codeplay

THE HETEROGENEOUS SYSTEMS EXPERTS

Programming GPUs with SYCL

Gordon Brown – Staff Software Engineer, SYCL

C++ Edinburgh – July 2016

Agenda

- Introduction to GPGPU
  - Why program GPUs?
  - CPU vs GPU architecture
  - General GPU programming tips
- SYCL for OpenCL
  - Overview
  - Features
- SYCL example
  - Vector add

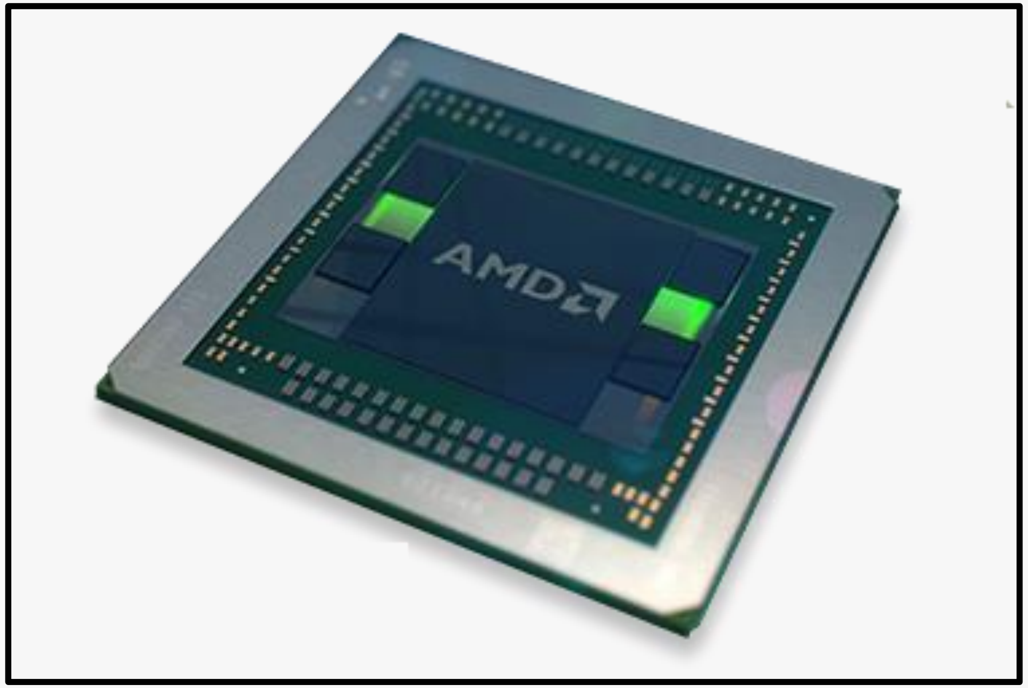
Introduction to GPGPU

Why Program GPUs?

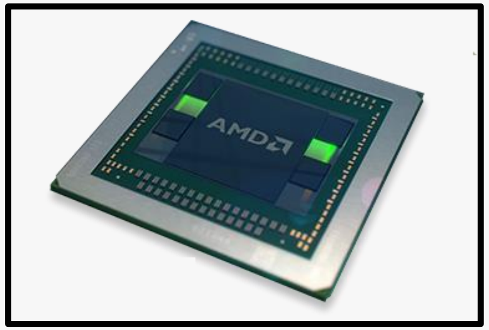
- Need for parallelism to gain performance
  - “Free lunch” provided by Moore’s law is over
  - Adding even more CPU cores is showing diminishing returns

- GPUs are extremely efficient for
  - Data parallel tasks
  - Arithmetic heavy computations

CPU vs GPU



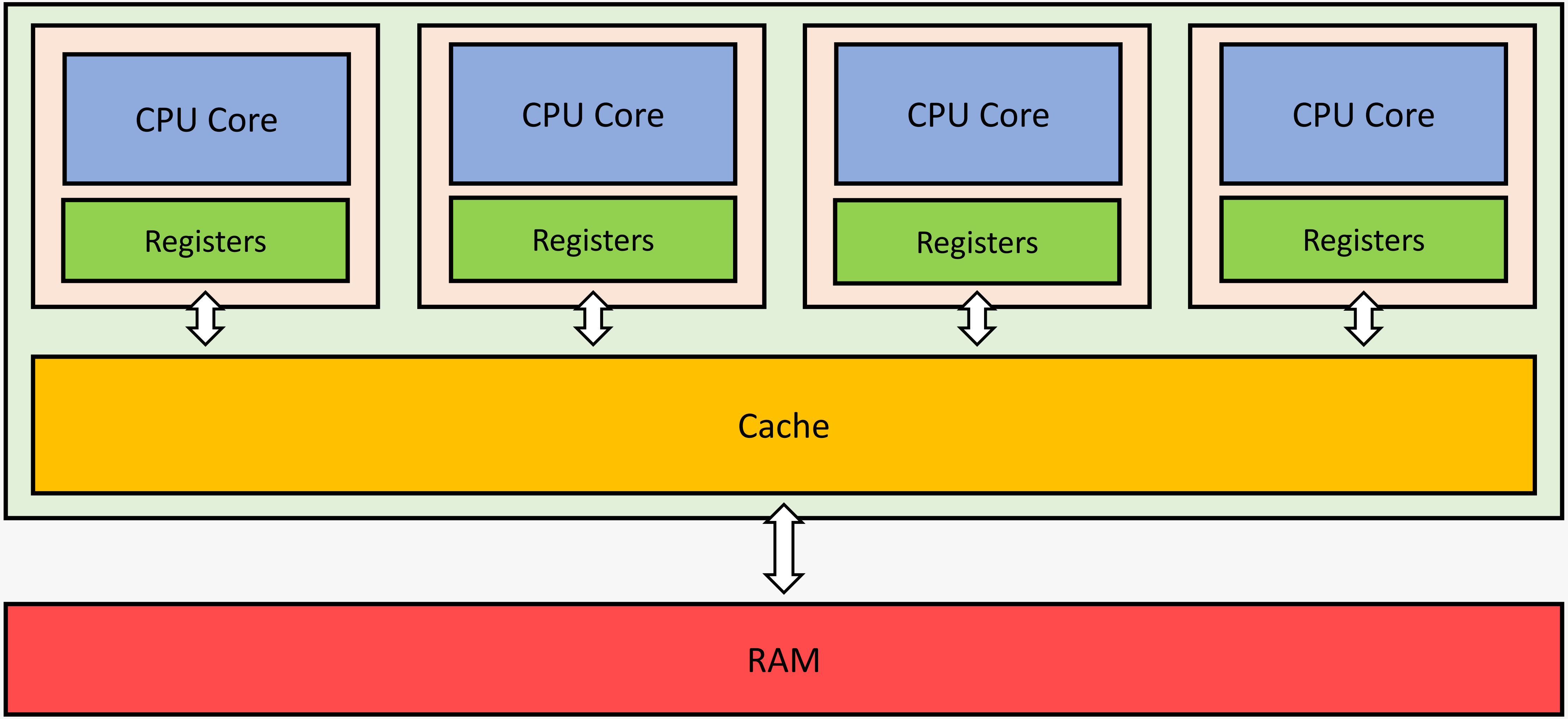
CPU vs GPU



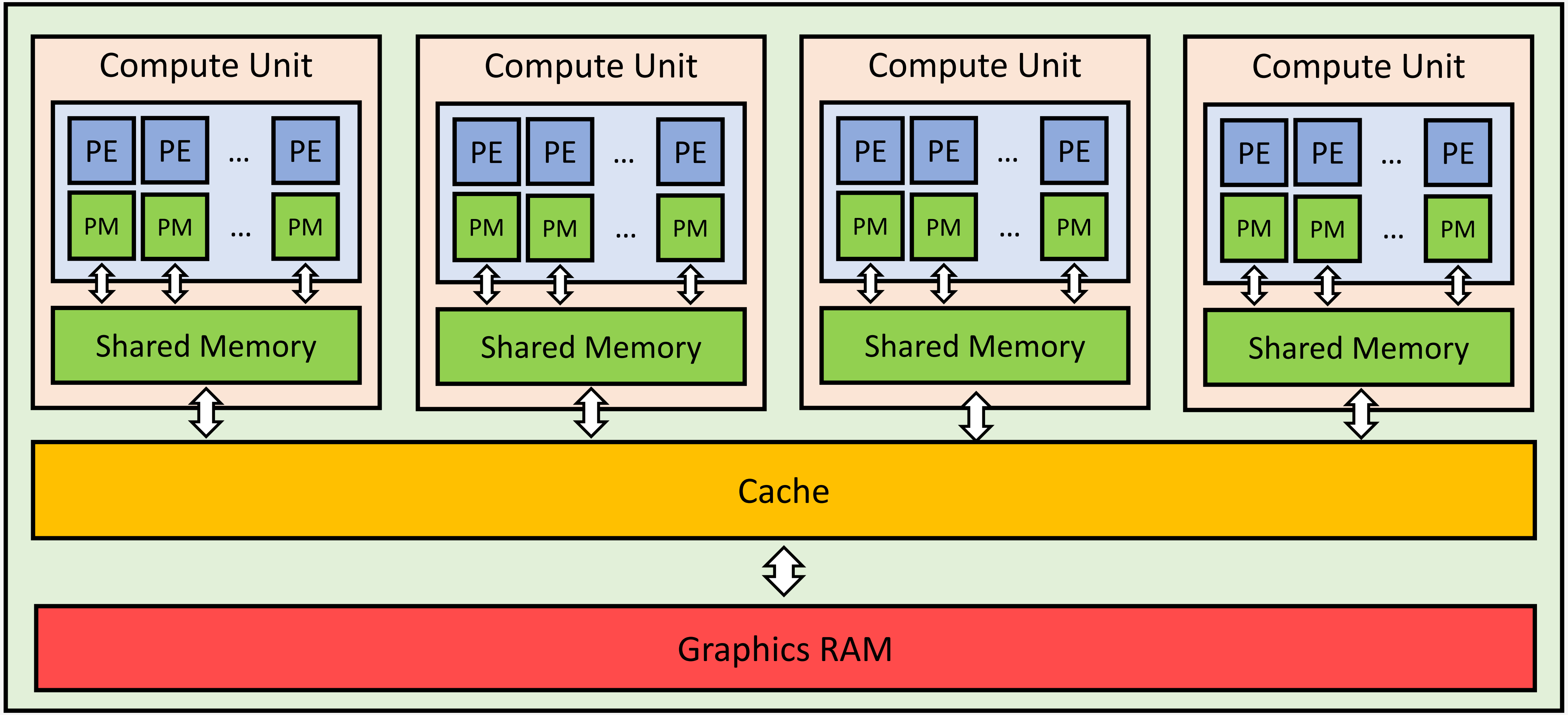
- Task parallelism
- Small number of large cores
- Separate instructions on each core independently
- Higher power consumption
- Lower memory bandwidth
- Random memory access

- Data parallelism
- Large number of small execution units
- Single instruction on all multiple execution units in lock-step
- Lower power consumption
- Higher memory bandwidth
- Sequential memory access

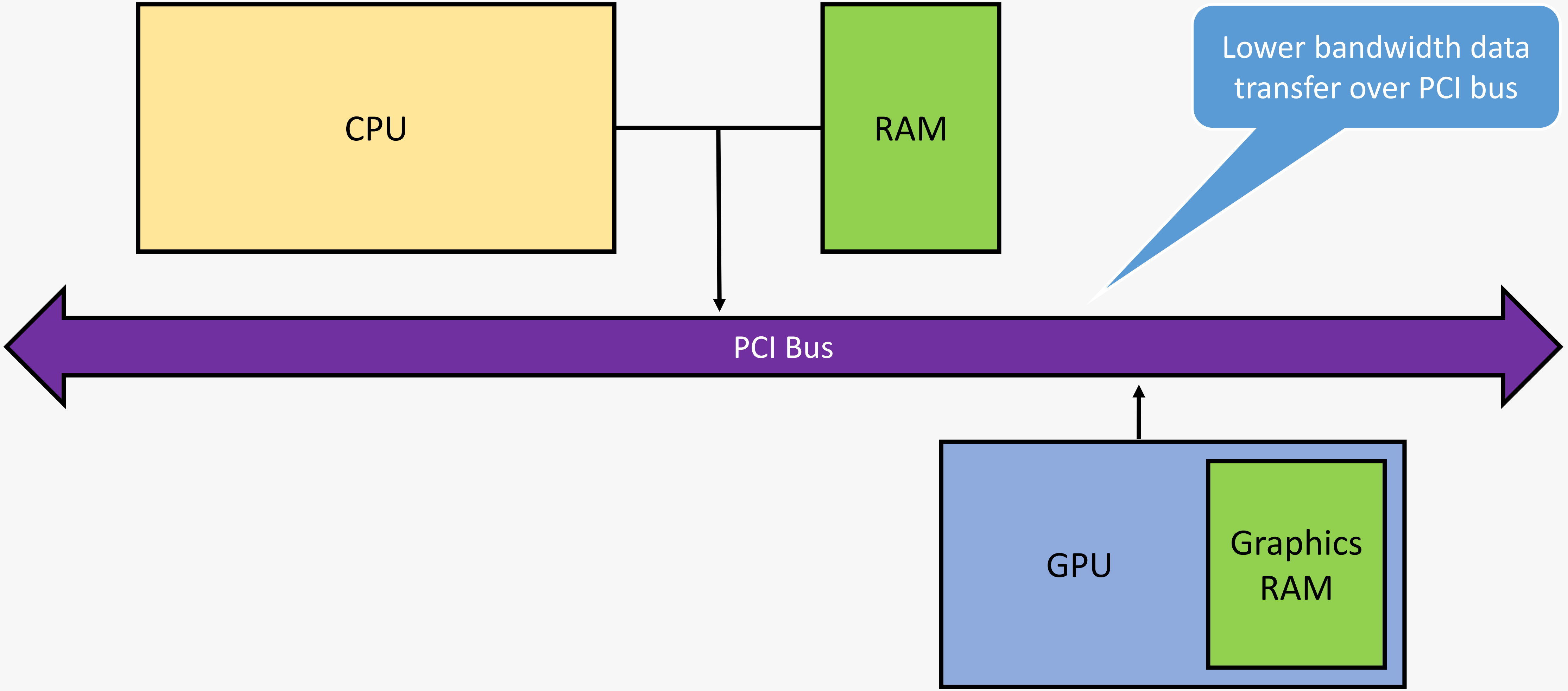
Common CPU Architecture



Common GPU Architecture



Common System Architecture

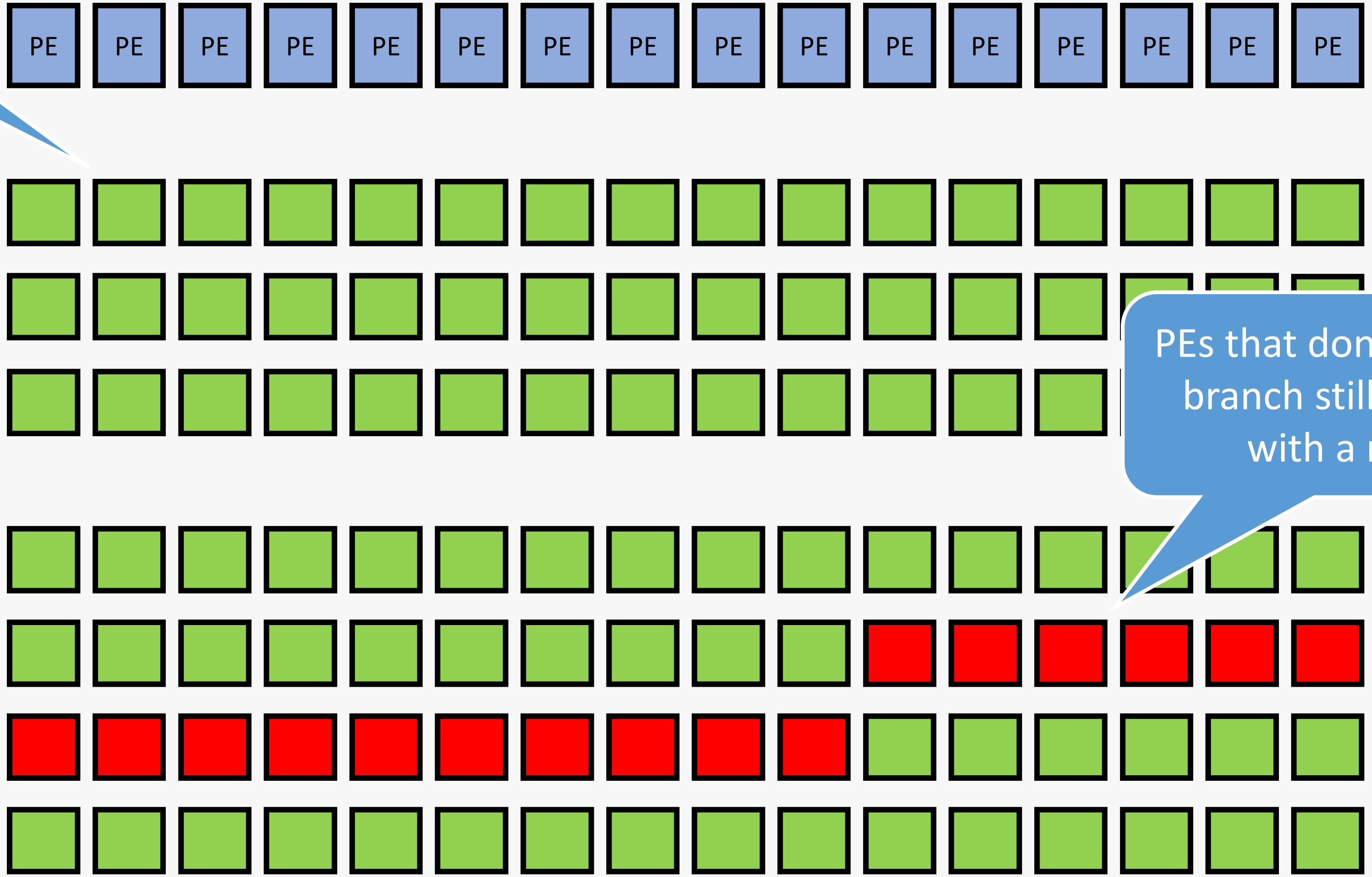


GPUs Execute in Lock-step

Waves of PEs are executed in lock-step

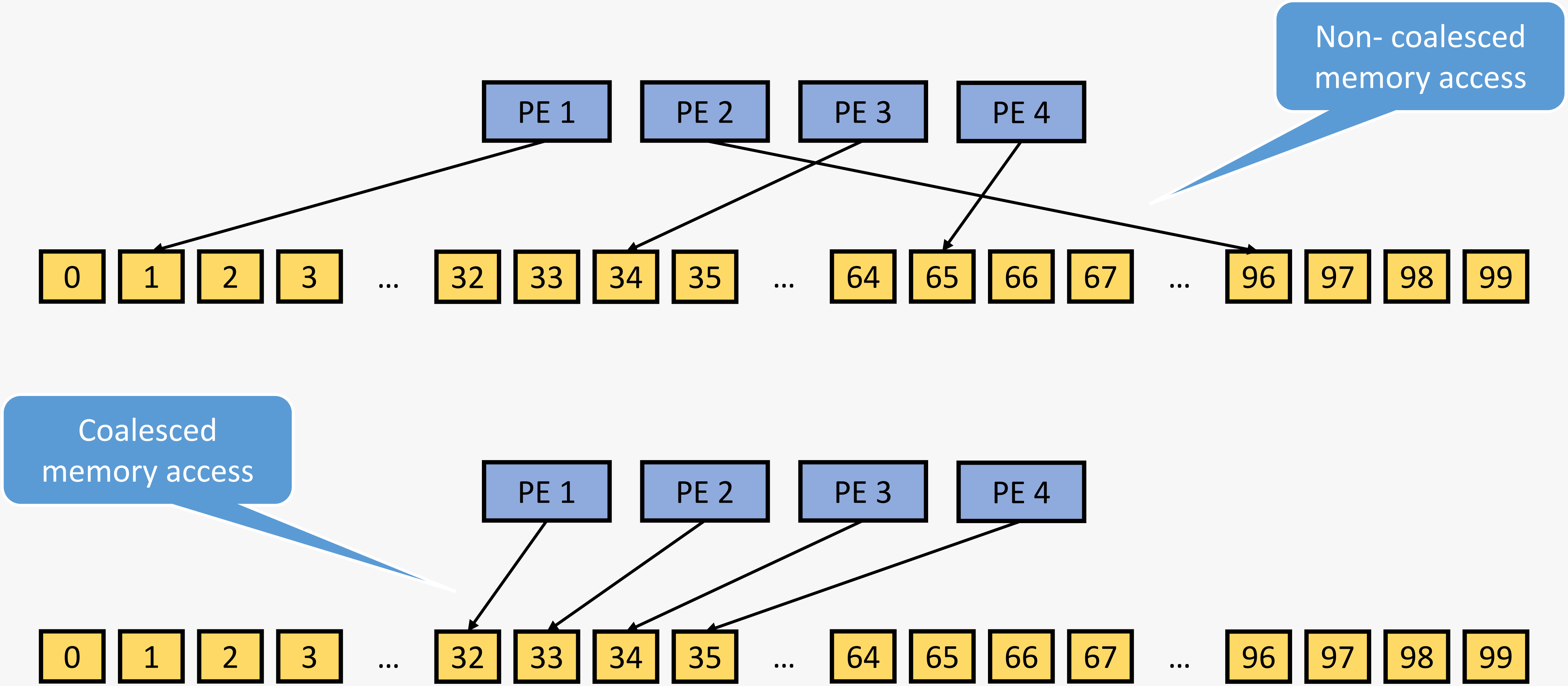
```
int v = 0;

v = foo(i);

a[i] = v;
```

```
if (i < 10)
{
  v = foo(i);
} else {
  v = bar(i);
}
a[i] = v
```



PEs that don’t follow a branch still execute with a mask

GPUs Access Memory Sequentially



General GPU Programming Tips

- Ensure the task is suitable
  - GPUs are most efficient for data parallel tasks
  - Performance gain from performing computation > cost of moving data
- Avoid branching
  - Waves of processing elements execute in lock-step
  - Both sides of branches execute with the other masked
- Avoid non-coalesced memory access
  - GPUs access memory more efficiently if accessed as contiguous blocks
- Avoid expensive data movement
  - The bottleneck in GPU programming is data movement between CPU and GPU memory
  - It’s important to have data as close to the processing as possible

SYCL for OpenCL

What is OpenCL?

- Allows you to write kernels that execute on accelerators
- Allows you to copy data between the host CPU and accelerators
- Supports a wide range of devices
- Comes in two components:
  - Host side C API for en-queueing kernels and copying data
  - Device side OpenCL C language for writing kernels

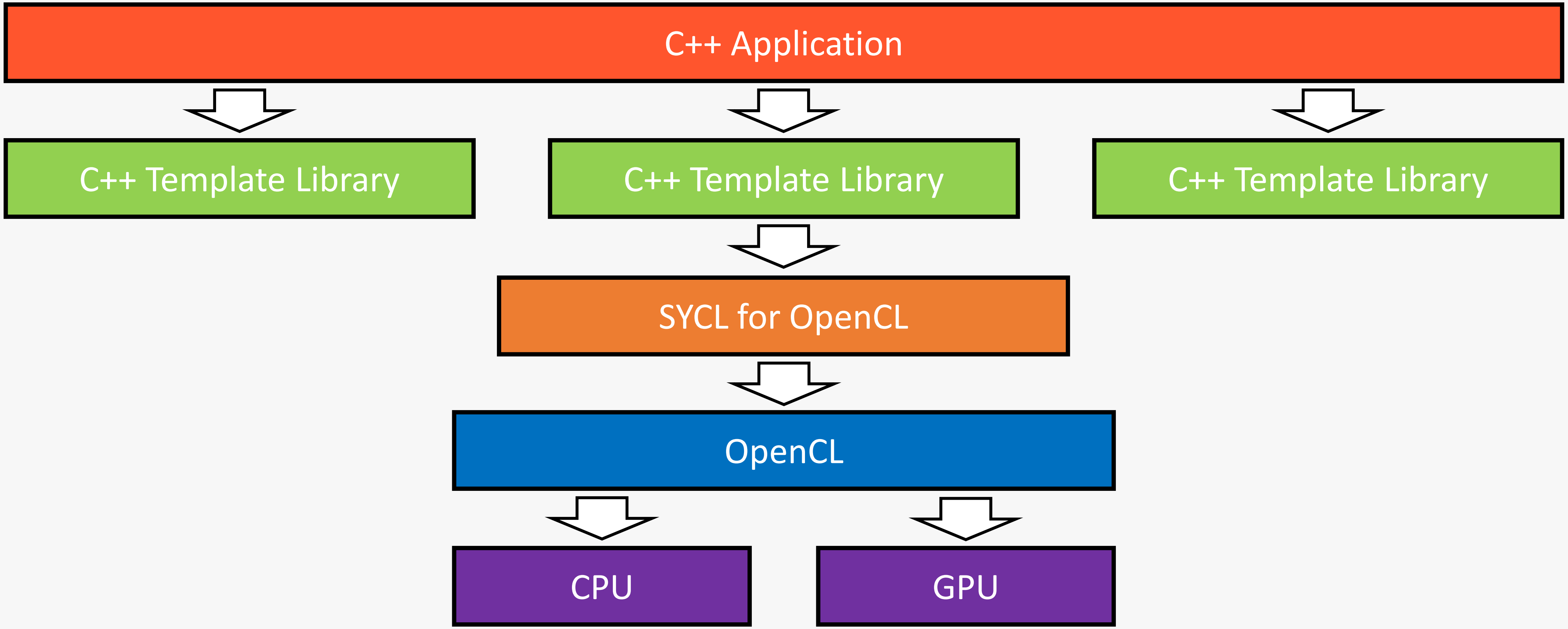
Motivation of SYCL

- Make heterogeneous programming more accessible
  - Provide a foundation for efficient and portable template algorithms
- Create a C++ for OpenCL ecosystem
  - Define an open portable standard
  - Provide the performance and portability of OpenCL
  - Base only on standard C++
- Provide a high-level shared source model
  - Provide a high-level abstraction over OpenCL boiler plate code
  - Allow C++ template libraries to target OpenCL
  - Allow type safety across host and device

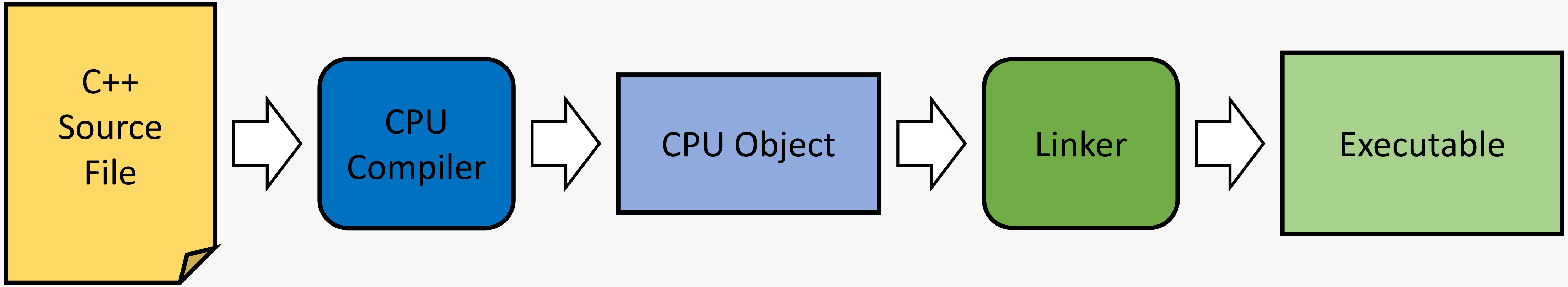
SYCL for OpenCL

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

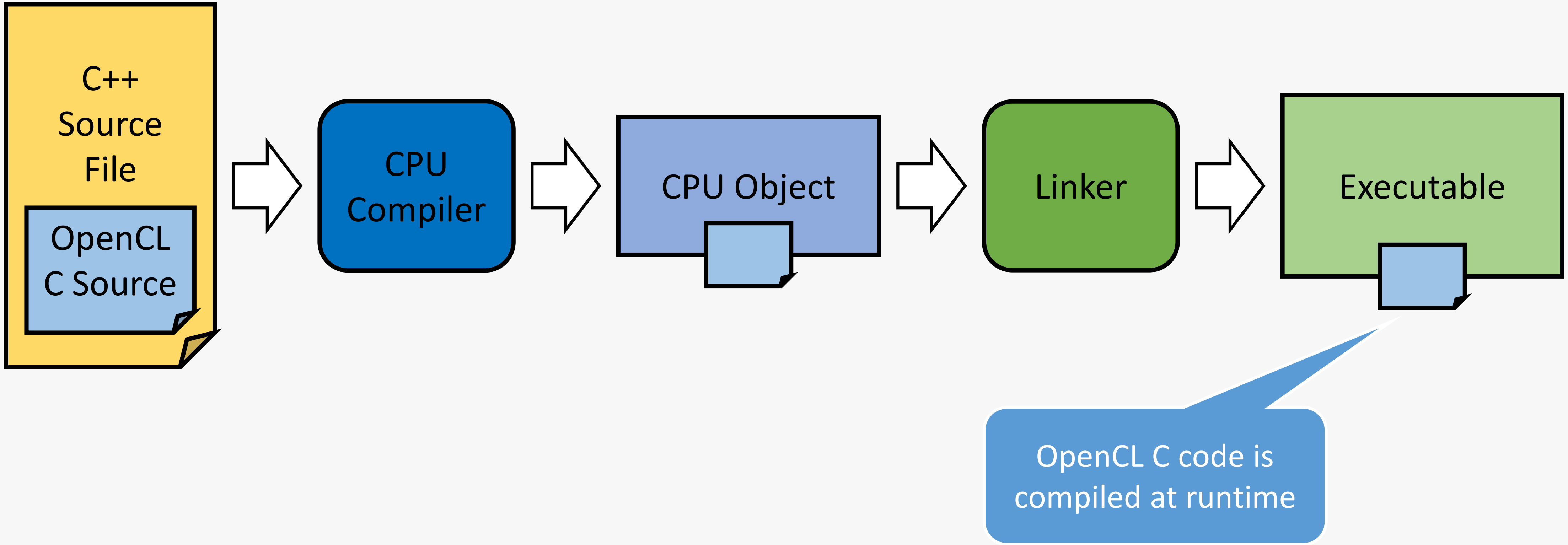
The SYCL Ecosystem



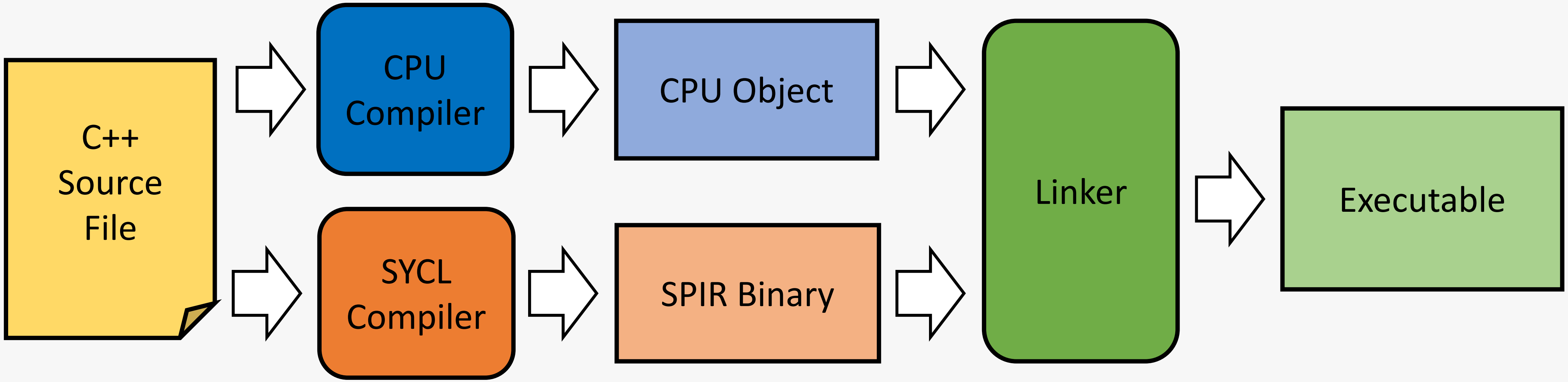
How Does Shared Source Work?: Regular C++ (Single Source)



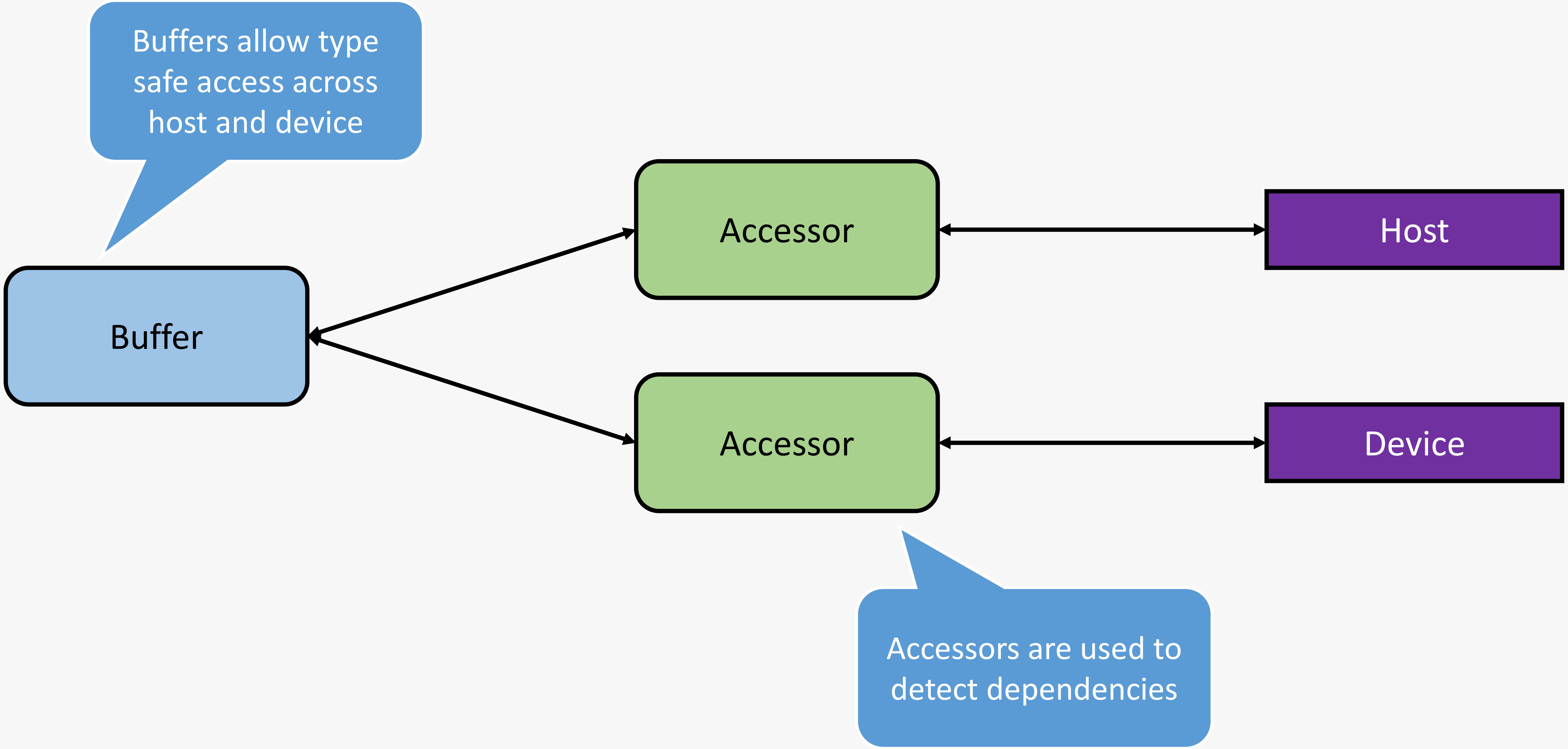
How Does Shared Source Work?: OpenCL (Separate Source)



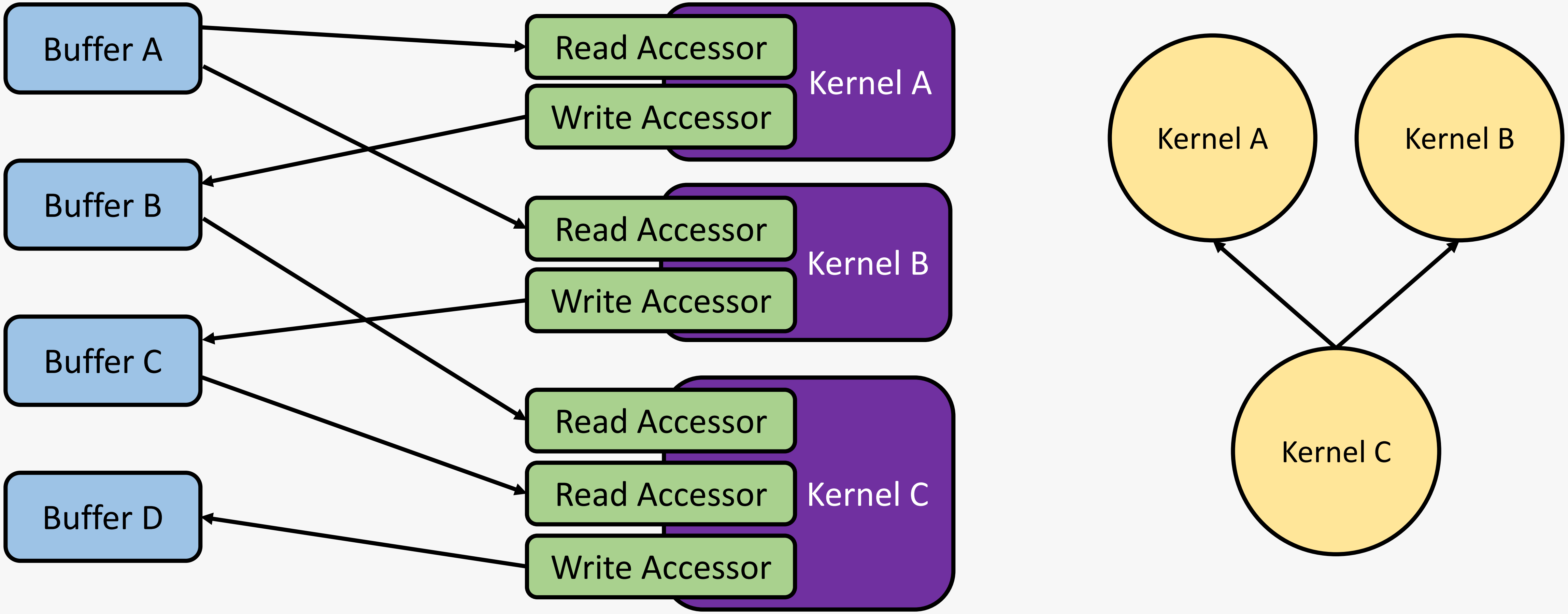
How Does Shared Source Work?: SYCL (Shared Source)



Separating Data & Access



Dependency Task Graphs



Supported Subset of C++ in Device Code

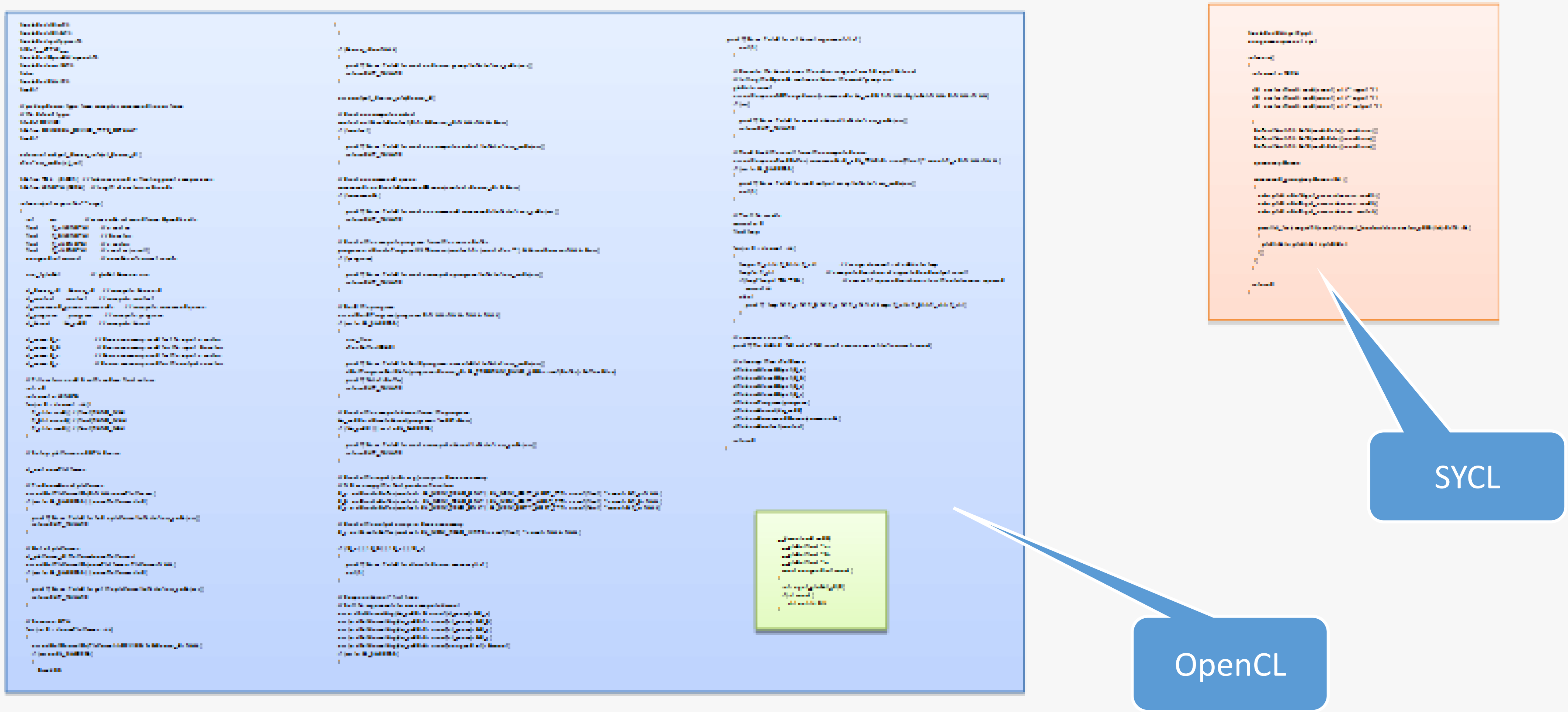
Supported Features

- Static polymorphism
- Lambdas
- Classes
- Operator overloading
- Templates
- Placement new

Non-supported features

- Dynamic polymorphism
- Dynamic allocation
- Exception handling
- RTTI
- Static variables
- Function pointers

Compact High-level API



Example: Vector Add

Example: Vector Add

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


template <typename T>
void parallel_add(T *inputA, T *inputB, T *output, size_t size) {













}
```

Include sycl.hpp for the whole SYCL runtime

Example: Vector Add

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


template <typename T>
void parallel_add(T *inputA, T *inputB, T *output, size_t size) {
  cl::sycl::buffer<T, 1> inputABuf(inputA, size);
  cl::sycl::buffer<T, 1> inputBBuf(inputB, size);
  cl::sycl::buffer<T, 1> outputBuf(output, size);







}
```

Create buffers to maintain the data across host and device

The buffers synchronise upon destruction

Example: Vector Add

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


template <typename T>
void parallel_add(T *inputA, T *inputB, T *output, size_t size) {
  cl::sycl::buffer<T, 1> inputABuf(inputA, size);
  cl::sycl::buffer<T, 1> inputBBuf(inputB, size);
  cl::sycl::buffer<T, 1> outputBuf(output, size);
  cl::sycl::queue defaultQueue;







}
```

Create a queue to en-queue work

Example: Vector Add

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


template <typename T>
void parallel_add(T *inputA, T *inputB, T *output, size_t size) {
  cl::sycl::buffer<T, 1> inputABuf(inputA, size);
  cl::sycl::buffer<T, 1> inputBBuf(inputB, size);
  cl::sycl::buffer<T, 1> outputBuf(output, size);
  cl::sycl::queue defaultQueue;
  defaultQueue.submit([&] (cl::sycl::handler &cgh) {






  });
}
```

Create a command group to define an asynchronous task

The scope of the command group is defined by a lambda

Example: Vector Add

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


template <typename T>
void parallel_add(T *inputA, T *inputB, T *output, size_t size) {
  cl::sycl::buffer<T, 1> inputABuf(inputA, size);
  cl::sycl::buffer<T, 1> inputBBuf(inputB, size);
  cl::sycl::buffer<T, 1> outputBuf(output, 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);



  });
}
```

Create accessors to give access to the data on the device

Example: Vector Add

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

template <typename T>
void parallel_add(T *inputA, T *inputB, T *output, size_t size) {
  cl::sycl::buffer<T, 1> inputABuf(inputA, size);
  cl::sycl::buffer<T, 1> inputBBuf(inputB, size);
  cl::sycl::buffer<T, 1> outputBuf(output, 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>(size)),
                                 [=](cl::sycl::id<1> idx) {

    }));
  });
}
```

Create a parallel_for
to define a kernel

Example: Vector Add

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

template <typename T>
void parallel_add(T *inputA, T *inputB, T *output, size_t size) {
  cl::sycl::buffer<T, 1> inputABuf(inputA, size);
  cl::sycl::buffer<T, 1> inputBBuf(inputB, size);
  cl::sycl::buffer<T, 1> outputBuf(output, 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>(size)),
                          [=](cl::sycl::id<1> idx) {
      outputPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
    }));
  });
}
```

You must provide a name for the lambda

Access the data via the accessor's subscript operator

Example: Vector Add

```
template <typename T>
void parallel_add(T *inputA, T *inputB, T *output, size_t size);

int main() {

  float inputA[count] = { /* input a */ };
  float inputB[count] = { /* input b */ };
  float output[count] = { /* output */ };

  parallel_add(inputA, inputB, output, count);
}
```

The result is stored in output upon returning from parallel_add

ComputeCpp™

Community Edition

Coming soon!

Watch this space:

http://sycl.tech/

We're Hiring!
codeplay.com/careers/

codeplay®
THE HETEROGENEOUS SYSTEMS EXPERTS

Thank You

@codeplaysoft

info@codeplay.com

codeplay.com