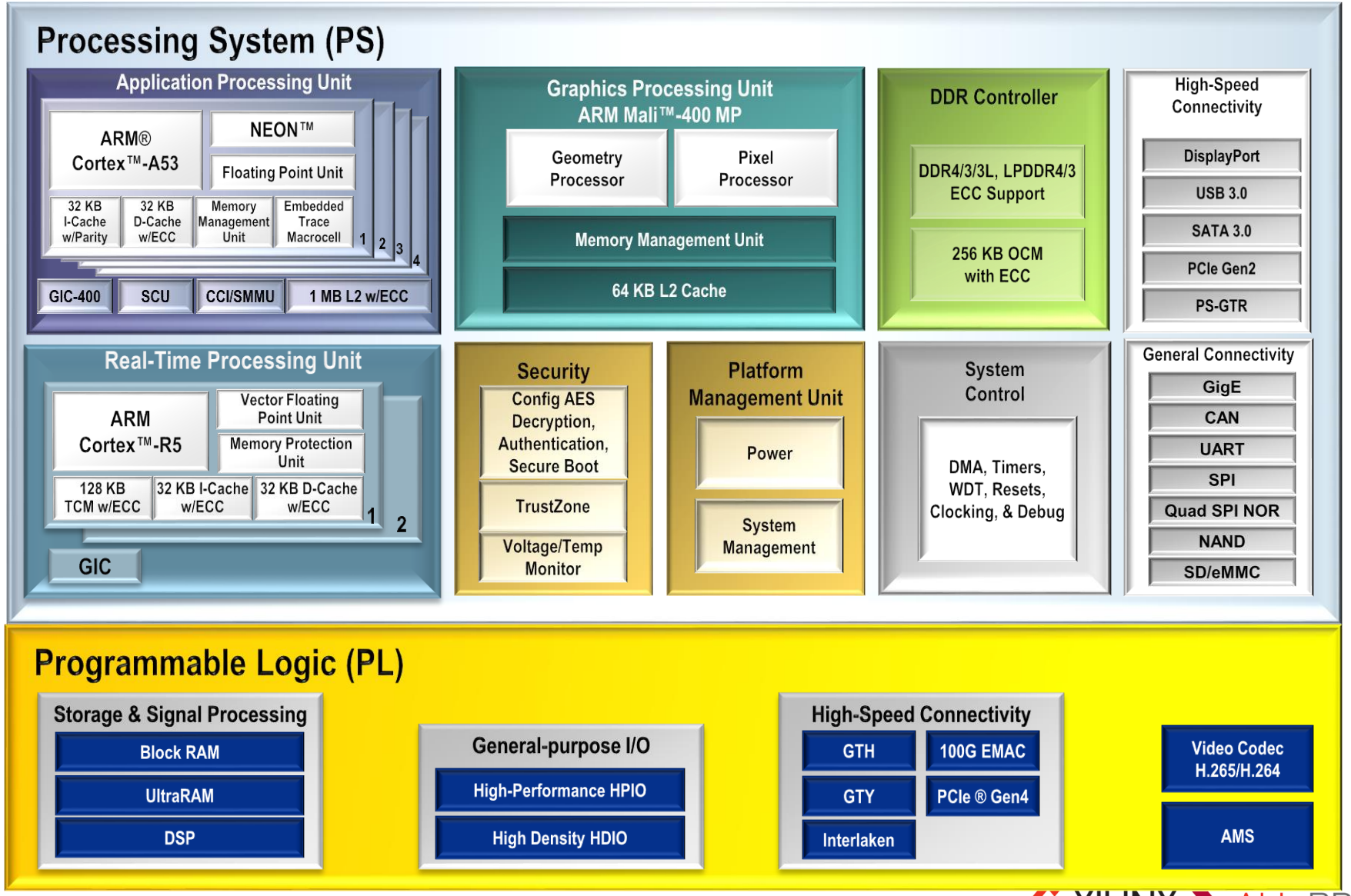
XILINX
ALL PROGRAMMABLE™

Experimenting with SYCL single-source post-modern C++ on Xilinx FPGA

Ronan Keryell & Lin-Ya Yu
Xilinx Research Labs
IWOCL DHPCC 2018/05/14

Deconstructivism in (hardware) architecture

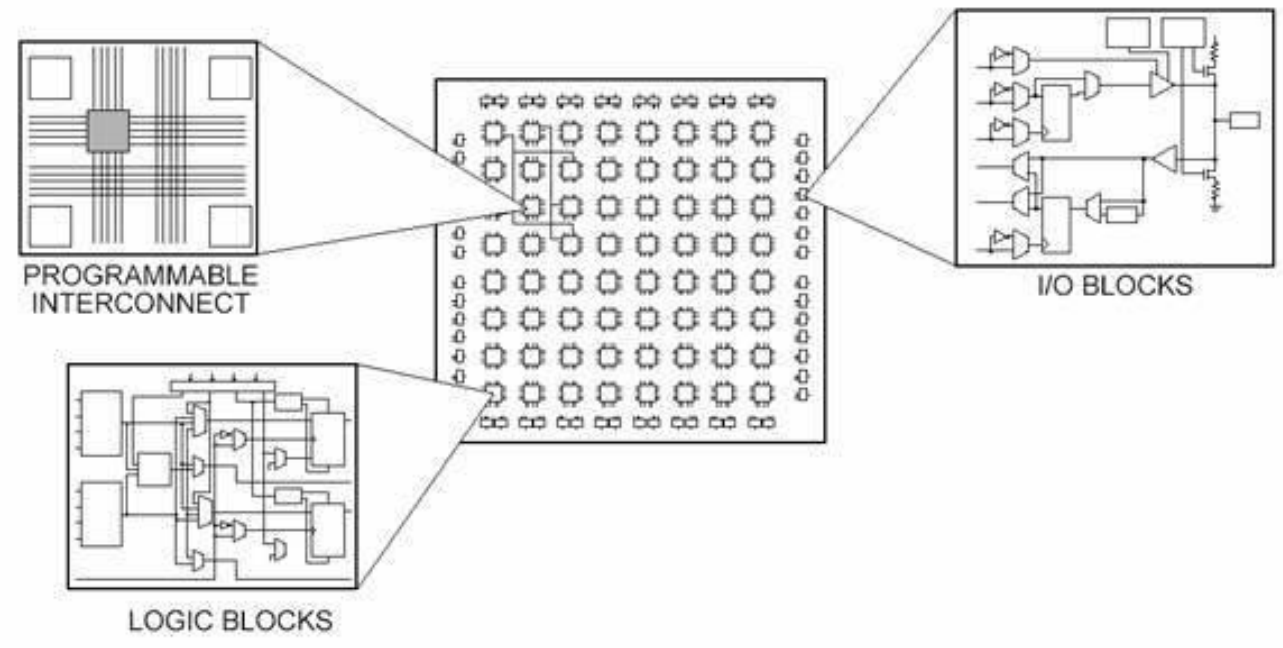
Typical modern MPSoC : Xilinx Zynq UltraScale+ MPSoC: All Programmable…



XILINX ALL PROGRAMMABLE.

Deconstructivism in post-modern architecture (for artistic reasons)

FPGA: extreme deconstructivism in (hardware) architecture



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

¿ How to program this ?

Position argument 1

Pick a language for unified heterogeneous computing?...

- Entry cost 

- ∃ thousands of dead parallel languages...

→Exit cost 



➔Use standard solutions with open source implementations

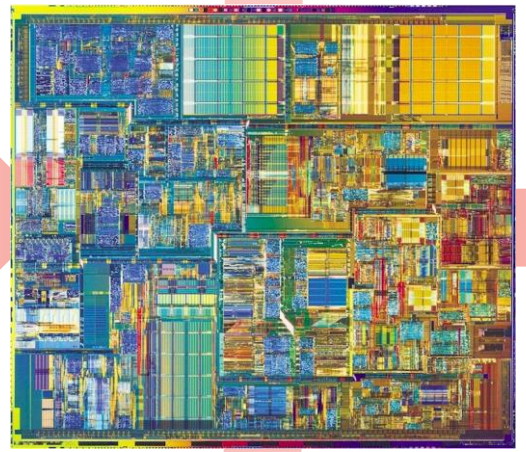
Khronos standards for heterogeneous systems

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

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

OpenGL|SC OpenVG OpenGL|ES Vulkan OpenGL OpenXR

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

¿ And what about post-modern C++ ?

Python/Modern C++/Old C++

- Python 3.6

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

- C++17

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

- 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 e =
v.begin();
      e != v.end();
      ++e)
  std::cout << *e << std::endl;
```

Back to Python…

- Python 3.x (interpreted):

```
def add(x, y):
  return x + y
print(add(2, 3))        # Output: 5
print(add("2", "3"))    # Output: 23
print(add(2, "Boom")) # Fails at run-time :-(
```

Modern C++ : like Python but with speed and type safety

- Python 3.x (interpreted):

```
def add(x, y):
  return x + y
print(add(2, 3))        # Output: 5
print(add("2", "3"))    # Output: 23
print(add(2, "Boom")) # Fails at run-time :-(
```

- Same in C++14 but compiled + static compile-time type-checking:

```
auto add = [] (auto x, auto y) { return x + y; };
std::cout << add(2, 3) << std::endl;        // 5
std::cout << add("2"s, "3"s) << std::endl; // 23
std::cout << add(2, "Boom"s) << std::endl; // Does not compile :-)
```

- Automatic type inference for terse generic programming and type safety
  - Without ~~template~~ keyword!

Generic variadic lambdas & operator interpolation

```
#include <iostream>
#include <string>
using namespace std::string_literals;
// Define an adder on anything.
// Use new C++14 generic variadic lambda syntax
auto add = [] (auto... args) {
  // Use new C++17 operator folding syntax
  return (... + args);
};
int main() {
 std::cout << "The result is: " << add(1, 2.5, 0xDeadBeefULL) << std::endl;
 std::cout << "The result is: " << add("begin"s, "end"s) << std::endl;
}
```

- Terse generic programming and type safety
  - Without ~~template~~ keyword!

Position argument 2: start with modern C++...

- Very successful & ubiquitous language
- Interoperability: seamless interaction with embedded world, libraries, OS...
- 2-line description by Bjarne Stroustrup
  - Direct mapping to hardware
  - Zero-overhead abstraction

⇒Unique existing position in embedded system to control the full stack!!!

- Full-stack ≡ combine both low-level aspects with high-level programming
  - Pay only for what you need
- Open-source production-grade compilers (GCC & Clang/LLVM) & tools
- Classes can be used to define Domain Specific Embedded Language (DSEL)
- Not directly targeting FPGA, GPU, DSP...
  - But extensible through classes (→ DSEL)

SYCL™

Matrix addition with producer/consumer tasks in 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 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;
}
```

- Type-safety & genericity
  - No type definition required inside kernels!
- Automatic data-transfers & compute overlap

SYCL 1.2.1 = Pure C++ based DSEL

- Modern C++ features available for OpenCL
  - Builds on the features of C++11, with additional support for C++14 and C++17
  - Enables ISO C++17 Parallel STL programs to be accelerated on OpenCL devices
  - Simplifies the porting of existing templated C++ Libraries and frameworks, i.e., Eigen, TensorFlow.....
- Generic heterogeneous computing model
  - On CPUs, GPUs, FPGAs……
  - Hierarchical parallelism
- Portability across platforms and compilers
- Single source programming model
  - Better type safety
  - Simpler and cleaner code
  - Compiled host and device code
- Asynchronous task graph
  - Describes implicitly with kernel tasks using buffers through accessors
  - Automatic overlap kernel executions and communications
- Only Queues needed to direct computations on devices
  - Runtime handles multiple platforms, devices, and context
- Provides the full OpenCL feature set
- Interoperability with multiple languages
  - OpenCL, OpenGL®, Vulkan®, OpenVX™, DirectX, and other vendor APIs, i.e., HLS C++ & RTL Xilinx FPGA kernels!
- Host fall-back
  - Easily develops and debugs applications on the host without a device
  - No specific compiler needed for experimenting on host

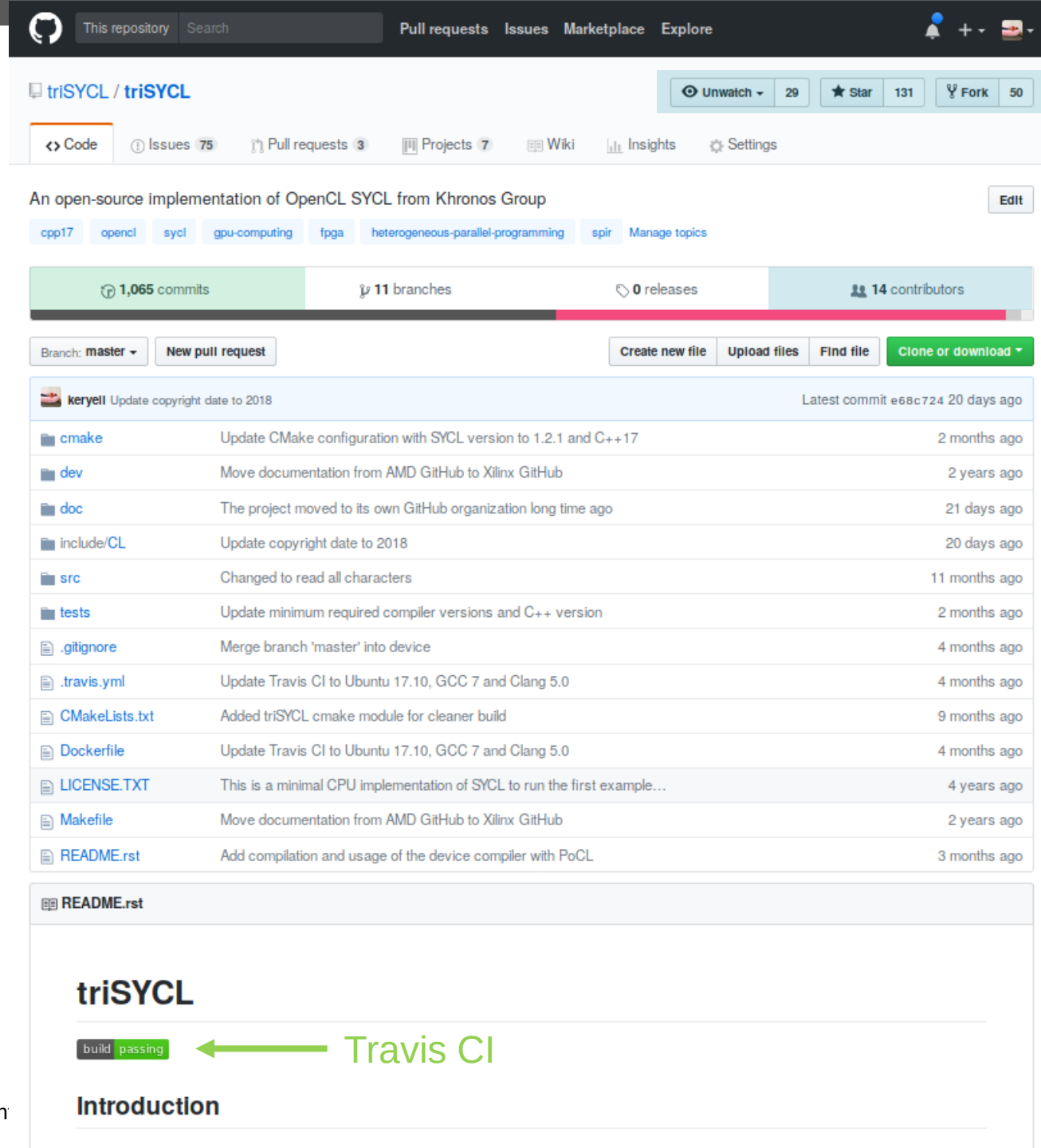
SYCL implementations

Known implementations of SYCL

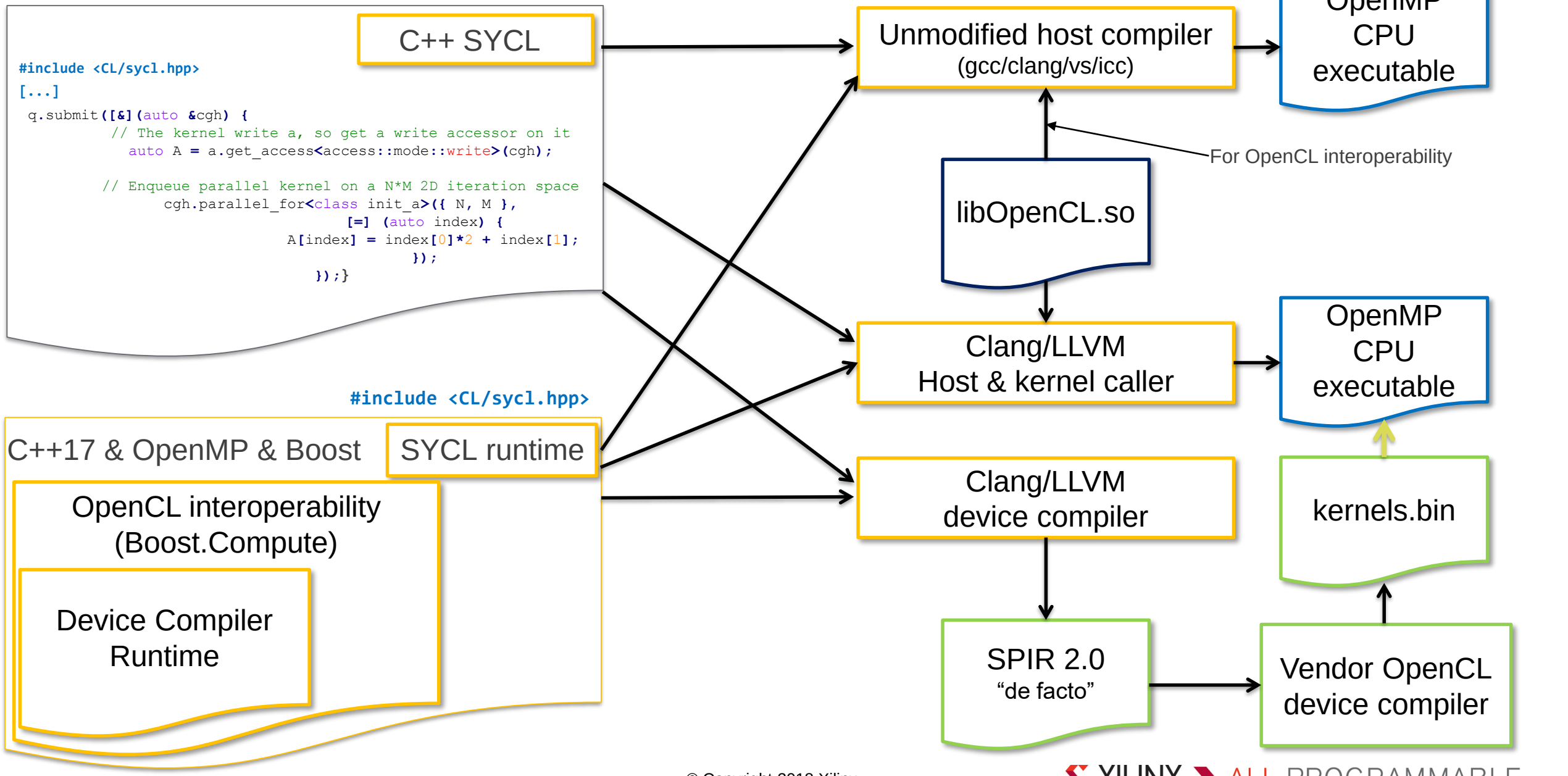
- ComputeCpp by Codeplay https://www.codeplay.com/products/computecpp
  - Most advanced SYCL 1.2.1 implementation, almost CTS compliant
  - Outlining compiler generating SPIR
  - Run on any OpenCL device and CPU, also prototype to Vulkan
  - Can run TensorFlow SYCL, Parallel STL, VisionCpp, SYCL BLAS…
- sycl-gtx https://github.com/ProGTX/sycl-gtx
  - Open source
  - No (outlining) compiler ➔ use some macros with different syntax
- triSYCL https://github.com/triSYCL/triSYCL

triSYCL

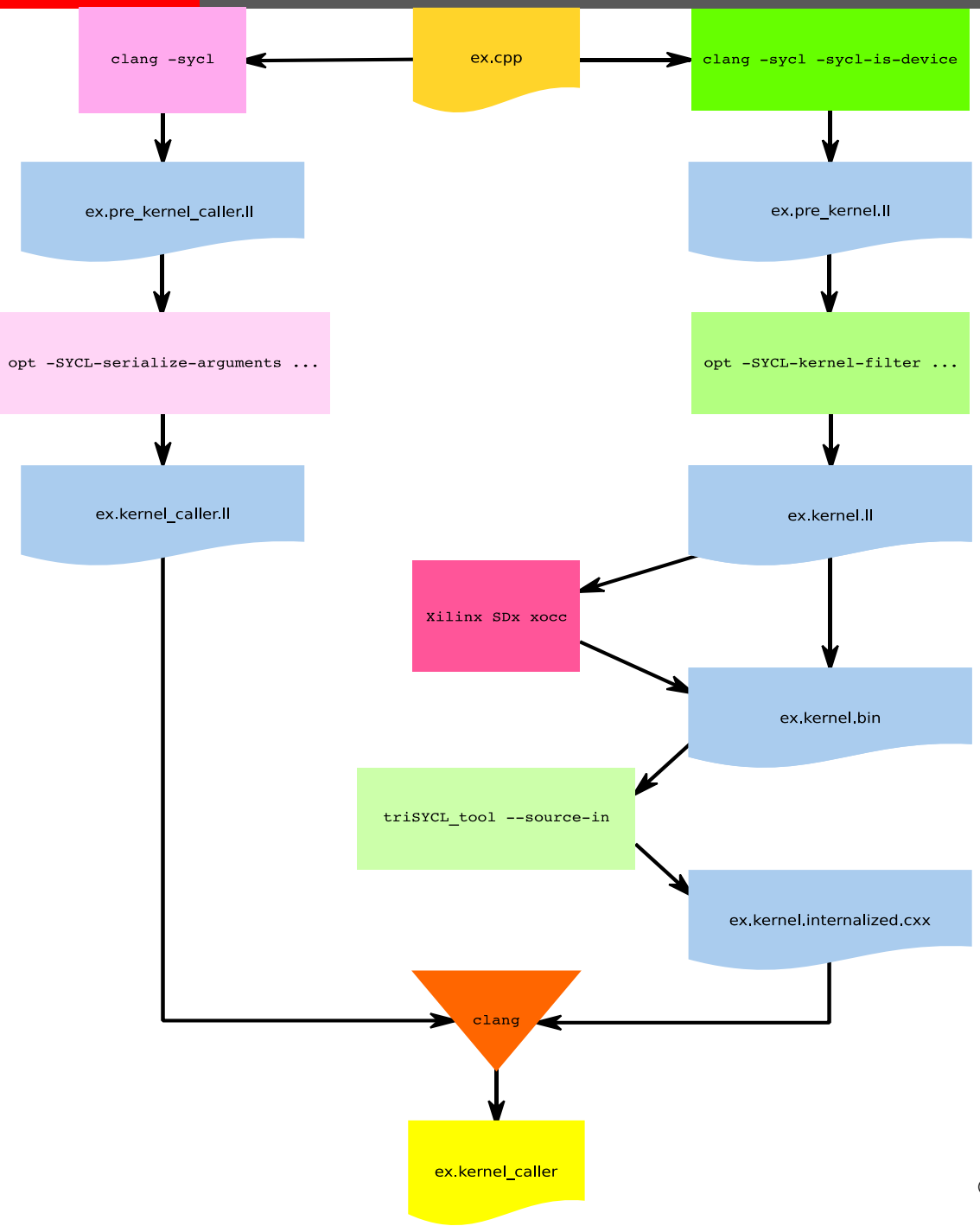
- Open Source SYCL 1.2.1/2.2
- Uses C++17 templated classes
- Used by Khronos to define the SYCL and OpenCL C++ standard
  - Languages are now too complex to be defined without implementing...
- On-going implementation started at AMD and now led by Xilinx
- https://github.com/triSYCL/triSYCL
- OpenMP for host parallelism
- Boost.Compute for OpenCL interaction
- Prototype of device compiler for Xilinx FPGA



https://github.com/triSYCL/triSYCL architecture



Low-level view of the device compiler workflow



- Modified Clang/LLVM 3.9
- Move to Clang/LLVM 7.0
  - Updated PoCL to support 7.0 version
- Makefile to control the compilation for now

Example of compilation to device (FPGA…)

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

#include <boost/test/minimal.hpp>

using namespace cl::sycl;

constexpr size_t N = 300;
using Type = int;

int test_main(int argc, char *argv[]) {
  buffer<Type> a { N };
  buffer<Type> b { N };
  buffer<Type> c { N };

  {
    auto a_b = b.get_access<access::mode::discard_write>();
    // Initialize buffer with increasing numbers starting at 0
    std::iota(a_b.begin(), a_b.end(), 0);
  }

  {
    auto a_c = c.get_access<access::mode::discard_write>();
    // Initialize buffer with increasing numbers starting at 5
    std::iota(a_c.begin(), a_c.end(), 5);
  }

  queue q { default_selector {} };
```

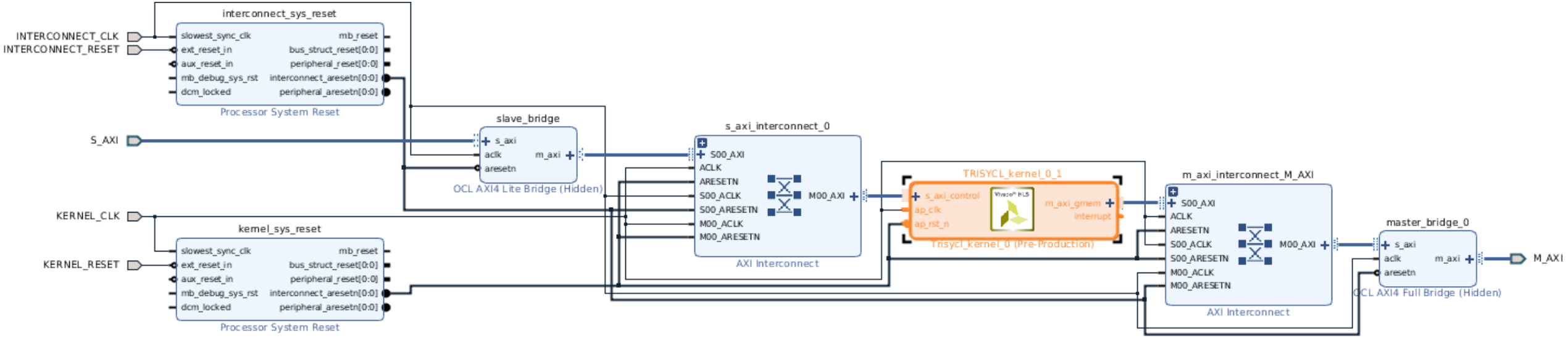
Current limitation: need to write this ➔

```
  // Launch a kernel to do the summation
  q.submit([&] (handler &cgh) {
      // Get access to the data
      auto a_a = a.get_access<access::mode::discard_write>(cgh);
      auto a_b = b.get_access<access::mode::read>(cgh);
      auto a_c = c.get_access<access::mode::read>(cgh);

      // A typical FPGA-style pipelined kernel
      cgh.single_task<class add>([=,
        d_a = drt::accessor<decltype(a_a)> { a_a },
        d_b = drt::accessor<decltype(a_b)> { a_b },
        d_c = drt::accessor<decltype(a_c)> { a_c }] {
        for (unsigned int i = 0 ; i < N; ++i)
          d_a[i] = d_b[i] + d_c[i];
      });
    });

  // Verify the result
  auto a_a = a.get_access<access::mode::read>();
  for (unsigned int i = 0 ; i < a.get_count(); ++i)
    BOOST_CHECK(a_a[i] == 5 + 2*i);

  return 0;
}
```

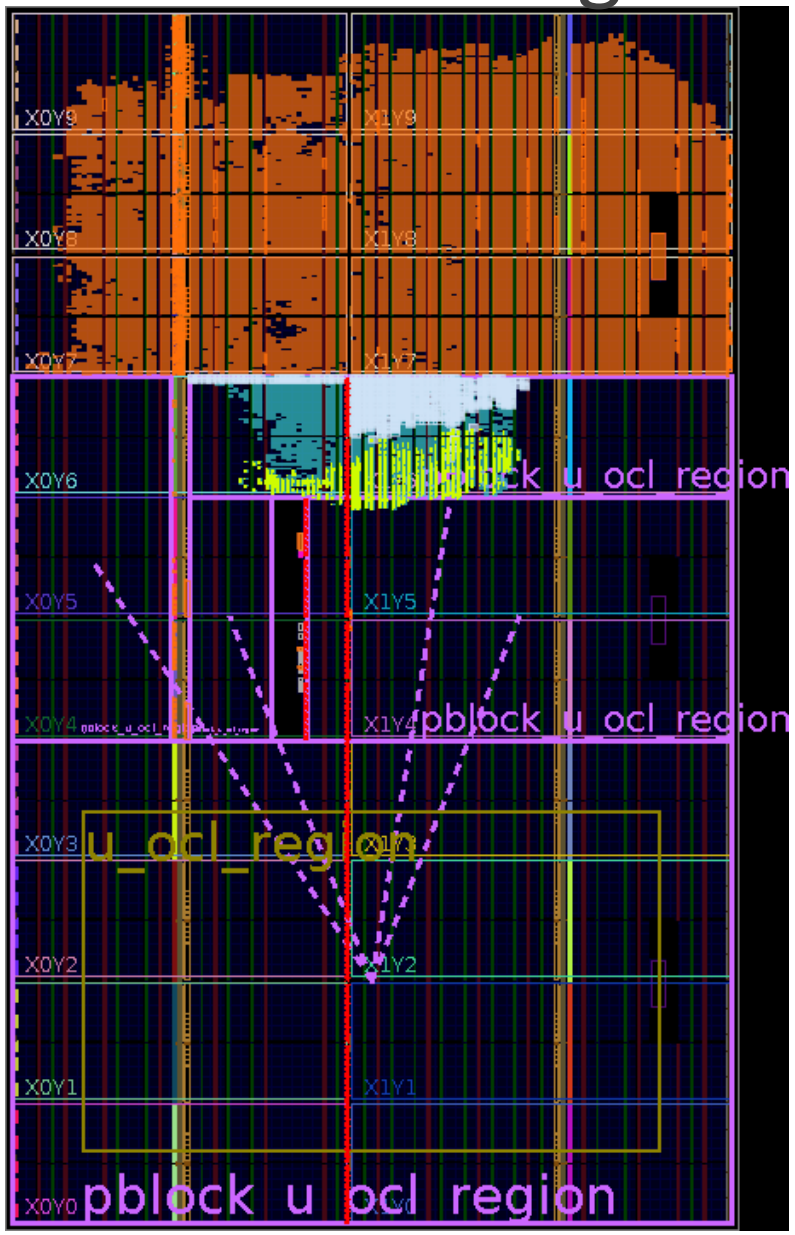
SPIR 2.0 “de facto” output with Clang 3.9.1

```
using; ModuleID = 'device_compiler/single_task_vector_add_drt.kernel.bc'

source_filename = "device_compiler/single_task_vector_add_drt.cpp"

target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"

target triple = "spir64"


declare i32 @__gxx_personality_v0(...)


; Function Attrs: noinline norecurse nounwind uwtable

define spir_kernel void @TRISYCL_kernel_0(i32 addrspace(1)* %f.0.0.0.val, i32 addrspace(1)*
%f.0.1.0.val, i32 addrspace(1)* %f.0.2.0.val) unnamed_addr #0 !kernel_arg_addr_space !3
!kernel_arg_type !4 !kernel_arg_base_type !4 !kernel_arg_type_qual !5 !kernel_arg_access_qual !6 {

entry:

  br label %for.body.i


for.body.i:                                       ; preds = %for.body.i, %entry

  %indvars.iv.i = phi i64 [ 0, %entry ], [ %indvars.iv.next.i, %for.body.i ]

  %arrayidx.i.i = getelementptr inbounds i32, i32 addrspace(1)* %f.0.1.0.val, i64 %indvars.iv.i

  %0 = load i32, i32 addrspace(1)* %arrayidx.i.i, align 4, !tbaa !7

  %arrayidx.i15.i = getelementptr inbounds i32, i32 addrspace(1)* %f.0.2.0.val, i64 %indvars.iv.i

  %1 = load i32, i32 addrspace(1)* %arrayidx.i15.i, align 4, !tbaa !7

  %add.i = add nsw i32 %1, %0

  %arrayidx.i13.i = getelementptr inbounds i32, i32 addrspace(1)* %f.0.0.0.val, i64 %indvars.iv.i

  store i32 %add.i, i32 addrspace(1)* %arrayidx.i13.i, align 4, !tbaa !7

  %indvars.iv.next.i = add nuw nsw i64 %indvars.iv.i, 1

  %exitcond.i = icmp eq i64 %indvars.iv.next.i, 300

  br i1 %exitcond.i, label %"_ZZZ9test_mainiPPcENK3$_1clERN2cl4sycl7handlerEENKUlvE_clEv.exit",
label %for.body.i

"_ZZZ9test_mainiPPcENK3$_1clERN2cl4sycl7handlerEENKUlvE_clEv.exit": ; preds = %for.body.i

  ret void

}


attributes #0 = { noinline norecurse nounwind uwtable "disable-tail-calls"="false" "less-precise-
fpmad"="false" "no-frame-pointer-elim"="false" "no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "stack-protector-buffer-size"="8"
"target-cpu"="x86-64" "target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false"
"use-soft-float"="false" }


!llvm.ident = !{!0}

!opencl.spir.version = !{!1}

!opencl.ocl.version = !{!2}


!0 = !{!"clang version 3.9.1 "}

!1 = !{i32 2, i32 0}

!2 = !{i32 1, i32 2}

!3 = !{i32 1, i32 1, i32 1}

!4 = !{!"int *", !"int *", !"int *"}

!5 = !{!"", !"", !""}

!6 = !{!"read_write", !"read_write", !"read_write"}

!7 = !{!8, !8, i64 0}

!8 = !{!"int", !9, i64 0}

!9 = !{!"omnipotent char", !10, i64 0}

!10 = !{!"Simple C++ TBAA"}
```

After Xilinx SDx 2017.2 xocc ingestion…



After Xilinx SDx 2017.2 xocc ingestion… FPGA layout!



FPGA-specific features and optimizations

Optimize for 1 element work-group

- Current device implementation focused on FPGA…
- Typical use case on FPGA
  - No concept of PE in CU… Everything can be generated
  - Typical application with only 1 work-group and 1 work-item
    - Any loop has to be explicitly added
    - Avoid OpenCL work-item offset overhead too…
  - triSYCL runtime generates 1 work-item per work-group + software work-group implementation in it
    - Better control
- Now generates `reqd_work_group_size(1, 1, 1)`
    - New LLVM pass `-reqd-workgroup-size-1`
    - Add metadata for work-group of size (1,1,1) in device compiler to reduce resources on device

```
; Function Attrs: noinline nounwind uwtable
define spir_kernel void @TRISYCL_kernel_0(i32 addrspace(1)* %f.0.0.0.val, i32 addrspace(1)* %f.0.1.0.val) unnamed_addr #1
!kernel_arg_addr_space !3 !kernel_arg_type !4 !kernel_arg_base_type !4 !kernel_arg_type_qual !5 !kernel_arg_access_qual !6
!reqd_work_group_size !7 {
```

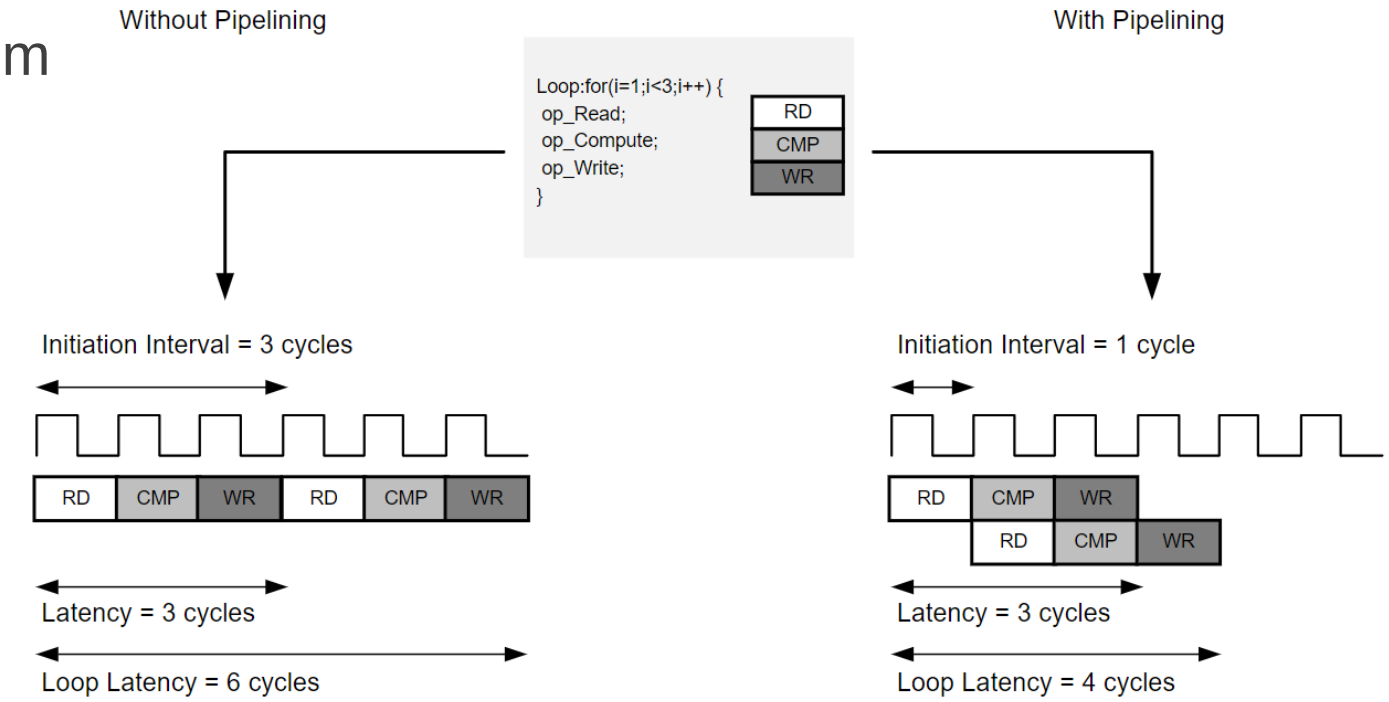
```
!7 = !{i32 1, i32 1, i32 1}
```

Let's try some SYCL vendor extensions…

- SYCL reserves `cl::sycl::vendor` namespace for some *vendor*
- Why not `cl::sycl::vendor::xilinx` to experiment with FPGA extensions?

Pipelining loops on FPGA

- Loop instructions sequentially executed by default
  - Loop iteration starts only after last operation from previous iteration
  - Sequential pessimism → idle hardware and loss of performance ☹
- → Use loop pipelining for more parallelism



- Efficiency measure in hardware realm: Initiation Interval (II)
  - Clock cycles between the starting times of consecutive loop iterations
  - II can be 1 if no dependency and short operations

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();
};
```

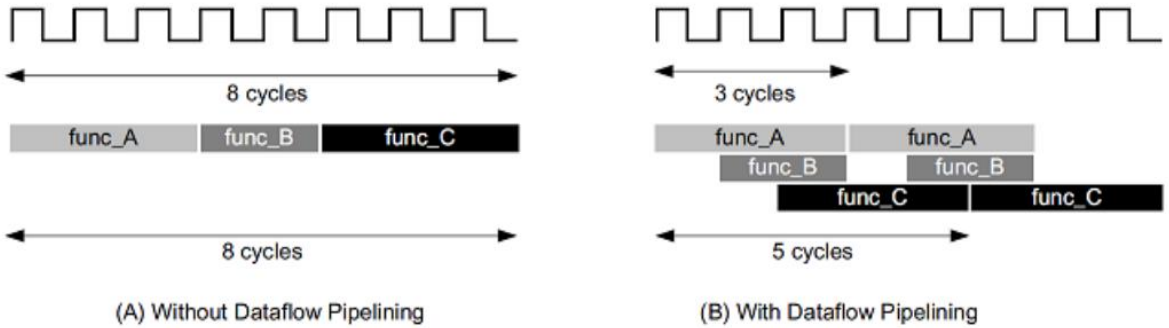
```
#ifdef TRISYCL_DEVICE
extern "C" {
  /// SSDM Intrinsics: dataflow operation
  void  ssdm_op_SpecDataflowPipeline(...)   attribute   ((nothrow, noinline, weak));
  /// SSDM Intrinsics: pipeline operation
  void _ssdm_op_SpecPipeline(...) __attribute__ ((nothrow, noinline, weak));
  /// SSDM Intrinsics: 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! ☺

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

```
void top (a,b,c,d) {
  ...
  func_A(a,b,i1);
  func_B(c,i1,i2);
  func_C(i2,d)

  return d;
}
```



- 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>([=,
                            d_a = drt::accessor<decltype(a_a)> { a_a },
                            d_b = drt::accessor<decltype(a_b)> { a_b }
                            ] {
                             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);
                               });
                           });
```

```
/** Apply dataflow execution on functions or loops

    With this mode, Xilinx tools analyze the dataflow dependencies
    between sequential functions or loops and create channels (based
    on ping-pong RAMs or FIFOs) that allow consumer functions or loops
    to start operation before the producer functions or loops have
    completed.

    This allows functions or loops to operate in parallel, which
    decreases latency and improves the throughput.

    \param[in] f is a function that functions or loops in f will be executed
    in a dataflow manner.
*/
auto dataflow = [] (auto functor) noexcept {
  /* SSDM instruction is inserted before the argument functor to guide xocc to
     do dataflow. */
  _ssdm_op_SpecDataflowPipeline(-1, "");
  functor();
};
```

- Use native C++ construct instead of alien `#pragma` or attributes (vendor OpenCL or HLS C++…)

```
#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! ☺

Another motivation for single-source feature…

- Limitations of OpenCL pointed for example by
  - "A Case for Better Integration of Host and Target Compilation When Using OpenCL for FPGAs." *Taylor Lloyd, Artem Chikin, Erick Ochoa, Karim Ali, José Nelson Amaral.* University of Alberta. FPL/FSP 2017 (27th International Conference on Field-Programmable Logic and Applications / Workshop on FPGAs for Software Programmers)

"Major Field-Programmable Gate Array (FPGA) vendors, such as Intel and Xilinx, provide toolchains for compiling Open Computing Language (OpenCL) to FPGAs. However, the separate host and device compilation approach advocated by OpenCL hides compiler optimization opportunities that can dramatically improve FPGA performance. This paper demonstrates the advantages of combined host and device compilation for OpenCL on FPGAs by presenting a series of transformations that require inter-compiler communication."

- https://ieeexplore.ieee.org/document/8084546
- www.fsp-workshop.org/2017/slides/2017_FSP_Combined_Compilation.pdf

Single-source brings more optimization

- Kernel code optimized according to host parameter value or data type
  - A host constant scalar can be inlined into the kernel
    - Save one API call to send the parameter to the kernel
  - A host constant array/tensor can be inlined into the kernel
    - Save one API call to send the parameter to the kernel
    - Replace memory access by direct constant computation
  - Dead-code elimination
  - …
- Single-source allows kernel fusion with (manual) metaprogramming
  - Kernel fusion heavily used in TensorFlow for example
  - ➔Kernel fusion leads to better performance by reducing the launch & memory overhead

➔Can lead to better performance compared to split-source (OpenCL, HLS C/C++…)

Single source SYCL

```
1  #include <CL/sycl.hpp>
2  #include <iostream>
3  #include <numeric>
4
5  #include <boost/test/minimal.hpp>
6
7  using namespace cl::sycl;
8
   …
13 constexpr size_t NUM_ROWS = 64;
14 constexpr size_t ELE_PER_ROW = 64;
15 constexpr size_t BLOCK_SIZE = NUM_ROWS * ELE_PER_ROW;
16 using Type = int;
17
18 template<typename T, typename U>
19 void readInput(T *buffer_in, const U &d_b) {
20 for(int i = 0; i < NUM_ROWS; ++i)
21   for (int j = 0; j < ELE_PER_ROW; ++j)
22     xilinx::pipeline([&] {
23           buffer_in[ELE_PER_ROW*i+j] = d_b[ELE_PER_ROW*i+j];
24     });
25 }
   …
49 int test_main(int argc, char *argv[]) {
50 constexpr int alpha = 3;
```

```
51 buffer<Type> a { BLOCK_SIZE };
52 buffer<Type> b { BLOCK_SIZE };
   …
55 // Initialize buffer with increasing numbers starting at 0
56 auto a_b = b.get_access<access::mode::discard_write>();
57 std::iota(a_b.begin(), a_b.end(), 0);
   …
64 // Construct the queue from the default OpenCL one.
65 queue q { default_selector {} };
66 // Launch a kernel to do the summation
67 q.submit([&] (handler &cgh) {
68   // Get access to the data
69   auto a_a = a.get_access<access::mode::discard_write>(cgh);
70   auto a_b = b.get_access<access::mode::read>(cgh);
71
72   // A typical FPGA-style pipelined kernel
73   cgh.single_task<class add>([=,
74           d_a = drt::accessor<decltype(a_a)> { a_a },
75           d_b = drt::accessor<decltype(a_b)> { a_b } ] {
           …
82           xilinx::dataflow([&] {
83                   readInput(buffer_in, d_b);
84                   compute(buffer_in, buffer_out, alpha);
85                   writeOutput(buffer_out, d_a);
86                   });    …
   }
```

Single source SYCL C++ syntax summary

- OpenCL C → SYCL C++
- 4+ files ⟶ 1 file
- 250+ lines ⟶ 98 lines
- Template functions, even kernels make things much more easy
  - Reuse the code!
- `constexpr` variables
  - Let compiler do the global host-device optimization for you!

Hardware & Software testing context

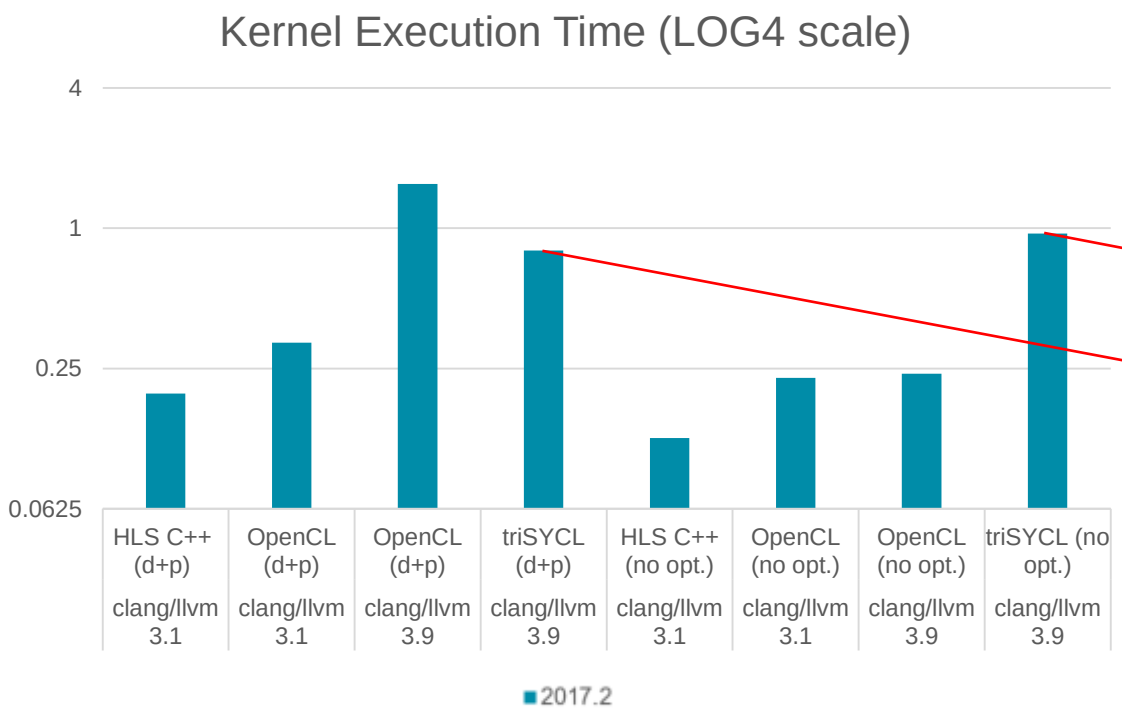
- CPU (Intel core i7-6700)
- FPGA (ADM-PCIE-7V3)
- Linux Ubuntu 17.04
- triSYCL device compiler using Clang/LLVM 3.9
- 2017.2 Xilinx xocc compiler using Clang/LLVM 3.1
- 2017.2 Xilinx xocc compiler using Clang/LLVM 3.9
- 2017.4 Xilinx xocc compiler using Clang/LLVM 3.1
- 2017.4 Xilinx xocc compiler using Clang/LLVM 3.9
- Xilinx SDx OpenCL runtime 2017.2

Experiments :

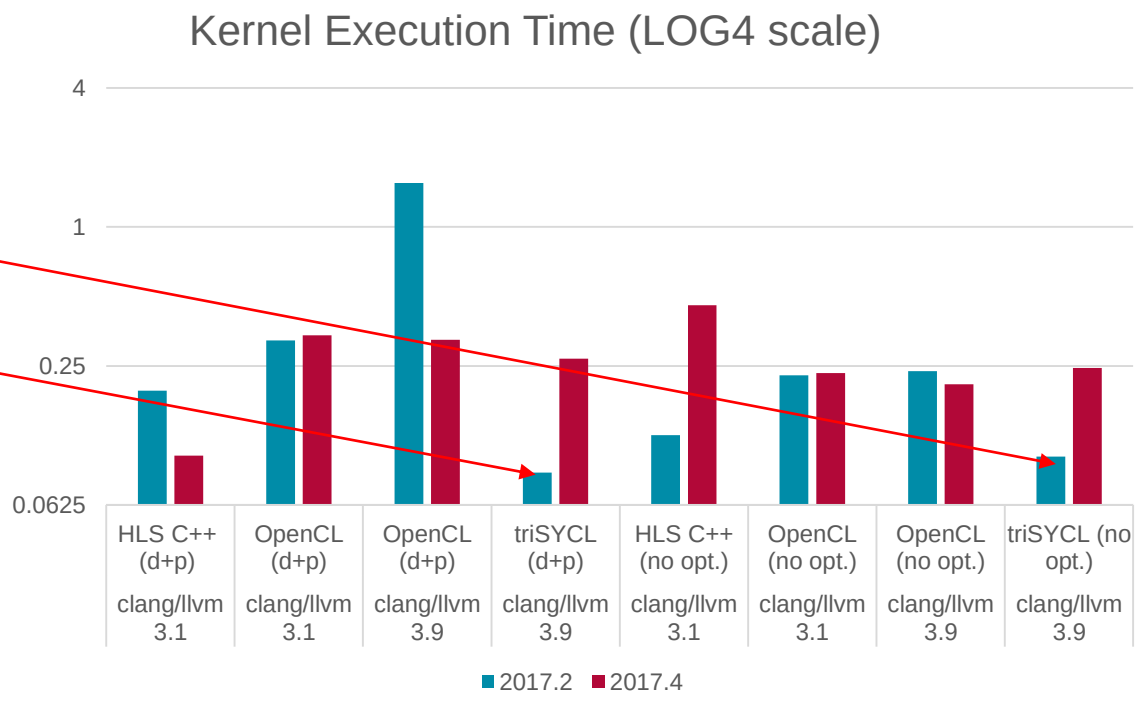
- triSYCL (optimized)
  - Single-source triSYCL on FPGA with Xilinx-specific optimizations
- triSYCL (non optimized)
  - Single-source triSYCL on FPGA
- HLS C++ (optimized) as built-in OpenCL kernel
  - Xilinx HLS C++ on FPGA with Xilinx-specific optimizations
- HLS C++ (non optimized) as built-in OpenCL kernel
  - Xilinx HLS C++ on FPGA
- OpenCL (optimized)
  - Targeting Xilinx OpenCL on FPGA with Xilinx-specific optimizations
- OpenCL (non optimized)
  - Targeting Xilinx OpenCL on FPGA

Performance on FPGA

- Read/Write row of 2D Array



November 2017…



February 2018

d: dataflow p: pipelining no opt: non optimized

Partitioning memories

- 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

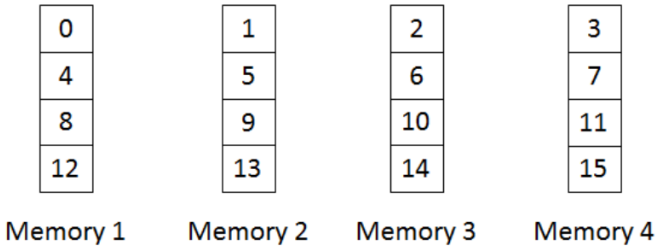


Figure 7-1: **Physical Layout of Buffer After Cyclic Partitioning**

- 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

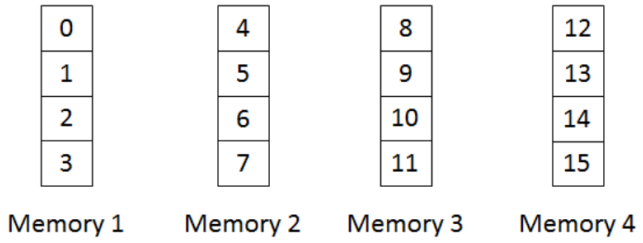


Figure 7-2: **Physical Layout of Buffer After Block Partitioning**

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

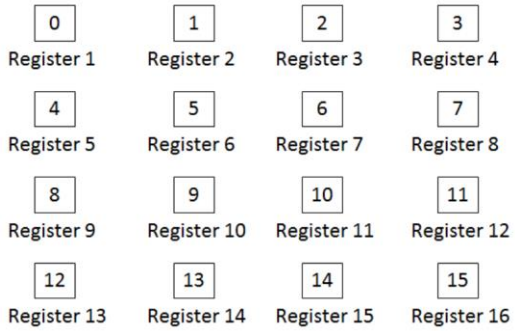


Figure 7-3: **Physical Layout of Buffer After Complete Partitioning**

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
…
```

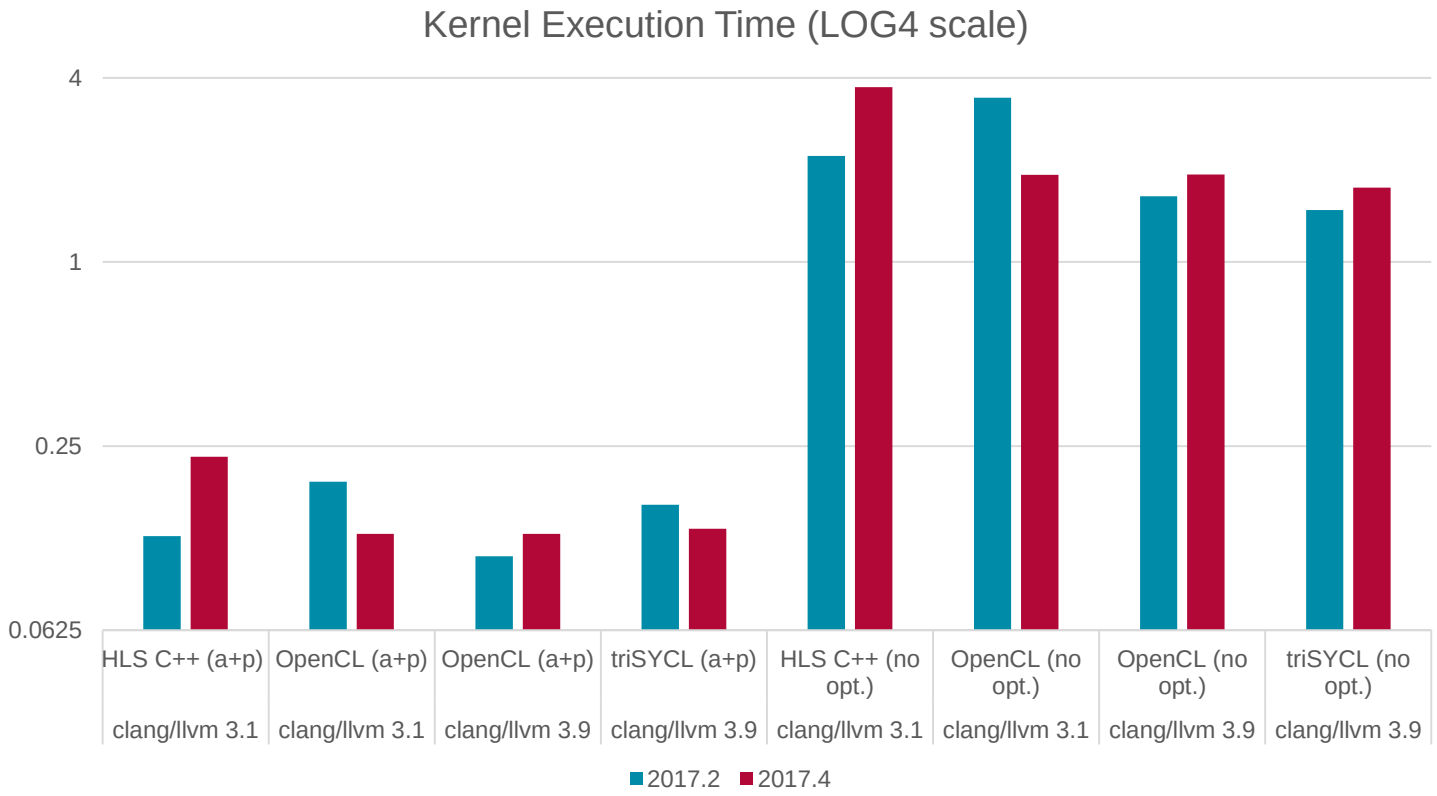
Xilinx HLS C++

```
//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

Performance on FPGA

- Array Block and Cyclic Partitioning with Matrix Multiplication



a: array partition p: pipelining no opt: non optimized

Conclusion

Conclusion: do it the standard way!

- C++ is used by millions of programmers and billions of end-users
  - Runs the world infrastructure!
  - 2018 Draper prize: Bjarne Stroustrup, for conceptualizing and developing the C++ programming language
- Real final applications for embedded systems:
  - Crazily optimized both on hosts and accelerators
  - Need to finely control even more heterogeneity in the future…
- Single-source & sYstem-wide C++ Language is compelling for the full application
- SYCL can cope with FPGA extensions for better control and performance
  - Pipelining, data-flow execution, fixed-point & arbitrary precision
- Pure C++ → easy implementation & CPU emulation
- SYCL can target full stack in a modern MP-SoC
  - Seamless integration of CPU, GPU, OpenAMP, MicroBlaze, accelerators… Not only OpenCL

And now we have a mascot…

Thanks to Dominic Agoro-Ombaka during Khronos F2F Montréal ☺ !

Bonus slides

Generic adder in 25 lines of SYCL & C++17

```
auto generic_adder = [] (auto... inputs) {
  auto a = boost::hana::make_tuple(buffer<typename decltype(inputs)::value_type>
    { std::begin(inputs),
      std::end(inputs) }...);
  auto compute = [] (auto args) {
    // f(... f(f(f(x1, x2), x3), x4) ..., xn)
    return boost::hana::fold_left(args, [] (auto x, auto y) { return x + y; });
  };
  auto size = a[0_c].get_count();
  auto pseudo_result = compute(boost::hana::make_tuple(*std::begin(inputs)...));
  using return_value_type = decltype(pseudo_result);
  buffer<return_value_type> output { size };
  queue {}.submit([&] (handler& cgh) {
      auto ka = boost::hana::transform(a, [&] (auto b) {
          return b.template get_access<access::mode::read>(cgh);
        });
      auto ko = output.template get_access<access::mode::discard_write>(cgh);

      cgh.parallel_for<class gen_add>(size, [=] (id<1> i) {
          auto operands = boost::hana::transform(ka, [&] (auto acc) {
              return acc[i];
            });
          ko[i] = compute(operands);
        });
  });
  return output.template get_access<access::mode::read_write>();
};
int main() {
  std::vector<int> u { 1, 2, 3 };
  std::vector<float> v { 5, 6, 7 };
  for (auto e : generic_adder(u, v))
    std::cout << e << ' ';
  std::cout << std::endl;
  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 };
  for (auto e : generic_adder(a, b, c))
    std::cout << e << ' ';
  std::cout << std::endl;
  return 0;
}
```

6 8 10

-52 14 1.75 17.125

Generic executor in 25 lines of SYCL & C++17

```
auto generic_executor = [] (auto op, auto... inputs) {
  auto a = boost::hana::make_tuple(buffer<typename decltype(inputs)::value_type>
    { std::begin(inputs),
      std::end(inputs) }...);
  auto compute = [&] (auto args) {
    // f(... f(f(f(x1, x2), x3), x4) ..., xn)
    return boost::hana::fold_left(args, op);
  };
  auto pseudo_result = compute(boost::hana::make_tuple(*std::begin(inputs)...));
  using return_value_type = decltype(pseudo_result);
  auto size = a[0_c].get_count();
  buffer<return_value_type> output { size };
  queue {}.submit([&] (handler& cgh) {
      auto ka = boost::hana::transform(a, [&] (auto b) {
          return b.template get_access<access::mode::read>(cgh);
        });
      auto ko = output.template get_access<access::mode::discard_write>(cgh);
      cgh.parallel_for<class gen_add>(size, [=] (id<1> i) {
          auto operands = boost::hana::transform(ka, [&] (auto acc) {
              return acc[i];
            });
          ko[i] = compute(operands);
        });
  });
  return output.template get_access<access::mode::read_write>();
};
int main() {
  std::vector<int> u { 1, 2, 3 };
  std::vector<float> v { 5, 6, 7 };
  for (auto e : generic_executor([] (auto x, auto y) { return x + y; }, u, v))
    std::cout << e << ' ';
  std::cout << std::endl;

  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 };
  for (auto e : generic_executor([] (auto x, auto y) { return 3*x - 7*y; },
                                 a, b, c))
    std::cout << e << ' ';
  std::cout << std::endl;
  return 0;
}
```

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

Modern metaprogramming as… hardware design tool

- Alternative implementation of

```
auto compute = [] (auto args) {
  return boost::hana::fold_left(args, [] (auto x, auto y) { return x + y; });
}; // 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

  ➔Various space/time/power trade-offs directly using metaprogramming! ☺

- Metaprogramming allows various implementations according to the types, sizes…
  - Kernel fusion, pipelined execution…
  - Codeplay VisionCpp, Eigen kernel fusion, Halide DSL…
  - In sync with C++ proposal on executors & execution contexts
- C++2a introspection & metaclasses will allow quite more!
  - Generative programming… http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0707r2.pdf
    - Mind-blowing… https://www.youtube.com/watch?v=4AfRAVcThyA
  - Express directly and specialize code for each PE of a TPU for example
- Imagine if SystemC was invented with C++2a instead of C++98…

ISO C++ SG7

- Study group 7 **Compile-time programming** (chair Chandler Carruth, Google)
- 2 different approaches already implemented before ratification
  - Experiment ahead & give feedback to committee
  - First focused on compile-time reflection capabilities, then expanded to compile-time programming in general
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0707r3.pdf

`$T` : metaclass reflecting type T, `$expr` : reflect expression `expr`

```
constexpr { // execute this at compile time
  for... (auto m : $T.variables()) // examine each member variable m in T
  if (m.name() == "xyzzy") // if there is one with name "xyzzy"
    -> { int plugh; } // then inject also an int named "plugh"
}
```

→ Can push SYCL Next to stratospheric levels ☺

  - Adapting datastructures to hardware specs… Automatic AOS/SOA transformations, remapping…

Experimenting with fixed-point types

Experimenting with fixed-point types

- Fixed-point implementation relies on existing ISO C++ CNL implementation
  - Interesting proposal of layered C++ library for fixed-point types by John McFarlane https://github.com/johnmcfarlane/cnl
  - Work well with triSYCL on CPU: the virtue of SYCL as pure C++ ☺
  - Can also works with Xilinx HLS C++ type `ap_int<>` in emulation on CPU
- Issue with triSYCL compiler about class constructors & address spaces
  - In Clang C++ object default constructors have parameter only in address space 0
  - Constructors, assignments, template deduction, overloading… need to work with other address spaces
    - Global, private, constant, generic
  - Looking at how OpenCL C++ solves these issues
  - Interesting to factorize out this code while upstreaming OpenCL C++ and SYCL compilers
  - Waiting for rebasing triSYCL on Clang/LLVM 7/ToT

Example of using fixed_point type in triSYCL

```
#include <CL/sycl.hpp>
  …
#include <cnl/fixed_point.h>
using namespace cl::sycl;
using namespace cnl;

constexpr size_t N = 1024;
using Type = fixed_point<char, -4>;
using HighType = fixed_point<int, -8>;

int test_main(int argc, char *argv[]) {
  …
  buffer<HighType> a { N };
  buffer<Type> b { std::begin(source_input),
          std::end(source_input) };
  buffer<Type> c { std::begin(source_input_1),
          std::end(source_input_1) };
  …
  for (int j = 0; j < 100; j++) {
    // Launch a kernel to do the operations
```

```
q.submit([&] (handler &cgh) {
  // Get access to the data
  auto a_a = a.get_access<
    access::mode::discard_write>(cgh);
  …
  // A typical FPGA-style pipelined kernel
  cgh.single_task<class add>([=,
    d_a = drt::accessor<decltype(a_a)> { a_a },
    d_b = drt::accessor<decltype(a_b)> { a_b },
    d_c = drt::accessor<decltype(a_c)> { a_c }] {
    decltype(d_a)::value_type sum = 0.0;
    for (unsigned int i = 0 ; i < N; ++i)
      sum += d_c[i];
    for (unsigned int i = 0 ; i < N; ++i)
      d_a[i] = d_b[i] * sum;
    });
  });
}
  …
return 0;
}
```

Hardware & Software testing context

- CPU (Intel core i7-6700)
- FPGA (ADM-PCIE-7V3)
- Linux Ubuntu 17.10
- triSYCL device compiler using Clang/LLVM 3.9
- 2017.2 Xilinx xocc compiler using Clang/LLVM 3.1
- 2017.2 Xilinx xocc compiler using Clang/LLVM 3.9
- Xilinx SDx OpenCL runtime 2017.2

Experiments :

- triSYCL fixed-point
  - Single-source triSYCL with CNL fixed-point type on FPGA
- triSYCL float
  - Single-source triSYCL with floating-point type on FPGA
- triSYCL ap_fixed interoperability
  - Interoperability with HLS C++ kernel using ap_fixed type on FPGA
- triSYCL float interoperability
  - Interoperability with HLS C++ kernel using floating-point type on FPGA
- HLS C++ float
  - Xilinx HLS C++ with floating-point type on FPGA
- HLS C++ ap_fixed
  - Xilinx HLS C++ with ap_fixed type on FPGA

Comparing Implementation Results for the Designs

Language	Type	LUT	LUTMem	REG	BRAM	DSP
triSYCL	float	2898	1348	3131	1	5
triSYCL	fixed_point	2469	1134	2752	1	1
HLS C++	float	1450	75	2517	1	5
HLS C++	ap_fixed	1651	76	2666	1	1