XILINX

C++20 on Xilinx FPGA with SYCL for Vitis

Luc Forget, Gauthier Harnisch, Ronan Keryell (rkeryell@xilinx.com),
Ralph Wittig

Xilinx Research Labs, San José, California

2021/10/28 @ CppCon 2021

Power wall & speed of light: the final frontier...

- Current physical limits
  - Power consumption
    - Cannot power-on all the transistors without melting (⇝ *dark silicon*)
    - Accessing memory consumes orders of magnitude more energy than a simple computation
    - Moving data inside a chip costs quite more than a computation
  - Speed of light
    - Accessing memory takes the time of 10^4+ CPU instructions
    - Even moving data across the chip (cache) is slow at 1+ GHz...

Power wall & speed of light: the final frontier...

- Current physical limits
  - Power consumption
    - Cannot power-on all the transistors without melting (⇝ *dark silicon*)
    - Accessing memory consumes orders of magnitude more energy than a simple computation
    - Moving data inside a chip costs quite more than a computation
  - Speed of light
    - Accessing memory takes the time of 10^4+ CPU instructions
    - Even moving data across the chip (cache) is slow at 1+ GHz...

Power wall & speed of light: the final frontier...

- Current physical limits
  - Power consumption
    - Cannot power-on all the transistors without melting (⇝ *dark silicon*)
    - Accessing memory consumes orders of magnitude more energy than a simple computation
    - Moving data inside a chip costs quite more than a computation
  - Speed of light
    - Accessing memory takes the time of 10^4+ CPU instructions
    - Even moving data across the chip (cache) is slow at 1+ GHz...

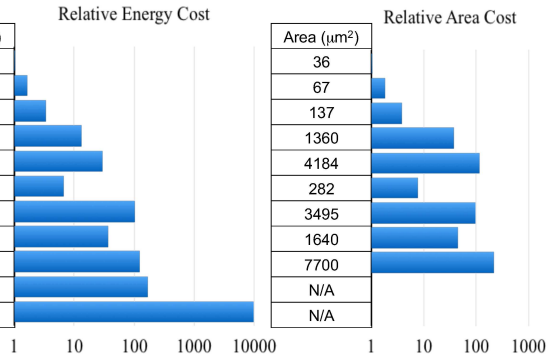
(Very old) 45nm technology characteristics

Tutorial on "*High-Performance Hardware for Machine Learning*", William Dally at NIPS, December 7th, 2015

`https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf`

Cost of Operations

Operation:	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

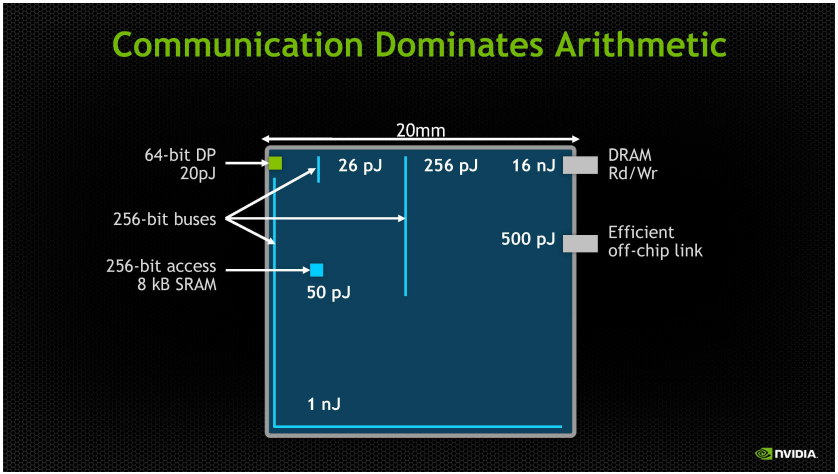


Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

Space-time traveling

"*Challenges for Future Computing Systems*", William J. Dally, January 19, 2015, HiPEAC 2015.

`http://www.cs.colostate.edu/~cs575dl/Sp2015/Lectures/Dally2015.pdf`



Implications of power wall & speed of light

- Change hardware and software
  - Use locality & hierarchy
  - Massive parallelism
- NUMA & distributed memories
  - ⇝ New memory address spaces (local, constant, global, non-coherent...)
  - ⇝ PiM (Processor-in-Memory), Near-Memory Processing (in 3D memory stack controller)...
- Specialize architecture
- Power on-demand only what is required

Implications of power wall & speed of light

- Change hardware and software
  - Use locality & hierarchy
  - Massive parallelism
- NUMA & distributed memories
  - ⇝ New memory address spaces (local, constant, global, non-coherent...)
  - ⇝ PiM (Processor-in-Memory), Near-Memory Processing (in 3D memory stack controller)...
- Specialize architecture
- Power on-demand only what is required

Implications of power wall & speed of light

- Change hardware and software
  - Use locality & hierarchy
  - Massive parallelism
- NUMA & distributed memories
  - ⇝ New memory address spaces (local, constant, global, non-coherent...)
  - ⇝ PiM (Processor-in-Memory), Near-Memory Processing (in 3D memory stack controller)...
- Specialize architecture
- Power on-demand only what is required

Implications of power wall & speed of light

- Change hardware and software
  - Use locality & hierarchy
  - Massive parallelism
- NUMA & distributed memories
  - ⇝ New memory address spaces (local, constant, global, non-coherent...)
  - ⇝ PiM (Processor-in-Memory), Near-Memory Processing (in 3D memory stack controller)...
- Specialize architecture
- Power on-demand only what is required

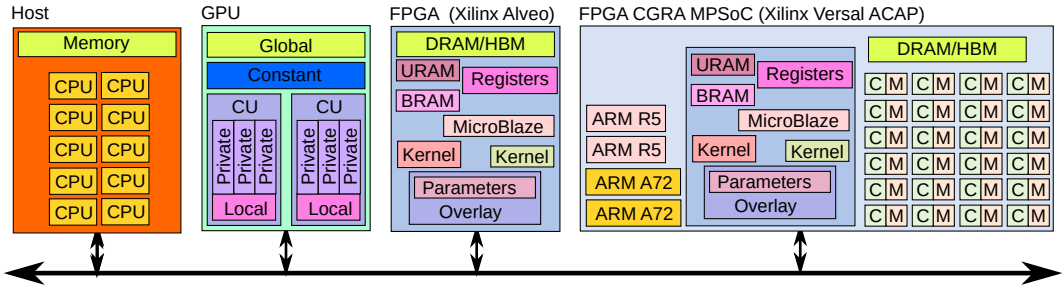
Implications of power wall & speed of light

- Change hardware and software
  - Use locality & hierarchy
  - Massive parallelism
- NUMA & distributed memories
  - ⇝ New memory address spaces (local, constant, global, non-coherent...)
  - ⇝ PiM (Processor-in-Memory), Near-Memory Processing (in 3D memory stack controller)...
- Specialize architecture
- Power on-demand only what is required

Nice take-away

The battery limitation may produce better programmers in the future ☺

Typical modern/future system



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

Outline

Remember C++ ?

2-line description by Bjarne Stroustrup

- Direct mapping to hardware
- Zero-overhead abstraction

- But what about heterogeneous computing?

Remember C++ ?

2-line description by Bjarne Stroustrup

- Direct mapping to hardware
- Zero-overhead abstraction

- But what about heterogeneous computing?

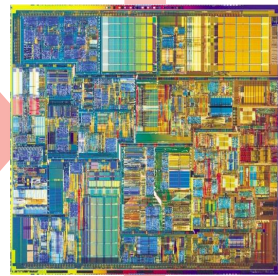
COLLADA

3D for the Web

- Real-time apps and games in-browser
- Efficiently delivering runtime 3D assets

WebGL glTF

NNEF OpenVX



OpenGL|SC OpenVG

Vulkan OpenGL|ES

OpenGL OpenXR

Vision and Neural Networks

- Tracking and odometry
- Scene analysis/understanding
- Neural Network inferencing

Real-time 2D/3D

- Virtual and Augmented Reality
- Cross-platform gaming and UI
- CG Visual Effects
- CAD and Product Design
- Safety-critical displays

SPIR SYCL

OpenCL

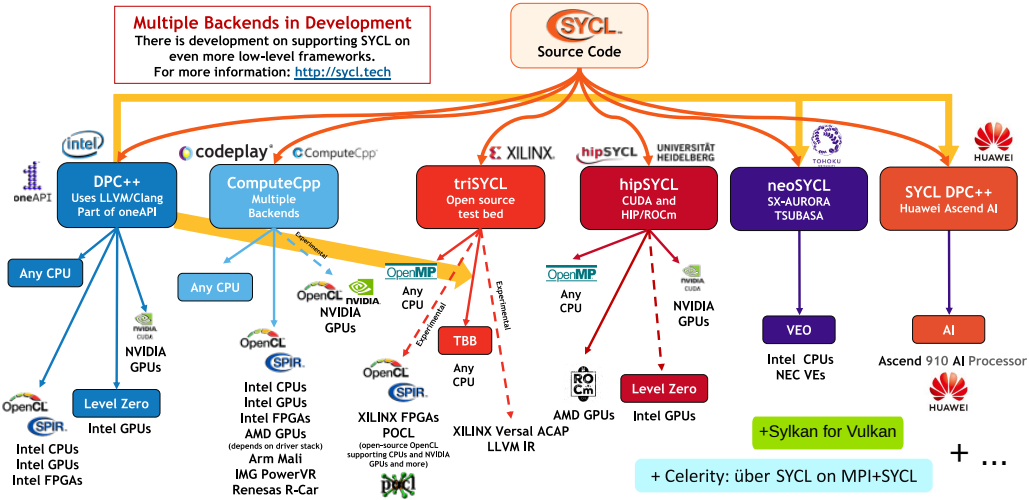
Parallel Computation

- Machine Learning acceleration
- Embedded vision processing
- High Performance Computing (HPC)

SYCL 2020 from Khronos Group, published on 2021-02-09

- `https://www.khronos.org/sycl`

SYCL ecosystem is growing



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

SYCL 2020 ≡ heterogeneous simplicity with modern C++

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

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

SYCL 2020 ≡ heterogeneous simplicity with modern C++

Abstract storage

- Host or device (remote) memory

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

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

SYCL 2020 ≡ heterogeneous simplicity with modern C++

Abstract storage

- Host or device (remote) memory

Code executed on device (“kernel”)

- “Single-source”
- Seamless integration in host code
- Type-safety
- Asynchronous execution

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

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

SYCL 2020 ≡ heterogeneous simplicity with modern C++

Abstract storage

- Host or device (remote) memory

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

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

Code executed on device (“kernel”)

- “Single-source”
- Seamless integration in host code
- Type-safety
- Asynchronous execution

Accessor

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

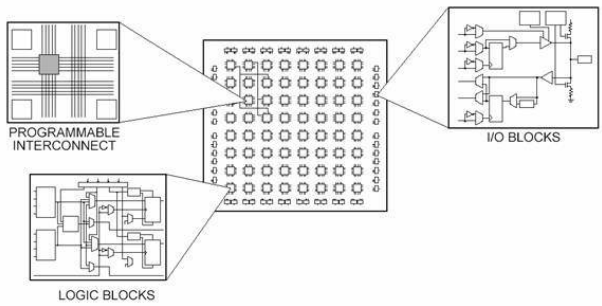
SYCL 2020 with unified shared memory (USM)

```cpp
// Using buffers and accessors

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

constexpr int N = 32;

int main () {
 sycl::buffer<int> buf { N };

 sycl::queue {}.submit([&](auto &h) {
  sycl::accessor a { buf, h, sycl::write_only,
                     sycl::no_init };
  h.parallel_for(N, [=](auto i) { a[i] = i; });
 });

 for (sycl::host_accessor a { buf }; auto e : a)
  std::cout << e << std::endl;
}
```

```cpp
// Using USM only

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

constexpr int N = 32;

int main () {
 sycl::queue q;
 int* a = sycl::malloc_shared<int>(N, q);

 q.parallel_for(N, [=](auto i) { a[i] = i; });
 q.wait();

 for (int i = 0; i < N; i++)
  std::cout << a[i] << std::endl;

 sycl::free(a, q);
}
```

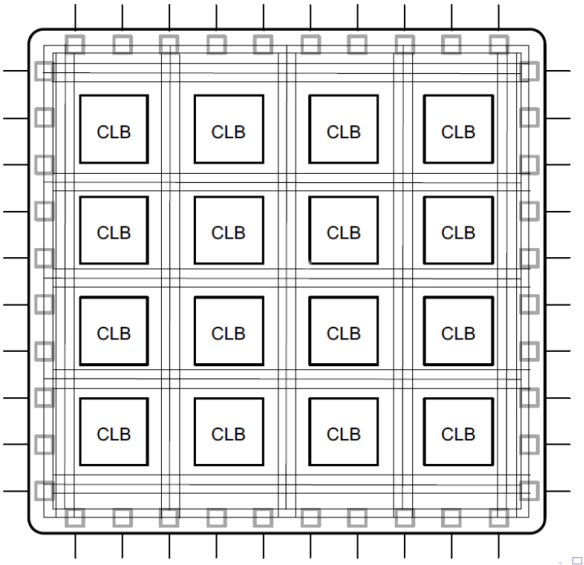
Outline

Deconstructivism in (hardware) architecture: FPGA

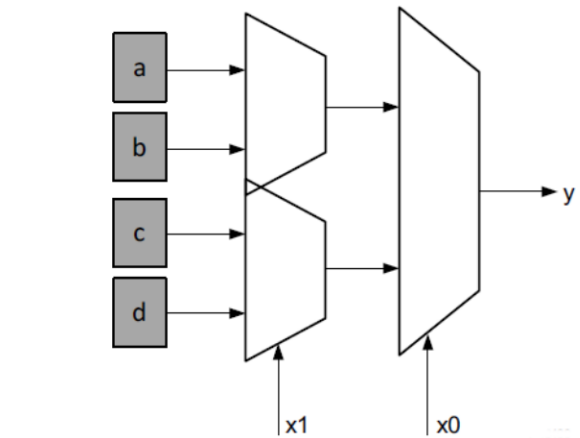


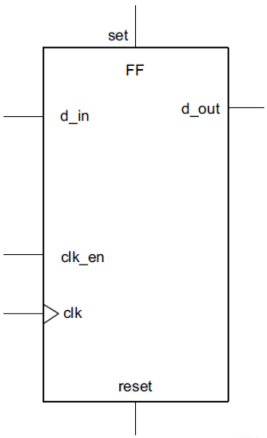
https://www.quora.com/What-is-FPGA-How-does-that-works

Basic architecture of Field-Programmable Gate Array (FPGA)



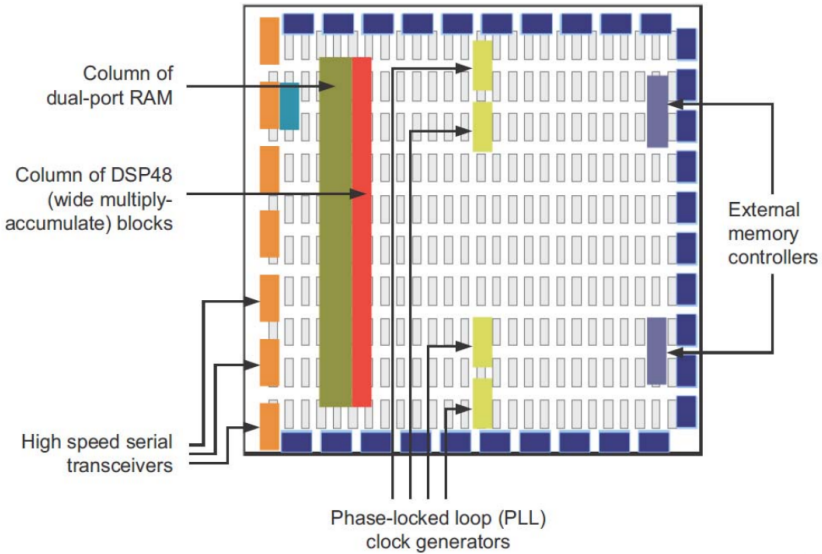
Basic architecture = Lookup Table + Flip-Flop storage + Interconnect



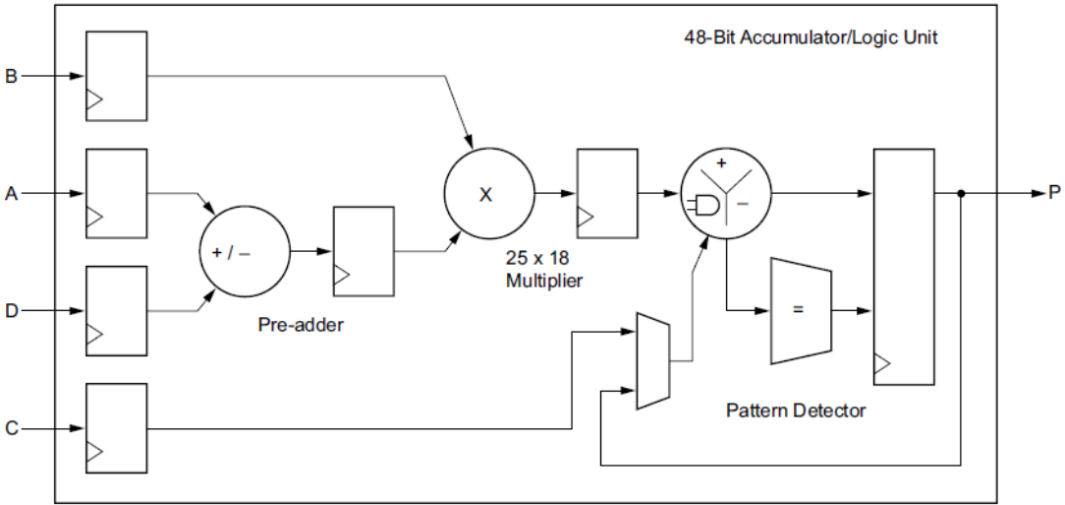


Typical Xilinx LUT have 6 inputs

Global view of programmable logic part



DSP48 block overview



Typical MPSoC with FPGA: Xilinx 7nm Versal AI Core VC1902

Versal Series Overview

Compute Engines

- Scalar Processors in every device
- Enhanced Programmable Logic
- New AI and enhanced DSP Engines

NoC and Memory

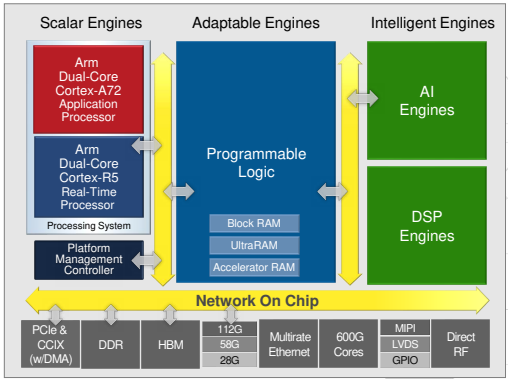
- High BW Network-on-Chip
- Hardened [LP]DDR4/5, and HBM

High-Speed Interfaces

- PCIe & CCIX up to Gen5
- Ethernet MAC up to 600Gbps

SerDes and RF

- SerDes Up to 112G PAM4
- Integrated ADC/DAC



Typical FPGA programming

- Bit-stream: vendor internal
- Hardware description language (HDL) describing behavior and structure, synchronous semantics
  - VHDL (1987) based on Ada syntax
  - Verilog (1984) based on C syntax

  Full control for extreme performance but very low productivity
- HLS (High-level synthesis)
  - Translate subset of C/C++/SystemC/Matlab/... into HDL

  Attract wider audience with higher productivity
- OpenCL C
  - Implemented as dialect of HLS

  Similar to other accelerators

None is single-source with host + accelerator

Typical FPGA programming

- Bit-stream: vendor internal
- Hardware description language (HDL) describing behavior and structure, synchronous semantics
  - VHDL (1987) based on Ada syntax
  - Verilog (1984) based on C syntax

  Full control for extreme performance but very low productivity
- HLS (High-level synthesis)
  - Translate subset of C/C++/SystemC/Matlab/... into HDL

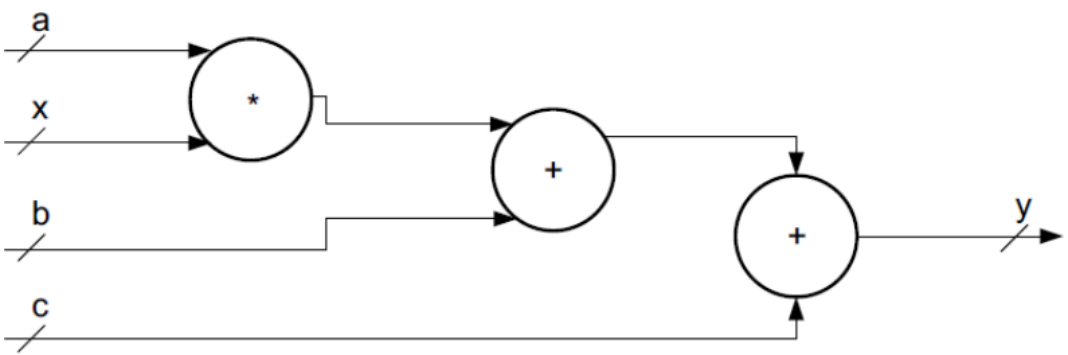
  Attract wider audience with higher productivity
- OpenCL C
  - Implemented as dialect of HLS

  Similar to other accelerators

None is single-source with host + accelerator

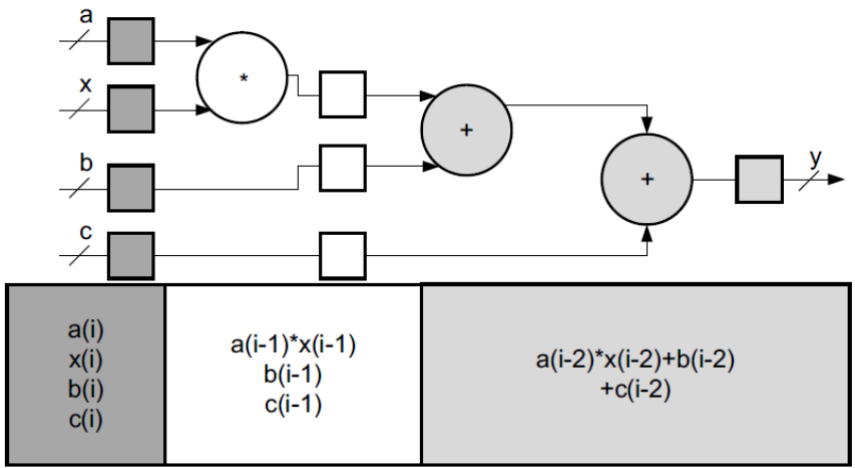
Compilation of C11/C++14 expressions in Xilinx Vitis HLS

```
y = a*x + b + c;
```

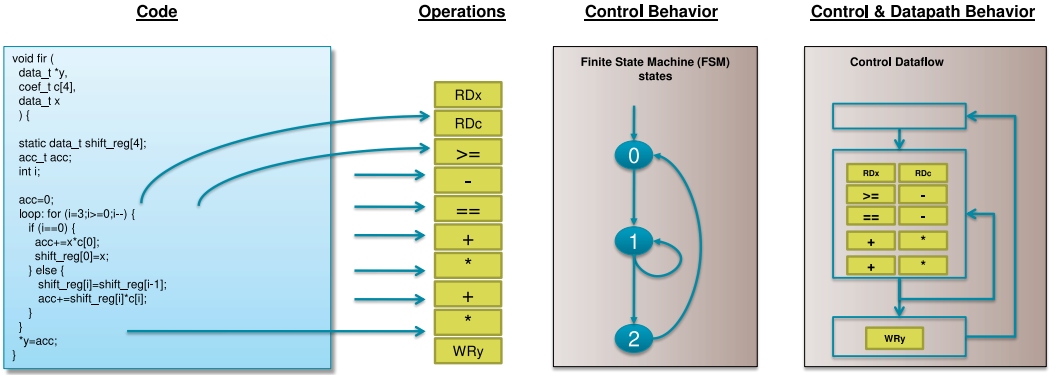


Compilation of expressions with pipelined execution

```
y = a*x + b + c;
```



HLS: control & datapath extraction



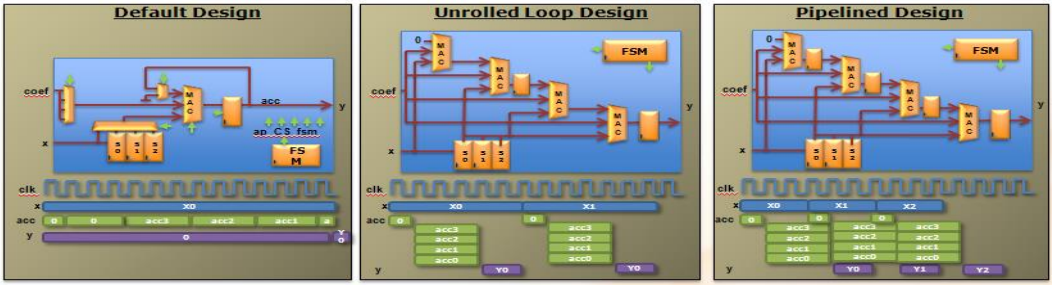
HLS: design exploration with directives/#pragma

```
loop: for (i=3;i>=0;i--) {
  if (i==0) {
    acc+=x*c[0];
    shift_reg[0]=x;
  } else {
    shift_reg[i]=shift_reg[i-1];
    acc+=shift_reg[i]*c[i];
  }
}
```

The same hardware is used for each iteration of the loop:
- Small area
- Long latency
- Low throughput

Different hardware is used for each iteration of the loop:
- Higher area
- Short latency
- Better throughput

Different iterations are executed concurrently:
- Higher area
- Short latency
- Best throughput

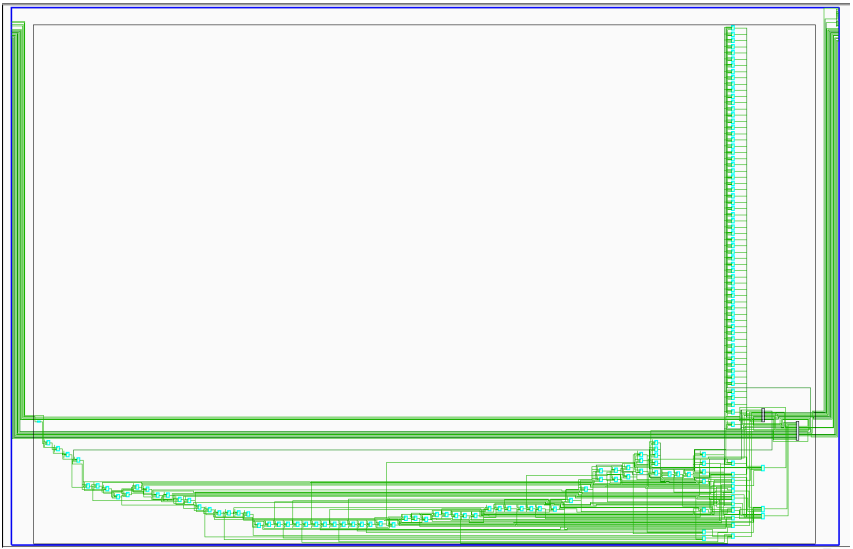


Outline

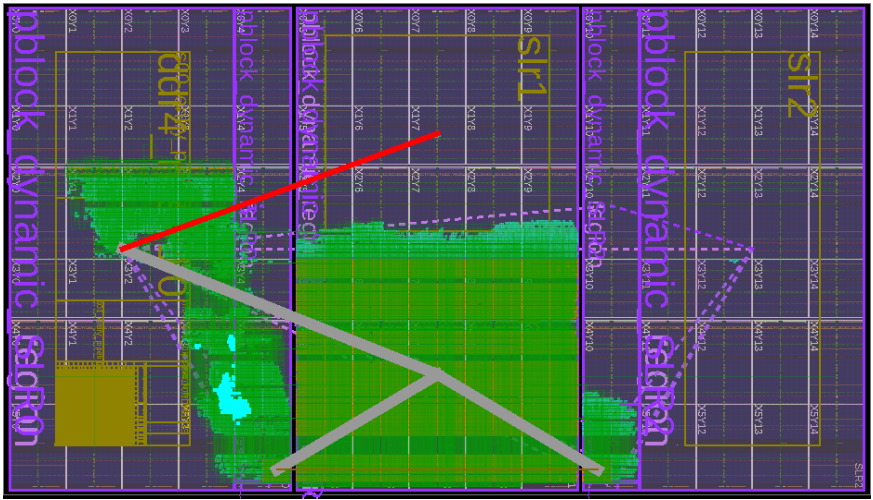
Compute the universal answer

```
// 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;
```

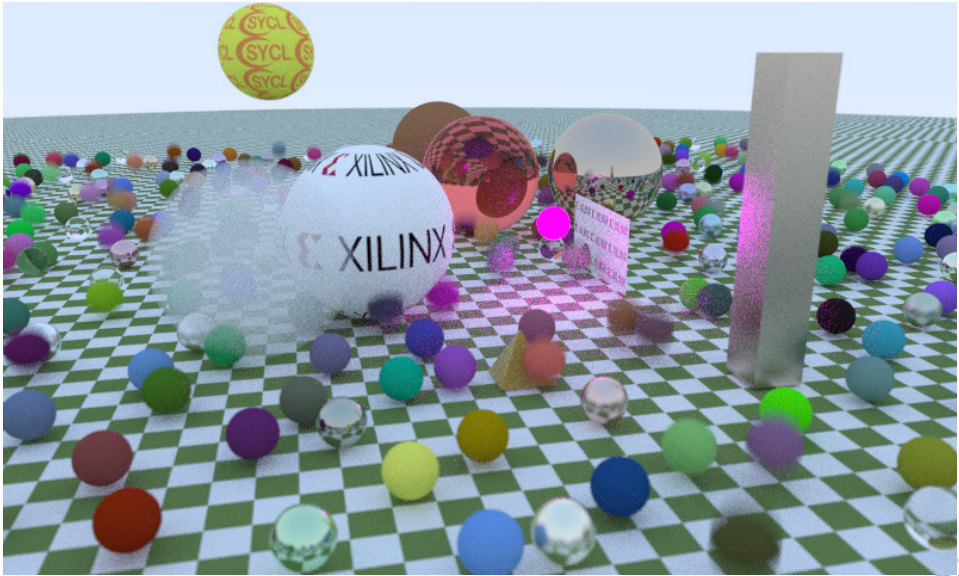
Schematics on Xilinx Alveo U200 FPGA PCIe card



Layout on Xilinx Alveo U200 FPGA PCIe card



Enable new application domains on FPGA: path tracing!



Path tracing 101

- "The Path to Path-Traced Movies", *Per H. Christensen (Pixar Animation Studios) and Wojciech Jarosz (Dartmouth College)*, Foundations and Trends in Computer Graphics and Vision, Vol. 10, No. 2 (2014), pp. 103–175.
  `https://cs.dartmouth.edu/wjarosz/publications/christensen16path.html`

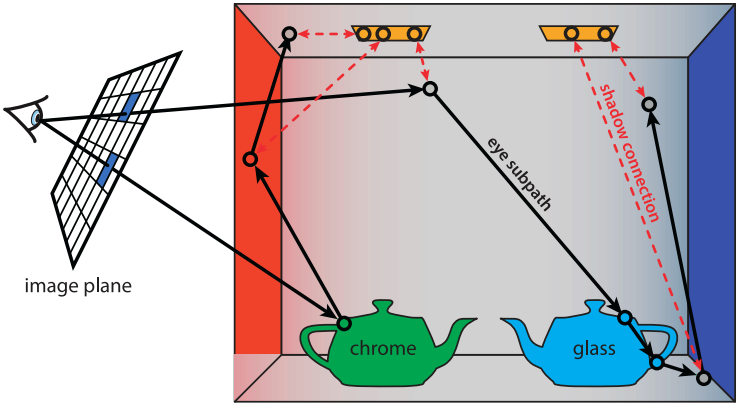


Figure 3.1: An illustration of tracing paths from the eye to the light sources in a Cornell box scene with two teapots.

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

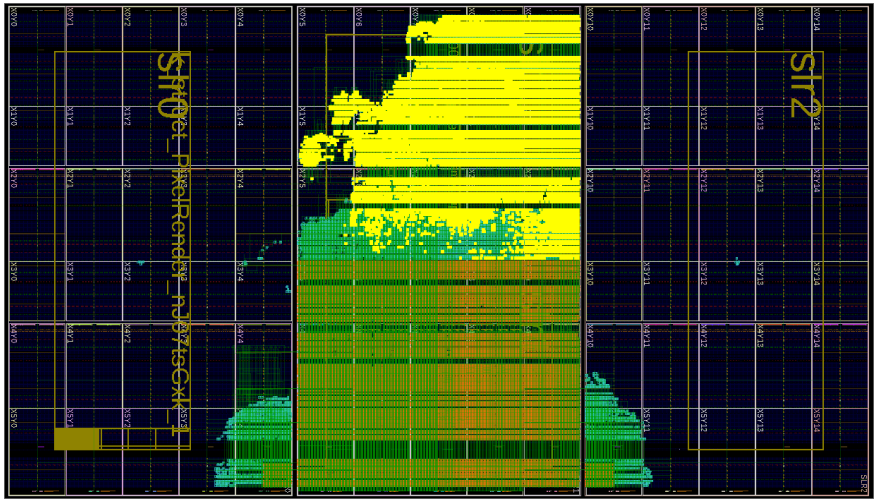
  https://raytracing.github.io/books/RayTracingInOneWeekend.html#surfacenormalsandmultipleobjects/anabstractionforhittableobjects

- 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 $\mathcal{O}(1)$ ☺

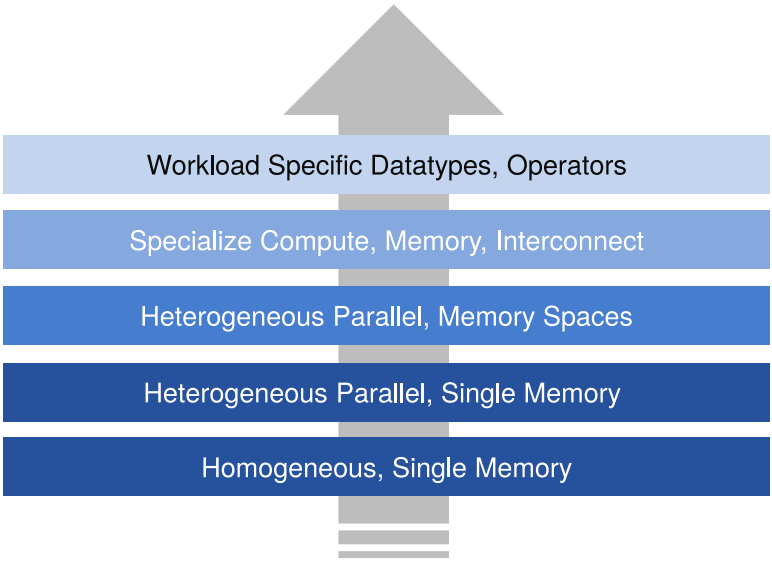
```
struct hittable {
  virtual bool hit(const ray& r, double t_min, double t_max,
                   hit_record& rec) const = 0;
};
struct sphere : hittable {
  virtual bool hit(const ray& r, double t_min, double t_max,
                   hit_record& rec) const override { ... };
};
struct rectangle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



Refinement levels with C++ abstractions



Domain Lib

Architecture Lib

C++SYCL

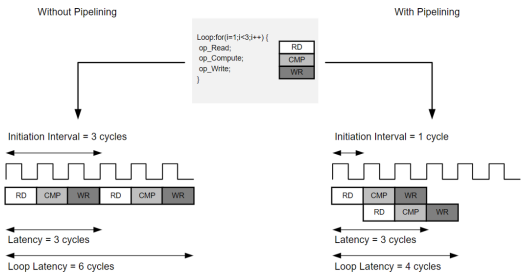
C++Parallel Lib

C++20

Pipelining loops on FPGA

- Loop instructions sequentially executed by default
- Loop iteration starts only after last operation from previous iteration
- Elemental operations are synthesized anyway...
- 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

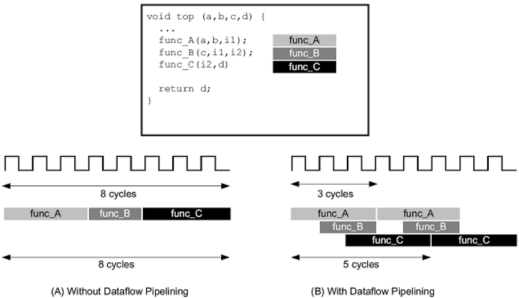
```
template<typename T, typename U>
void compute(T (&buffer_in)[BLOCK_SIZE],
             U (&buffer_out)[BLOCK_SIZE]) {
  for(int i = 0; i < NUM_ROWS; ++i) {
    for (int j = 0; j < WORD_PER_ROW; ++j) {
      sycl::ext::xilinx::pipeline([&] {
        int inTmp = buffer_in[WORD_PER_ROW*i+j];
      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

- Use native C++ construct instead of alien #pragma or attribute (vendor OpenCL or HLS C++...)
- Compatible with metaprogramming
- Implementation
  - No need for specific parser/tool-chain!
  - Just use lambda + intrinsics! ☺

```
#if defined(__SYCL_DEVICE_ONLY__)
#define __SYCL_DEVICE_ANNOTATE(...) __attribute__((annotate(__VA_ARGS__)))
#else
#define __SYCL_DEVICE_ANNOTATE(...)
#endif
template <typename IIType = AutoII, typename rewindtype = NoRewindPipeline,
          PipelineStyle pipelineType = PipelineStyle::stall, typename T>
__SYCL_DEVICE_ANNOTATE("xilinx_pipeline", IIType::value, rewindtype::value, pipelineType)
__SYCL_ALWAYS_INLINE void pipeline(T &&functor) { std::forward<T>(functor)(); }
```

Dataflow optimization on FPGA

- On CPU
  - Functions are executed sequentially
- On FPGA
  - Functions are implemented in hardware. . .
  - They coexist!
- Possible to execute them in parallel! ☺
- Even better when in a loop



- Dataflow execution mode
  - Each function scheduled as soon as data is available
  - Using FIFOs to forward data

Decorating code for dataflow execution in triSYCL

```
cgh.single_task<class add>([=] {
    int buffer_in[BLOCK_SIZE];
    int buffer_out[BLOCK_SIZE];
    vendor::xilinx::dataflow([&] {
        readInput(buffer_in, d_b);
        compute(buffer_in, buffer_out);
        writeOutput(buffer_out, d_a);
      });
  });
```

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

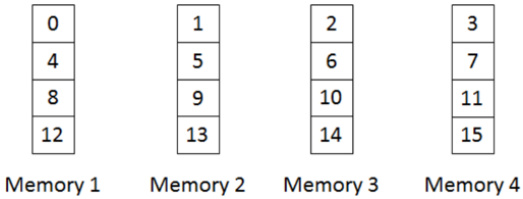
- Compatible with metaprogramming
- Implementation
  - No need for specific parser/tool-chain!
  - Just use lambda + intrinsics! ☺

```
template <typename T>
__SYCL_DEVICE_ANNOTATE("xilinx_dataflow")
__SYCL_ALWAYS_INLINE void dataflow(T &&functor) {
  std::forward<T>(functor)();
}
}
```

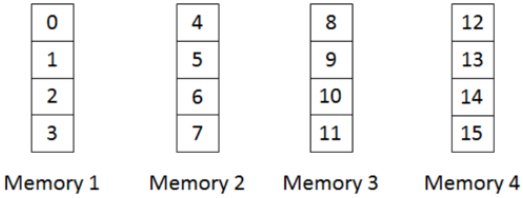
Partitioning memories *(I)*

- Remember memory bank conflicts on Cray vector computers 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



Partitioning memories (II)



- 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

- Complete partitioning
  - Extreme distribution
  - Extreme bandwidth
  - Low latency

partition_array class in triSYCL use case (I)

Enhanced std::array

```
cgh.single_task<class add>([=] {
  // 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;
  [...]
});
```

- Xilinx Vivado HLS C++

```
int A[MAX_DIM * MAX_DIM];
int B[MAX_DIM * MAX_DIM];
int C[MAX_DIM * MAX_DIM];
#pragma HLS ARRAY_PARTITION variable=A dim=1 \
                            cyclic factor=64
#pragma HLS ARRAY_PARTITION variable=B dim=1 \
                            block factor=64
```

- Xilinx Vitis OpenCL C

```
int A[MAX_DIM * MAX_DIM]
__attribute__((xcl_array_partition
               (cyclic, MAX_DIM, 1)));
int B[MAX_DIM * MAX_DIM]
__attribute__((xcl_array_partition
               (block, MAX_DIM, 1)));
int C[MAX_DIM * MAX_DIM];
```

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

- 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;
    }));
});
```

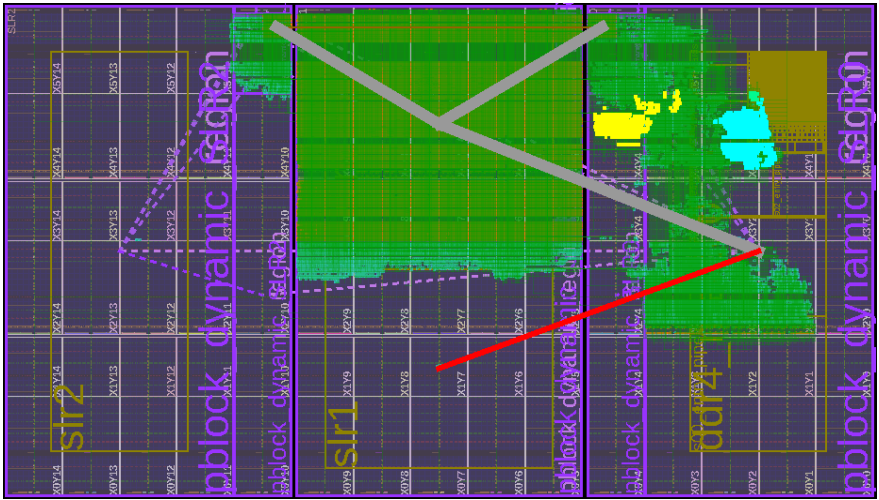
- Just ignored when not targeting Xilinx FPGA

Generic executor in 25 lines of SYCL, C++20 & Boost.Hana

```
static selector_defines::CompiledForDeviceSelector selector;
/* A generic function taking any number of arguments of any type
   and folding them with a given generic operator */
auto generic_executor(auto op, auto... inputs) {
  // Use a tuple of heterogeneous buffers to wrap the inputs
  auto bufs = boost::hana::make_tuple(
      sycl::buffer{std::begin(inputs), std::end(inputs)}...);
  // The element-wise computation
  auto compute =
    [=](auto args) { return boost::hana::fold_left(args, op); };
  // Use the range of the first argument as the range
  // of the result and computation
  auto size = bufs[0_c].size();
  // Infer the type of the output from 1 computation on inputs
  using return_value_type = decltype(
    compute(boost::hana::make_tuple(*std::begin(inputs)...)));
  // Create a buffer to return the result
  sycl::buffer<return_value_type> output{size};
  // Submit a command-group to the device
  sycl::queue{selector}.submit([&](sycl::handler &cgh) {
    // Define the data used as a tuple of read accessors
    auto ka = boost::hana::transform(bufs, [&](auto b) {
      return sycl::accessor{b, cgh, sycl::read_only};
    });
    // Data are produced to a write accessor to the output buffer
    sycl::accessor ko{output, cgh, sycl::write_only, sycl::no_init};
    // Define the kernel
    cgh.single_task([=] {
      for (int i = 0; i < size; ++i)
        sycl::ext::xilinx::pipeline([&] {
          // Pack operands an elemental computation in a tuple
          auto operands = boost::hana::transform(
            ka, [&](auto acc) { return acc[i]; });
          // Assign computation on the operands to the elemental result
          ko[i] = compute(operands);
        });
    });
  });
  // Return the output buffer
  return output;
};

int main() {
  std::vector<std::int8_t> u{1, 2, 3};
  std::vector<std::int16_t> v{5, 6, 7};
  // Do not use std::plus because it forces the same type for both operands
  auto res = generic_executor([](auto x, auto y) { return x + y; }, u, v);
  for (sycl::host_accessor a{res, sycl::read_only};
       auto e : std::span{&a[0], a.size()})
    std::cout << e << ' ';
  std::cout << std::endl;
  // Just for kidding
  std::vector<double> a{1, 2.5, 3.25, 10.125};
  std::set<char> b{5, 6, 7, 2};
  std::list<float> c{-55, 6.5, -7.5, 0};
  auto res2 = generic_executor(
    [](auto x, auto y) { return 3 * x - 7 * y; }, a, b, c);
  for (sycl::host_accessor a{res2, sycl::read_only};
       auto e : std::span{&a[0], a.size()})
    std::cout << e << ' ';
  std::cout << std::endl;
}
```

```
6 8 10
352 -128 -44.25 -55.875
```

Layout of generic executor on Xilinx Alveo U200 FPGA PCIe card



Modern metaprogramming as. . . hardware design tool

- Alternative implementation of

```
auto compute = [] (auto args) {
  return boost::hana::fold_left(args, [] (auto a, auto b) { return a + b; });
}; // f(... f(f(f(x1, x2), x3), x4) ..., xn)
```

  - Possible to use other Boost.Hana algorithms to add some hierarchy in the computation (Wallace's tree...)
  - Or to sort by type to minimize the hardware usage starting with "smallest" types
- Metaprogramming allows various implementations according to the types, sizes. . .
  - Kernel fusion, pipelined execution. . .
  - Codeplay VisionCpp, Eigen kernel fusion, Halide DSL. . .
- Introspection & metaclasses will allow quite more!
  - Generative programming. . .
  - Express directly and specialize code for each PE of a CGRA for example
- Imagine if SystemC was invented with this future C++ instead of C++98. . .

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 (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 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

Inclusive heterogeneous computing. . .

No transistors left behind!

Intel CPU (C++) + Xilinx FPGA (Vitis) + Nvidia GPU (CUDA)

```cpp
#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.size(), [=](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.size(); ++i)
   std::cout << acc[i] << ", ";
 std::cout << std::endl;
}
```

```
clang++ -std=c++20 -fsycl -fsycl-unnamed-lambda -fsycl-targets=nvptx64-nvidia-cuda-sycldevice,fpga64_hls_hw FPGA_GPU_CPU.cpp -o FPGA_GPU_CPU
```

oneAPI DPC++ + triSYCL https://github.com/triSYCL/sycl

Intel CPU (C++) + Xilinx FPGA (Vitis) + Nvidia GPU (CUDA)

- 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!

```
#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.size(), [=](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.size(); ++i)
   std::cout << acc[i] << ", ";
 std::cout << std::endl;
}
```

```
clang++ -std=c++20 -fsycl -fsycl-unnamed-lambda -fsycl-targets=nvptx64-nvidia-cuda-sycldevice,fpga64_hls_hw FPGA_GPU_CPU.cpp -o FPGA_GPU_CPU
```

oneAPI DPC++ + triSYCL `https://github.com/triSYCL/sycl`

Resources

- `https://www.khronos.org/sycl`
- `https://sycl.tech` community content
- `https://github.com/triSYCL/sycl` Open-source fusion triSYCL + DPC++ to Xilinx FPGA
- `https://github.com/intel/llvm` Intel oneAPI DPC++ open-source SYCL being up-streamed to Clang/LLVM

Khronos puns and pronunciation explained

SYCL

sickle [ˈsi-kəl]

SPIR

spear [ˈspir]

Conclusion

- Disclaimer: all this an on-going research project ☺
  - Lot of features are still missing
  - Use Xilinx Vivado & Vitis for any FPGA work non-focused on C++20
- Heterogeneous computing is everywhere and here to stay
  - *"The next decade will see a Cambrian explosion of novel computer architectures"*, Hennessy & Patterson, 2019
  - Embedded & full systems & HPC machines add other complexity levels...
- SYCL C++ standard from Khronos Group
  - Pure modern C++ DSEL for 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
  - No user locked-in!
  - Various implementations with back-ends & interoperability with other ecosystems (Vulkan, OpenCL, OpenMP, proprietary: CUDA, HIP, Level 0, TBB...)
- Still not happy? Do it the standard way! Participate to the standards and to open-source implementations to have a real impact! ☺

Conclusion

- Disclaimer: all this an on-going research project ☺
  - Lot of features are still missing
  - Use Xilinx Vivado & Vitis for any FPGA work non-focused on C++20
- Heterogeneous computing is everywhere and here to stay
  - *“The next decade will see a Cambrian explosion of novel computer architectures”*, Hennessy & Patterson, 2019
  - Embedded & full systems & HPC machines add other complexity levels...
- SYCL C++ standard from Khronos Group
  - Pure modern C++ DSEL for 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
  - No user locked-in!
  - Various implementations with back-ends & interoperability with other ecosystems (Vulkan, OpenCL, OpenMP, proprietary: CUDA, HIP, Level 0, TBB...)
- Still not happy? Do it the standard way! Participate to the standards and to open-source implementations to have a real impact! ☺

Conclusion

- Disclaimer: all this an on-going research project ☺
  - Lot of features are still missing
  - Use Xilinx Vivado & Vitis for any FPGA work non-focused on C++20
- Heterogeneous computing is everywhere and here to stay
  - *"The next decade will see a Cambrian explosion of novel computer architectures"*, Hennessy & Patterson, 2019
  - Embedded & full systems & HPC machines add other complexity levels...
- SYCL C++ standard from Khronos Group
  - Pure modern C++ DSEL for 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
  - No user locked-in!
  - Various implementations with back-ends & interoperability with other ecosystems (Vulkan, OpenCL, OpenMP, proprietary: CUDA, HIP, Level 0, TBB...)
- Still not happy? Do it the standard way! Participate to the standards and to open-source implementations to have a real impact! ☺

Conclusion

- Disclaimer: all this an on-going research project ☺
  - Lot of features are still missing
  - Use Xilinx Vivado & Vitis for any FPGA work non-focused on C++20
- Heterogeneous computing is everywhere and here to stay
  - *"The next decade will see a Cambrian explosion of novel computer architectures"*, Hennessy & Patterson, 2019
  - Embedded & full systems & HPC machines add other complexity levels...
- SYCL C++ standard from Khronos Group
  - Pure modern C++ DSEL for 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
  - No user locked-in!
  - Various implementations with back-ends & interoperability with other ecosystems (Vulkan, OpenCL, OpenMP, proprietary: CUDA, HIP, Level 0, TBB...)
- Still not happy? Do it the standard way! Participate to the standards and to open-source implementations to have a real impact! ☺

Conclusion

- Disclaimer: all this an on-going research project ☺
  - Lot of features are still missing
  - Use Xilinx Vivado & Vitis for any FPGA work non-focused on C++20
- Heterogeneous computing is everywhere and here to stay
  - *"The next decade will see a Cambrian explosion of novel computer architectures"*, Hennessy & Patterson, 2019
  - Embedded & full systems & HPC machines add other complexity levels...
- SYCL C++ standard from Khronos Group
  - Pure modern C++ DSEL for 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
  - No user locked-in!
  - Various implementations with back-ends & interoperability with other ecosystems (Vulkan, OpenCL, OpenMP, proprietary: CUDA, HIP, Level 0, TBB...)
- Still not happy? Do it the standard way! Participate to the standards and to open-source implementations to have a real impact! ☺