SYCL and Codeplay

Michael Wong (Codeplay Software, VP of Research and Development), Andrew Richards, CEO

ISOCPP.org Director, VP http://isocpp.org/wiki/faq/wg21#michael-wong

Head of Delegation for C++ Standard for Canada

Vice Chair of Programming Languages for Standards Council of Canada

Chair of WG21 SG5 Transactional Memory
Chair of WG21 SG14 Games Dev/Low Latency/Financial Trading/Embedded

Editor: C++ SG5 Transactional Memory Technical Specification

Editor: C++ SG1 Concurrency Technical Specification

http:://wongmichael.com/about

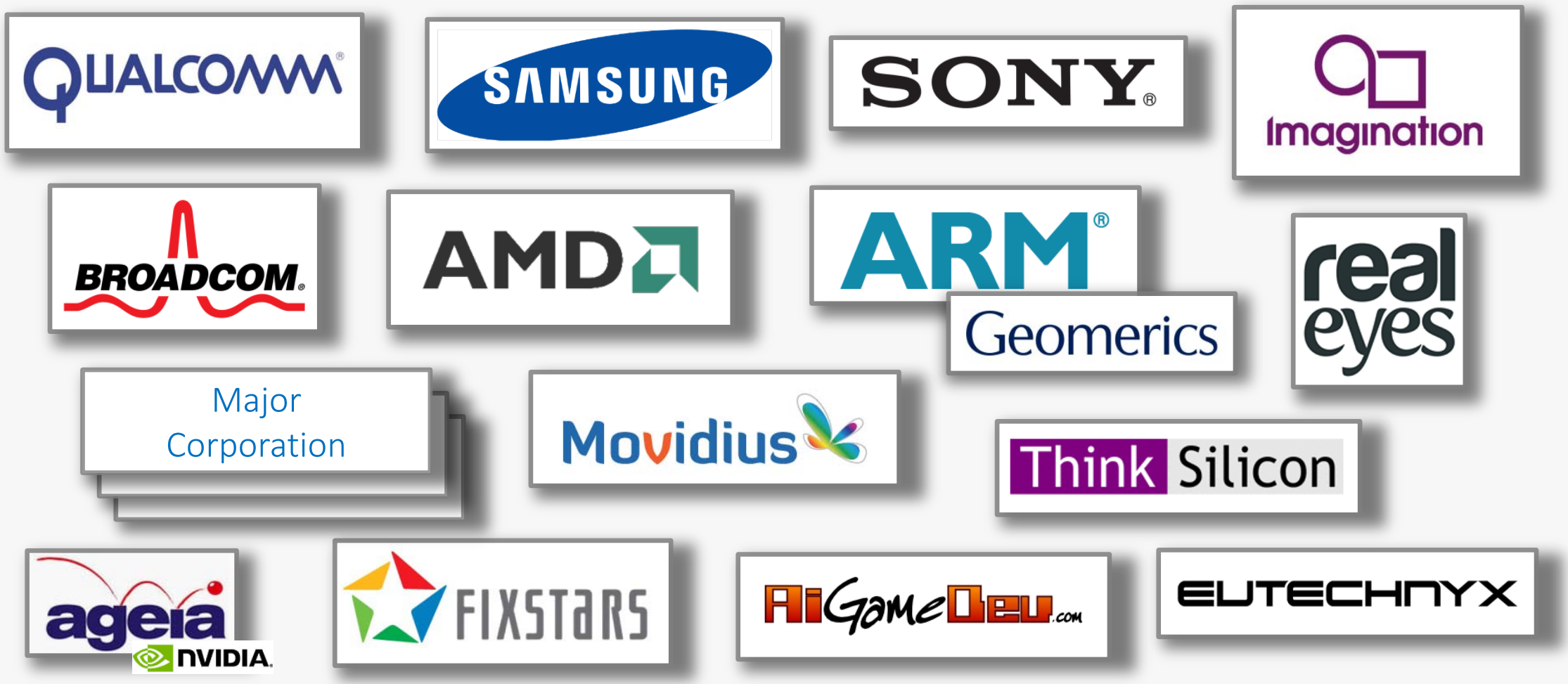
Agenda

- Introduction to Codeplay
- SYCL: The open Khronos standard
  - A comparison of Heterogeneous Programming Models
  - SYCL Design Philosophy: C++ end to end model for HPC and consumers
- The ecosystem:
  - VisionCpp
  - Parallel STL
  - TensorFlow, Machine Vision, Neural Networks, Self-Driving Cars
- Codeplay ComputeCPP Community Edition: Free Download

Achievements

- Established 2002
- Producing heterogeneous compilers & tools
- Over 15 non-CPU cores e.g. GPU, VPU, DSP, …
- Produced C/C++ for GPUs before Nvidia
- **ComputeSuite** – a toolbox for heterogeneous systems

Customers & Partners



Driving the Standards

ISO C++

- Chair SG14
  - Low Latency/Embedded/Games/Simulation/Financial
- Chair of SG5 (Transactional Memory)
- Editor of SG1 Concurrency Spec

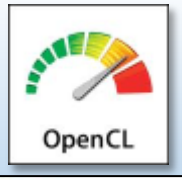
Member of

- British Standards Institute delegation of Technical Experts to C/C++
- Head of delegation of Standards Council of Canada to C/C++
- Director and VP of ISOCPP.org

KHRONOS GROUP

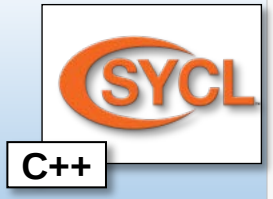
Contributor to:

- OpenCL
- OpenVX
- Safety-Critical
- SPIR-V & Vulkan



SYCL

- Chair
- Spec Editor



HSA FOUNDATION

Tools

- Chair
- Spec Editor

System Runtime

- Chair
- Spec Editor

Engineering Leadership

- Andrew Richards CEO
  - 22 years of Gaming & Graphics development
  - Contribution to OpenCL, SPIR, SYCL and HSA
- Michael Wong VP R&D
  - Joined from IBM, opening Canada office
  - 15 years of C++ leadership, holding many key positions
- 55 Staff in Edinburgh
  - All University Graduates incl. 6 PhD + 10 Masters
  - University Partnerships for EngD & PhD

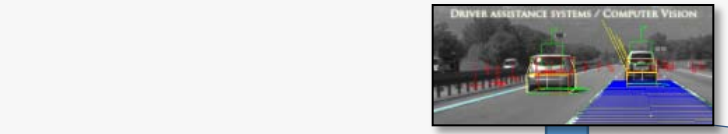
Codeplay: world expert in Heterogeneous software platform for self-driving cars, AI/machine learning/neural networks, computer vision, data centres, graphics, mobile devices, with Open Standards

KHRONOS GROUP
CONNECTING SOFTWARE TO SILICON

ISO International Organization for Standardization

THE C++ PROGRAMMING LANGUAGE

Major Corporation



Vulkan

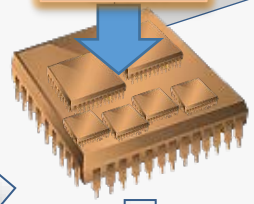
OpenCL

SYCL

ComputeSuite

ComputeCpp
ComputeAorta
ComputeVisionCpp

LLVM



OpenCV

OpenVX

TensorFlow

ISO 26262

Algorithms

Software/Middleware
Tools & layers to Standards

Processors IP
CPU, GPU, DSP, VPU, ViSP

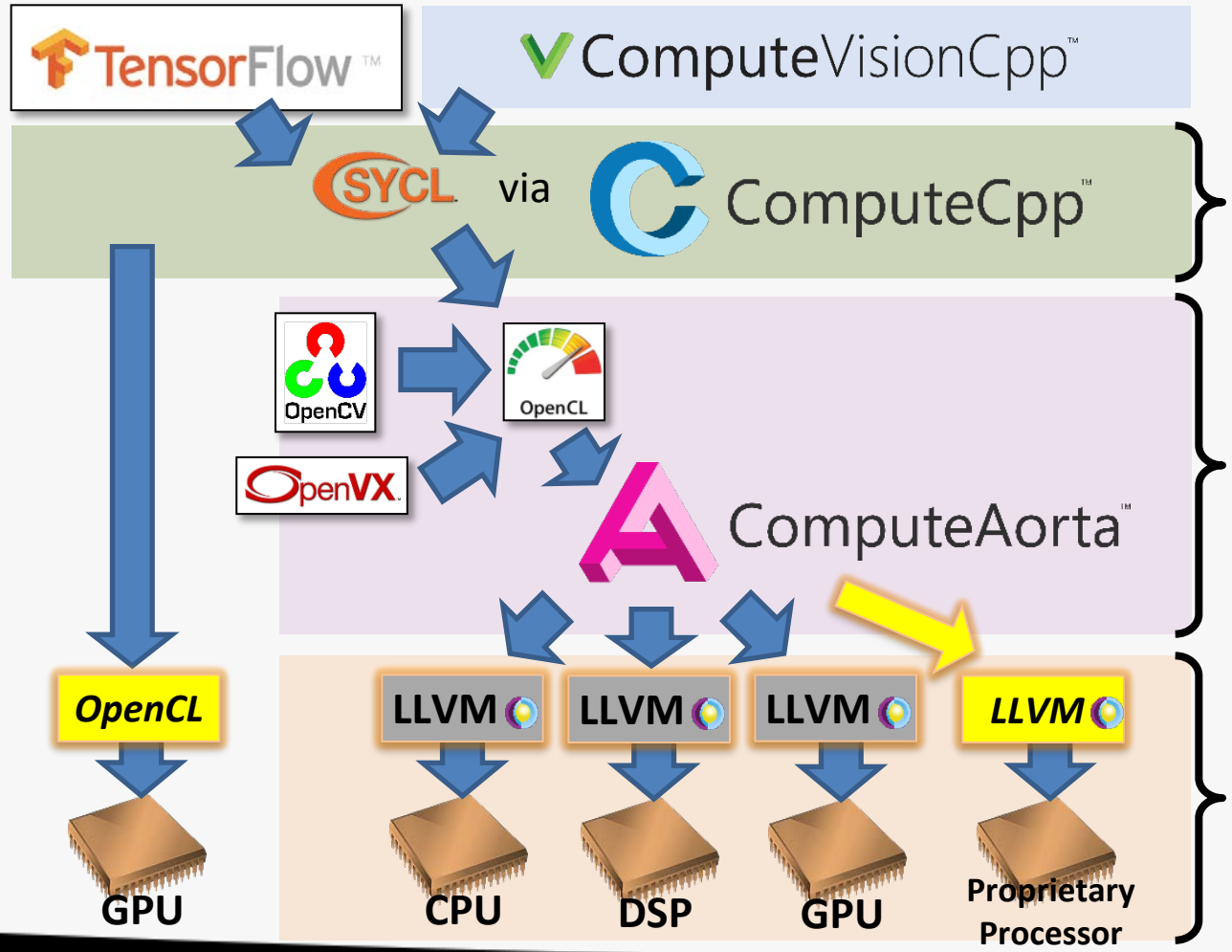
Semiconductors

Module Integration

ISO C99/11 or C++ 11/14/17/20 compliant

QUALCOMM SAMSUNG SONY Imagination BROADCOM AMD ARM Geomerics real eyes Movidius Think Silicon agea NVIDIA FIXSTARS AiGameDev.com EUTECHNYX

Computer Suite: Layers of standards



Solution - ComputeCpp ™
OpenCL for legacy system channelled to existing library. New languages supported

Solution - ComputeAorta ™
Inclusion of extra LLVM for proprietary processors

Solution – LLVM updates
Small effort for LLVM port. Original libraries utilized

Adaptable
- New languages supported
- Easy to hook-in pre-existing solutions
- Easy to add more cores and more compilers

Agenda

- Introduction to Codeplay
- SYCL: The open Khronos standard
  - A comparison of Heterogeneous Programming Models
  - SYCL Design Philosophy: C++ end to end model for HPC and consumers
- The ecosystem:
  - VisionCpp
  - Parallel STL
  - TensorFlow, Machine Vision, Neural Networks, Self-Driving Cars
- Codeplay ComputeCPP Community Edition: Free Download

SYCL v1.2 release

IWOCL, May 2015

SYCL is not magic

SYCL is a practical, open, royalty-free standard to deliver high performance software on today's highly-parallel systems

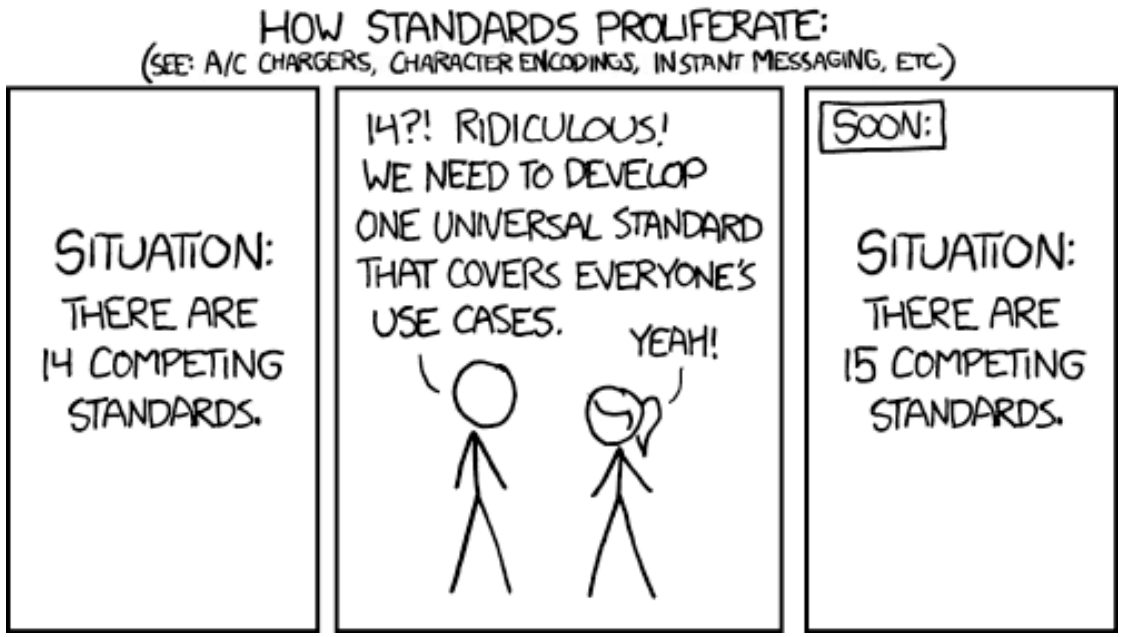
What is SYCL for?

- **Modern C++ lets us separate the what from the how :**
  - We want to separate **what** the user wants to do: *science, computer vision, AI ...*
  - And enable the **how** to be: *run fast on an OpenCL device*
- **Modern C++ supports and encourages this separation**

What we want to achieve

- **We want to enable a C++ ecosystem for OpenCL:**
  - C++ template libraries
  - Tools: compilers, debuggers, IDEs, optimizers
  - Training, example programs
  - Long-term support for current and future OpenCL features

Why a new standard?



http://imgs.xkcd.com/comics/standards.png

- **There are already very established ways to map C++ to parallel processors**
  - So we follow the established approaches
- **There are specifics to do with OpenCL we need to map to C++**
  - We have worked hard to be an *enabler* for other C++ parallel standards
- **We add no more than we need to**

What features of OpenCL do we need?

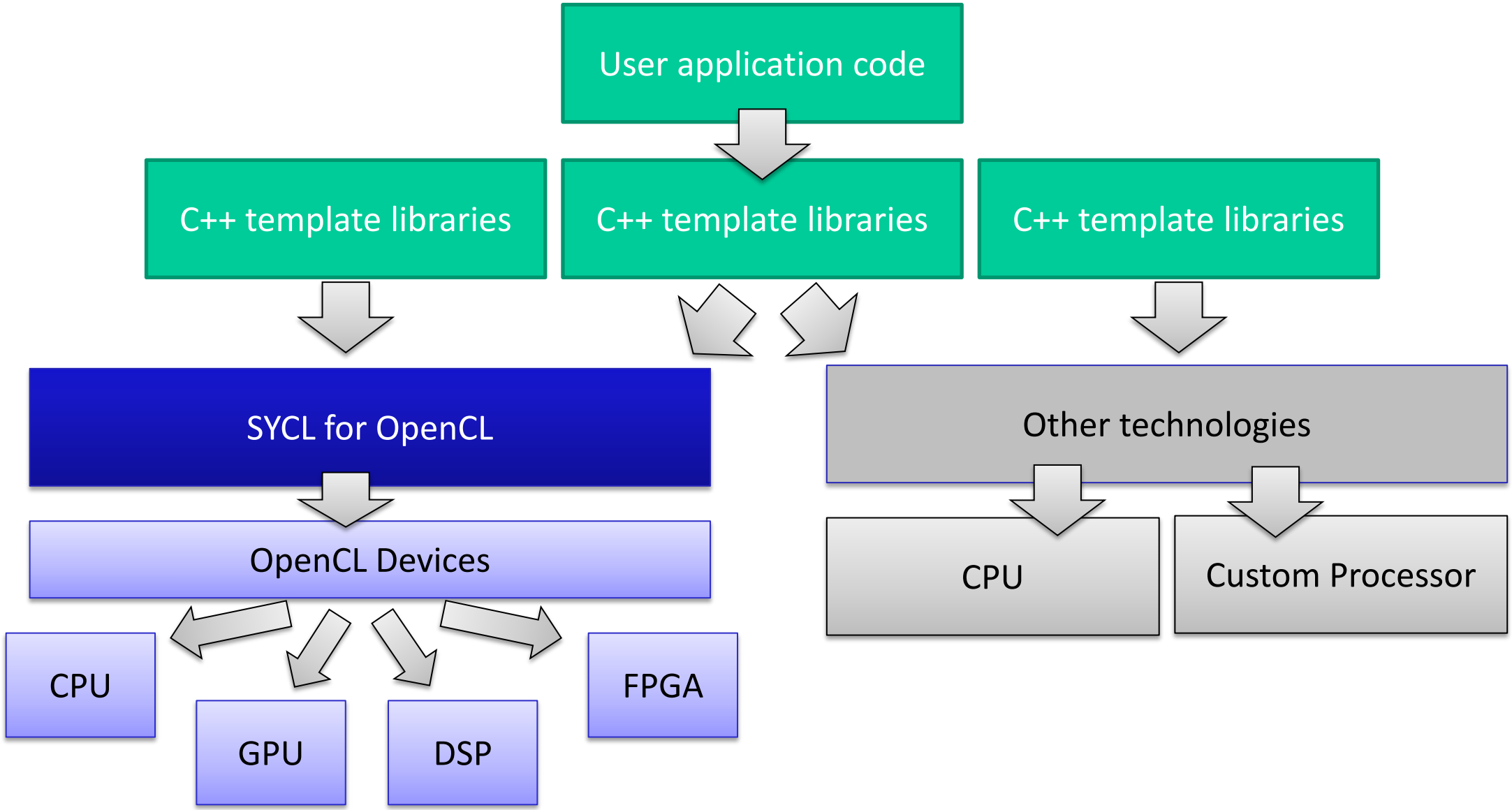
- **We want to enable all OpenCL features in C++ with SYCL**
  - Images, work-groups, barriers, constant/global/local/private memory
  - Memory sharing: mapping and DMA
  - Platforms, contexts, events, queues
  - Support wide range of OpenCL devices: CPUs, GPUs, FPGAs, DSPs...
- **We want to make it easy to write high-performance OpenCL code in C++**
  - SYCL code in C++ must use memory and execute kernels efficiently
  - We must provide developers with all the optimization options they have in OpenCL
- **We want to enable OpenCL C code to interoperate with C++ SYCL code**
  - Sharing of contexts, memory objects etc

KHRONOS GROUP

How do we bring OpenCL features to C++?

- **Key decisions:**
  - We will not add any language extensions to C++
  - We will work with existing C++ compilers
  - We will provide the full OpenCL feature-set in C++

OpenCL / SYCL Stack



Example SYCL Code

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

int main ()
{
….
    // Device buffers
    buffer<float, 1 > buf_a(array_a, range<1>(count));
    buffer<float, 1 > buf_b(array_b, range<1>(count));
    buffer<float, 1 > buf_c(array_c, range<1>(count));
    buffer<float, 1 > buf_r(array_r, range<1>(count));
    queue myQueue;
    myQueue.submit([&](handler& cgh)
    {
              // Data accessors
              auto a = buf_a.get_access<access::read>(cgh);
              auto b = buf_b.get_access<access::read>(cgh);
              auto c = buf_c.get_access<access::read>(cgh);
              auto r = buf_r.get_access<access::write>(cgh);
              // Kernel
              cgh.parallel_for<class three_way_add>(count, [=](id<> i)
                  {
                      r[i] = a[i] + b[i] + c[i];
                  })
              );
          });
….
}
```

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

void func (float *array_a, float *array_b, float *array_c,
           float *array_r, size_t count)
{
    buffer<float, 1 > buf_a(array_a, range<1>(count));
    buffer<float, 1 > buf_b(array_b, range<1>(count));
    buffer<float, 1 > buf_c(array_c, range<1>(count));
    buffer<float, 1 > buf_r(array_r, range<1>(count));
    queue myQueue (gpu_selector);

    myQueue.submit([&](handler& cgh)
    {
        auto a = buf_a.get_access<access::read>(cgh);
        auto b = buf_b.get_access<access::read>(cgh);
        auto c = buf_c.get_access<access::read>(cgh);
        auto r = buf_r.get_access<access::write>(cgh);

        cgh.parallel_for<class three_way_add>(count, [=](id<1> i)
        {
            r[i] = a[i] + b[i] + c[i];
        });
     });
}
```

#include the SYCL header file

Encapsulate data in SYCL *buffers* which be mapped or copied to or from OpenCL devices

Create a *queue*, preferably on a GPU, which can execute *kernels*

Submit to the queue all the work described in the handler lambda that follows

Create *accessors* which encapsulate the type of access to data in the buffers

Execute in parallel the work over an *ND range* (in this case 'count')

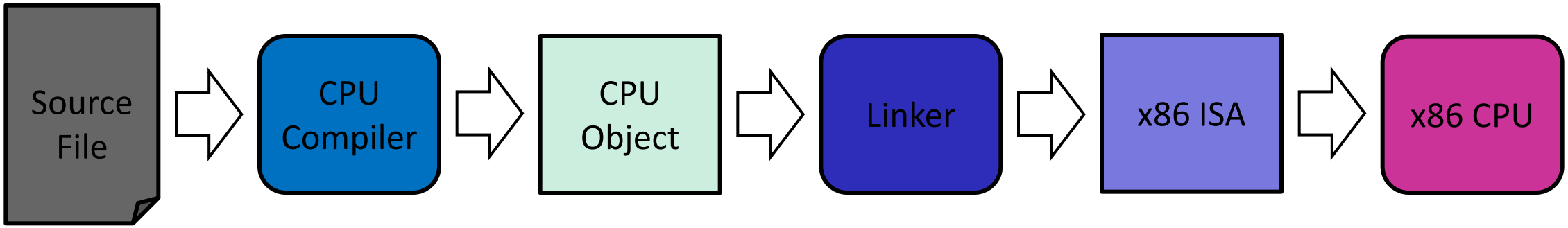
This code is executed in parallel on the device

How did we come to our decisions?

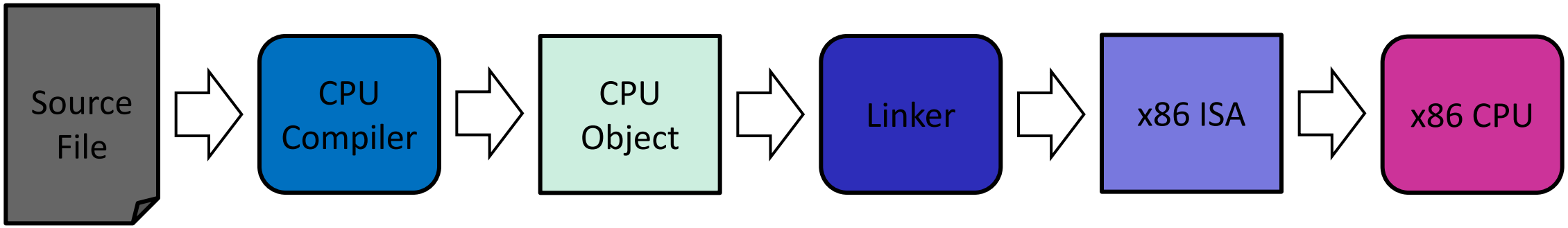
What was our thinking?

How do we offload code to a heterogeneous device?

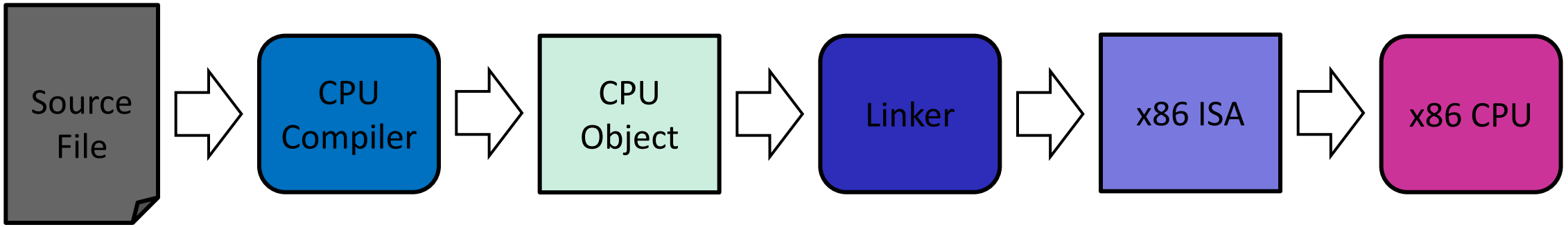
Compilation Model

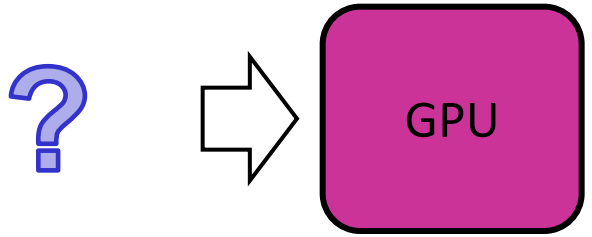


Compilation Model



Compilation Model

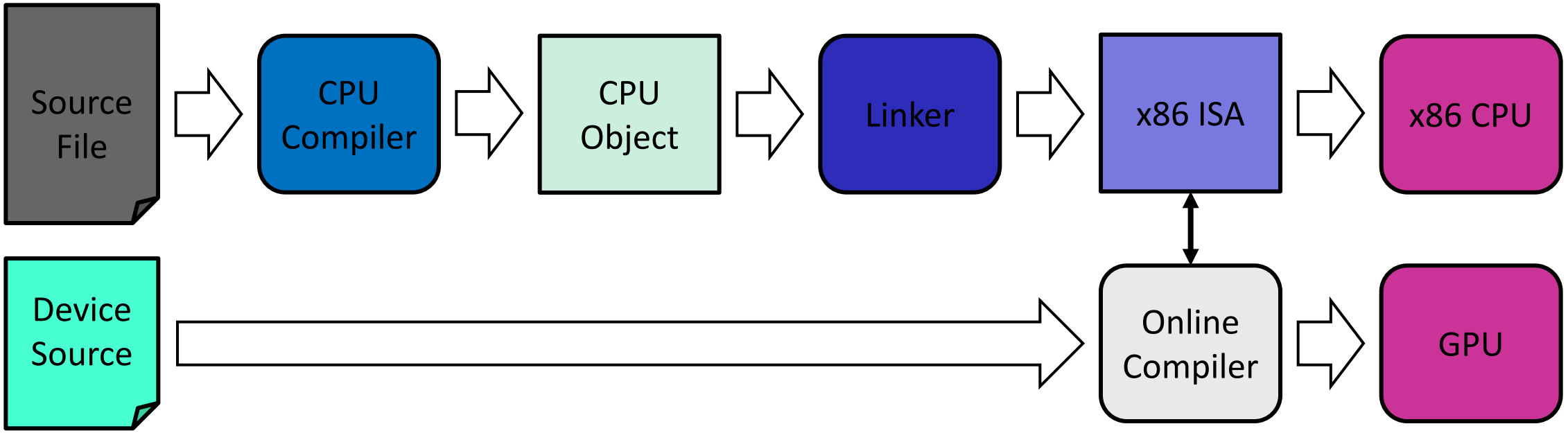




How can we compile source code for a sub architectures?

- Separate source
- Single source

Separate Source Compilation Model



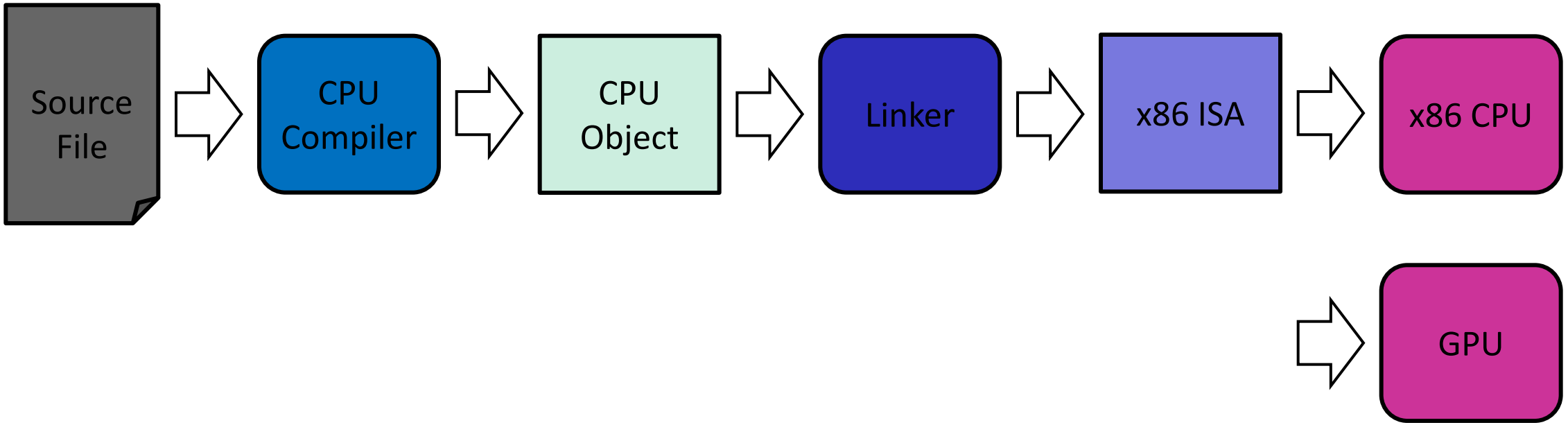

```
float *a, *b, *c;
…
kernel k = clCreateKernel(…, “my_kernel”, …);
clEnqueueWriteBuffer(…, size, a, …);
clEnqueueWriteBuffer(…, size, a, …);
clEnqueueNDRange(…, k, 1, {size, 1, 1}, …);
clEnqueueWriteBuffer(…, size, c, …);
```

Here we’re using OpenCL as an example

```
void my_kernel(__global float *a, __global float *b,
               __global float *c) {
  int id = get_global_id(0);
  c[id] = a[id] + b[id];
}
```

KHRONOS GROUP

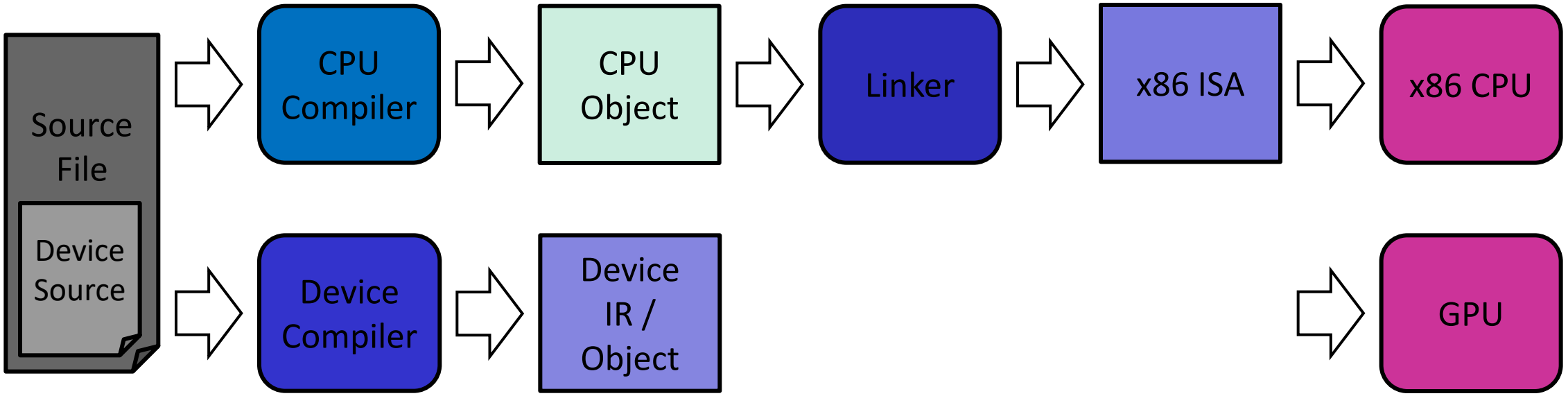
Single Source Compilation Model




```
array_view<float> a, b, c;
extent<2> e(64, 64);

parallel_for_each(e, [=](index<2> idx) restrict(amp) {
  c[idx] = a[idx] + b[idx];
});
```

Here we are using C++ AMP as an example

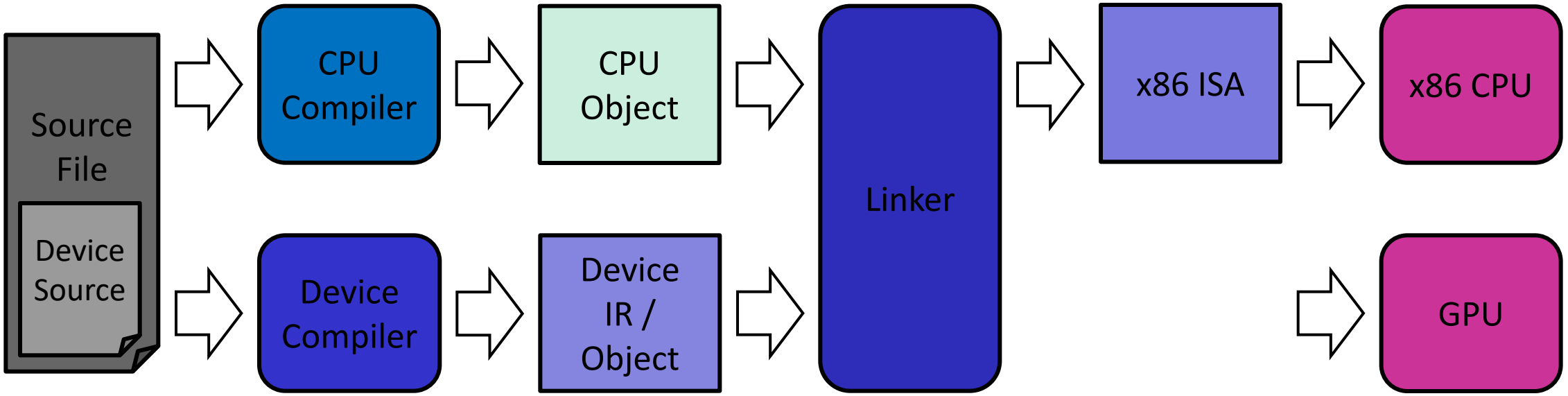
Single Source Compilation Model




```
array_view<float> a, b, c;
extent<2> e(64, 64);

parallel_for_each(e, [=](index<2> idx) restrict(amp) {
  c[idx] = a[idx] + b[idx];
});
```

Here we are using C++ AMP as an example

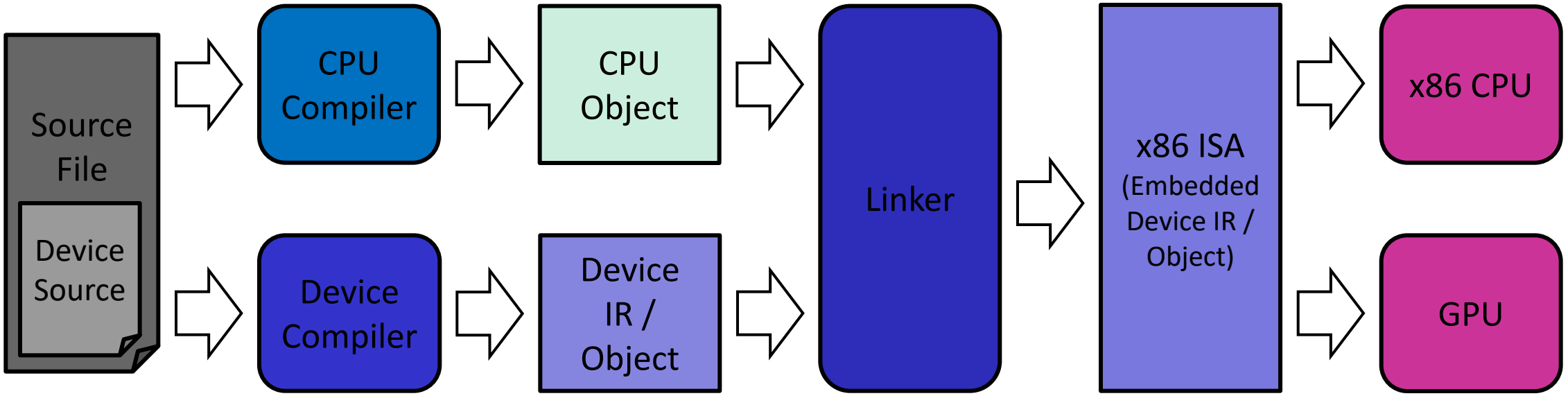
Single Source Compilation Model




```
array_view<float> a, b, c;
extent<2> e(64, 64);

parallel_for_each(e, [=](index<2> idx) restrict(amp) {
  c[idx] = a[idx] + b[idx];
});
```

Here we are using C++ AMP as an example

Single Source Compilation Model




```
array_view<float> a, b, c;
extent<2> e(64, 64);

parallel_for_each(e, [=](index<2> idx) restrict(amp) {
  c[idx] = a[idx] + b[idx];
});
```

Here we are using C++ AMP as an example

Benefits of Single Source

- Device code is written in the same source file as the host CPU code
- Allows compile-time evaluation of device code
- Supports type safety across host CPU and device
- Supports generic programming
- Removes the need to distribute source code

Describing Parallelism

How do you represent the different forms of parallelism?

- Directive vs explicit parallelism
- Task vs data parallelism
- Queue vs stream execution

Directive vs Explicit Parallelism

Examples:

- OpenMP, OpenACC

Implementation:

- Compiler transforms code to be parallel based on pragmas

Examples:

- SYCL, CUDA, TBB, Fibers, C++11 Threads

Implementation:

- An API is used to explicitly enqueuer one or more threads

Here we're using OpenMP as an example

```
vector<float> a, b, c;

#pragma omp parallel for
for(int i = 0; i < a.size(); i++) {
  c[i] = a[i] + b[i];
}
```

Here we're using C++ AMP as an example

```
array_view<float> a, b, c;
extent<2> e(64, 64);
parallel_for_each(e, [=](index<2> idx)
restrict(amp) {
  c[idx] = a[idx] + b[idx];
});
```

Task vs Data Parallelism

Examples:

- OpenMP, C++11 Threads, TBB

Implementation:

- Multiple (potentially different) tasks are performed in parallel

Here we're using TBB as an example

```
vector<task> tasks = { … };

tbb::parallel_for_each(tasks.begin(),
    tasks.end(), [=](task &v) {
  task();
});
```

Examples:

- C++ AMP, CUDA, SYCL, C++17 ParallelSTL

Implementation:

- The same task is performed across a large data set

Here we're using CUDA as an example

```
float *a, *b, *c;
cudaMalloc((void **)&a, size);
cudaMalloc((void **)&b, size);
cudaMalloc((void **)&c, size);

vec_add<<<64, 64>>>(a, b, c);
```

Queue vs Stream Execution

Examples:

- C++ AMP, CUDA, SYCL, C++17 ParallelSTL

Implementation:

- Functions are placed in a queue and executed once per enqueuer

Examples:

- BOINC, BrookGPU

Implementation:

- A function is executed on a continuous loop on a stream of data

Here we're using CUDA as an example

```
float *a, *b, *c;
cudaMalloc((void **)&a, size);
cudaMalloc((void **)&b, size);
cudaMalloc((void **)&c, size);

vec_add<<<64, 64>>>(a, b, c);
```

Here we're using BrookGPU as an example

```
reduce void sum (float a<>,
                 reduce float r<>) {
  r += a;
}
float a<100>;
float r;
sum(a,r);
```

Data Locality & Movement

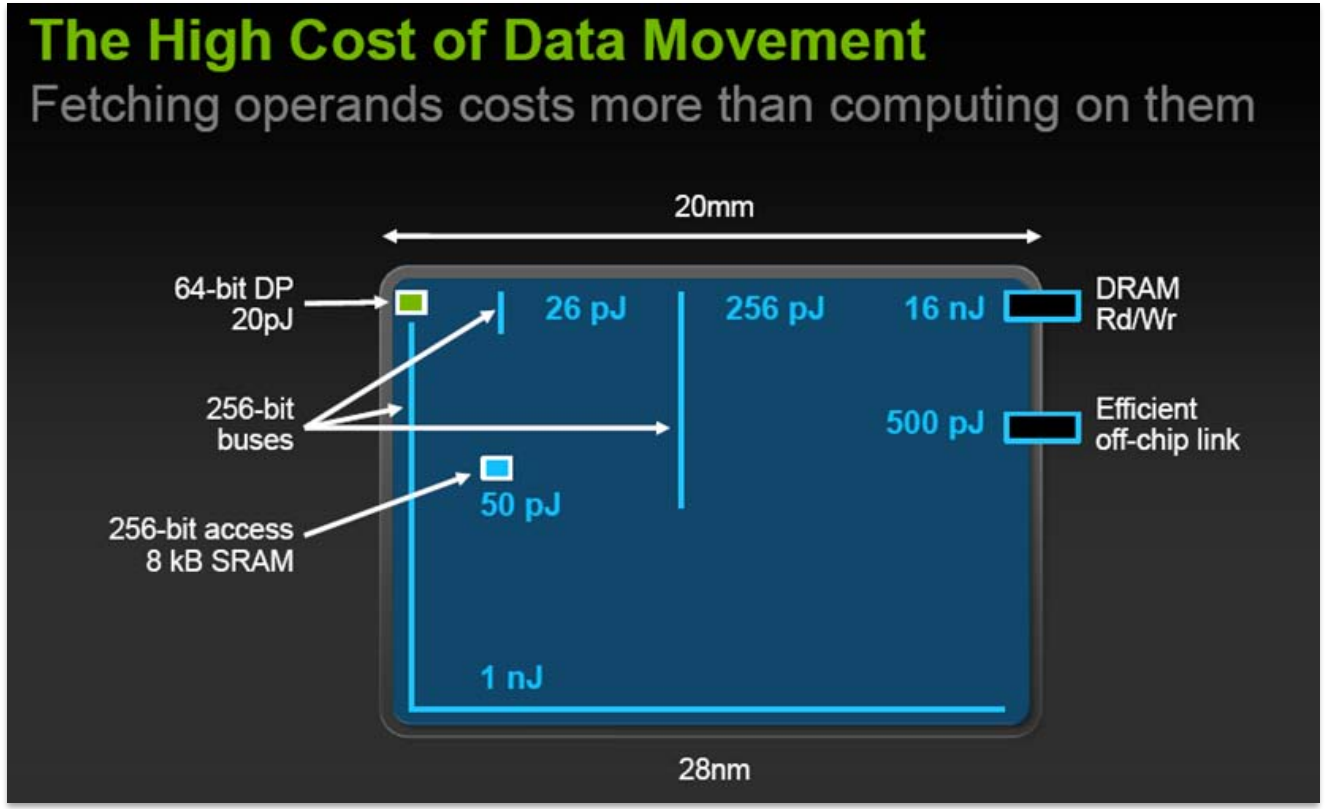
One of the biggest limiting factor in heterogeneous computing

- Cost of data movement in time and power consumption

Cost of Data Movement

- **It can take considerable time to move data to a device**
  - This varies greatly depending on the architecture
- **The bandwidth of a device can impose bottlenecks**
  - This reduces the amount of throughput you have on the device
- **Performance gain from computation > cost of moving data**
  - If the gain is less than the cost of moving the data it's not worth doing
- **Many devices have a hierarchy of memory regions**
  - Global, read-only, group, private
  - Each region has different size, affinity and access latency
  - Having the data as close to the computation as possible reduces the cost

Cost of Data Movement



Credit: Bill Dally, Nvidia, 2010

- 64bit DP Op:
  - 20pJ
- 4x64bit register read:
  - 50pJ
- 4x64bit move 1mm:
  - 26pJ
- 4x64bit move 40mm:
  - 1nJ
- 4x64bit move DRAM:
  - 16nJ

How do you move data from the host CPU to a device?

- Implicit vs explicit data movement

Implicit vs Explicit Data Movement

Examples:

- SYCL, C++ AMP

Implementation:

- Data is moved to the device implicitly via cross host CPU / device data structures

Examples:

- OpenCL, CUDA, OpenMP

Implementation:

- Data is moved to the device via explicit copy APIs

Here we're using C++ AMP as an example

```
array_view<float> ptr;
extent<2> e(64, 64);
parallel_for_each(e, [=](index<2> idx)
restrict(amp) {
  ptr[idx] *= 2.0f;
});
```

Here we're using CUDA as an example

```
float *h_a = { … }, d_a;
cudaMalloc((void **)&d_a, size);
cudaMemcpy(d_a, h_a, size,
  cudaMemcpyHostToDevice);
vec_add<<<64, 64>>>(a, b, c);
cudaMemcpy(d_a, h_a, size,
  cudaMemcpyDeviceToHost);
```

KHRONOS GROUP

How do you address memory between host CPU and device?

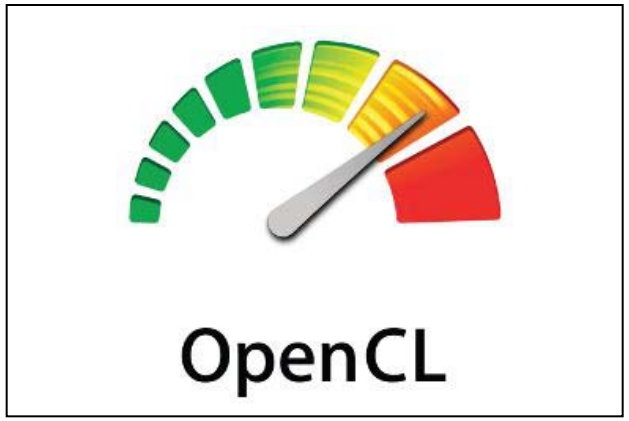
- Multiple address space
- Non-coherent single address space
- Cache coherent single address space

Comparison of Memory Models

- **Multiple address space**
  - SYCL 1.2, C++AMP, OpenCL 1.x, CUDA
  - Pointers have keywords or structures for representing different address spaces
  - Allows finer control over where data is stored, but needs to be defined explicitly
- **Non-coherent single address space**
  - SYCL 2.2, HSA, OpenCL 2.x , CUDA 4, OpenMP
  - Pointers address a shared address space that is mapped between devices
  - Allows the host CPU and device to access the same address, but requires mapping
- **Cache coherent single address space**
  - SYCL 2.2, HSA, OpenCL 2.x, CUDA 6, C++,
  - Pointers address a shared address space (hardware or cache coherent runtime)
  - Allows concurrent access on host CPU and device, but can be inefficient for large data

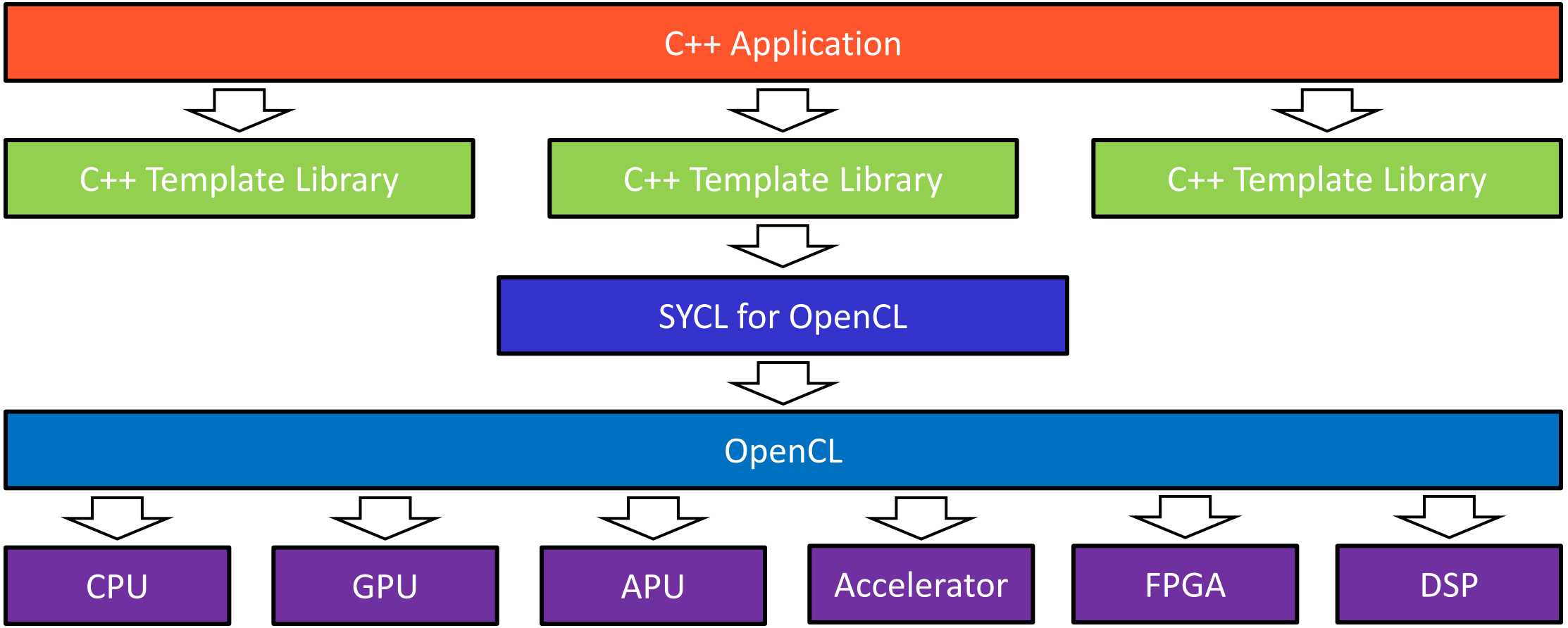
SYCL: A New Approach to Heterogeneous Programming in C++

SYCL for OpenCL



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

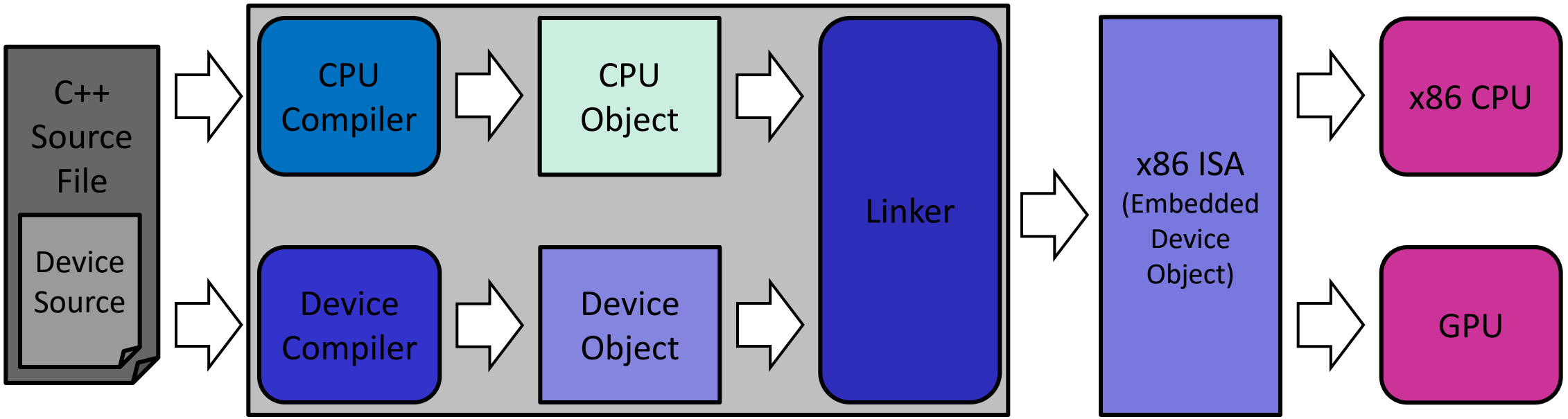
The SYCL Ecosystem



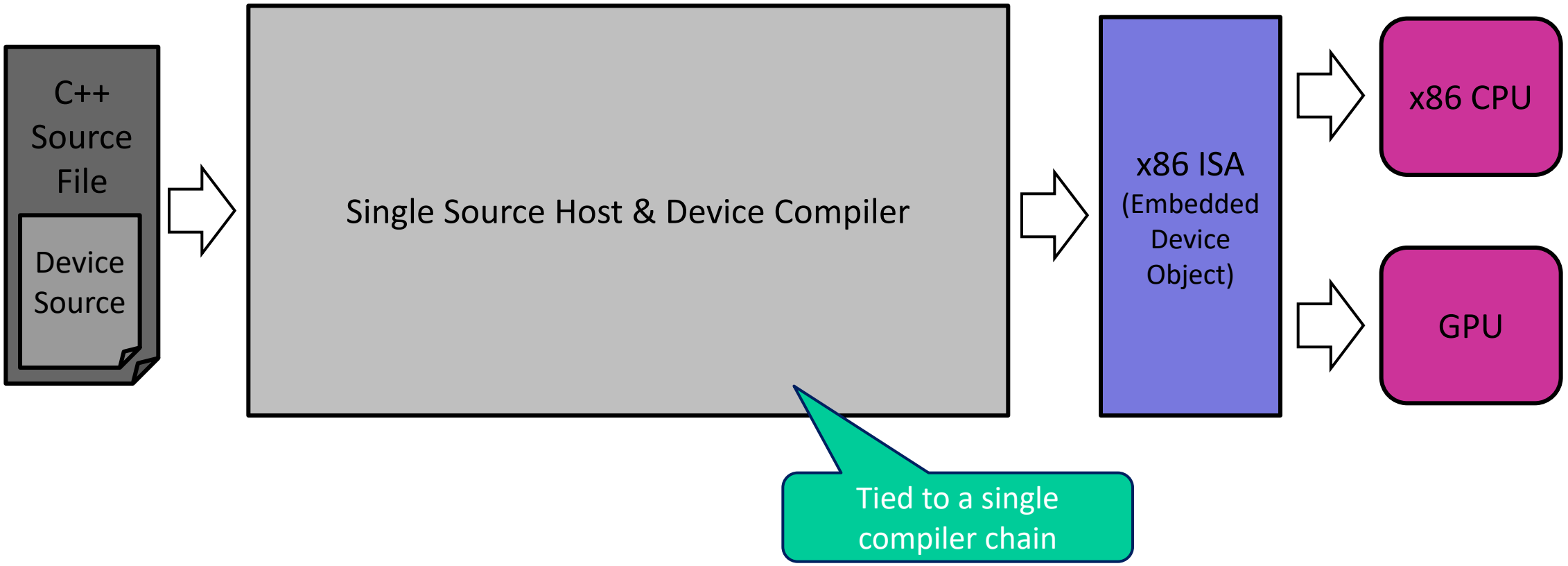
How does SYCL improve heterogeneous offload and performance portability?

- SYCL is entirely standard C++
- SYCL compiles to SPIR
- SYCL supports a multi compilation single source model

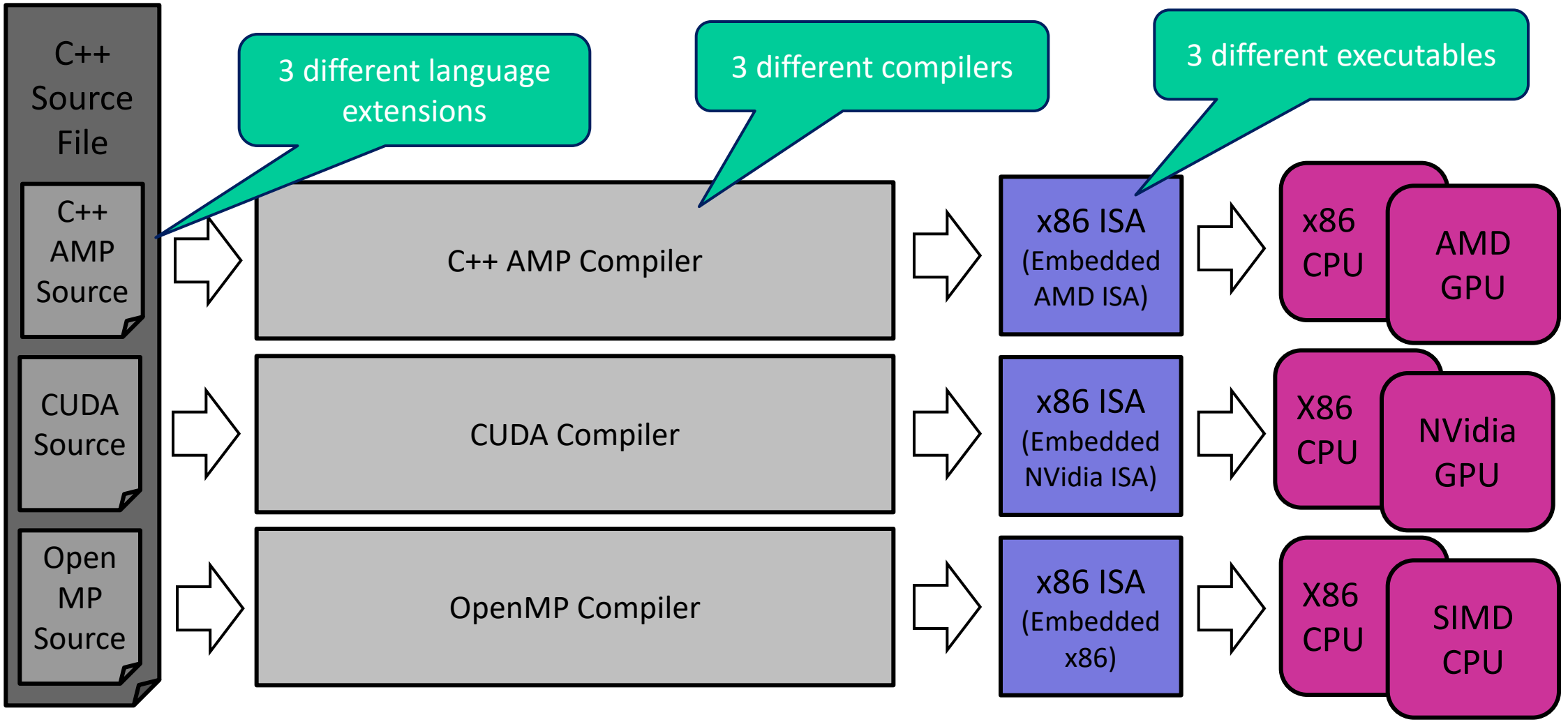
Single Compilation Model



Single Compilation Model



Single Compilation Model



KHRONOS GROUP

SYCL is Entirely Standard C++

```
__global__ vec_add(float *a, float *b, float *c) {
  return c[i] = a[i] + b[i];
}

float *a, *b, *c;
vec_add<
```

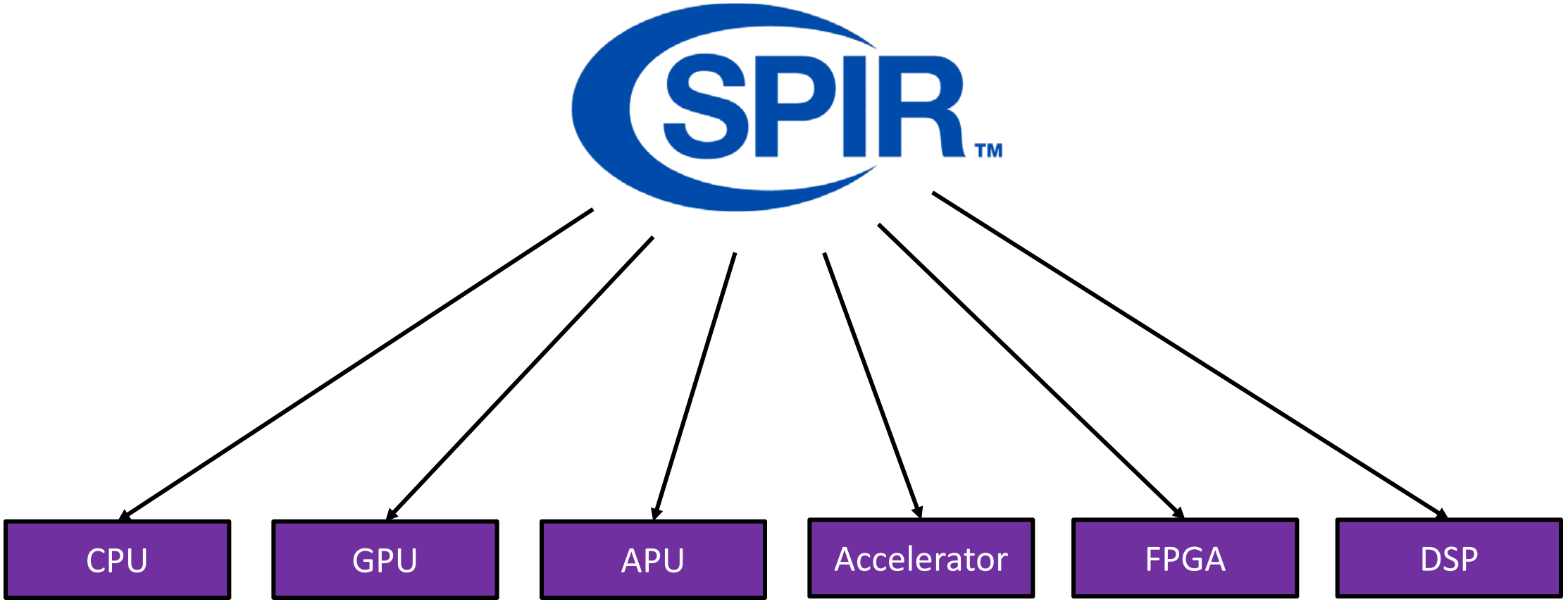
```
vector<float> a, b, c;

#pragma parallel_for
                    size(); i++) {
```

```
array_view<float> a, b, c;
extent<2> e(64, 64);

parallel_for_each(e, [=](index<2> idx) restrict(amp) {
  c[idx] = a[idx] + b[idx];
});
```

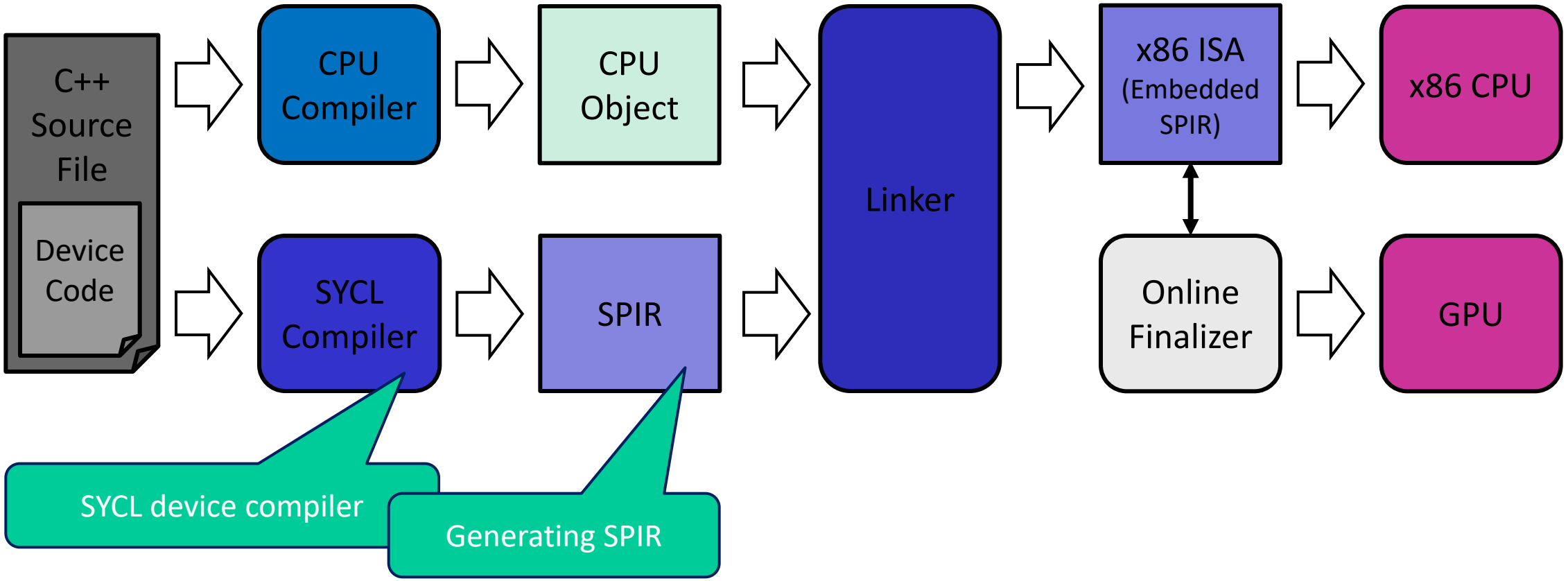
```
cgh.parallel_for<class vec_add>(range, [=](cl::sycl::id<2> idx) {
  c[idx] = a[idx] + c[idx];
}));
```

KHRONOS GROUP

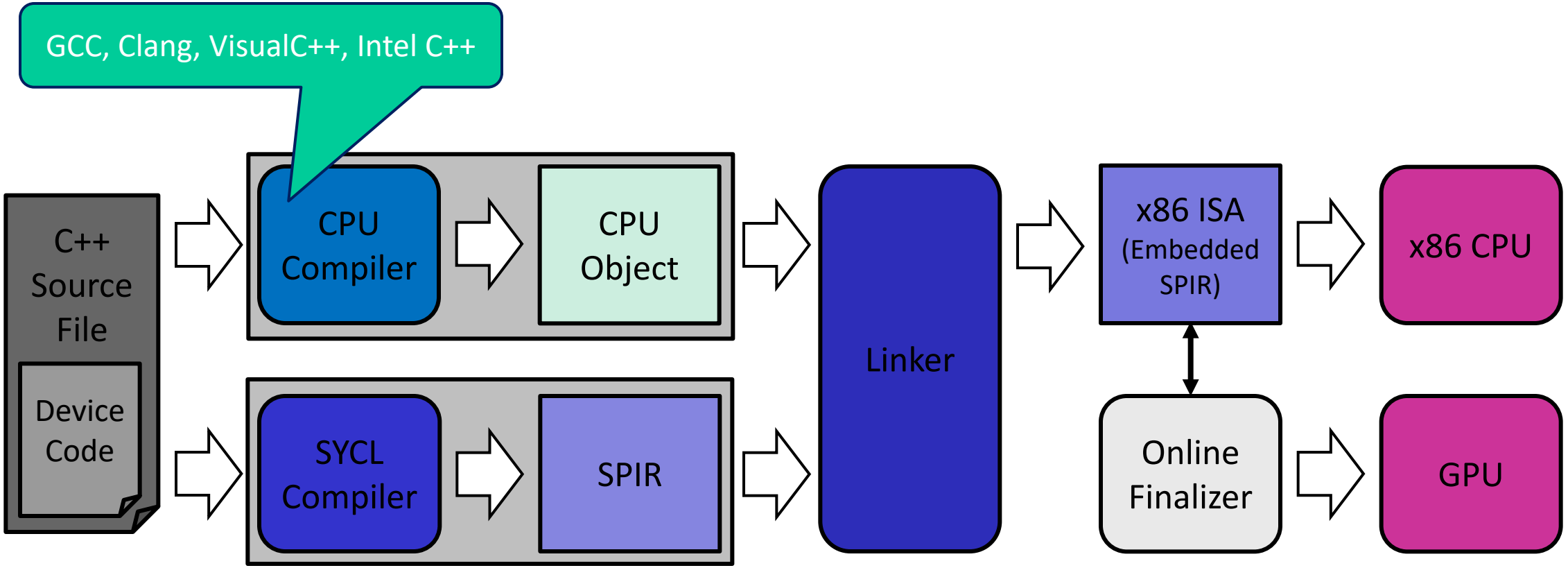
SYCL Targets a Wide Range of Devices with SPIR



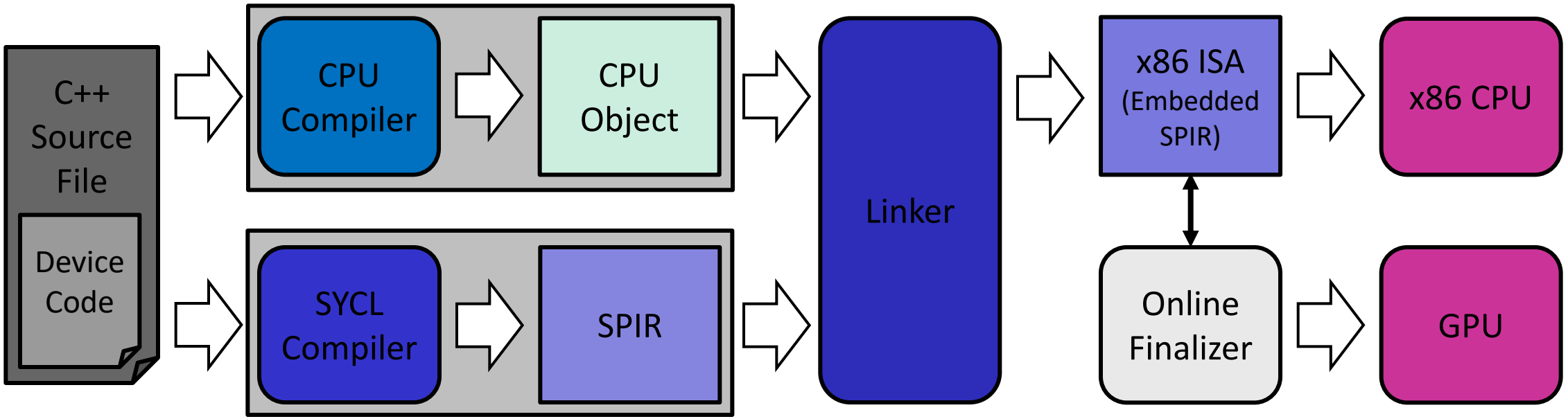
Multi Compilation Model



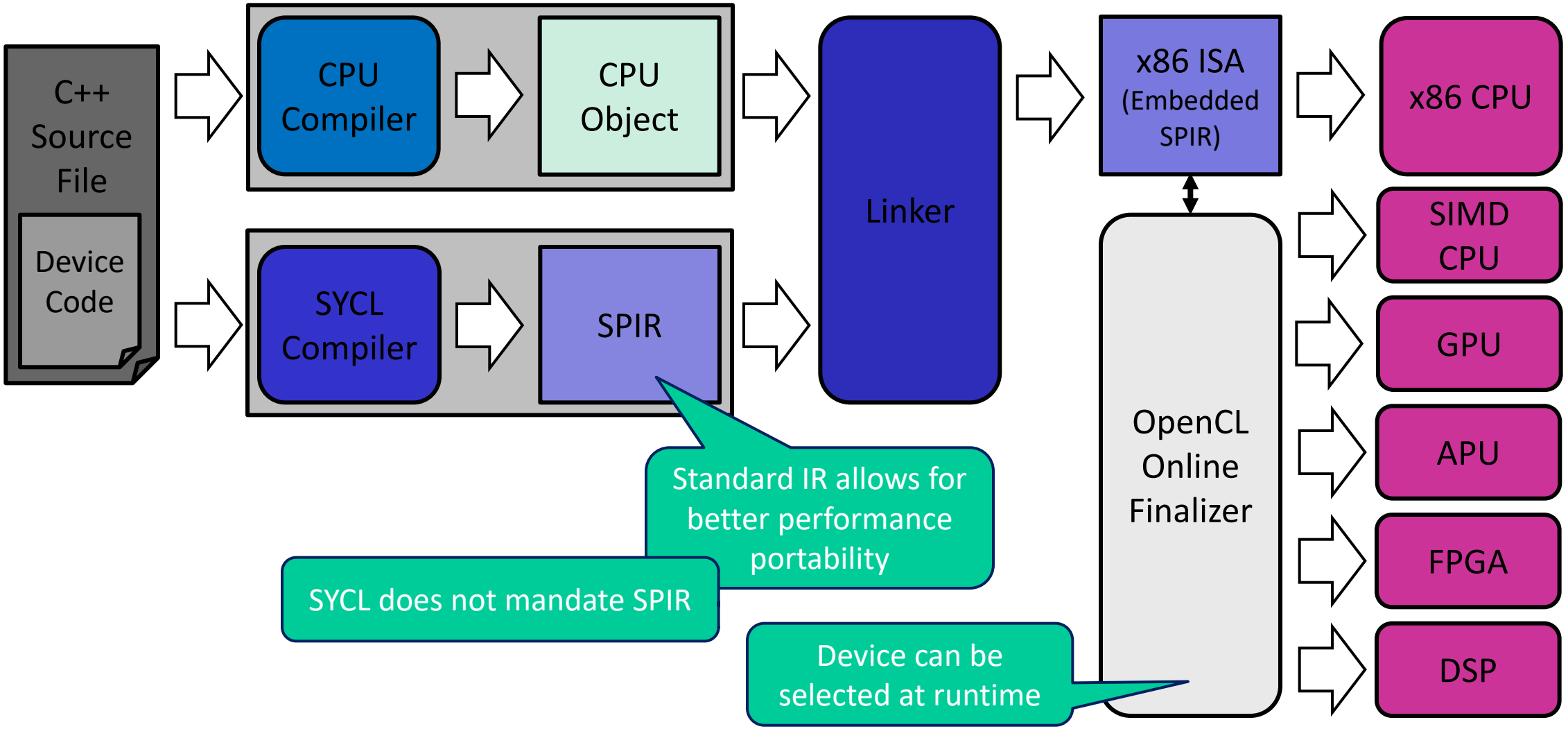
Multi Compilation Model



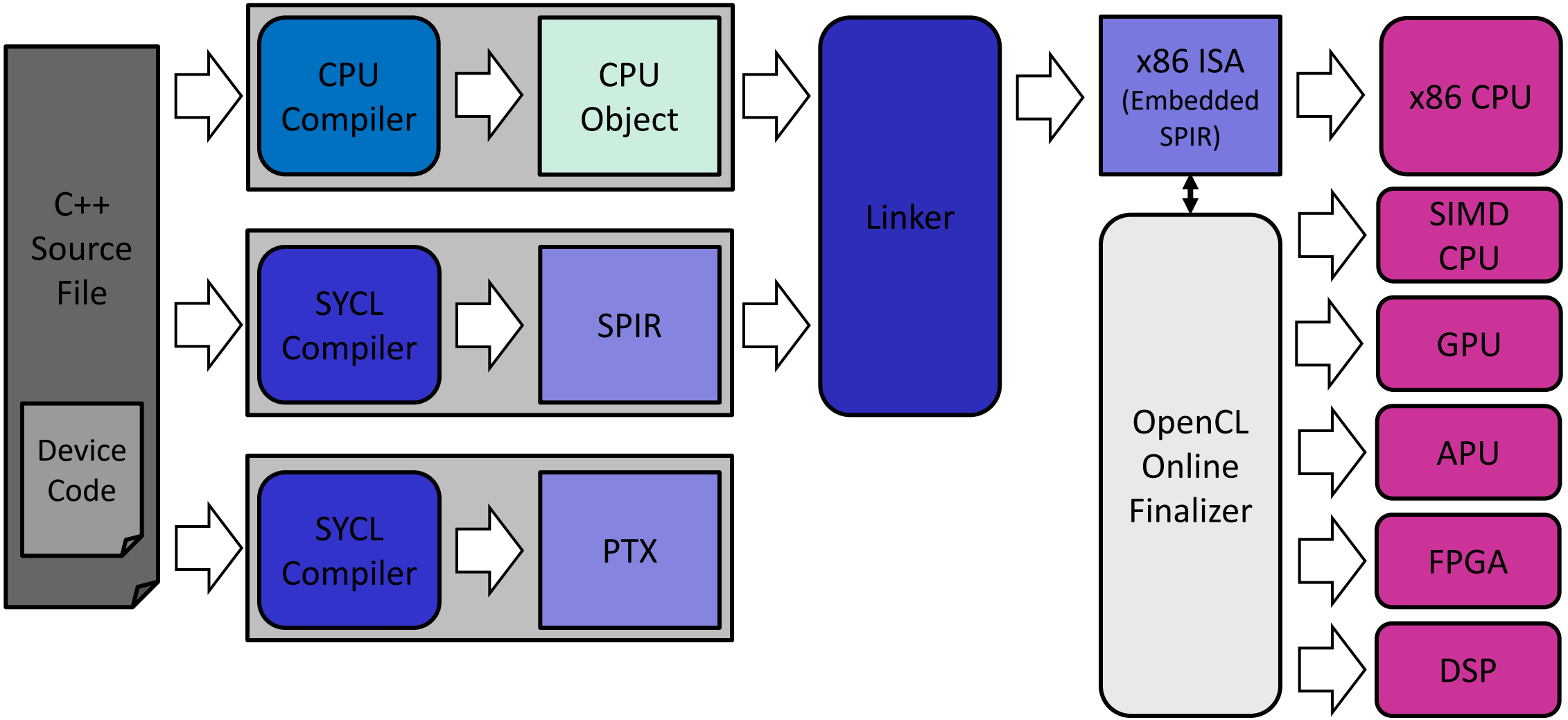
Multi Compilation Model



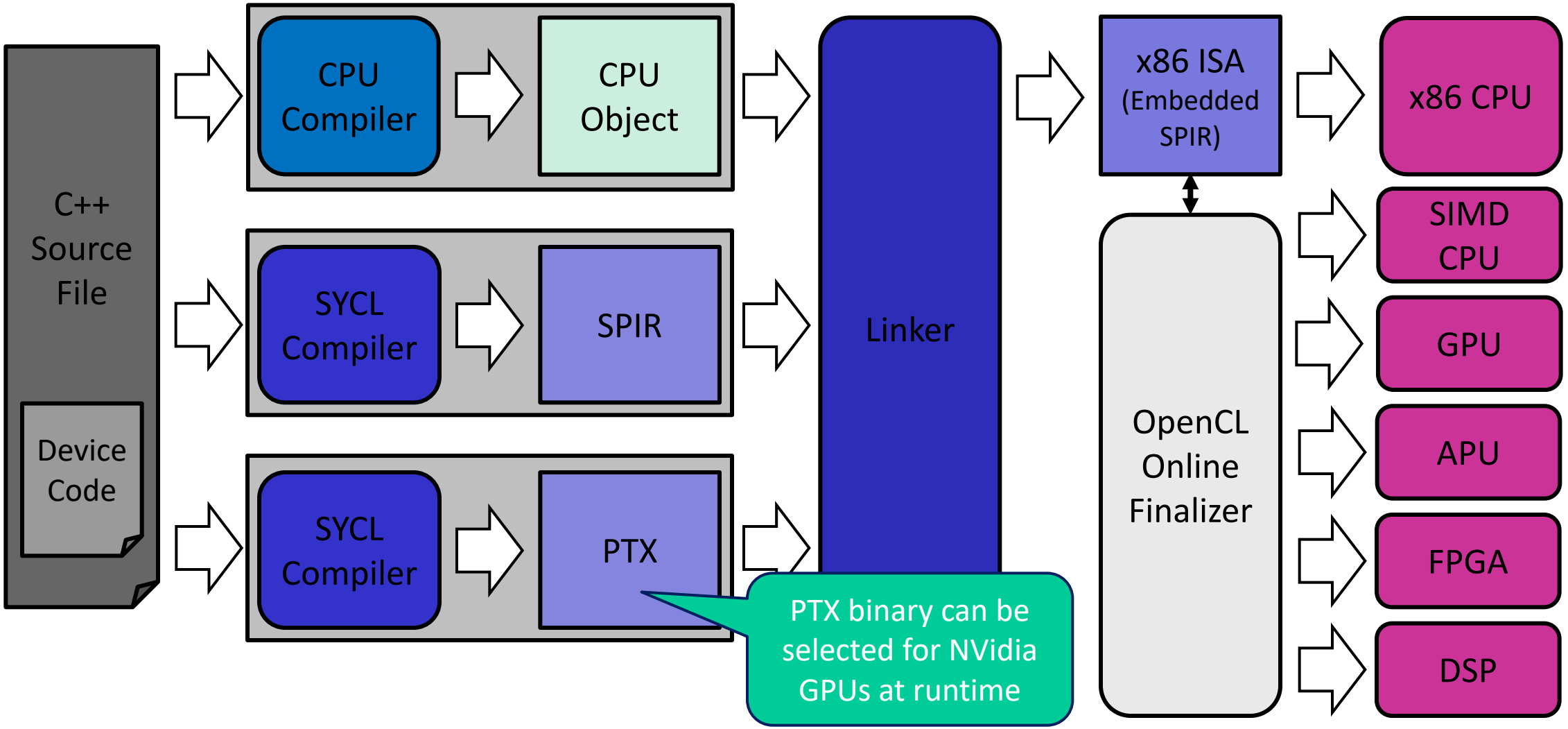
Multi Compilation Model



Multi Compilation Model



Multi Compilation Model



How does SYCL support different ways of representing parallelism?

- SYCL is an explicit parallelism model
- SYCL is a queue execution model
- SYCL supports both task and data parallelism

Representing Parallelism

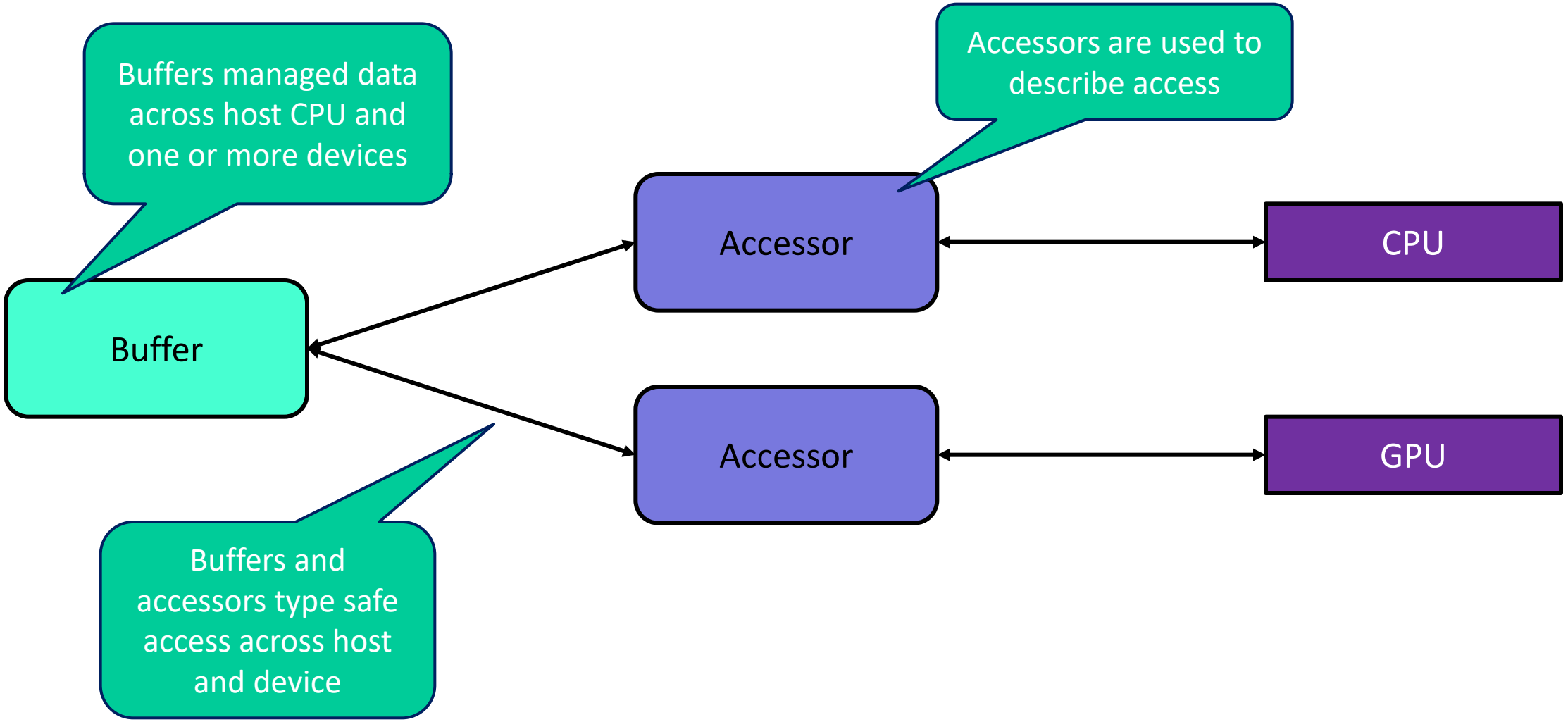
```
cgh.single_task([=](){

  /* task parallel task executed once*/

});
```

```
cgh.parallel_for(range<2>(64, 64), [=](id<2> idx){

  /* data parallel task executed across a range */

});
```

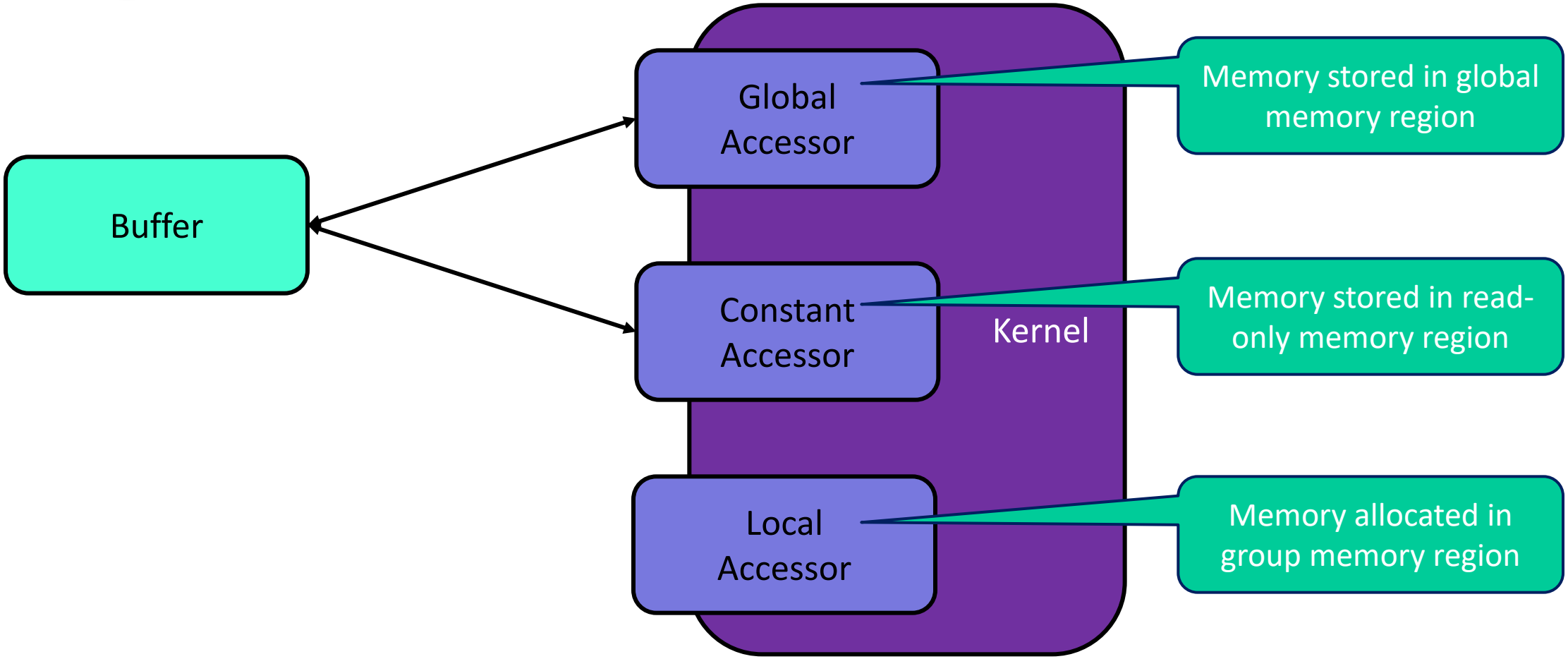
How does SYCL make data movement more efficient?

- SYCL separates the storage and access of data
- SYCL can specify where data should be stored/allocated
- SYCL creates automatic data dependency graphs

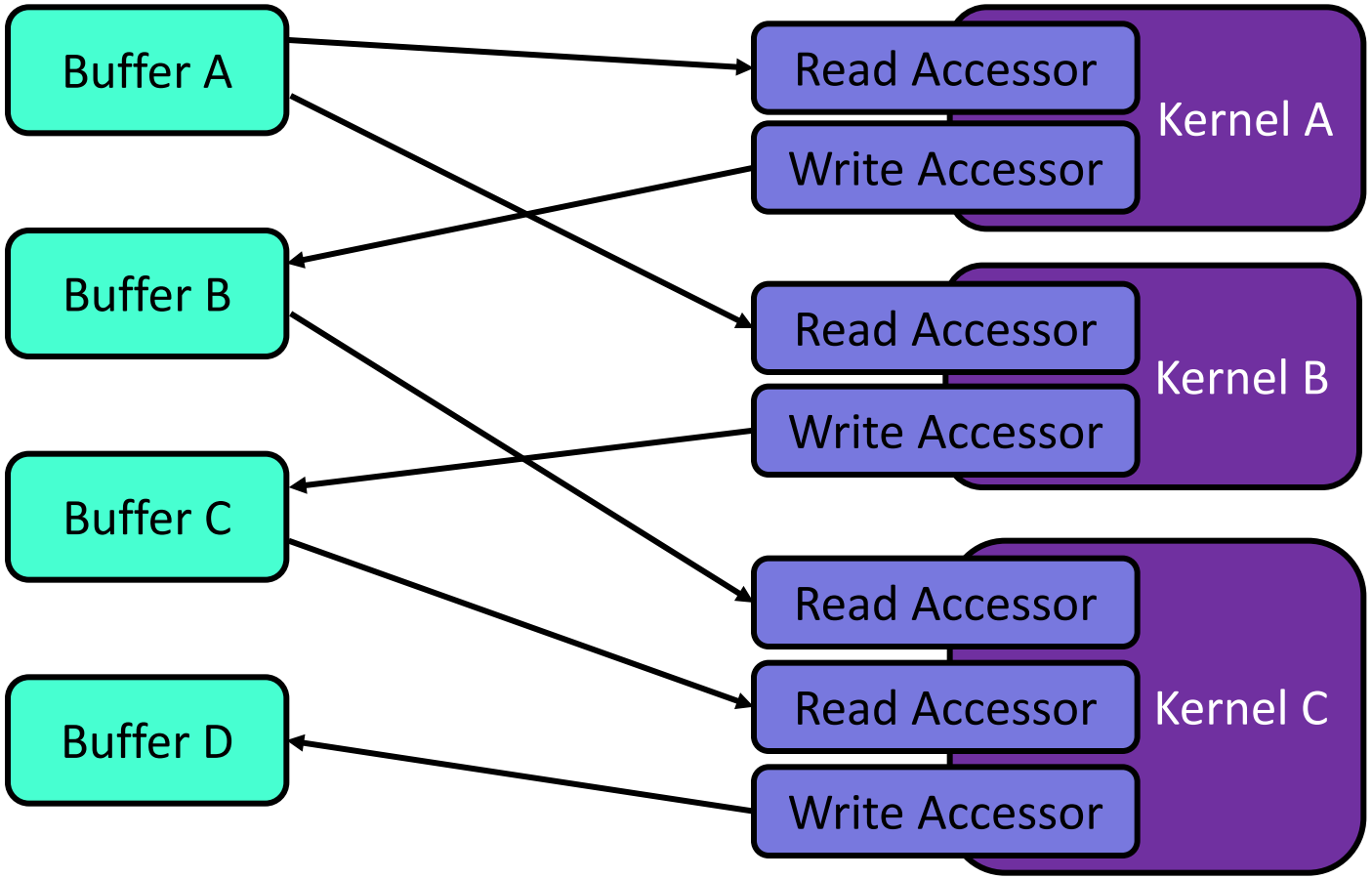
Separating Storage & Access

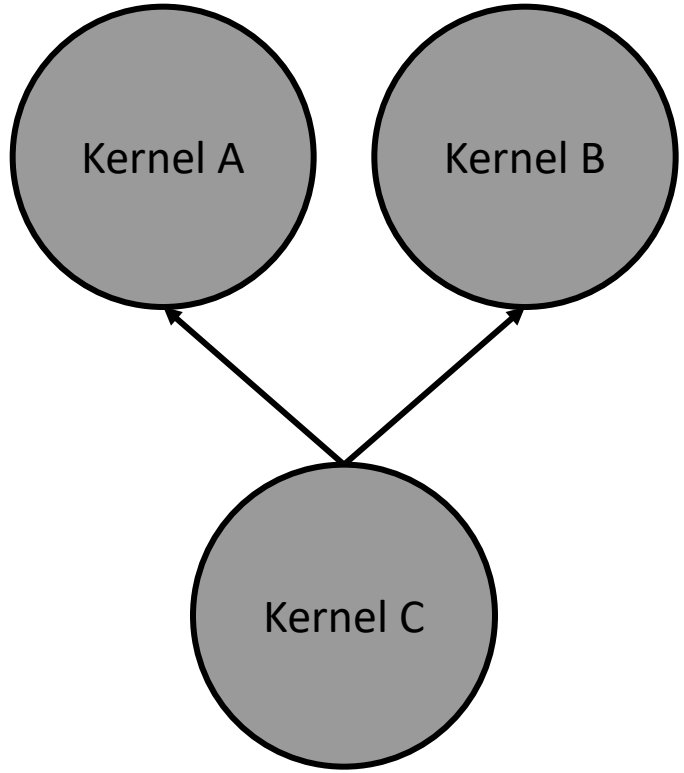


Storing / Allocating Memory in Different Regions



Data Dependency Task Graphs





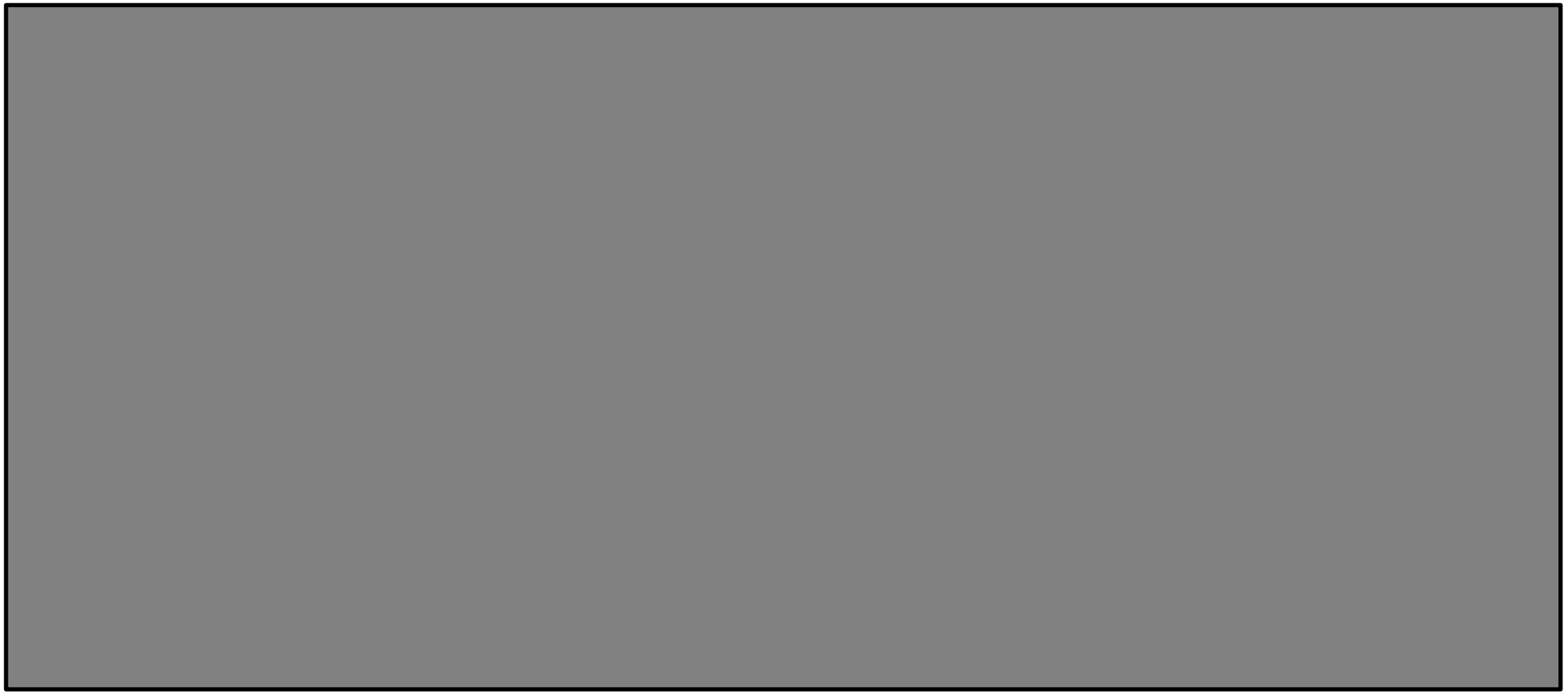
Benefits of Data Dependency Graphs

- **Allows you to describe your problems in terms of relationships**
  - Don’t need to en-queue explicit copies
- **Removes the need for complex event handling**
  - Dependencies between kernels are automatically constructed
- **Allows the runtime to make data movement optimizations**
  - Pre-emptively copy data to a device before kernels
  - Avoid unnecessarily copying data back to the host after execution on a device
  - Avoid copies of data that you don’t need

So what does SYCL look like?

- Here is a simple example SYCL application; a vector add

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, count);
}
```

Single-source vs C++ kernel language

- **Single-source: a single-source file contains both host and device code**
  - Type-checking between host and device
  - A single template instantiation can create all the code to kick off work, manage data and execute the kernel
    - **e.g. sort<MyClass> (myData);**
  - The approach taken by C++ 17 Parallel STL as well as SYCL

- **C++ kernel language**
  - Matches standard OpenCL C
  - Proposed for OpenCL v2.1
  - Being considered as an addition for SYCL v2.1

Why ‘name’ kernels?

- **Enables implementers to have multiple, different compilers for host and different devices**
  - With SYCL, software developers can choose to use the best compiler for CPU and the best compiler for each individual device they want to support
  - The resulting application will be highly optimized for CPU *and* OpenCL devices
  - Easy-to-integrate into existing build systems

- **Only required for C++11 lambdas, not required for C++ *functors***
  - Required because lambdas don’t have a name to enable linking between different compilers

KHRONOS GROUP

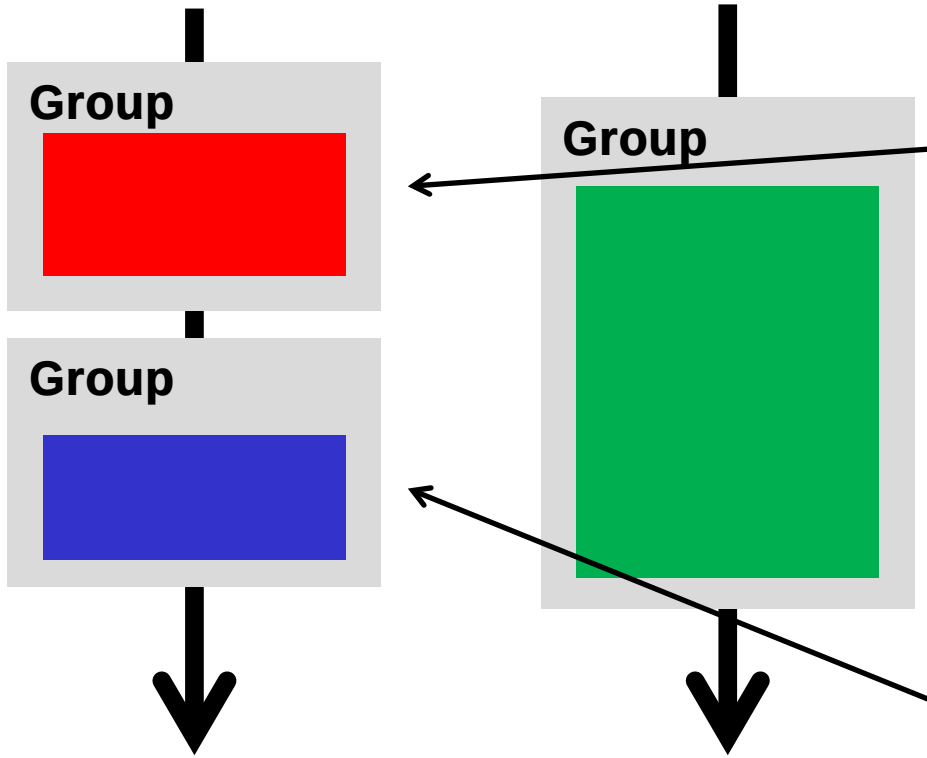
Buffers/images/accessors vs shared pointers

- **OpenCL v1.2 supports a wide range of different devices and operating systems**
  - All shared data must be encapsulated in OpenCL memory objects: buffers and images
  - To enable SYCL to achieve maximum performance of OpenCL, we follow OpenCL's memory model approach
  - But, we apply OpenCL's memory model to C++ with buffers, images and accessors
  - Separation of data storage and data access

Hierarchical parallelism

- A whole new approach
- Enables high-performance, portable C++ template algorithms to work across CPUs, GPUs and other devices easily
- Is really just syntactical

Task Graph Deduction



Efficient Scheduling

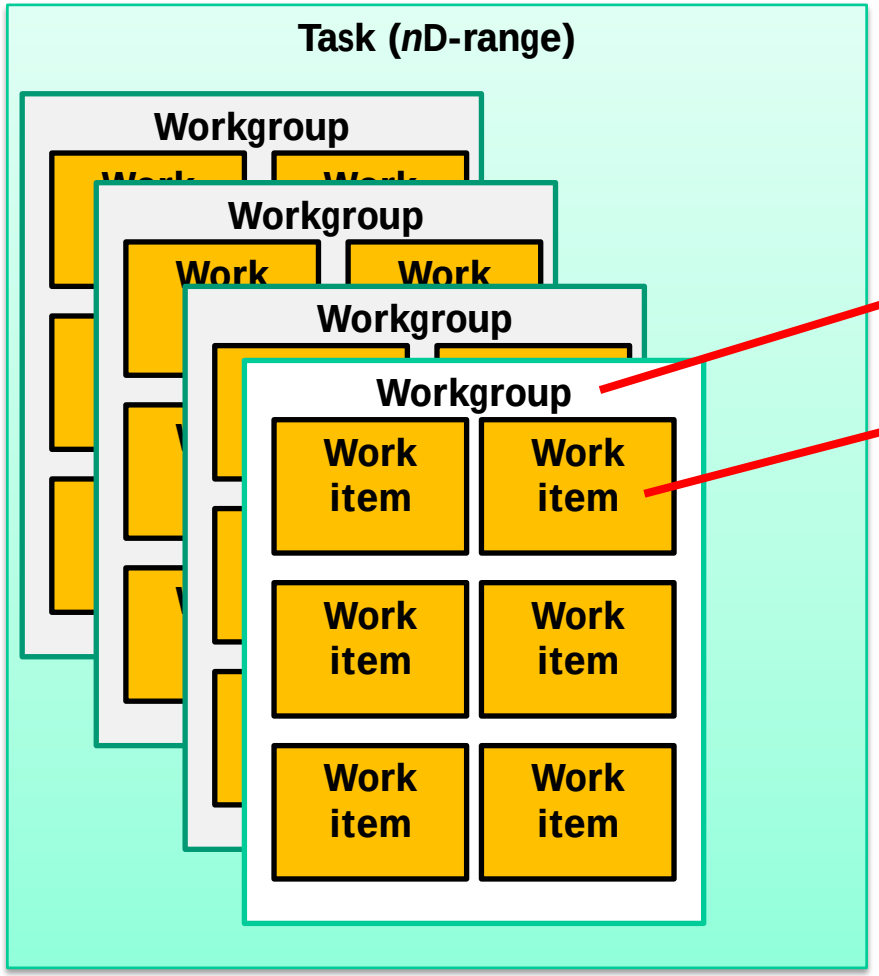
```
const int n_items = 32;
range<1> r(n_items);

int array_a[n_items] = { 0 };
int array_b[n_items] = { 0 };

buffer<int, 1> buf_a(array_a, range<1>(r));
buffer<int, 1> buf_b(array_b, range<1>(r));

queue q;
q.submit([&](handler& cgh)
{
  auto acc_a = buf_a.get_access<read_write>(cgh);
           algorithm_a s(acc_a);
           cgh.parallel_for(n_items, s);
});

q.submit([&](handler& cgh)
{
  auto acc_b = buf_b.get_access<read_write>(cgh);
           algorithm_b s(acc_b);
           cgh.parallel_for(n_items, s);
});


q.submit([&](handler& cgh)
{
  auto acc_a = buf_a.get_access<read_write>(cgh);
           algorithm_c s(acc_a);
           cgh.parallel_for(n_items, s);
});
```

KHRONOS GROUP

Data access with *accessors*

- **Encapsulates the difference between data storage and data access**
- **Allows creation of a parallel task graph with schedule, synchronization and data movement**
- **Enables devices to use optimal access to data**
  - Including having different pointer sizes on the device to those on the host
  - Allows usage of different address spaces for different data
- **Enhanced with call-graph-duplication (for C++ pointers) and explicit pointer classes**
  - To enable direct pointer-like access to data on the device
- **Portable, because accessors can be implemented as raw pointers**

Hierarchical Data Parallelism




```
buffer<int> my_buffer(data, range<1>(10));

q.submit([&](handler& cgh)
{
   auto in_access = my_buffer.access<cl::sycl::access:read>(cgh);
   auto out_access = my_buffer.access<cl::sycl::access:write>(cgh);

   cgh.parallel_for_workgroup<class hierarchical>
      (nd_range<1>(range<1> (size), range<1> (groupsize)), (group<1> grp)
   {
      parallel_for_workitem(grp, [=](item<1> tile)
      {
         out_access[tile] = in_access[tile] * 2;
      });
   }));
});
```

Advantages:

1. Easy to understand the concept of work-groups
2. Performance-portable between CPU and GPU
3. No need to think about barriers (automatically deduced)
4. Easier to compose components & algorithms

KHRONOS GROUP

What can I do with SYCL?

Anything you can do with C++!

With the performance and portability of OpenCL

Progress report on the SYCL vision

- ✓ Open, royalty-free standard: released
- ✓ Conformance testsuite: going into adopters package

- ➢ Open-source implementation: in progress (triSYCL)
- ➢ Commercial, conformant implementation: in progress
- ➢ C++ 17 Parallel STL: open-source in progress

- Template libraries for important C++ algorithms: getting going
- Integration into existing parallel C++ libraries: getting going

KHRONOS GROUP

Building the SYCL for OpenCL ecosystem

- **To deliver on the full potential of high-performance heterogeneous systems**
  - We need the libraries
  - We need integrated tools
  - We need implementations
  - We need training and examples

- **An open standard makes it much easier for people to work together**
  - SYCL is a group effort
  - We have designed SYCL for maximum ease of integration

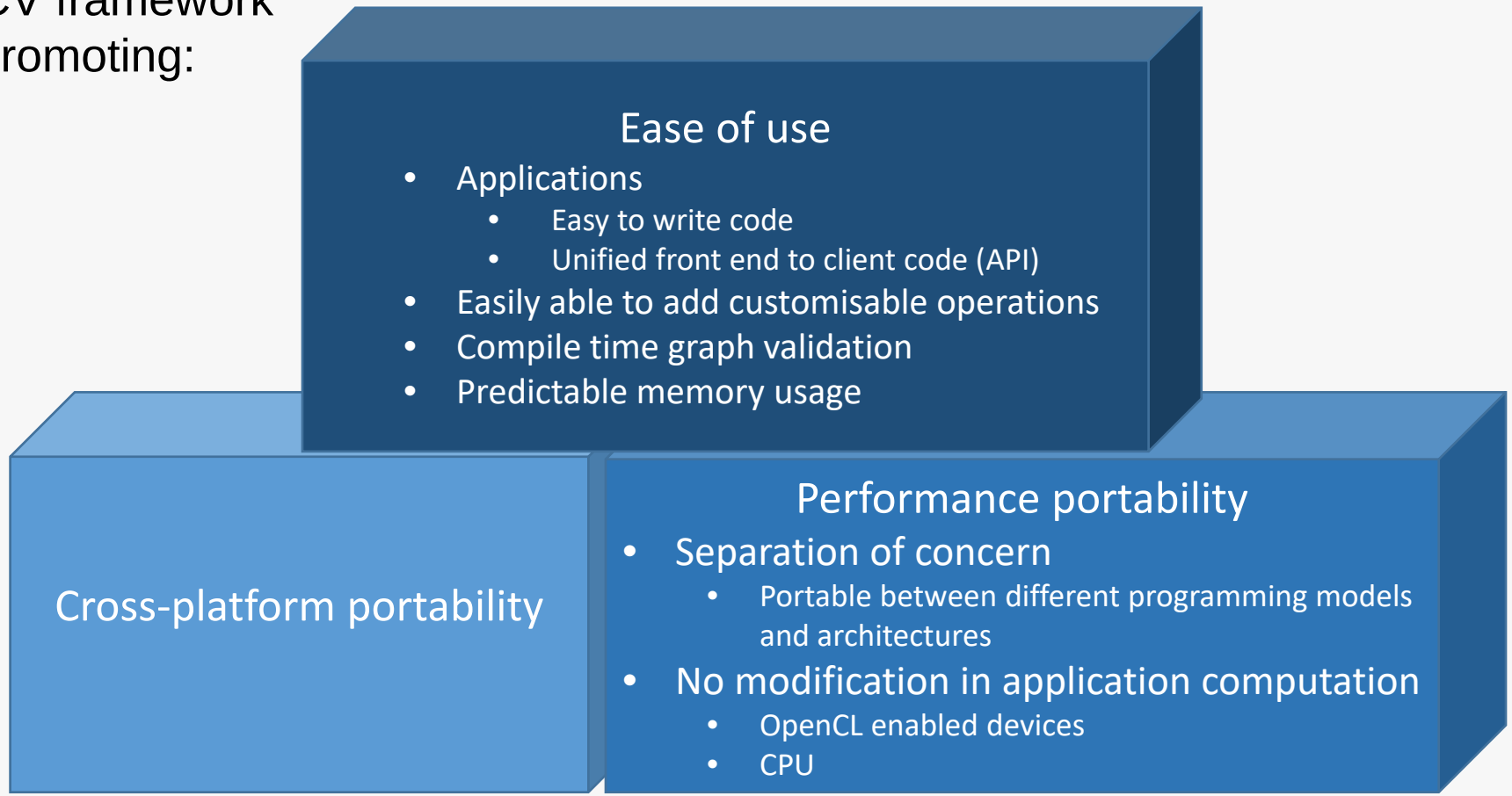
Agenda

- **Introduction to Codeplay**
- **SYCL: The open Khronos standard**
  - A comparison of Heterogeneous Programming Models
  - SYCL Design Philosophy: C++ end to end model for HPC and consumers
- **The ecosystem:**
  - VisionCpp
  - Parallel STL
  - TensorFlow, Machine Vision, Neural Networks, Self-Driving Cars
- **Codeplay ComputeCPP Community Edition: Free Download**

VisionCpp™

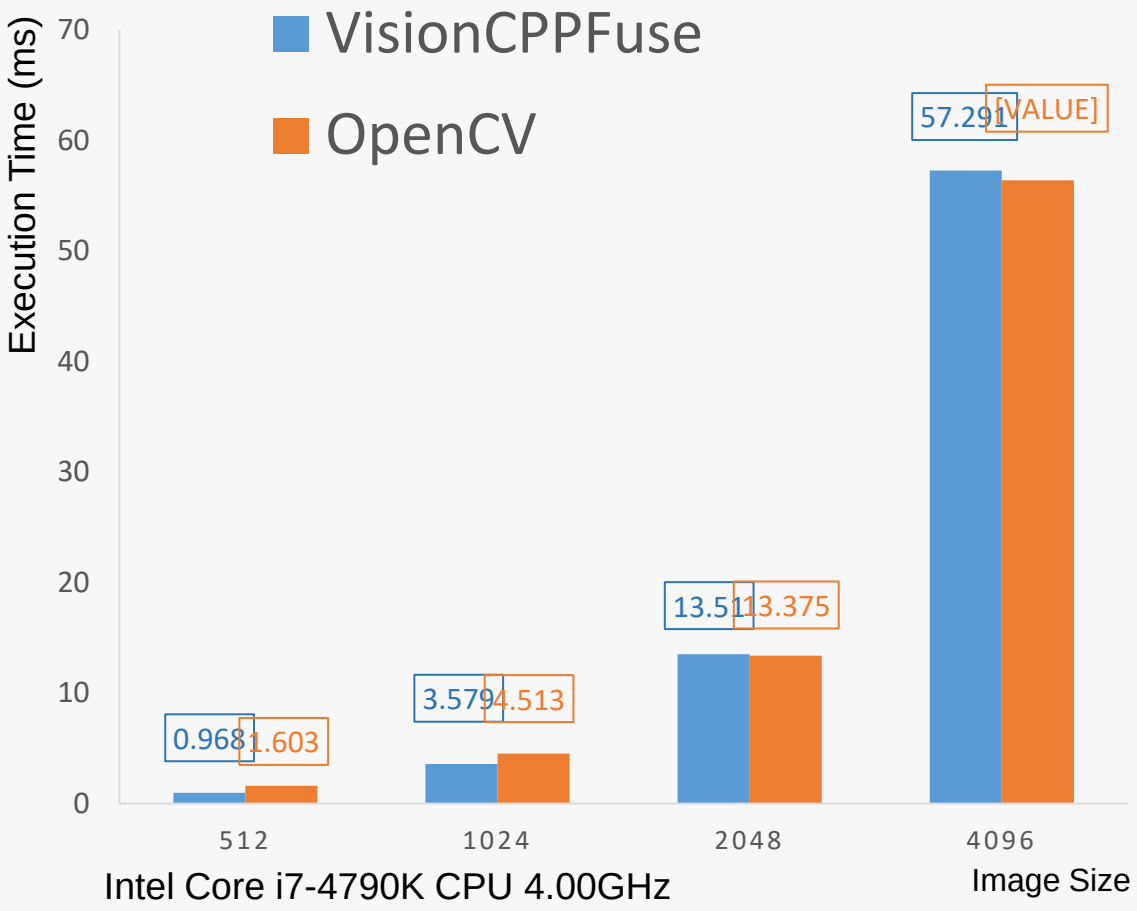
Using SYCL to Develop Vision Tools

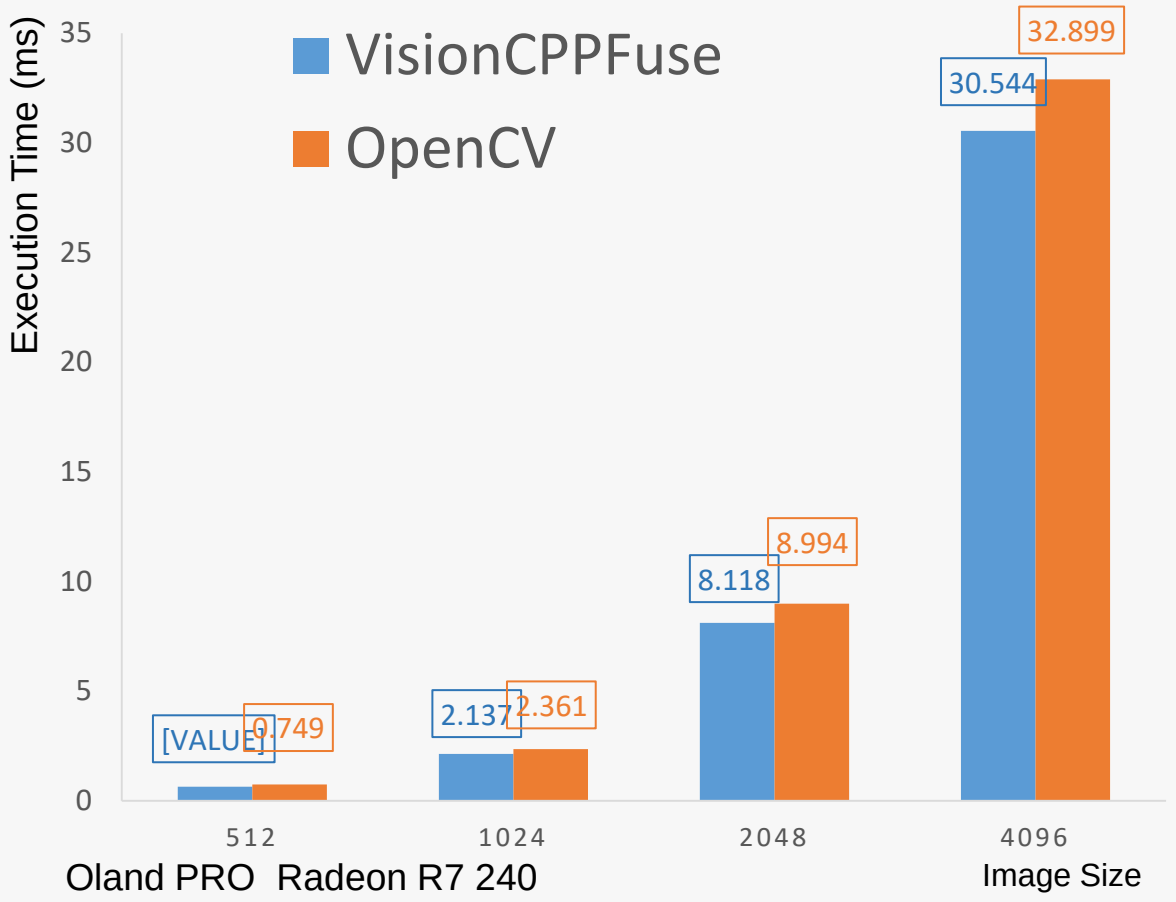
A high-level CV framework for OpenCL promoting:



VisionCpp™

A worthy addition to your tool kit





Parallel STL: Democratizing Parallelism in C++

- Various libraries offered STL-like interface for parallel algorithms
  - Thrust, Bolt, libstdc++ Parallel Mode, AMP algorithms
- In 2012, two separate proposals for parallelism to C++ standard:
  - NVIDIA (N3408), based on Thrust (CUDA-based C++ library)
  - Microsoft and Intel (N3429), based on Intel TBB andPPL/C++AMP
- Made joint proposal (N3554) suggested by SG1
  - Many working drafts for N3554, N3850, N3960, N4071, N4409
- Final proposal P0024R2 accepted for C++17 during Jacksonville

Existing implementations

- Following the evolution of the document
  - Microsoft: http://parallelstl.codeplex.com
  - HPX: http://stellar-group.github.io/hpx/docs/html/hpx/manual/parallel.html
  - HSA: http://www.hsafoundation.com/hsa-for-math-science
  - Thibaut Lutz: http://github.com/t-lutz/ParallelSTL
  - NVIDIA: http://github.com/n3554/n3554
  - Codeplay: http://github.com/KhronosGroup/SyclParallelSTL

What is Parallelism TS v1 adding?

- A set of execution policies and a collection of parallel algorithms
  - The exception_list object
  - The **Execution Policies**
  - Paragraphs explaining the conditions for parallel algorithms
  - New parallel algorithms

Sorting with the STL

A sequential sort

```
std :: vector <int > data = { 8, 9, 1, 4 };
std :: sort ( std :: begin ( data ), std :: end( data ));
if ( std :: is_sorted ( data )) {
    cout << " Data is sorted ! " << endl ;
}
```

A parallel sort

```
std :: vector <int > data = { 8, 9, 1, 4 };
std :: sort ( std :: par , std :: begin ( data ), std :: end ( data ));
if ( std :: is_sorted ( data )) {
    cout << " Data is sorted ! " << endl ;
}
```

- **`par`** is an object of an *Execution Policy*
- The sort will be executed in parallel using an implementation-defined method

The SYCL execution policy

```
std :: vector <int > data = { 8, 9, 1, 4 };
std :: sort ( sycl_policy , std :: begin (v), std :: end (v));
if ( std :: is_sorted ( data )) {
    cout << " Data is sorted ! " << endl ;
}
```

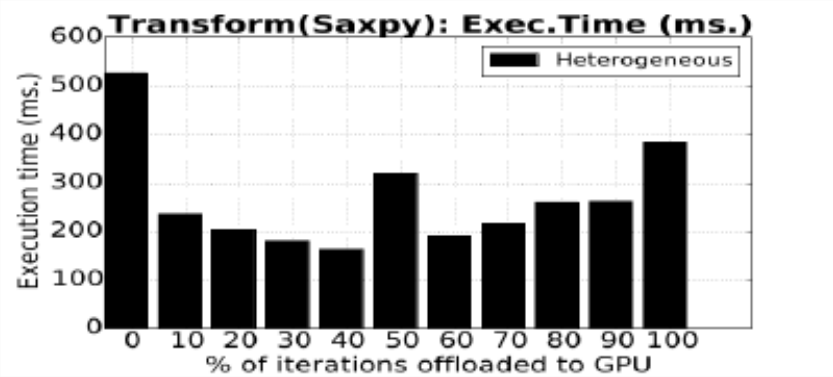
```
template <typename KernelName = DefaultKernelName >
class sycl_execution_policy {
public :
    using kernelName = KernelName ;
    sycl_execution_policy () = default ;
    sycl_execution_policy (cl :: sycl :: queue q);
    cl :: sycl :: queue get_queue () const ;
};
```

- sycl_policy is an *Execution Policy*
- data is a standard stl::vector
- Technically, will use the device returned by *default_selector*

Heterogeneous load balancing

Dynamic decision of heterogeneous balancing

- Percentage offloading is runtime value
- Developers can create runtime evaluation functions
  - → Depending on worlkload
  - → Depending on platform
  - → Depending on user-input



Future work

Parallel STL is an Open Source Project!

- You can contribute algorithms
- Or new policies
- Or device-specific optimizations/algorithms!

https://github.com/KhronosGroup/SyclParallelSTL

Other projects – in progress

TensorFlow: Google's machine learning library

Eigen: C++ linear algebra template library

+ others …

Conclusion

- Heterogeneous computing has been coming for a while now
- C++ is a prominent language for doing this
- SYCL, HPX, Agency, HCC, KoKKos, Raja allows you to program a wide range of heterogeneous devices with standard C++
- ComputeCpp is available now for you to download and use

Agenda

- Introduction to Codeplay
- SYCL: The open Khronos standard
  - A comparison of Heterogeneous Programming Models
  - SYCL Design Philosophy: C++ end to end model for HPC and consumers
- The ecosystem:
  - VisionCpp
  - Parallel STL
  - TensorFlow, Machine Vision, Neural Networks, Self-Driving Cars
- Codeplay ComputeCPP Community Edition: Free Download

Community Edition

Available now for free!

Visit:

computecpp.codeplay.com

- Open source SYCL projects:
  - ComputeCpp SDK - Collection of sample code and integration tools
  - SYCL ParallelSTL – SYCL based implementation of the parallel algorithms
  - VisionCpp – Compile-time embedded DSL for image processing
  - Eigen C++ Template Library – Compile-time library for machine learning

All of this and more at: http://sycl.tech

codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

Thank you for your time
Any questions?



@codeplaysoft info@codeplay.com codeplay.com