel oneAPI DPC++ implementation
  - SYCL C++ runtime
    - Reuse various back-ends
  - Rely on OpenCL host API of Xilinx XRT to control FPGA
  - Clang front-end to deal with splitting host/device code
  - LLVM passes to massage device code for various targets (address-spaces...)
- Merge from historical triSYCL
  - Specific runtime to handle Xilinx FPGA decorations
  - LLVM passes to massage IR for Xilinx FPGA
    - Translate C++-generated decorations to Xilinx HLS decorations
    - Rename kernels to be Xilinx HLS-compliant
    - Translate LLVM IR 14 down to LLVM IR 6.x digested by Vitis
    - ...
- Python script to drive Xilinx Vitis `v++` from Clang driver
  - Kernel compiling from LLVM IR (which is an unsupported feature...)
  - Link kernels together

Outline

Conclusion

- Disclaimer: triSYCL is an on-going research project ☺
  - Lot of features are still missing
  - Use Xilinx Vivado & Vitis for any FPGA work non-focused on C++20
- SYCL C++ standard from Khronos Group
  - Pure modern C++ DSEL for simpler heterogeneous computing
  - Single-source & single-language to metaprogram the full architecture across device boundaries
  - It's not magic, it's modern C++! ☺ 3nm devices deserve 3nm C++! ☺
  - Target emulation, debug & co-design on CPU for free
- Open-source + open standards
  - Modern heterogeneous system: several accelerators from various vendors
  - Various implementations with back-ends & interoperability with other ecosystems
  - No user locked-in!
- Even more interesting since AMD announced will of Xilinx acquisition: CPU, GPU, FPGA, CGRA...

Conclusion

- Disclaimer: triSYCL is an on-going research project ☺
  - Lot of features are still missing
  - Use Xilinx Vivado & Vitis for any FPGA work non-focused on C++20
- SYCL C++ standard from Khronos Group
  - Pure modern C++ DSEL for simpler heterogeneous computing
  - Single-source & single-language to metaprogram the full architecture across device boundaries
  - It's not magic, it's modern C++! ☺ 3nm devices deserve 3nm C++! ☺
  - Target emulation, debug & co-design on CPU for free
- Open-source + open standards
  - Modern heterogeneous system: several accelerators from various vendors
  - Various implementations with back-ends & interoperability with other ecosystems
  - No user locked-in!
- Even more interesting since AMD announced will of Xilinx acquisition: CPU, GPU, FPGA, CGRA...

Conclusion

- Disclaimer: triSYCL is an on-going research project ☺
  - Lot of features are still missing
  - Use Xilinx Vivado & Vitis for any FPGA work non-focused on C++20
- SYCL C++ standard from Khronos Group
  - Pure modern C++ DSEL for simpler heterogeneous computing
  - Single-source & single-language to metaprogram the full architecture across device boundaries
  - It's not magic, it's modern C++! ☺ 3nm devices deserve 3nm C++! ☺
  - Target emulation, debug & co-design on CPU for free
- Open-source + open standards
  - Modern heterogeneous system: several accelerators from various vendors
  - Various implementations with back-ends & interoperability with other ecosystems
  - No user locked-in!
- Even more interesting since AMD announced will of Xilinx acquisition: CPU, GPU, FPGA, CGRA...

Conclusion

- Disclaimer: triSYCL is an on-going research project ☺
  - Lot of features are still missing
  - Use Xilinx Vivado & Vitis for any FPGA work non-focused on C++20
- SYCL C++ standard from Khronos Group
  - Pure modern C++ DSEL for simpler heterogeneous computing
  - Single-source & single-language to metaprogram the full architecture across device boundaries
  - It's not magic, it's modern C++! ☺ 3nm devices deserve 3nm C++! ☺
  - Target emulation, debug & co-design on CPU for free
- Open-source + open standards
  - Modern heterogeneous system: several accelerators from various vendors
  - Various implementations with back-ends & interoperability with other ecosystems
  - No user locked-in!
- Even more interesting since AMD announced will of Xilinx acquisition: CPU, GPU, FPGA, CGRA...