XILINX
ALL PROGRAMMABLE™

SYCL C++ and OpenCL interoperability experimentation with triSYCL

Anastasios Doumoulakis, Ronan Keryell & Kenneth O'Brien

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 very slow at 1+ GHz...

Power wall & speed of light: implications

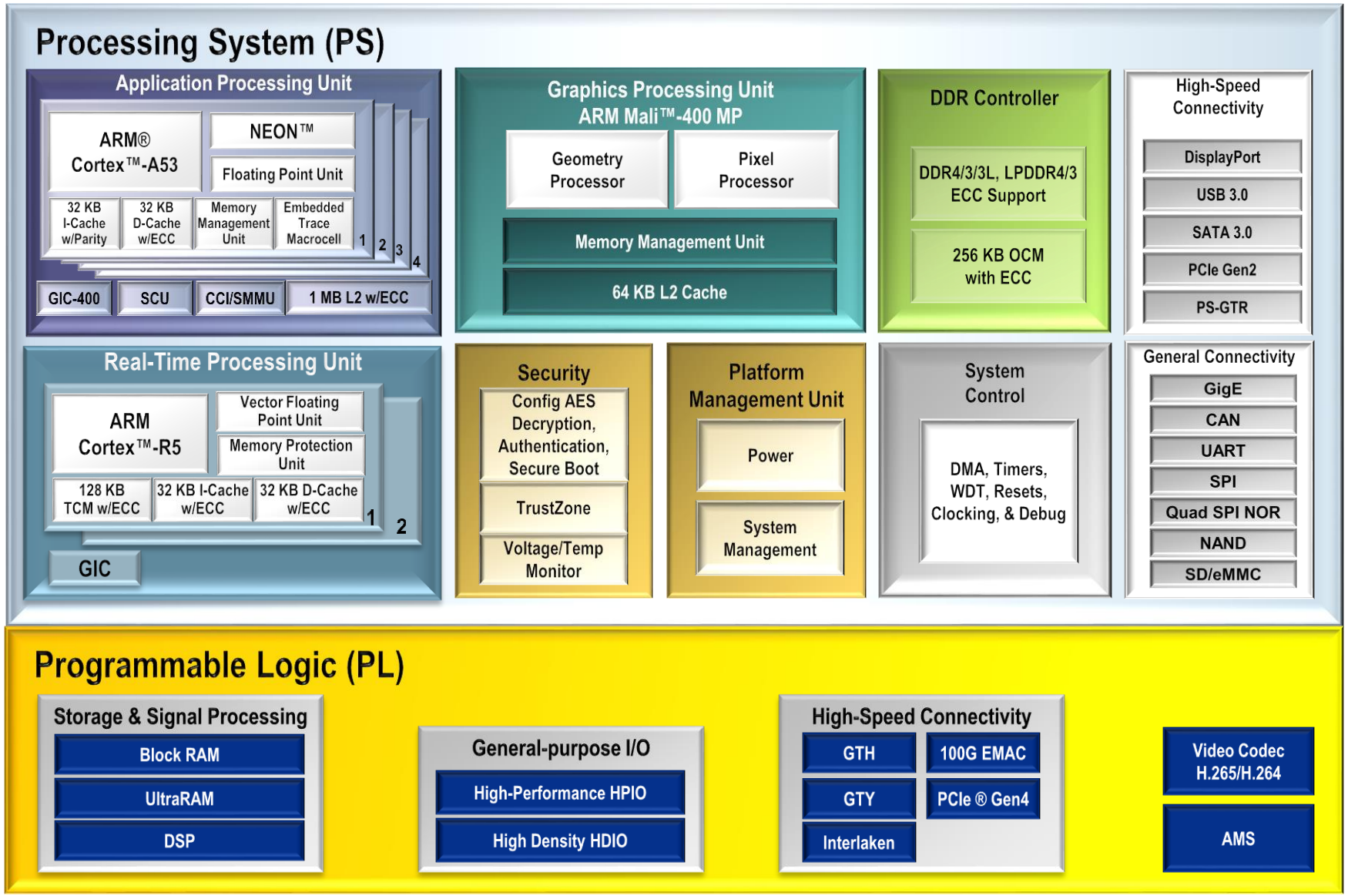
- Change hardware and software
  - Use locality & hierarchy
  - Massive parallelism
- NUMA (non-uniform memory access) & distributed memories

  → 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 pressure may produce better programmers in the future ☺

FPGA in MPSoC: Zynq UltraScale+ MPSoC Overview



¿¿¿ How to avoid Frankenstein programming ???

Khronos standards for heterogeneous systems

Connecting Software to Silicon

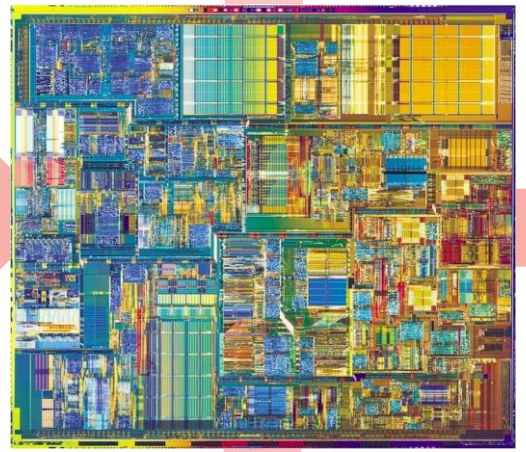
COLLADA

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

'VR Initiative'

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

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

OpenCL SPIR SYCL

KHRONOS GROUP

XILINX ALL PROGRAMMABLE

PROMOTER MEMBERS

KHRONOS GROUP

Over 100 members worldwide
Any company is welcome to join

SYCL

SYCL 2.2 = pure C++17 DSEL

- Implement concepts useful for heterogeneous computing
- Asynchronous task graph
- Hierarchical parallelism & kernel-side enqueue
- Queues to direct computations on devices
- Single-source programming model
  - Take advantage of CUDA on steroids & OpenMP simplicity and power
  - Compiled for host *and* device(s)
  - Enabling the creation of C++ higher level programming models & C++ templated libraries
  - System-level programming (SYstemCL)
- Buffers to define location-independent storage
- Accessors to express usage for buffers and pipes: read/write/...
  - No explicit data motion
  - Automatic overlapping of communication/computation
- Hierarchical storage
  - Rely on C++ allocator to specify storage (SVM...)
  - Usual OpenCL-style global/local/private
- Most modern C++ features available for OpenCL
  - Programming interface based on abstraction of OpenCL components (data management, error handling...)
  - Provide OpenCL interoperability
- Directly executable DSEL
  - Host fall-back & emulation for free
  - No specific compiler needed for experimenting on host
  - Debug and symmetry for SIMD/multithread on host

Complete example of matrix addition in OpenCL SYCL

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

using namespace cl::sycl;

constexpr size_t N = 2;
constexpr size_t M = 3;
using Matrix = float[N][M];

// Compute sum of matrices a and b into c
int main() {
 Matrix a = { { 1, 2, 3 }, { 4, 5, 6 } };
 Matrix b = { { 2, 3, 4 }, { 5, 6, 7 } };

 Matrix c;

 {// Create a queue to work on default device
  queue q;
  // Wrap some buffers around our data
  buffer A { &a[0][0], range { N, M } };
  buffer B { &b[0][0], range { N, M } };
  buffer C { &c[0][0], range { N, M } };

  // Enqueue some computation kernel task
  q.submit([&](handler& cgh) {
   // Define the data used/produced, type-safe
   auto ka = A.get_access<access::mode::read>(cgh);
   auto kb = B.get_access<access::mode::read>(cgh);
   auto kc = C.get_access<access::mode::write>(cgh);
   // Create & call kernel named "mat_add"
   cgh.parallel_for<class mat_add>(range { N, M },
      [=](id<2> i) { kc[i] = ka[i] + kb[i]; }
   );
  }); // End of our commands for this queue
 } // End scope, wait for the buffers to be released
 // Copy back the buffer data with RAII behaviour.
 std::cout << "c[0][2] = " << c[0][2] << std::endl;
 return 0;
}
```

Producer/consumer for matrix addition in OpenCL SYCL

```
#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

    // Create a queue to work on default device
    queue q;
    // 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.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.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 an asynchronous kernel to compute matrix addition c = a
+ b
    q.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];
                        });
    });
    /* Request an access to read c from the host-side. The SYCL
runtime
       ensures that c is ready when the accessor is returned */
    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;
}
```

triSYCL open-source implementation

triSYCL

- Open-source SYCL implementation
- On-going implementation started at AMD and now led by Xilinx
- Used by Khronos committee to define the SYCL & OpenCL C++ standard
  - Languages are now too complex to be defined without implementing...
- https://github.com/triSYCL/triSYCL

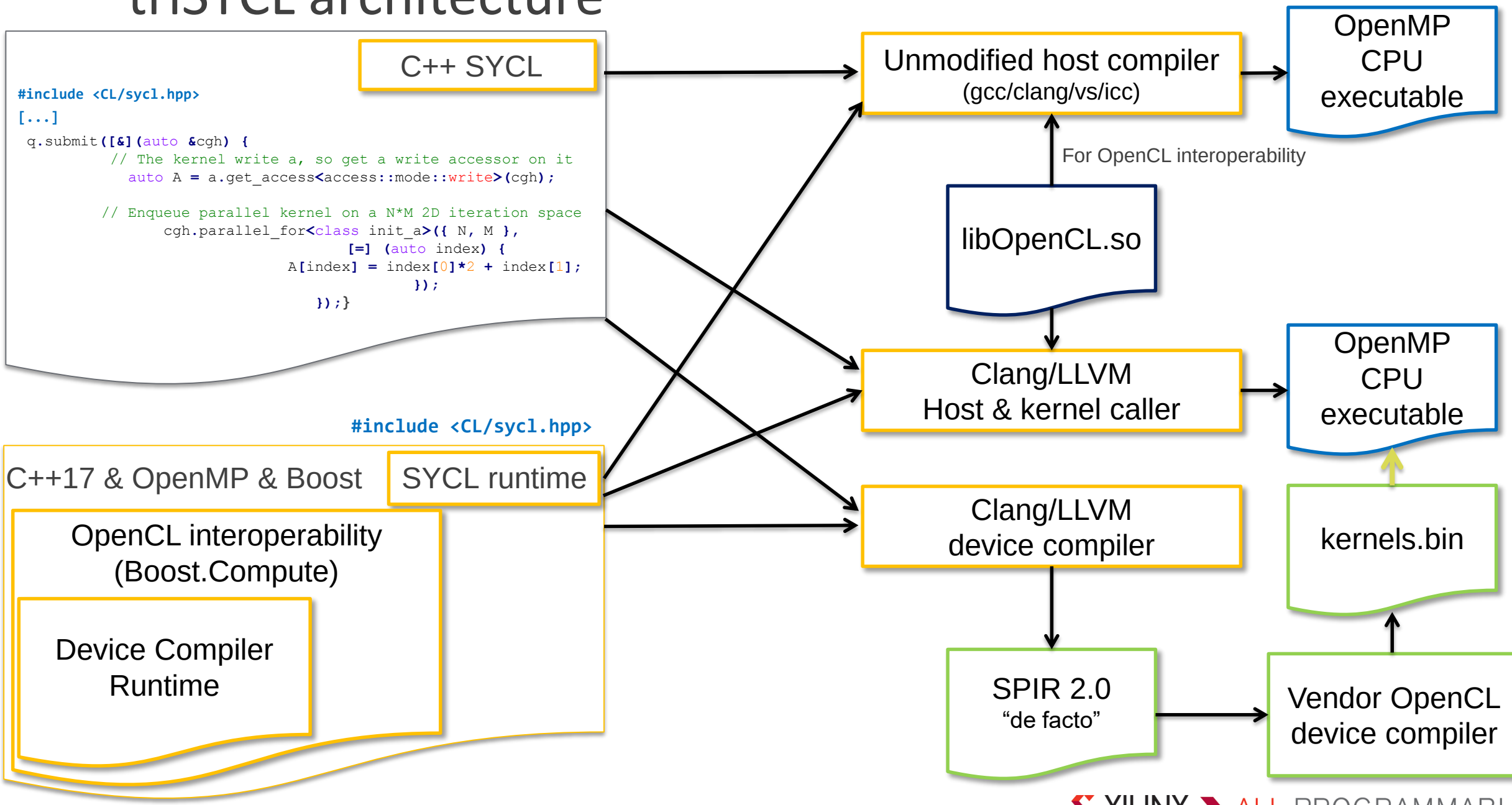
  ≈ 10 contributors

  ∃ private Git repositories for future Khronos & experimental Xilinx versions

triSYCL (more details…)

- Pure C++ implementation & CPU-only implementation for now
  - Templated C++17 classes
  - Use OpenMP for computation on CPU + `std::thread` for task graph
  - Rely on STL & Boost for zen style
  - CPU emulation for free
    - Quite useful for debugging
  - More focused on correctness than performance for now (array bound checking...)
- Provide OpenCL-interoperability mode
  - Reuse existing OpenCL code
- Some extensions
  - Xilinx blocking pipes (useful on FPGA with Independent Forward Progress guarantee)
- On-going outlining compiler based on open-source Clang/LLVM compiler

triSYCL architecture



SYCL OpenCL interoperability

OpenCL interoperability of SYCL: low-hanging fruits…

- Writing OpenCL C or OpenCL C++ kernels is painful
  - Not single-source like CUDA or SYCL
  - Need explicit buffer management & data transfers
  - No type safety between host and kernel code
- SYCL *is* single-source but provides *also* OpenCL interoperability mode
  - Can call existing OpenCL kernels *too*!
  - Keep task-graph goodies and accessors *at the same time*
  - *No* need for explicit data transfers
- Can replace usual C++ OpenCL wrappers

∃ OpenCL built-in kernels…

- OpenCL built-in kernels are *very* common in FPGA world
- Written in Verilog/VHDL or Vivado HLS C++
  - But with SDAccel OpenCL-compatible kernel interface
- OpenCL host API useful as standard host-device interaction
- Typical use cases
  - Kernel libraries
  - Linear algebra
  - Machine learning
  - Computer vision (Xilinx reVISION)
  - Direct access to hardware: wire-speed Ethernet…
- Not possible to use SYCL single-source mode with built-in kernels…
- …But SYCL OpenCL interoperability mode can simplify usage of these kernels
  - Alternative to other wrappers

Binary Neural Network for image classification on FPGA

- Vision application using 1-bit data
  - Efficient on FPGA! ☺
- Kernels written in Vivado HLS C++
- Convolution weights directly in the program
  - No need to load them
  - Partial evaluation of the program ☺
- Use OpenCL host API
  - Transfer data to/from FPGA
  - Launch kernels

Binary Neural Network for image classification

- Original version with own OpenCL C++ wrapper

```
queue.enqueueWriteBuffer(inputBuffer, CL_TRUE, 0, numInputElems, inputVector, NULL, &event);
kernel.setArgs(0, inputBuffer);
kernel.setArgs(1, outputBuffer);
kernel.setArgs(2, false);
kernel.setArgs(3, 0);
kernel.setArgs(4, 0);
kernel.setArgs(5, 0);
kernel.setArgs(6, 0L);
kernel.setArgs(7, count);
queue.enqueueTask(kernel, NULL, &event);
queue.enqueueReadBuffer(outputBuffer, CL_TRUE, 0, numOutputElems, outputVector, NULL, &event);
```

- Value of SYCL interoperability mode?

Binary Neural Network for image classification on FPGA PCIe card

```
buffer<long> inputBuffer { inputVector, numInputElems };
  {
    buffer<long> outputBuffer { outputVector, numOutputElems };
    queue.submit([&] (handler &cgh) {
      cgh.set_args(inputBuffer.get_access<access::mode::read>(cgh),
                   outputBuffer.get_access<access::mode::write>(cgh),
                   false, 0, 0, 0, 0L, count);
      cgh.single_task(kernel);
    });
  } // Implicit communication & write-back on buffer destruction
```

Performance results (ms) of the BNN application for 2 host API:

	Setup	*Memory*	*Build*	*Kernel*	*Total*
OpenCL	6490.22	.08	4598.1	4.011	11092.43
SYCL	6411.32	.002	4903.65	3.91	11325.57

Hand-written number recognition using L2 norm

- Recognition of handwritten digits in 8-bit grey 28×28 images
- 500 images compared against reference set of 5000 labelled digit images
- Pixel-wise L2 norm
- 3 versions of the program https://github.com/a-doumoulakis/triSYCL_knn
  - Pure single-source SYCL
  - SYCL with OpenCL interoperability
  - Pure OpenCL (host API + kernel) using `CL/cl2.hpp`
- Difficult to find platform supporting all the requirements ☹
  - AMD GPU: supports SPIR but not recent Linux & X11 server
  - nVidia: supports OpenCL 2.0 & modern OS but not SPIR

Hand-written number recognition using L2 norm : SYCL

```
int search_image(buffer<int>& training, buffer<int>& res_buffer,
                 const Img& img, queue& q) {
  {
    buffer<int> A { std::begin(img.pixels), std::end(img.pixels) };
    // Compute the L2 distance between an image and each one from the
    // training set
    q.submit([&] (handler &cgh) {
        // These accessors lazily trigger data transfers between host
        // and device only if necessary. For example "training" is
        // only transfered the first time the kernel is executed.
        auto train = training.get_access<access::mode::read>(cgh);
        auto ka = A.get_access<access::mode::read>(cgh);
        auto kb = res_buffer.get_access<access::mode::write>(cgh);
        // Launch a kernel with training_set_size work-items
        cgh.parallel_for<class KnnKernel>(range<1> { training_set_size },
                                          [=] (id<1> index) {
            decltype(ka)::value_type diff = 0;
            // For each pixel
            for (auto i = 0; i != pixel_number; i++) {
              auto toAdd = ka[i] - train[index[0]*pixel_number + i];
              diff += toAdd*toAdd;
            }
            kb[index] = diff;
          });
      });
  }

  auto r = res_buffer.get_access<access::mode::read>();

  // Find the image with the minimum distance
  auto min_image = std::min_element(std::begin(result), std::end(result));

  // Test if we found the good digit
  return
    training_set[std::distance(std::begin(result), min_image)].label ==
img.label;
}

int main(int argc, char* argv[]) {
  training_set = slurp_file("data/trainingsample.csv");
  validation_set =  slurp_file("data/validationsample.csv");
  buffer<int> training_buffer = get_buffer(training_set);
  buffer<int> result_buffer { result, training_set_size };

  // A SYCL queue to send the heterogeneous work-load to
  queue q;

  int correct = 0;

  // Match each image from the validation set against the images from
  // the training set
  for (auto const& img : validation_set)
    correct += search_image(training_buffer, result_buffer, img, q);

  [...]
  return 0;
}
```

Hand-written number recognition using L2 norm : SYCL OpenCL

```
int search_image(buffer<int>& training, const Img& img, queue& q, const kernel& k) {
  int res[training_set_size];

  {
    buffer<int> A { std::begin(img.pixels), std::end(img.pixels) };
    buffer<int> B { res, training_set_size };
    // Compute the L2 distance between an image and each one from the
    // training set
    q.submit([&] (handler &cgh) {
        // Set the kernel arguments. The accessors lazily trigger data
        // transfers between host and device only if necessary. For
        // example "training" is only transfered the first time the
        // kernel is executed.
        cgh.set_args(training.get_access<access::mode::read>(cgh),
                     A.get_access<access::mode::read>(cgh),
                     B.get_access<access::mode::discard_write>(cgh),
                     int { training_set_size }, int { pixel_number });
        // Launch the kernel with training_set_size work-items
        cgh.parallel_for(training_set_size, k);
      });
    // The destruction of B here waits for kernel execution and copy
    // back the data to res
  }

  // Find the image with the minimum distance
  auto min_image = std::min_element(std::begin(res), std::end(res));

  // Test if we found the correct digit
  return
    training_set[std::distance(std::begin(res), min_image)].label == img.label;
}

int main(int argc, char* argv[]) {
  int correct = 0;
  training_set = slurp_file("data/trainingsample.csv");
  validation_set =  slurp_file("data/validationsample.csv");
  buffer<int> training_buffer = get_buffer(training_set);
```

```
  // A SYCL queue to send the heterogeneous work-load to
  queue q { boost::compute::system::default_queue() };

  // Use real OpenCL program for the kernel
  auto program = boost::compute::program::create_with_source(R"(
    __kernel void kernel_compute(__global const int* trainingSet,
                                 __global const int* data,
                                 __global int* res, int setSize, int dataSize) {
      int diff, toAdd, computeId;
      computeId = get_global_id(0);
      if (computeId < setSize) {
        diff = 0;
        for (int i = 0; i < dataSize; i++) {
            toAdd = data[i] - trainingSet[computeId*dataSize + i];
            diff += toAdd * toAdd;
        }
        res[computeId] = diff;
      }
    }
    )", boost::compute::system::default_context());

  program.build();

  // Construct a SYCL kernel from OpenCL kernel to be used in
  // interoperability mode
  kernel k { boost::compute::kernel { program, "kernel_compute"} };

  // Match each image from the validation set against the images from
  // the training set
  for (auto const & img : validation_set)
    correct += search_image(training_buffer, img, q, k);

  [...]

  return 0;
}
```

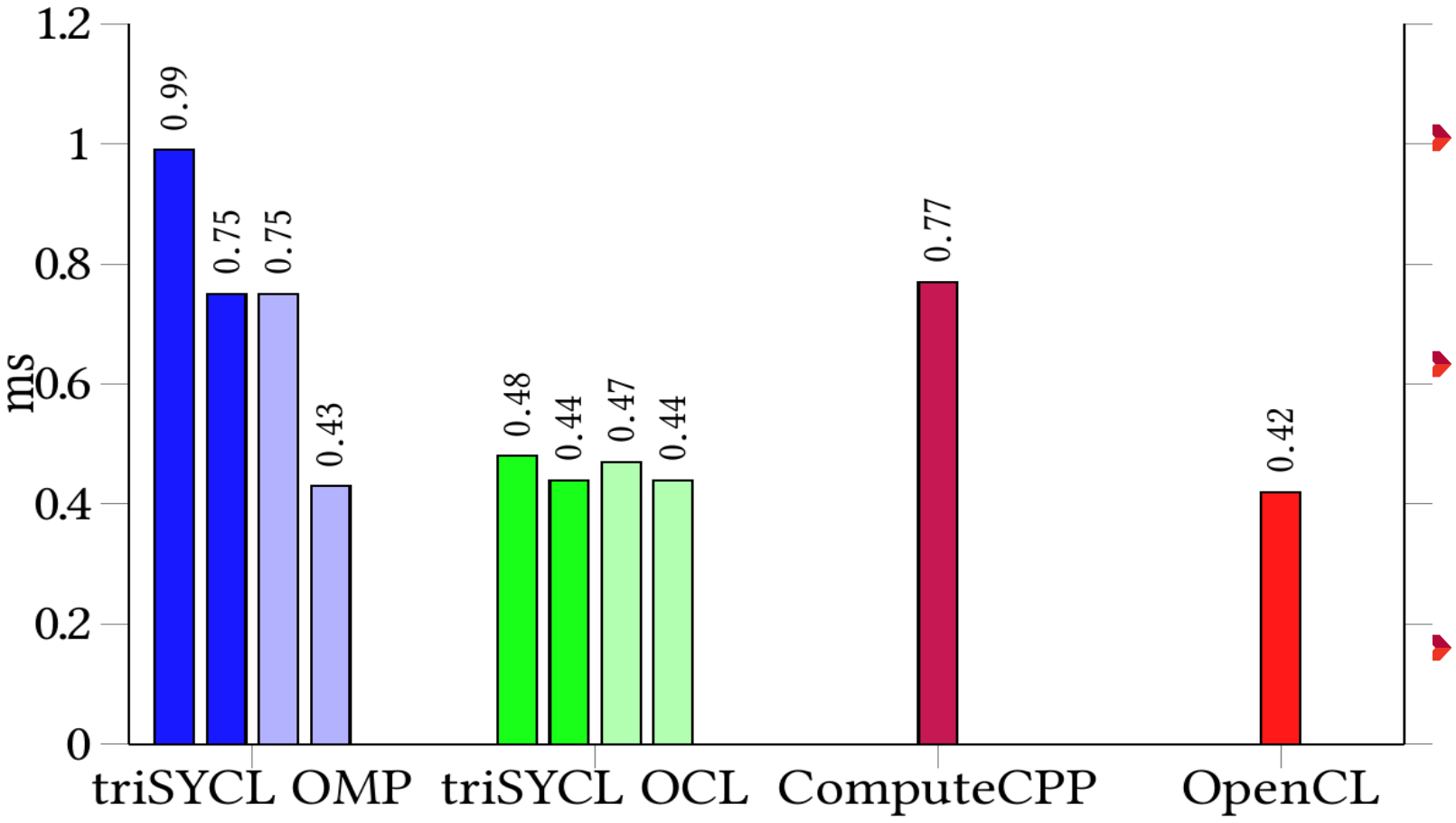
Software stack & hardware context

- Only using CPU Intel Core i7-6700
- Linux Ubuntu 16.10
- GCC 6.2.0
- Clang/LLVM 3.8.1
- ComputeCpp 0.1.3 CE
- OpenCL on CPU: Intel OpenCL runtime 16.1.1 and Intel OpenCL SDK
- Profiling with Intel VTune

Software experiment

- triSYCL OpenMP
  - Single-source triSYCL targeting OpenMP on CPU
- ComputeCPP
  - Single-source SYCL with ComputeCpp to SPIR and Intel OpenCL on CPU
- triSYCL OpenCL
  - Interoperability with Intel OpenCL on CPU
- OpenCL
  - Running "pure" OpenCL code with Intel OpenCL on CPU
- TRISYCL_NOASYNC
  - Disable asynchronous execution of kernels and thread usage
  - Not SYCL conformant
- Disclaimer
  - Quick experiment without optimizing
  - early release software

Hand-written number recognition using L2 norm



Darker: GCC/GCC NOASYNC, lighter Clang/Clang NOASYNC

- Intel CPU OpenCL implementation on top of TBB
  - Efficient thread execution recycling 7 threads
- OpenMP
  - GCC uses `libgomp`
  - Clang more efficient with AVX2 SIMD vectorization + `libomp` donated by Intel
- Using NOASYNC gives estimation thread overhead
  - 500 kernels = 500 `std::task` creations…
  - Each one launching 8 OpenMP threads: 4000 thread creations…
- ComputeCpp behaves reasonably
  - Not officially supported with SPIR on top of Intel CPU OpenCL
  - Probably no SIMD through SPIR-Intel path yet

Performance issues…

- triSYCL started as an experimenting workbench for the specification…
- Clean and understandable modern C++ code
  - Direct mapping of SYCL tasks to raw C++ `std::thread` + `std::condition_variable`
  - No mapping from SYCL task-graph down to OpenCL event-base graph
- … but with more and more users, performance is now an issue
- → Important overhead for small kernels!
- Lazy buffer transfers are implemented
- Refactoring needed for the asynchronous task graph
  - Use std::async & thread pools
  - Map OpenCL kernels to OpenCL event-based dependencies

Conclusion

Conclusion

- SYCL = pure modern C++ DSeL for single-source heterogeneous computing
  - High productivity through host-devices type-safety
  - Generic parallel programming model + OpenCL interoperability
- OpenCL interoperability mode: useful for plain OpenCL programmers!
  - Recycle existing OpenCL-ish kernels (FPGA HLS…)
  - Simplify host code with modern C++
    - Task graph
    - Implicit buffer transfers through accessors
- triSYCL: pure C++17, OpenMP and Boost for CPU & OpenCL-compatible devices
  - Open standards need open-source implementations for acceptance
  - Reasonable performance in OpenCL interoperability mode
  - On-going implementation of device compiler with Clang/LLVM to SPIR for full single-source experience
  - Subliminal message to vendors: please provide OpenCL & SPIR support for recent OS ☹
- Open-source on-going implementation… lot of optimization opportunities!
  - Join us and improve your skills on C++17/C++20, Boost, Clang/LLVM, OpenCL, SPIR, OS, FPGA…