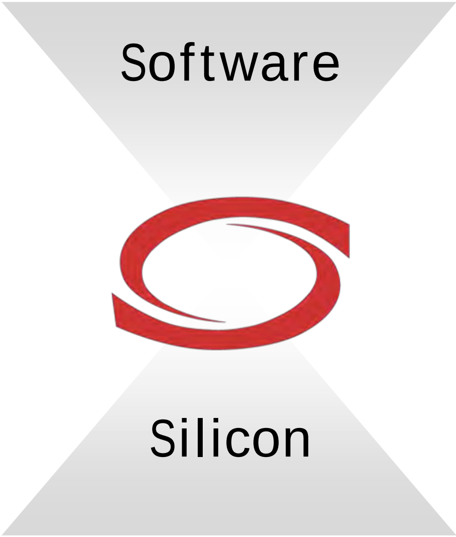
KHRONOS™
GROUP

SYCL™

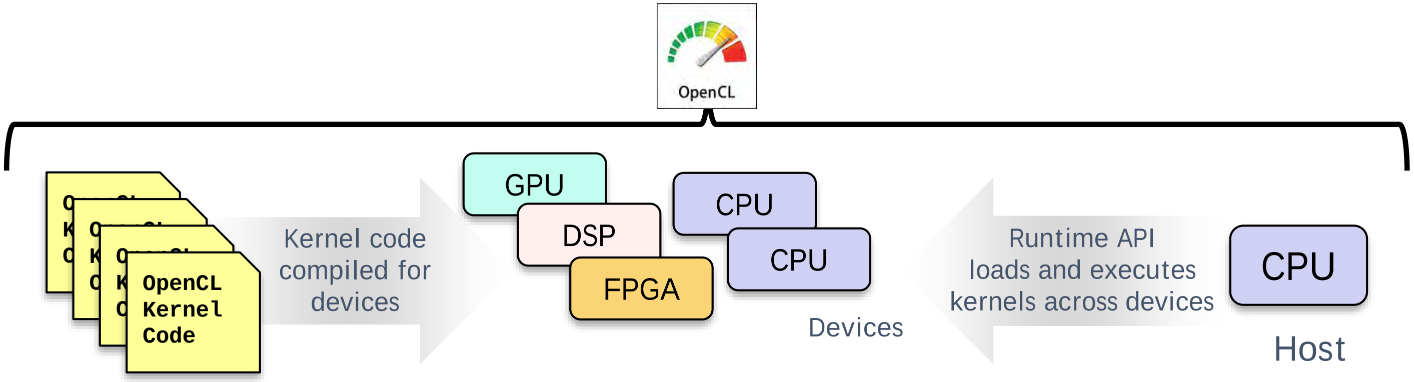
Khronos APIs Connect Software to Silicon



Khronos is an International Industry Consortium of over 100 companies creating royalty-free, open standard APIs to enable software to access hardware acceleration for 3D graphics, Virtual and Augmented Reality, Parallel Computing, Neural Networks and Vision Processing

OpenCL - Low-level Parallel Programing

- Low-level, explicit programming of heterogeneous parallel compute resources
  - One code tree can be executed on CPUs, GPUs, DSPs and FPGA …
- OpenCL C or C++ language to write kernel programs to execute on any compute device
  - Platform Layer API - to query, select and initialize compute devices
  - Runtime API - to build and execute kernels programs on multiple devices
- The programmer gets to control:
  - What programs execute on what device
  - Where data is stored in various speed and size memories in the system
  - When programs are run, and what operations are dependent on earlier operations



OpenCL 2.2 Released in May 2017

2011

OpenCL 1.2

Becomes industry baseline for heterogeneous parallel computing

2013

OpenCL 2.0

Enables new class of hardware
SVM
Generic Addresses
On-device dispatch

2015

OpenCL 2.1
SPIR-V 1.0

SPIR-V in Core
Kernel Language Flexibility

2017

OpenCL 2.2
SPIR-V 1.2

OpenCL C++ Kernel Language
Static subset of C++14
Templates and Lambdas

SPIR-V 1.2
OpenCL C++ support

Pipes
Efficient device-scope communication between kernels

Code Generation Optimizations
- Specialization constants at SPIR-V compilation time
- Constructors and destructors of program scope global objects
- User callbacks can be set at program release time

https://www.khronos.org/opencl/

OpenCL Ecosystem

Hardware Implementers
Desktop/Mobile/Embedded/FPGA

OpenCL 2.2 - Top to Bottom C++

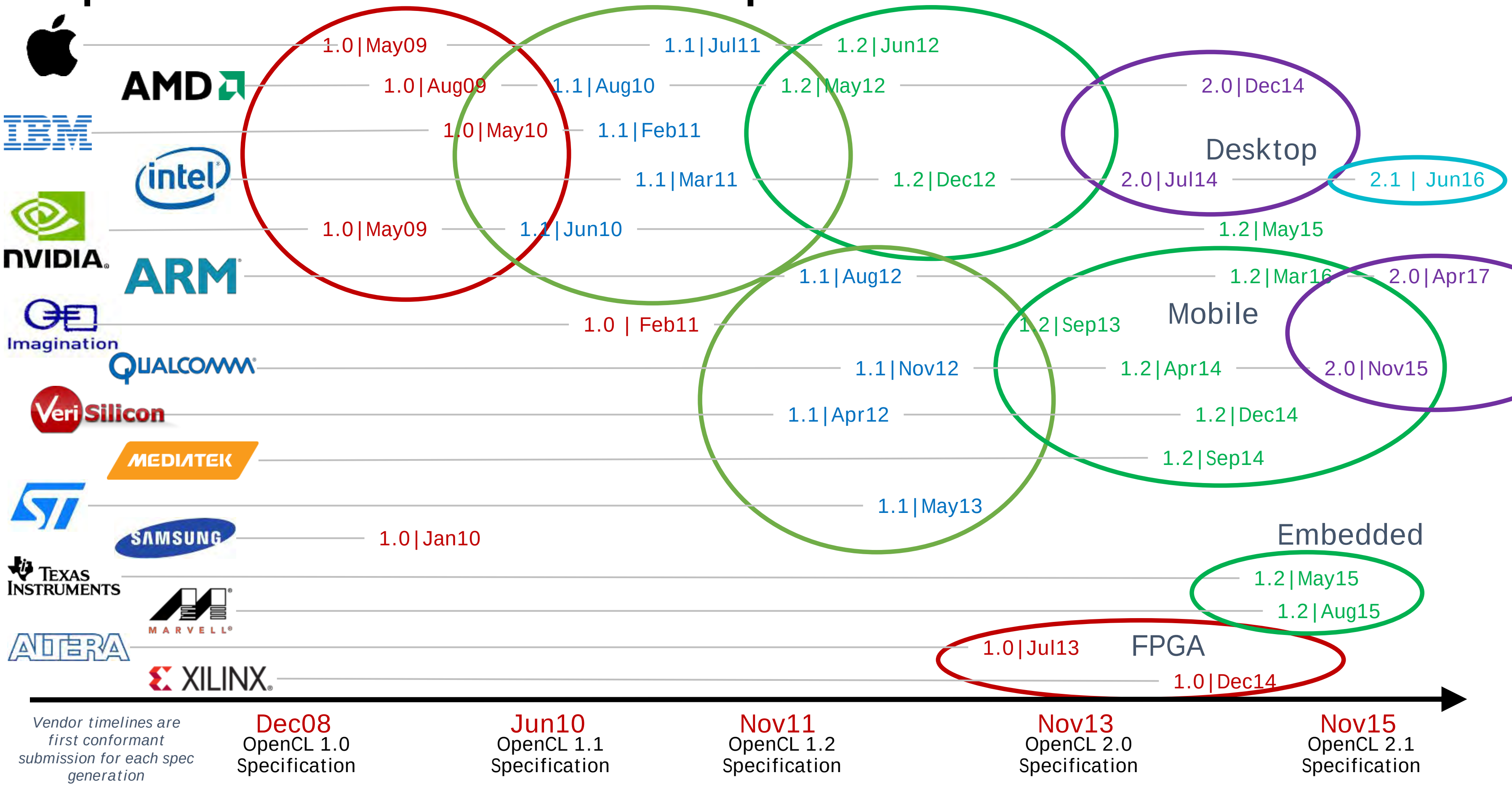
Single Source C++ Programming

Core API and Language Specs

Portable Kernel Intermediate Language

100s of applications using OpenCL acceleration
Rendering, visualization, video editing, simulation, image processing, vision and neural network inferencing

OpenCL Conformant Implementations



OpenCL as Language/Library Backend

Caffe — C++ based Neural network framework

Halide — Language for image processing and computational photography

C++ AMP (Accelerated Massive Parallelism with Microsoft Visual C++) — MulticoreWare open source project on Bitbucket

SYCL — Single Source C++ Programming for OpenCL

aparapi — Java language extensions for parallelism

OpenCV — Vision processing open source project

OpenACC (Directives for Accelerators) — Compiler directives for Fortran, C and C++

TensorFlow — Open source software library for machine learning

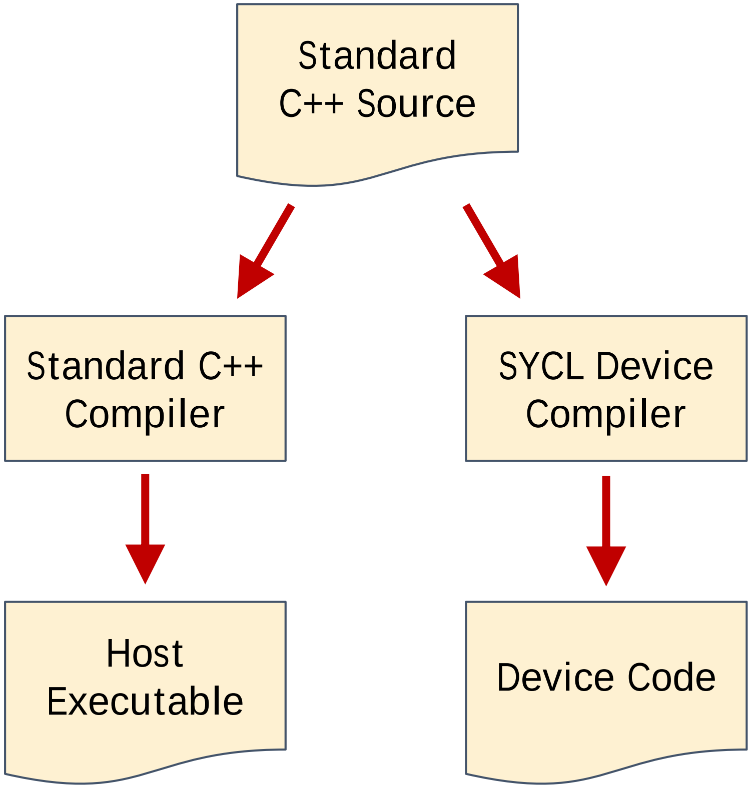
OpenCL

Hundreds of languages, frameworks and projects using OpenCL to access vendor-optimized, heterogeneous compute runtimes

Over 4,000 GitHub repositories using OpenCL: tools, applications, libraries, languages - up from 2,000 two years ago

What is SYCL?

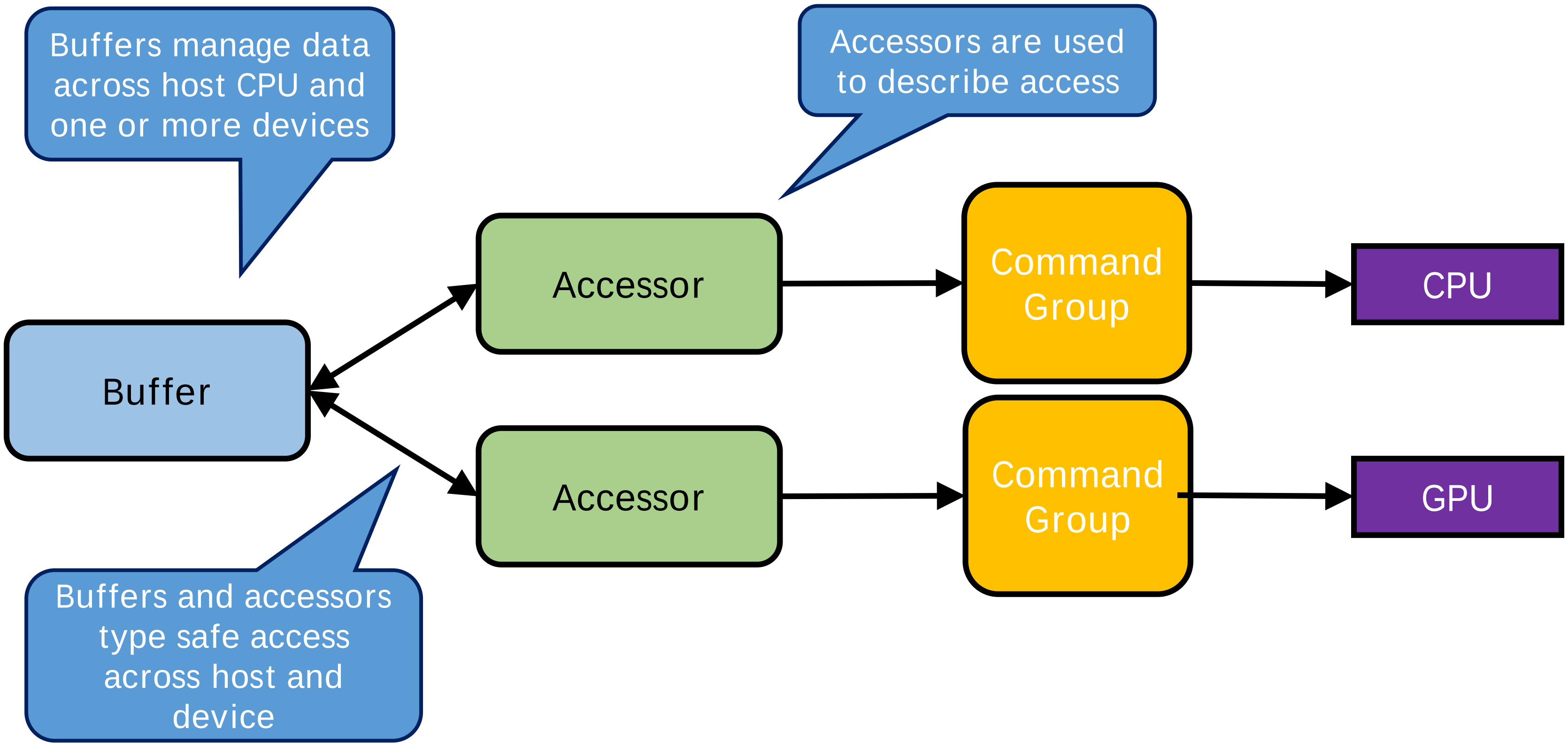
- High-level C++ abstraction layer for OpenCL
  - Full coverage for all OpenCL features
  - Interop to enable existing OpenCL code with SYCL
- Single-source compilation
- Automatic scheduling of data movement



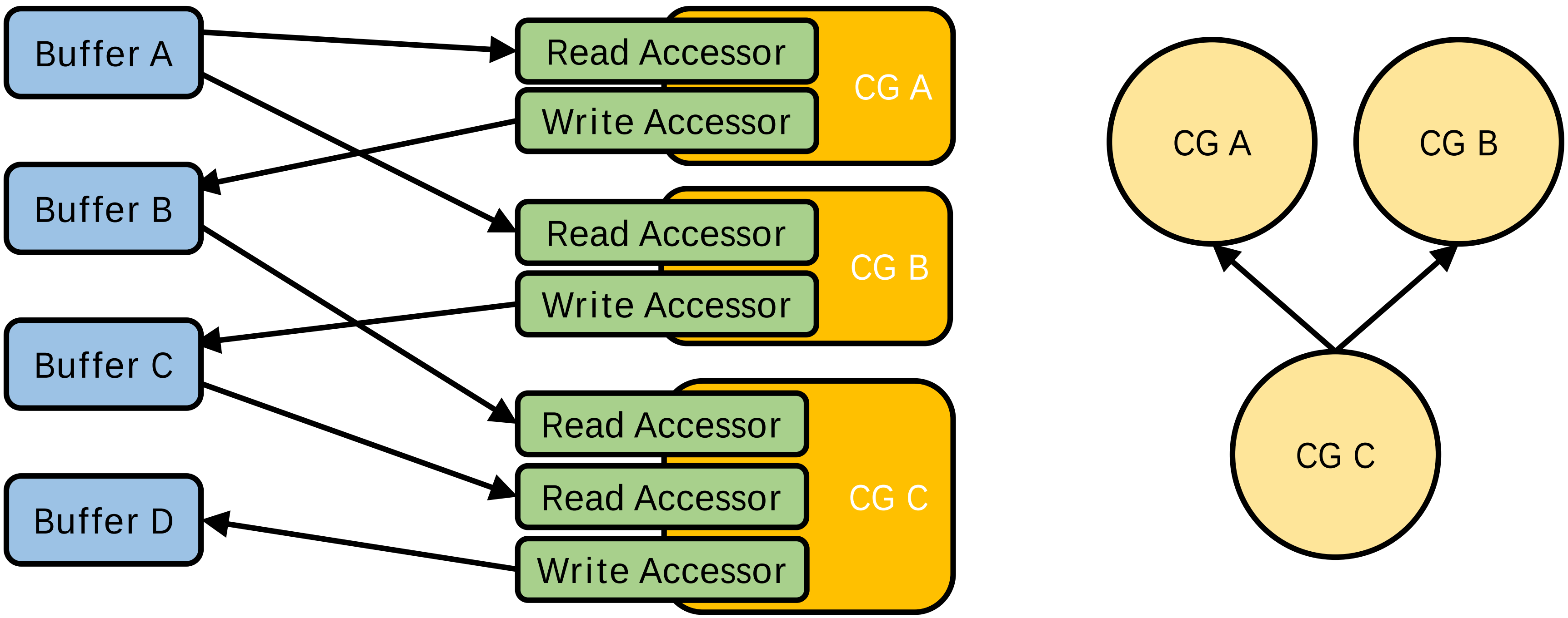
SYCL Example

```
// Create a device queue.
cl::sycl::queue device_queue;

// Create buffers.
cl::sycl::range<1> n_items{array_size};
cl::sycl::buffer<cl::sycl::cl_int, 1> in_buffer(in.data(),
n_items);
cl::sycl::buffer<cl::sycl::cl_int, 1> out_buffer(out.data(),
n_items);

// Submit a kernel and associated data movement operations.
device_queue.submit([&](cl::sycl::handler &cgh) {
 // Defines the kernels access requirements.
 auto in_accessor =
in_buffer.get_access<cl::sycl::access::mode::read>(cgh);
 auto out_accessor =
out_buffer.get_access<cl::sycl::access::mode::write>(cgh);
```

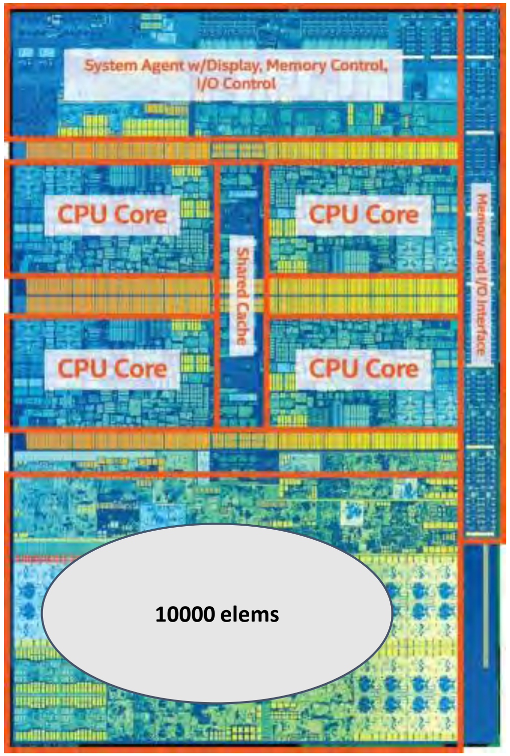
Separating Storage & Access



Data Dependency Task Graphs



What can I do with a Parallel For Each?



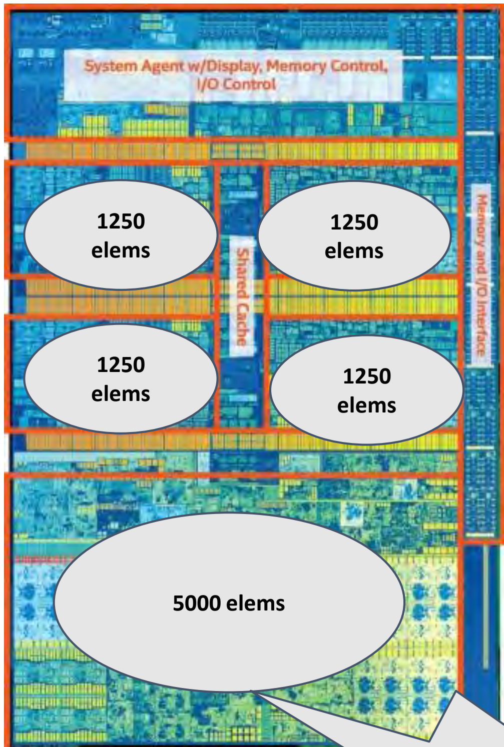
Intel Core i7 7th generation

```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::fill_n(sycl_policy,
            std::begin(v1), nElems, 1);

std::for_each(sycl_named_policy
                 <class KernelName>,
                std::begin(v), std::end(v),
                 [=](float f) { f * f + f });
```

Workload is distributed on the GPU cores

(mileage may vary, implementation-specific behaviour)

What can I do with a Parallel For Each?



Intel Core i7 7th g

Experimental!

```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::fill_n(sycl_heter_policy(cpu, gpu, 0.5),
            std::begin(v1), nElems, 1);

std::for_each(sycl_heter_policy<class kName>
                (cpu, gpu, 0.5),
              std::begin(v), std::end(v),
              [=](float f) { f * f + f });
```

Workload is distributed on all cores!

(mileage may vary, implementation-specific behaviour)

Parallel overloads available in SYCL Parallel STL

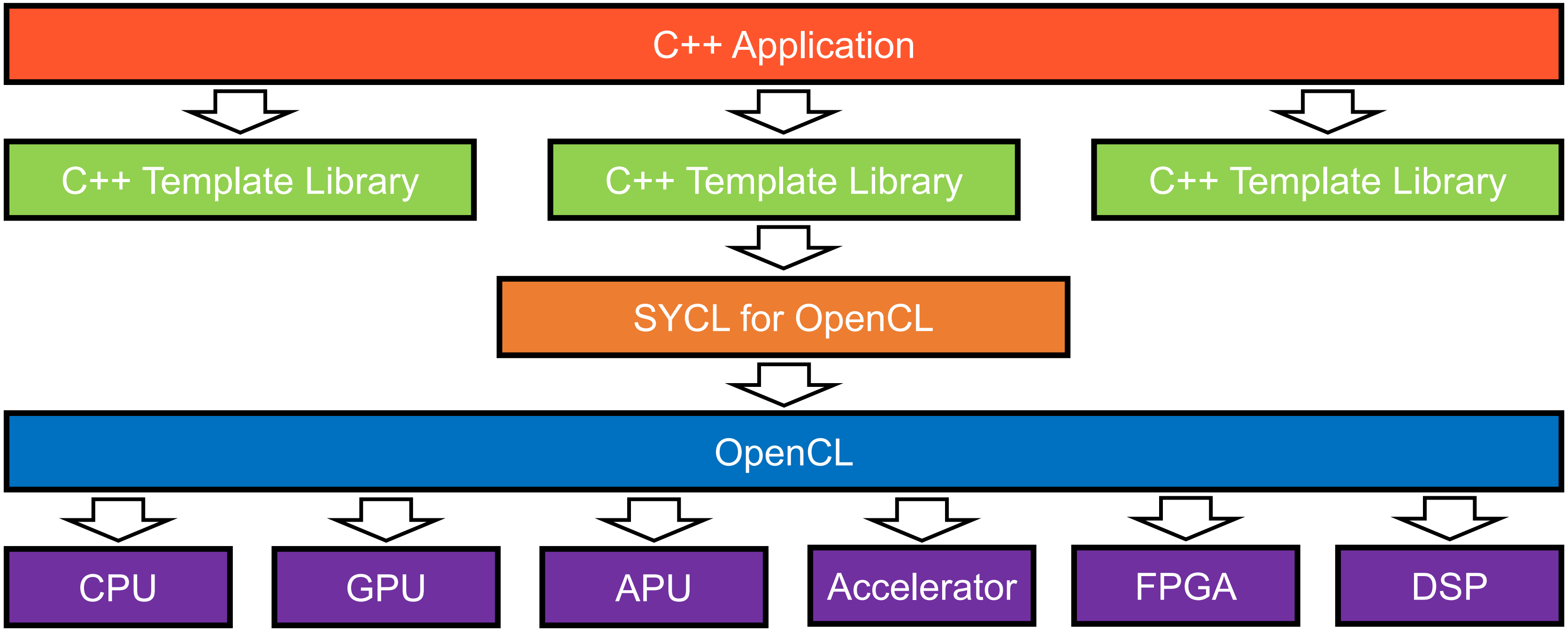
Table 1 — Table of parallel algorithms

adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if ✓	copy_n	count
count_if ✓	equal	exclusive_scan ✓	fill ✓
fill_n	find ✓	find_end	find_first_of
find_if ✓	find_if_not ✓	for_each ✓	for_each_n ✓
generate	generate_n	includes	inclusive_scan ✓
inner_product ✓	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce ✓	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort ✓
stable_partition	stable_sort	swap_ranges	transform ✓
transform_exclusive_scan	transform_inclusive_scan	transform_reduce ✓	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

SYCL for OpenCL

- Cross-platform, single-source, high-level, C++ programming layer
  - Built on top of OpenCL and based on standard C++14

The SYCL Ecosystem



Example: Vector Add

Example: Vector Add

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


template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> &out) {











}
```

Example: Vector Add

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


template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> &out) {
  cl::sycl::buffer<T, 1> inputABuf(inA.data(), out.size());
  cl::sycl::buffer<T, 1> inputBBuf(inB.data(), out.size());
  cl::sycl::buffer<T, 1> outputBuf(out.data(), out.size());







}
```

The buffers synchronise upon destruction

Example: Vector Add

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


template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> &out) {
  cl::sycl::buffer<T, 1> inputABuf(inA.data(), out.size());
  cl::sycl::buffer<T, 1> inputBBuf(inB.data(), out.size());
  cl::sycl::buffer<T, 1> outputBuf(out.data(), out.size());
  cl::sycl::queue defaultQueue;









}
```

Example: Vector Add

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


template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> &out) {
  cl::sycl::buffer<T, 1> inputABuf(inA.data(), out.size());
  cl::sycl::buffer<T, 1> inputBBuf(inB.data(), out.size());
  cl::sycl::buffer<T, 1> outputBuf(out.data(), out.size());
  cl::sycl::queue defaultQueue;
  defaultQueue.submit([&] (cl::sycl::handler &cgh) {







  });
}
```

Create a command group to define an asynchronous task

Example: Vector Add

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


template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> &out) {
  cl::sycl::buffer<T, 1> inputABuf(inA.data(), out.size());
  cl::sycl::buffer<T, 1> inputBBuf(inB.data(), out.size());
  cl::sycl::buffer<T, 1> outputBuf(out.data(), out.size());
  cl::sycl::queue defaultQueue;
  defaultQueue.submit([&] (cl::sycl::handler &cgh) {
    auto inputAPtr = inputABuf.get_access<cl::sycl::access::read>(cgh);
    auto inputBPtr = inputBBuf.get_access<cl::sycl::access::read>(cgh);
    auto outputPtr = outputBuf.get_access<cl::sycl::access::write>(cgh);




  });
}
```

Example: Vector Add

```
#include <CL/sycl.hpp>
template <typename T> kernel;

template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> &out) {
  cl::sycl::buffer<T, 1> inputABuf(inA.data(), out.size());
  cl::sycl::buffer<T, 1> inputBBuf(inB.data(), out.size());
  cl::sycl::buffer<T, 1> outputBuf(out.data(), out.size());
  cl::sycl::queue defaultQueue;
  defaultQueue.submit([&] (cl::sycl::handler &cgh) {
    auto inputAPtr = inputABuf.get_access<cl::sycl::access::read>(cgh);
    auto inputBPtr = inputBBuf.get_access<cl::sycl::access::read>(cgh);
    auto outputPtr = outputBuf.get_access<cl::sycl::access::write>(cgh);
    cgh.parallel_for<kernel<T>>(cl::sycl::range<1>(out.size())),
                                        [=](cl::sycl::id<1> idx) {

    }));
  });
}
```

You must provide a name for the lambda

Create a parallel_for to define the device code

Example: Vector Add

```
#include <CL/sycl.hpp>
template <typename T> kernel;

template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> &out) {
  cl::sycl::buffer<T, 1> inputABuf(inA.data(), out.size());
  cl::sycl::buffer<T, 1> inputBBuf(inB.data(), out.size());
  cl::sycl::buffer<T, 1> outputBuf(out.data(), out.size());
  cl::sycl::queue defaultQueue;
  defaultQueue.submit([&] (cl::sycl::handler &cgh) {
    auto inputAPtr = inputABuf.get_access<cl::sycl::access::read>(cgh);
    auto inputBPtr = inputBBuf.get_access<cl::sycl::access::read>(cgh);
    auto outputPtr = outputBuf.get_access<cl::sycl::access::write>(cgh);
    cgh.parallel_for<kernel<T>>(cl::sycl::range<1>(out.size())),
                                    [=](cl::sycl::id<1> idx) {
      outputPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
    }));
  });
}
```

Example: Vector Add

```
template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> out);

int main() {

  std::vector<float> inputA = { /* input a */ };
  std::vector<float> inputB = { /* input b */ };
  std::vector<float> output = { /* output */ };

  parallel_add(inputA, inputB, output);

  ...
}
```

Implementing Parallel STL with SYCL

```
/* sycl_execution_policy.
* The sycl_execution_policy enables algorithms to be executed using
*  a SYCL implementation.
*/
template <class KernelName = DefaultKernelName>
class sycl_execution_policy {
  cl::sycl::queue m_q;
 public:
  // The kernel name when using lambdas
  using kernelName = KernelName;
  sycl_execution_policy() = default;
  sycl_execution_policy(cl::sycl::queue q) : m_q(q){};
  sycl_execution_policy(const sycl_execution_policy&) = default;
  // Returns the name of the kernel as a string
  std::string get_name() const { return typeid(kernelName).name(); };
  // Returns the queue, if any
  cl::sycl::queue get_queue() const { return m_q; }
```

Creates a SYCL policy using an existing queue

Typeid information only valid for debugging

Implementing Parallel STL with SYCL

Iterator can be any RandomAccess tag

Functions can take C++ iterators or SYCL-specific iterators

```
/* for_each
 */
template <class Iterator, class UnaryFunction>
void for_each(Iterator b, Iterator e, UnaryFunction f) {
  impl::for_each(*this, b, e, f);
}
```

For_each member function on the policy forwards to implementation

```
template <class ExecutionPolicy, class Iterator, class UnaryFunction>
void for_each(ExecutionPolicy &sep, Iterator b, Iterator e, UnaryFunction op) {
 {
   cl::sycl::queue q(sep.get_queue());
   auto device = q.get_device();
   size_t localRange =
     device.get_info<cl::sycl::info::device::max_work_group_size>();
   auto bufI = sycl::helpers::make_buffer(b, e);
   auto vectorSize = bufI.get_count();
   size_t globalRange = sep.calculateGlobalSize(vectorSize, localRange);
```

Obtain the queue from the policy

Obtain device parameters

Prepare allocations on device

Continues...

```
  auto f = [vectorSize, localRange, globalRange, &bufI, op](
      cl::sycl::handler &h) mutable {
    cl::sycl::nd_range<1> r{
      cl::sycl::range<1>{std::max(globalRange, localRange)},
      cl::sycl::range<1>{localRange}};
    auto aI = bufI.template get_access<cl::sycl::access::mode::read_write>(h);
    h.parallel_for<typename ExecutionPolicy::kernelName>(
      r, [aI, op, vectorSize](cl::sycl::nd_item<1> id) {
       if (id.get_global(0) < vectorSize) {
        op(aI[id.get_global(0)]);
       }
      });
  };
  q.submit(f);
}
```

Device Lambda

User functor

Submit for execution on the device

Demo Results - Running std::sort

(Running on Intel i7 6600 CPU & Intel HD Graphics 520)

size	2^16	2^17	2^18	2^19
std::seq	0.27031s	0.620068s	0.669628s	1.48918s
std::par	0.259486s	0.478032s	0.444422s	1.83599s
std::unseq	0.24258s	0.413909s	0.456224s	1.01958s
sycl_execution_policy	0.273724s	0.269804s	0.277747s	0.399634s

Eigen Linear Algebra Library

SYCL backend in mainline
Focused on Tensor support, providing support for machine learning/CNNs
Equivalent coverage to CUDA
Working on optimization for various hardware architectures (CPU, desktop and mobile GPUs)
https://bitbucket.org/eigen/eigen/

TensorFlow

- SYCL backend support for all major CNN operations
- Complete coverage for major image recognition networks
  - GoogLeNet, Inception-v2, Inception-v3, ResNet, ….
- Ongoing work to reach 100% operator coverage and optimization for various hardware architectures (CPU, desktop and mobile GPUs)
- https://github.com/tensorflow/tensorflow

SYCL Ecosystem

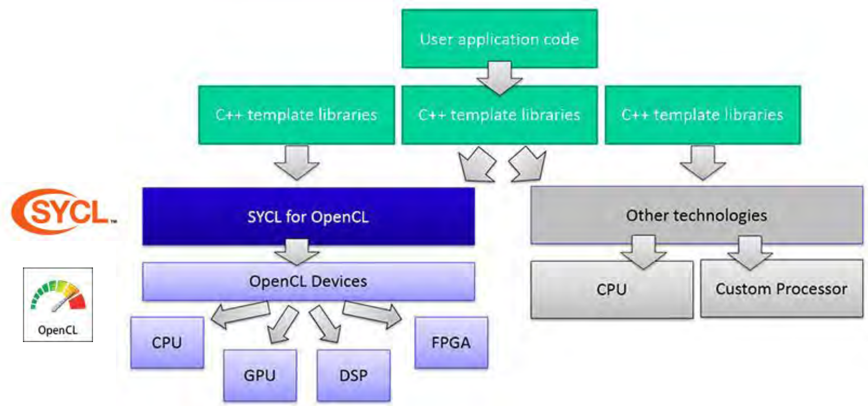
- Single-source heterogeneous programming using STANDARD C++
  - Use C++ templates and lambda functions for host & device code
  - Layered over OpenCL
- Fast and powerful path for bring C++ apps and libraries to OpenCL
  - C++ Kernel Fusion - better performance on complex software than hand-coding
  - Halide, Eigen, Boost.Compute, SYCLBLAS, SYCL Eigen, SYCL TensorFlow, SYCL GTX
  - triSYCL, ComputeCpp, VisionCpp, ComputeCpp SDK …
- More information at http://sycl.tech

Developer Choice

The development of the two specifications are aligned so code can be easily shared between the two approaches

C++ Kernel Language
Low Level Control
'GPGPU'-style separation of device-side kernel source code and host code

Single-source C++
Programmer Familiarity
Approach also taken by C++ AMP and OpenMP



Agenda

- Use the Proper abstractions?
- Why the rush to Massive Parallelism
- What Now?
- Hello World from C++11/14/17 Parallelism
- SYCL: C++ Heterogeneous (GPU) Programming
- Bonus: Executors
-

Agenda

Brief history

Use cases

Design

Examples

Future work

What are executors?

invoke async parallel algorithms future::then post

defer define_task_block dispatch asynchronous operations strand<>

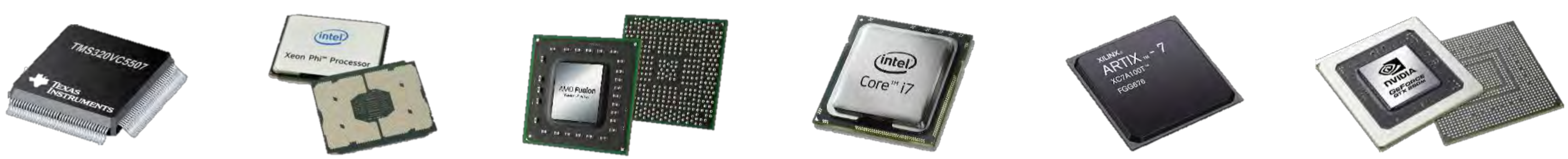
Unified interface for execution

SYCL / OpenCL / CUDA / HCC

OpenMP / MPI

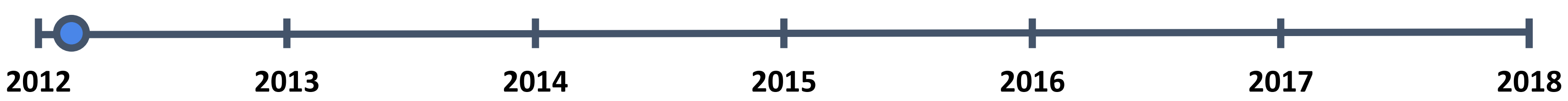
C++ Thread Pool

Boost.Asio / Networking TS

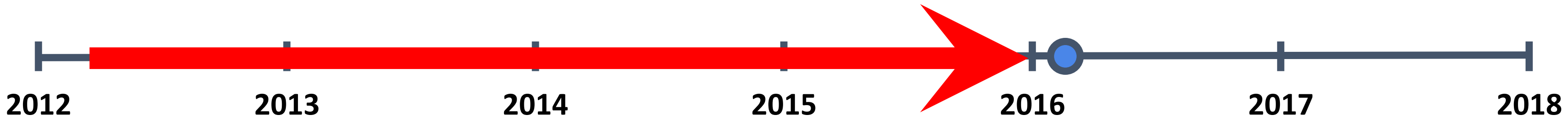


Brief history

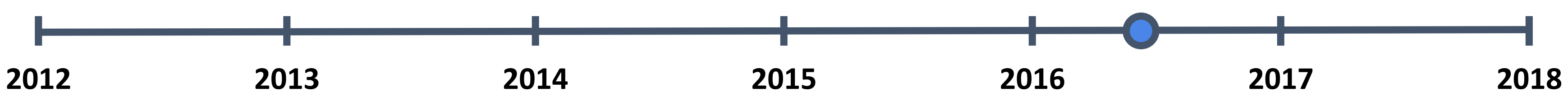
- First executor proposal published in 2012



- Between 2012 and 2016 many more papers were published
- The work centred on four main proposals:
  - N4414: Executors and schedulers, revision 5
  - P0058r1: An Interface for Abstracting Execution
  - P0113r0: Executors and Asynchronous Operations, Revision 2
  - P0285r0: Using customization points to unify executors



- At the Oulu 2016 meeting the executors sub group was formed
- The focus of the sub group was to bring together the various use cases of executors and form a unified proposal



- Since then the group has published multiple papers proposing a unified executors proposals:
  - P0443r2: A Unified Executors Proposal for C++
  - P0761r0: Executors Design Document