



Towards Heterogeneous Programming in C++

Gordon Brown – Staff Software Engineer, SYCL

Michael Wong – VP of R&D, Chair of SG5/SG14

Andrew Richards – CEO Codeplay

CppCon2016 – September 2016

Acknowledgement Disclaimer

Numerous people internal and external to the original C++/Khronos group, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedbacks to form part of this talk.

I even lifted this acknowledgement and disclaimer from some of them.

But I claim all credit for errors, and stupid mistakes.

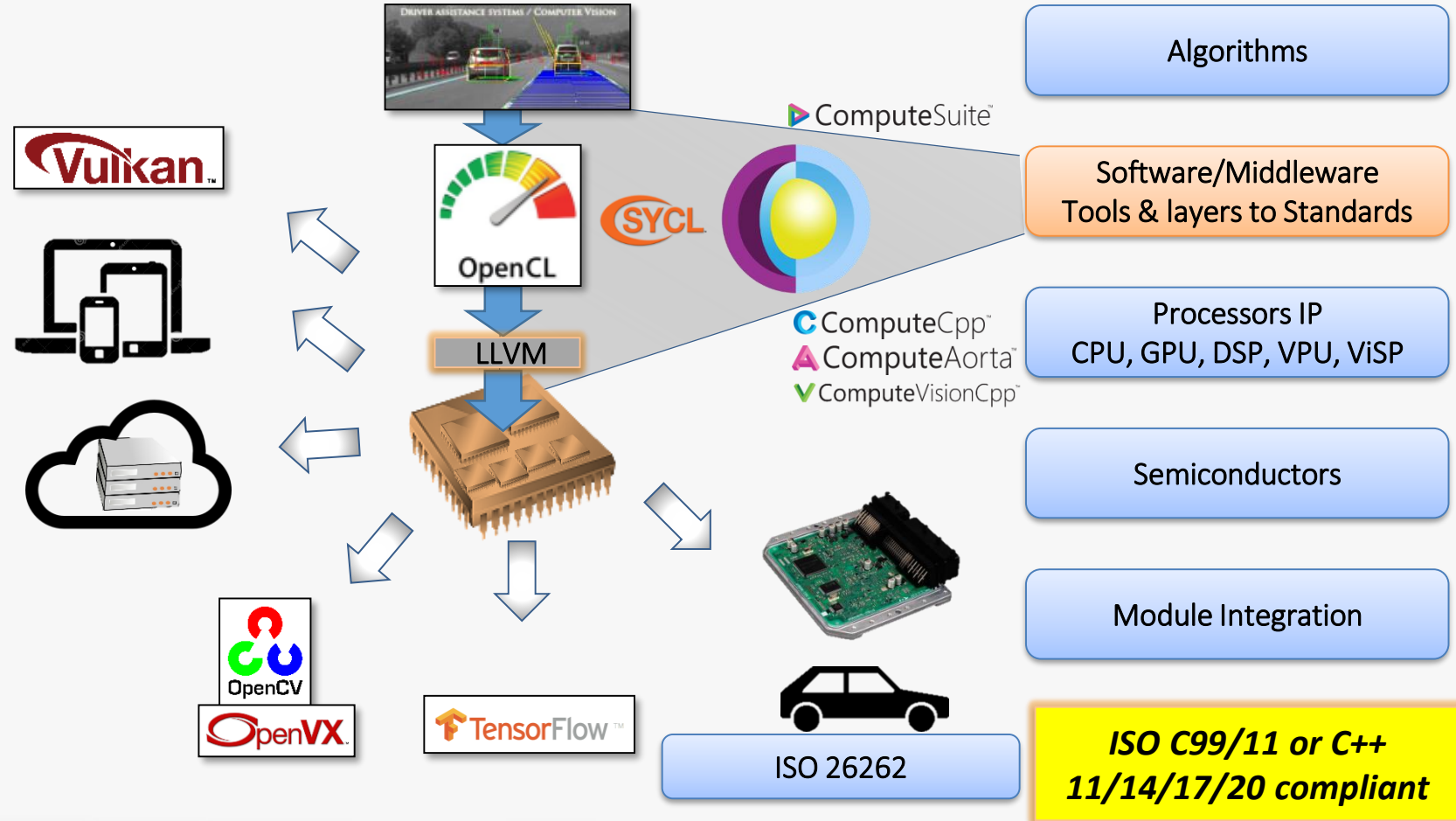
These are mine, all mine!

Legal Disclaimer

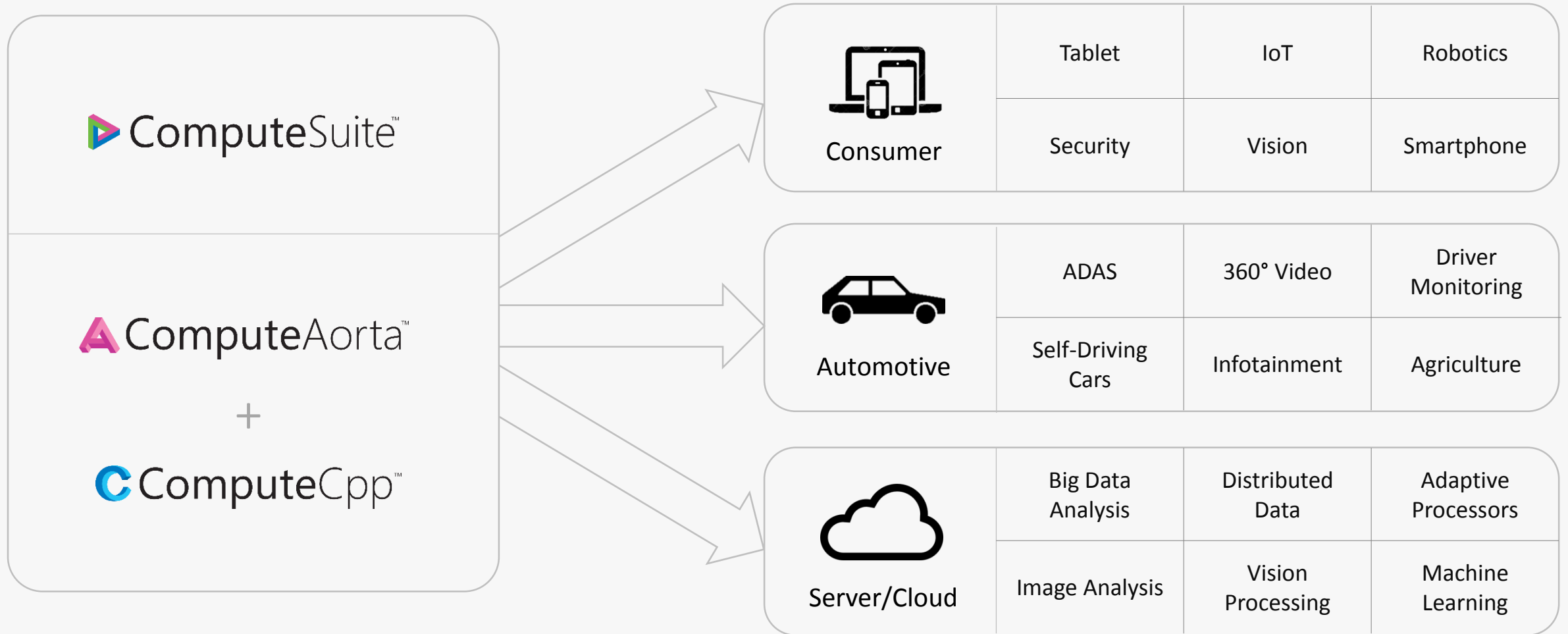
This work represents the view of the author and does not necessarily represent the view of Codeplay.

Other company, product, and service names may be trademarks or service marks of others.

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



Codeplay Products



Why am I here talking to you today?

- As of the C++ standards meeting in Jacksonville, Feb 2016, we now have a mandate to bring heterogeneous computing to C++

What is heterogeneous computing?

- Heterogeneous computing involves gaining performance through the utilisation of systems which make use of more than one kind of processor each with specialized processing capabilities to handle particular tasks

So why should you be interested?

- Heterogeneous computing is everywhere, it's driving the technology of the future

So why should you be interested?



- Heterogeneous computing is everywhere, it's driving the technology of the future

What would I like you to take away from today?

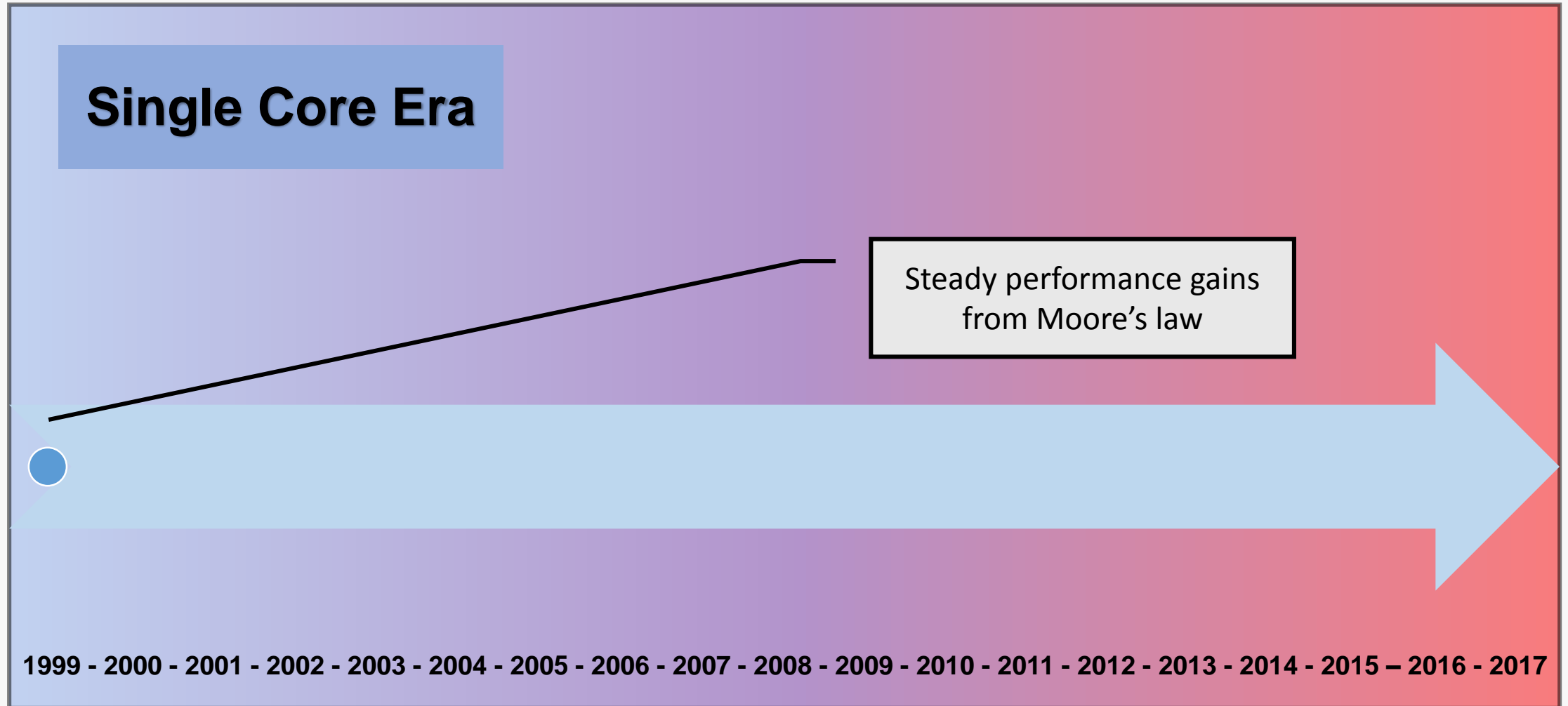
- A vision of the future of heterogeneous programming in C++

Outline

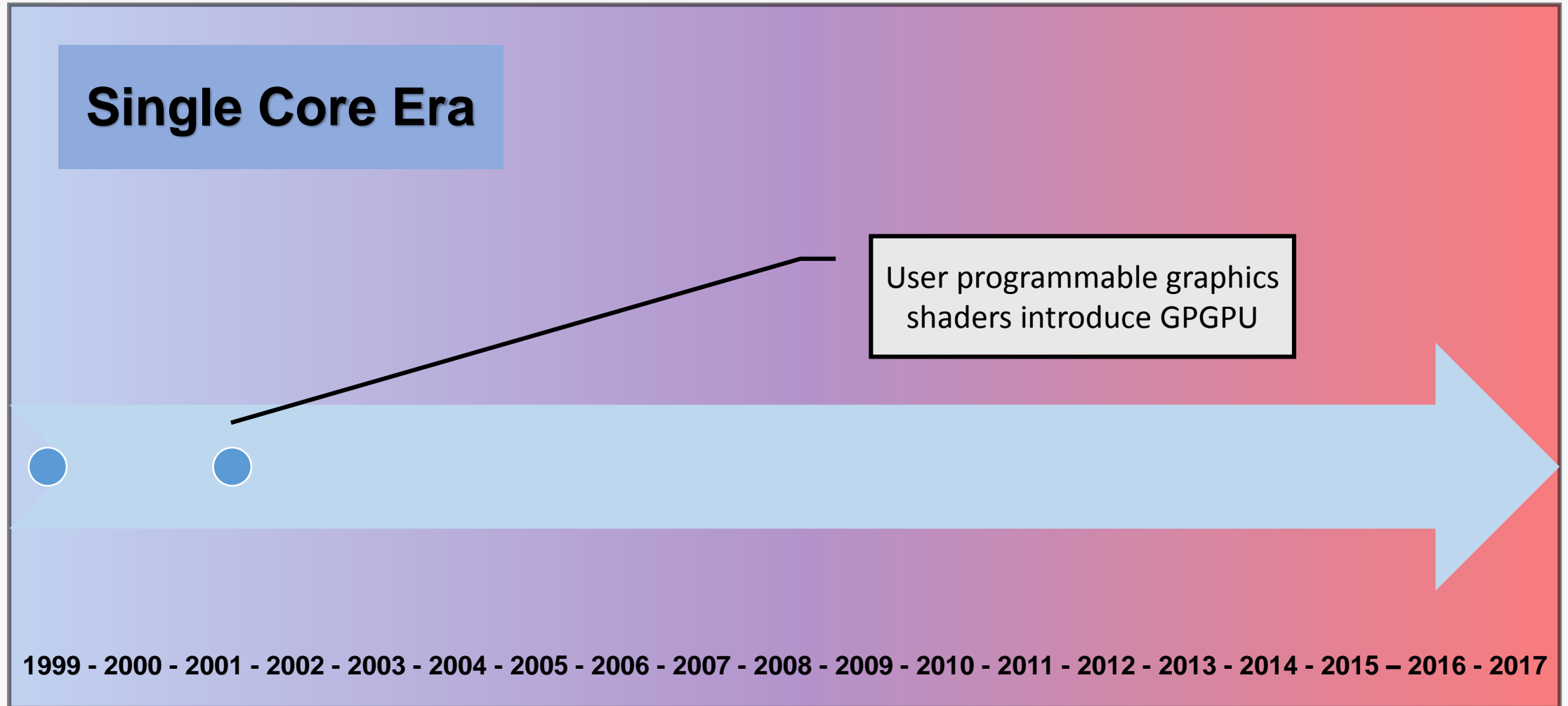
- How did we get here?
- The rise of C++ in heterogeneous computing
- Understanding the challenges of the heterogeneous era
- SYCL: A new approach to heterogeneous programming in C++
- The future of heterogeneous programming in C++

How did we get here?

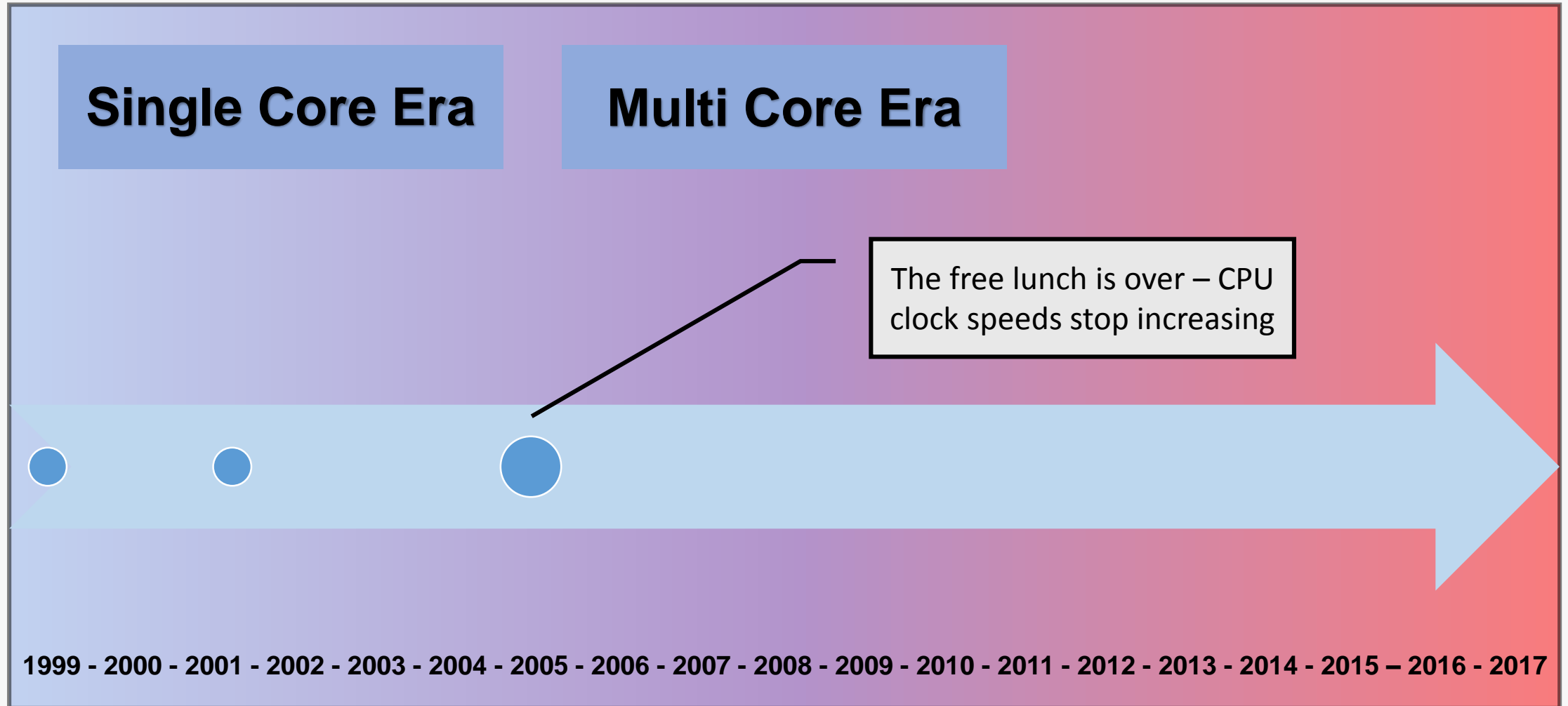
History of Heterogeneous Computing



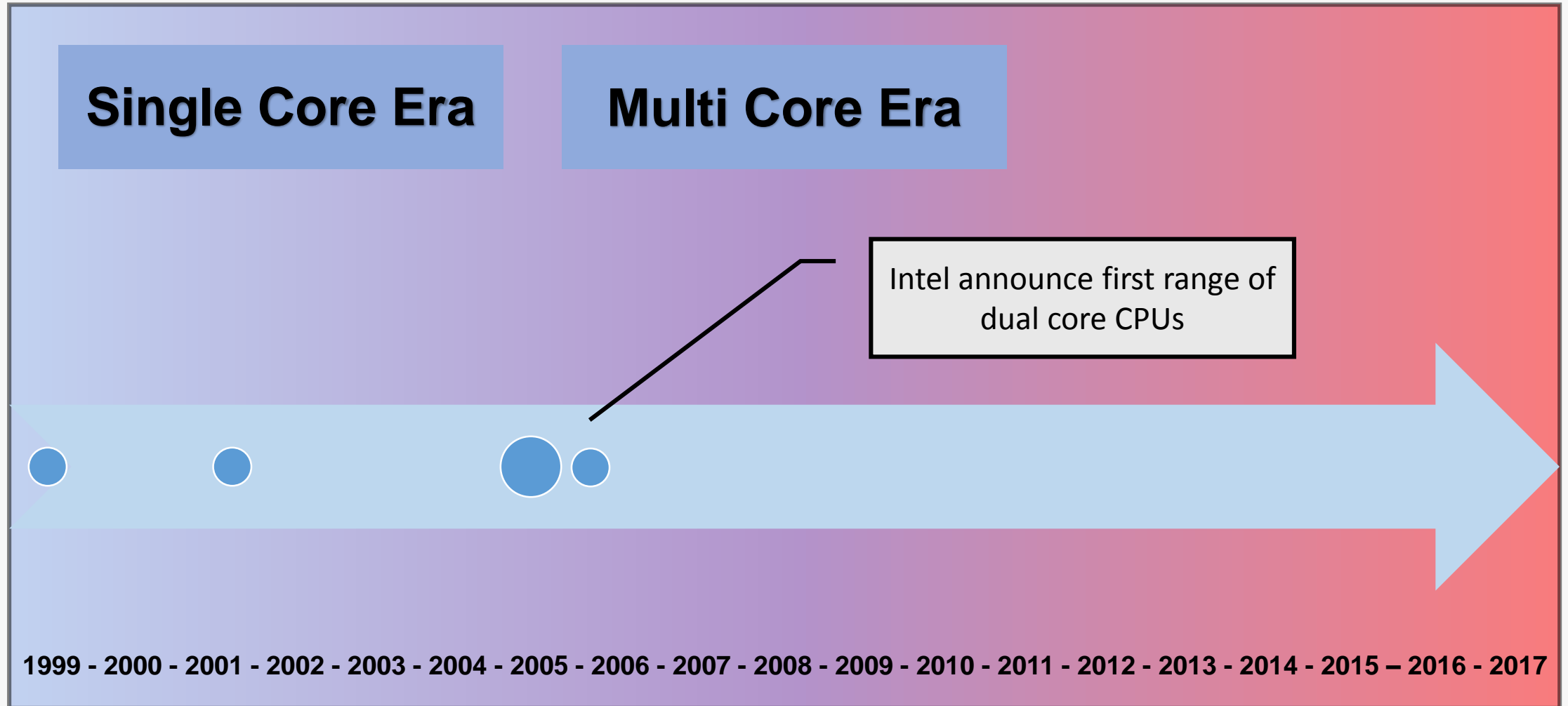
History of Heterogeneous Computing



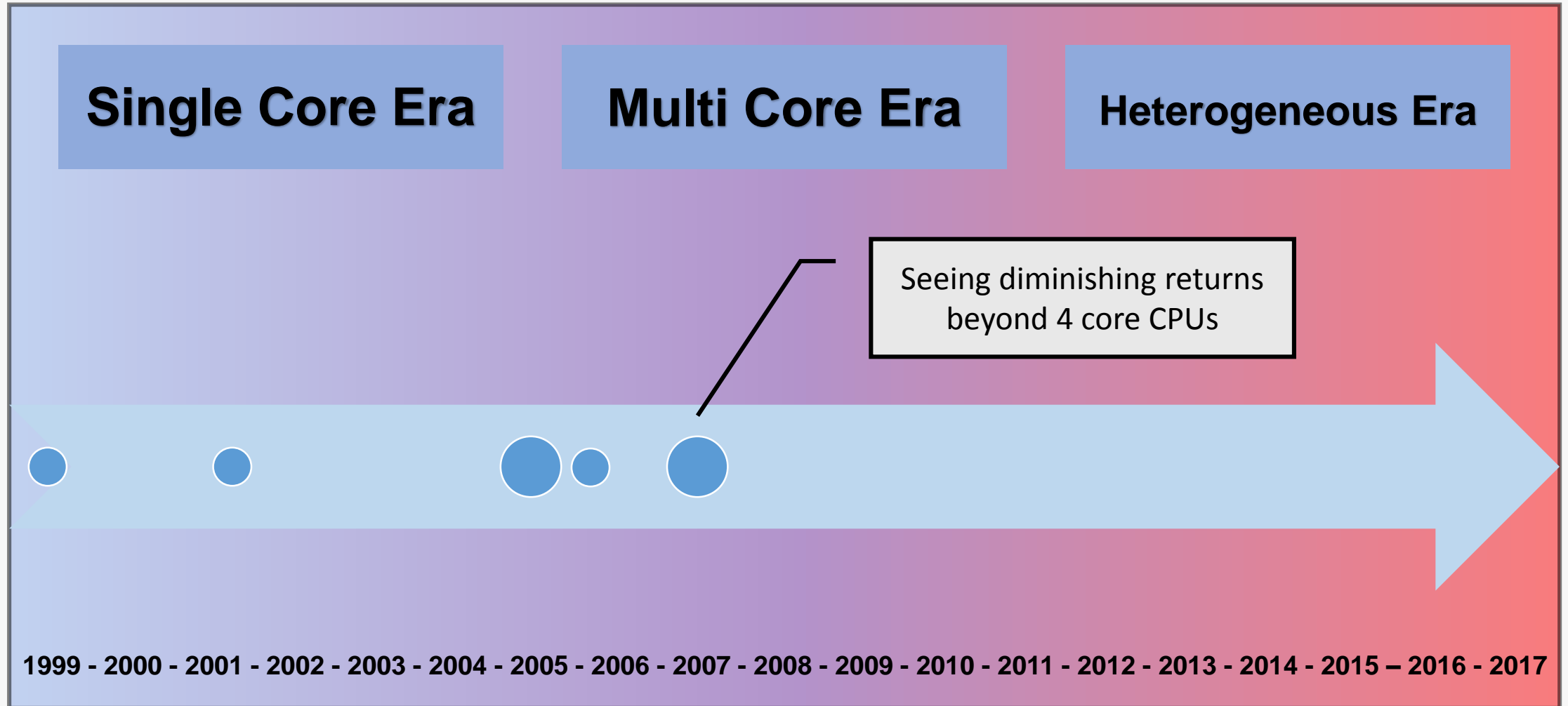
History of Heterogeneous Computing



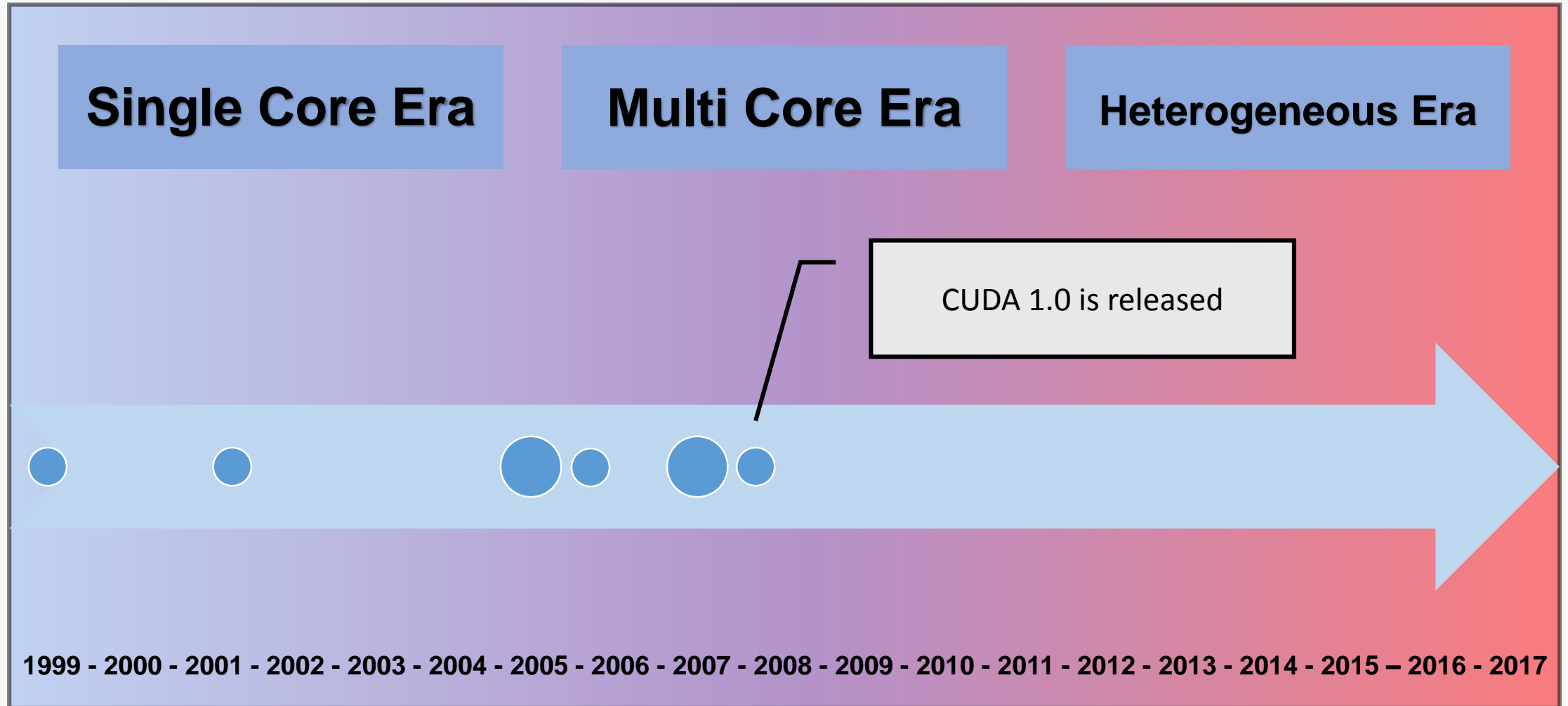
History of Heterogeneous Computing



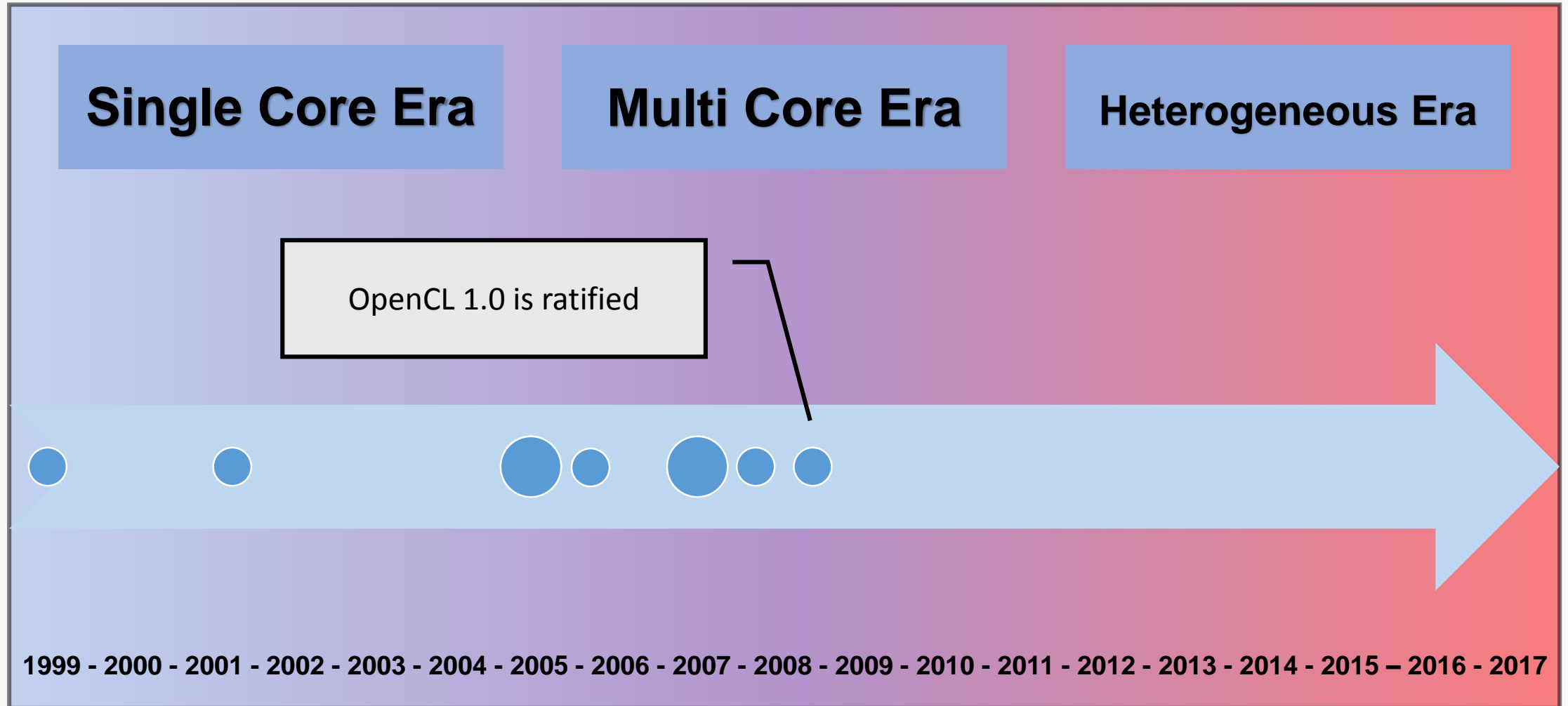
History of Heterogeneous Computing



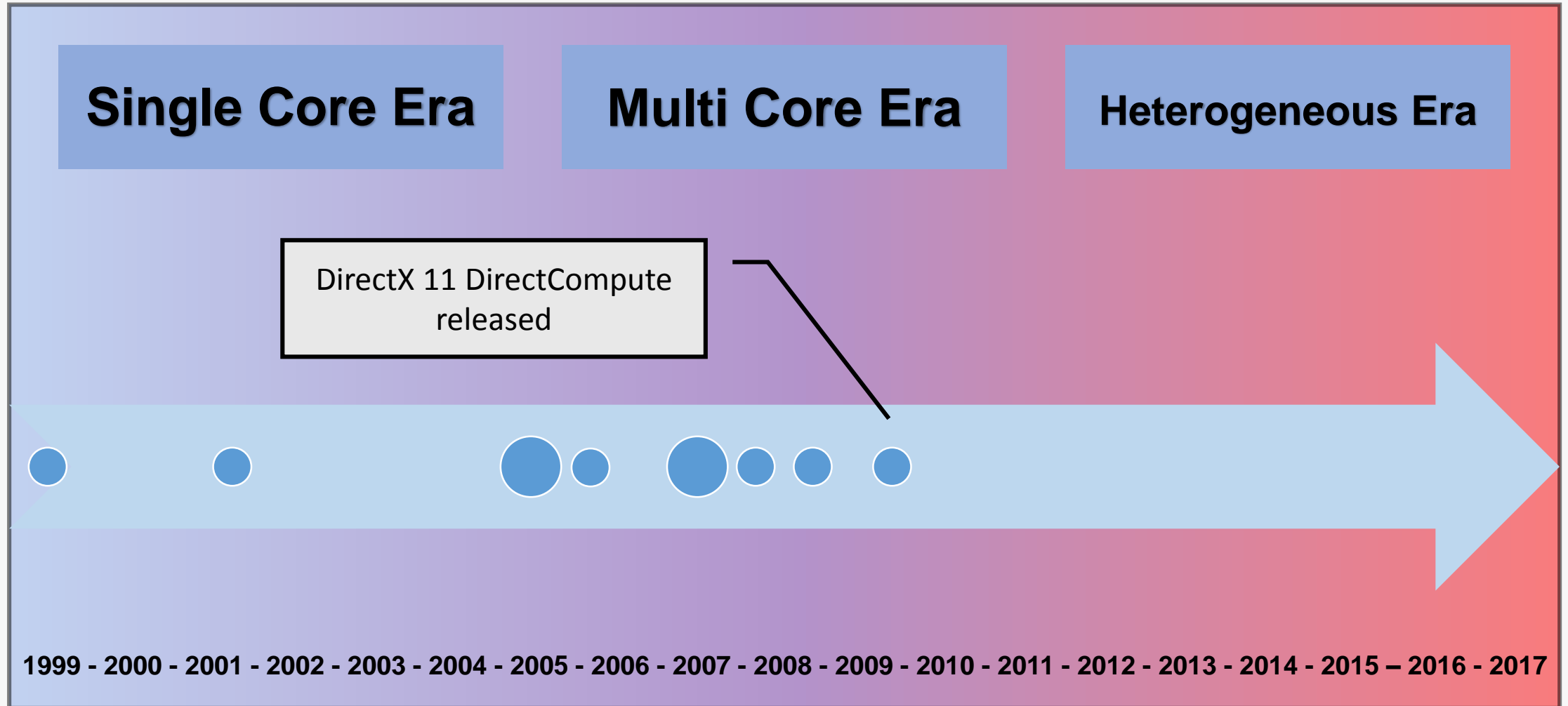
History of Heterogeneous Computing



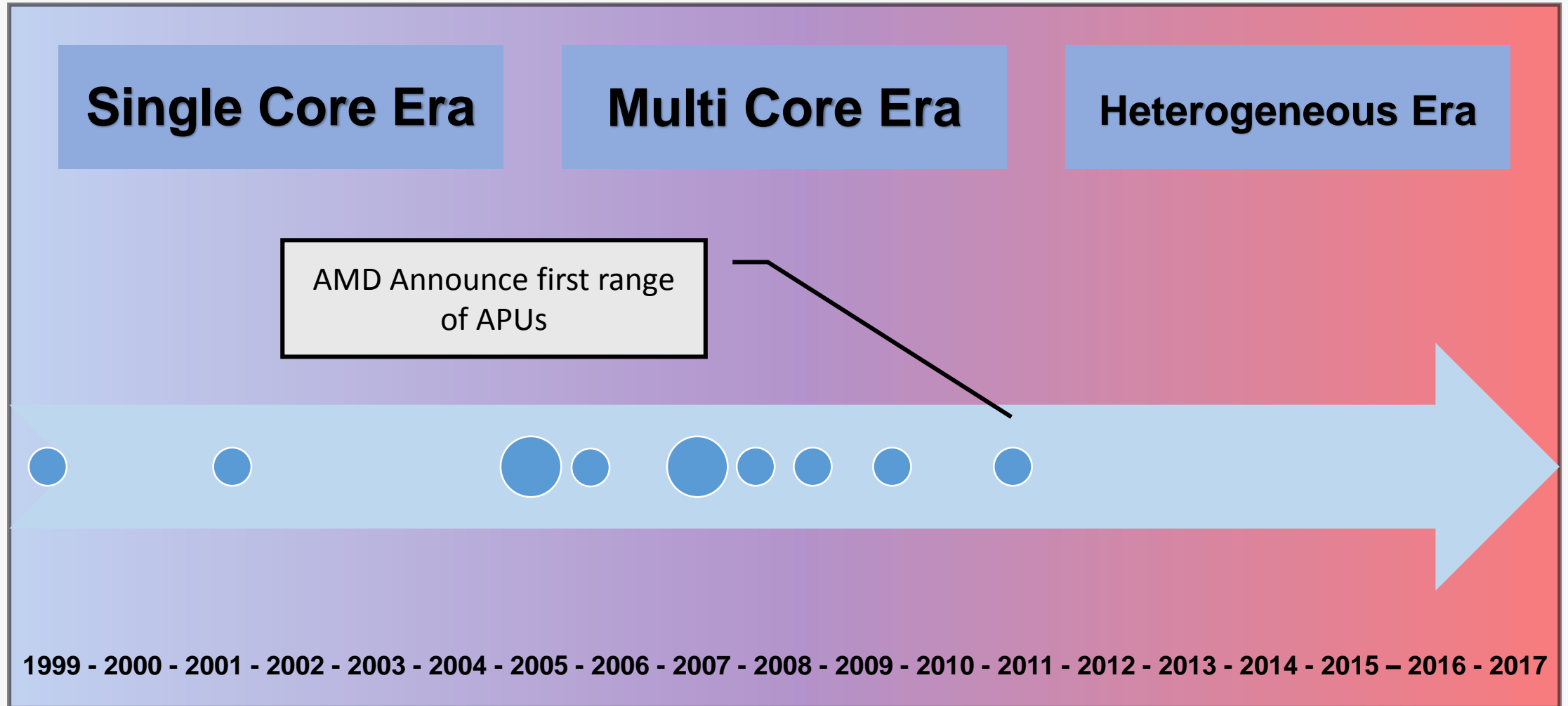
History of Heterogeneous Computing



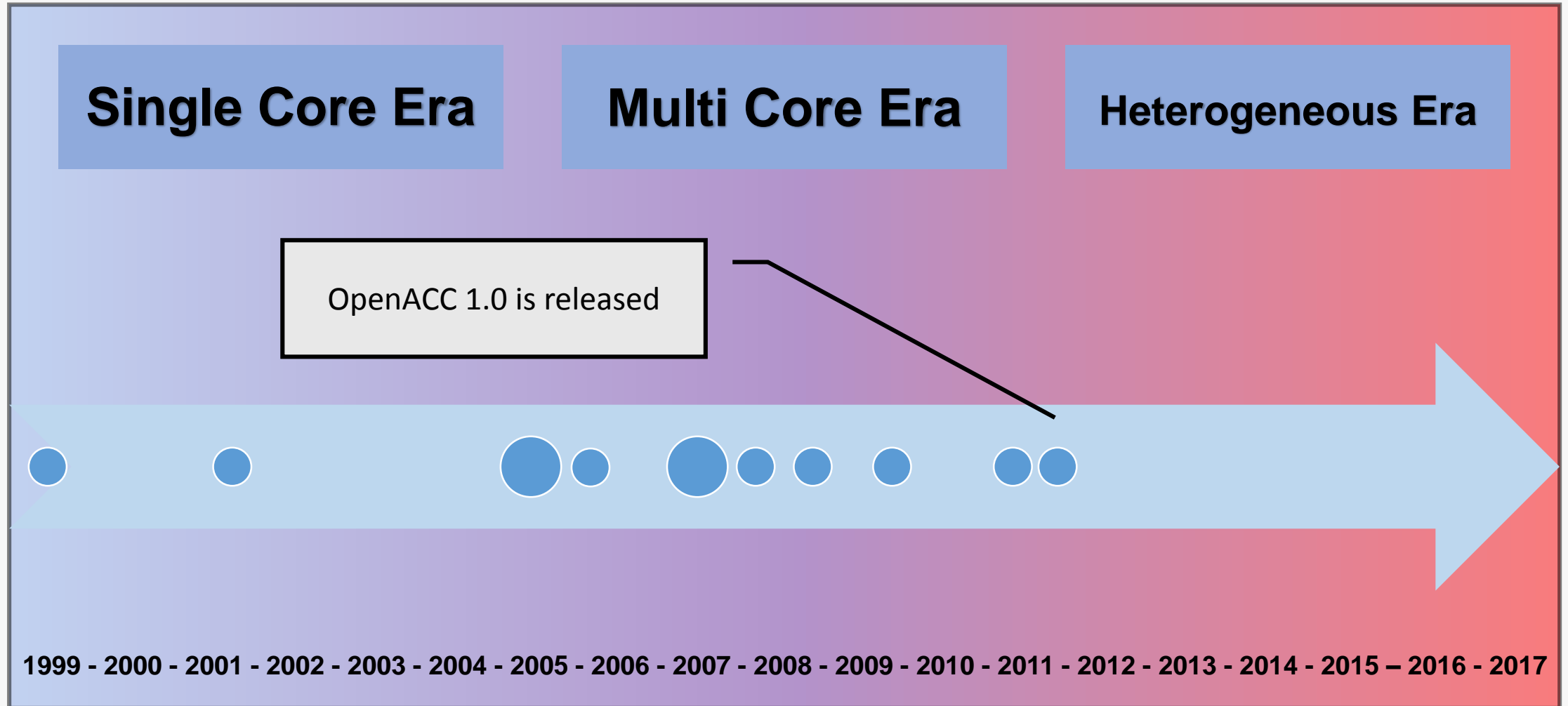
History of Heterogeneous Computing



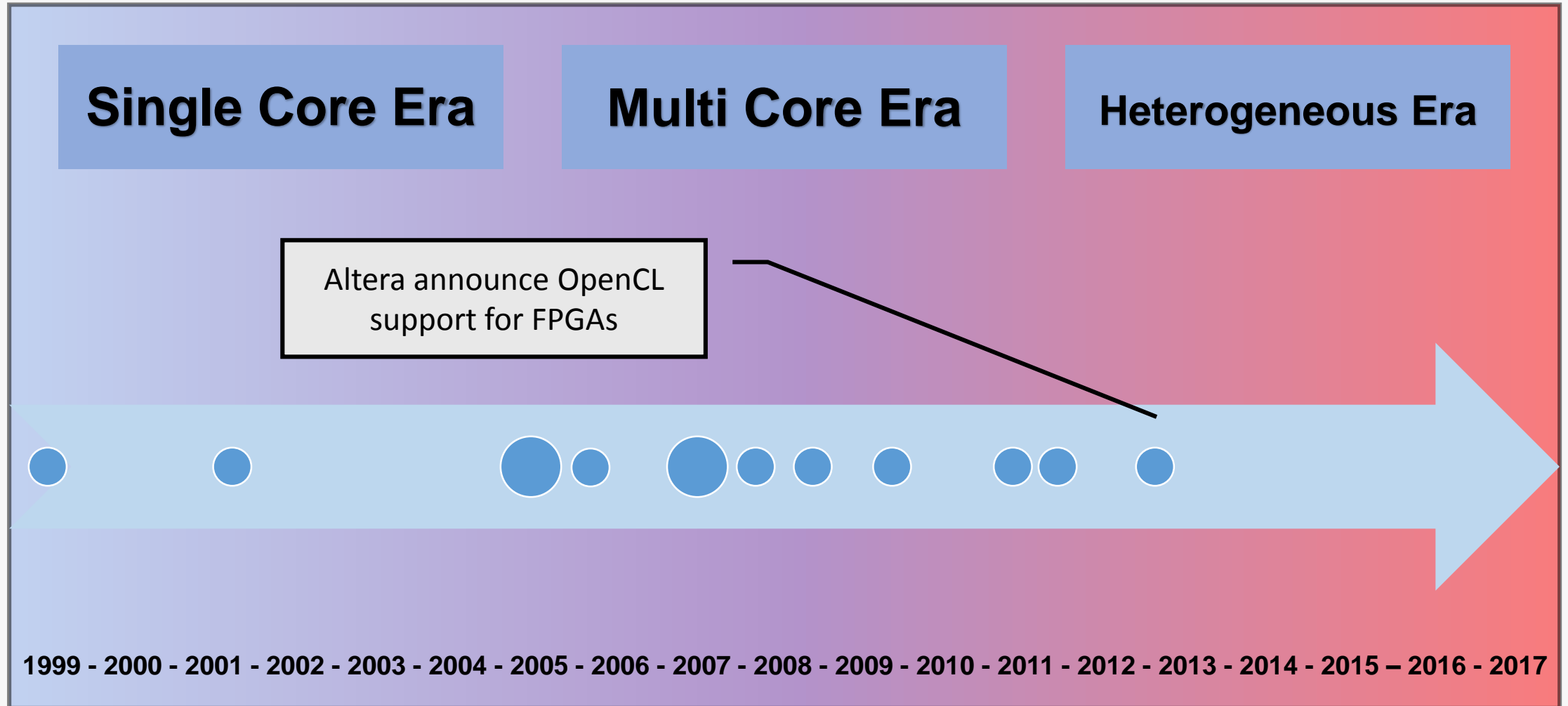
History of Heterogeneous Computing



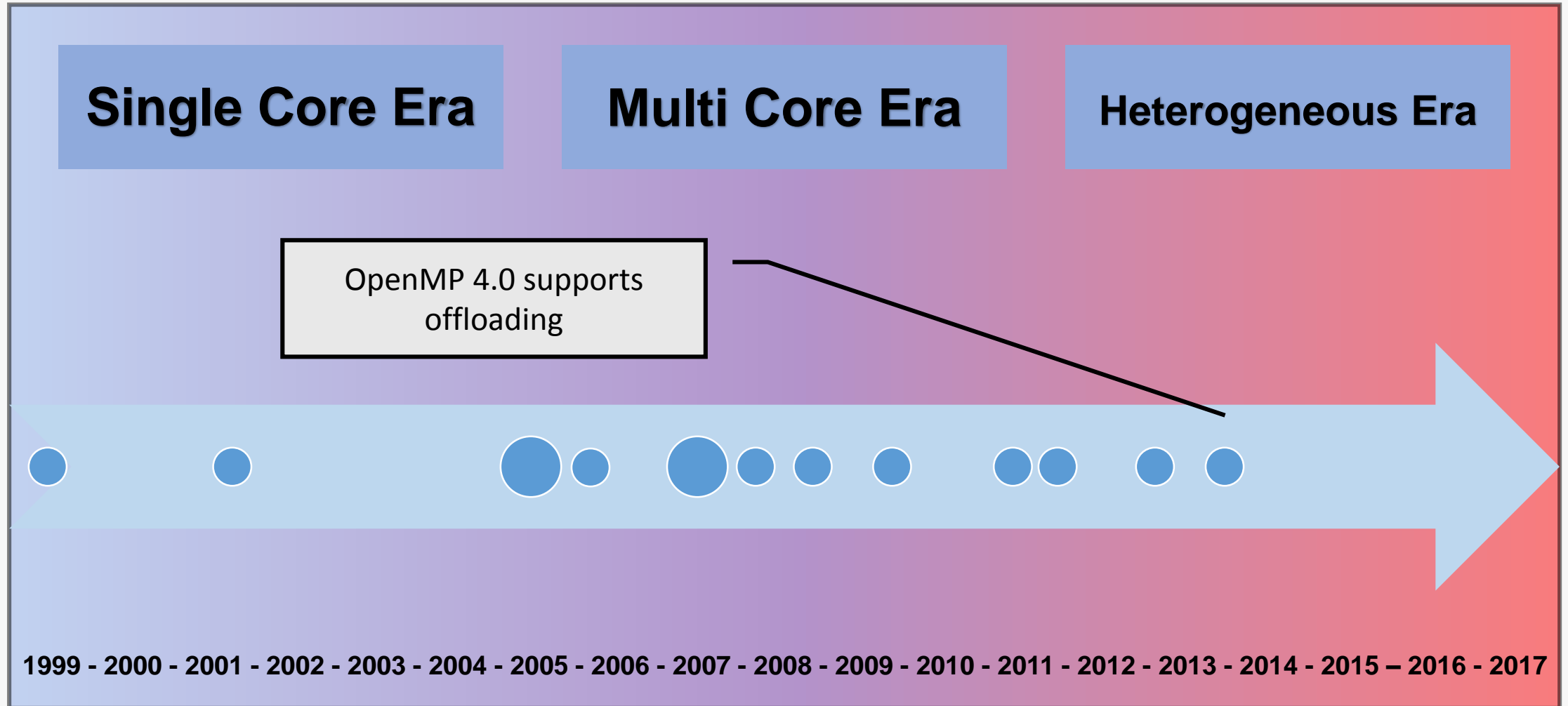
History of Heterogeneous Computing



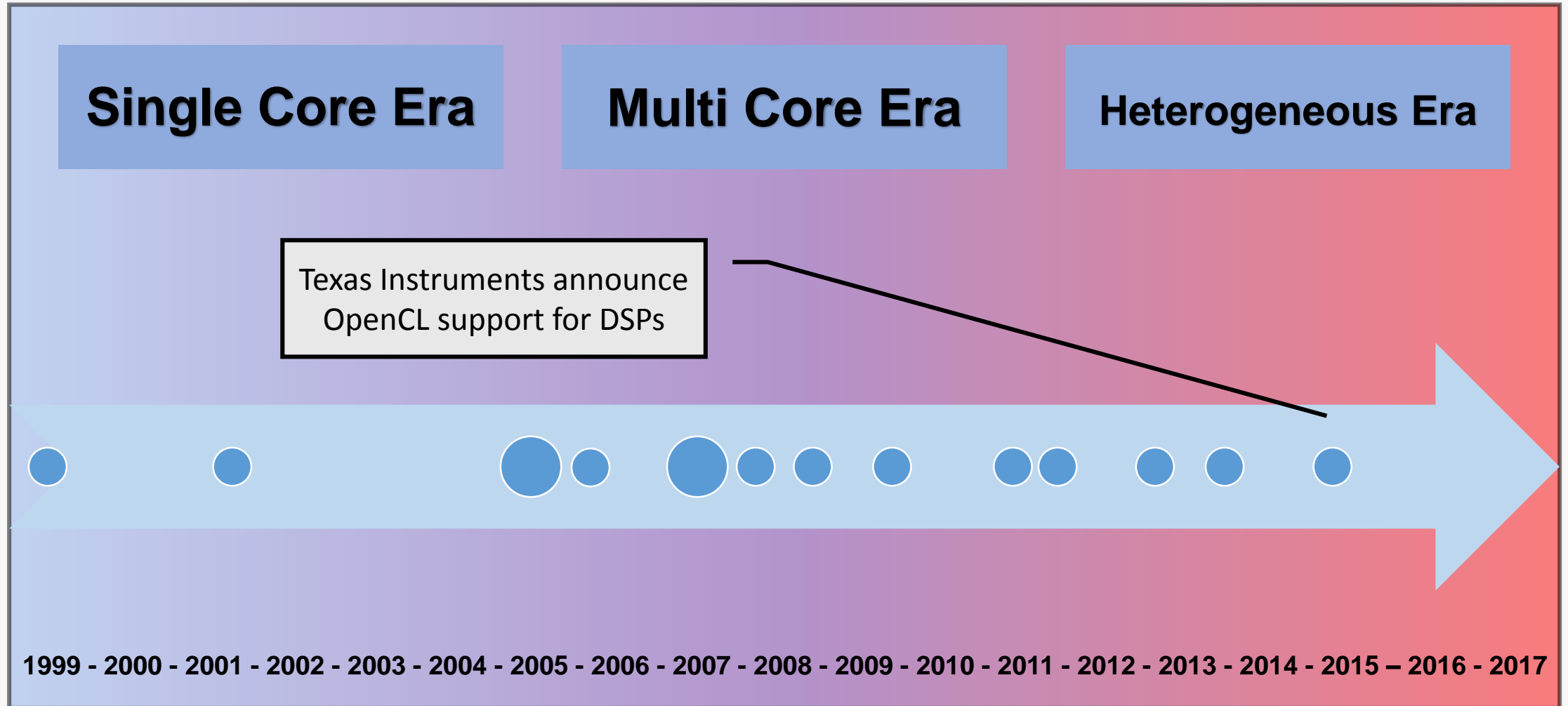
History of Heterogeneous Computing



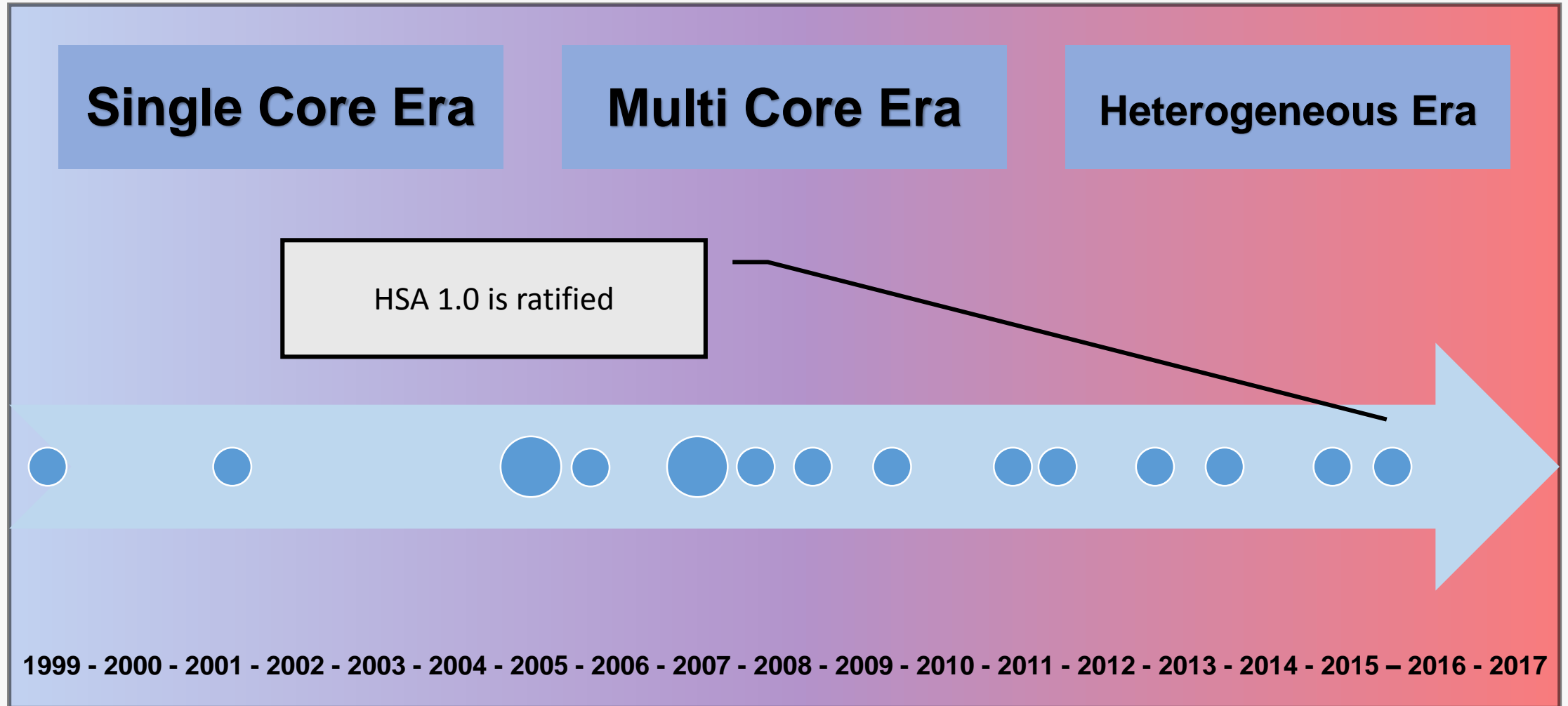
History of Heterogeneous Computing



History of Heterogeneous Computing



History of Heterogeneous Computing

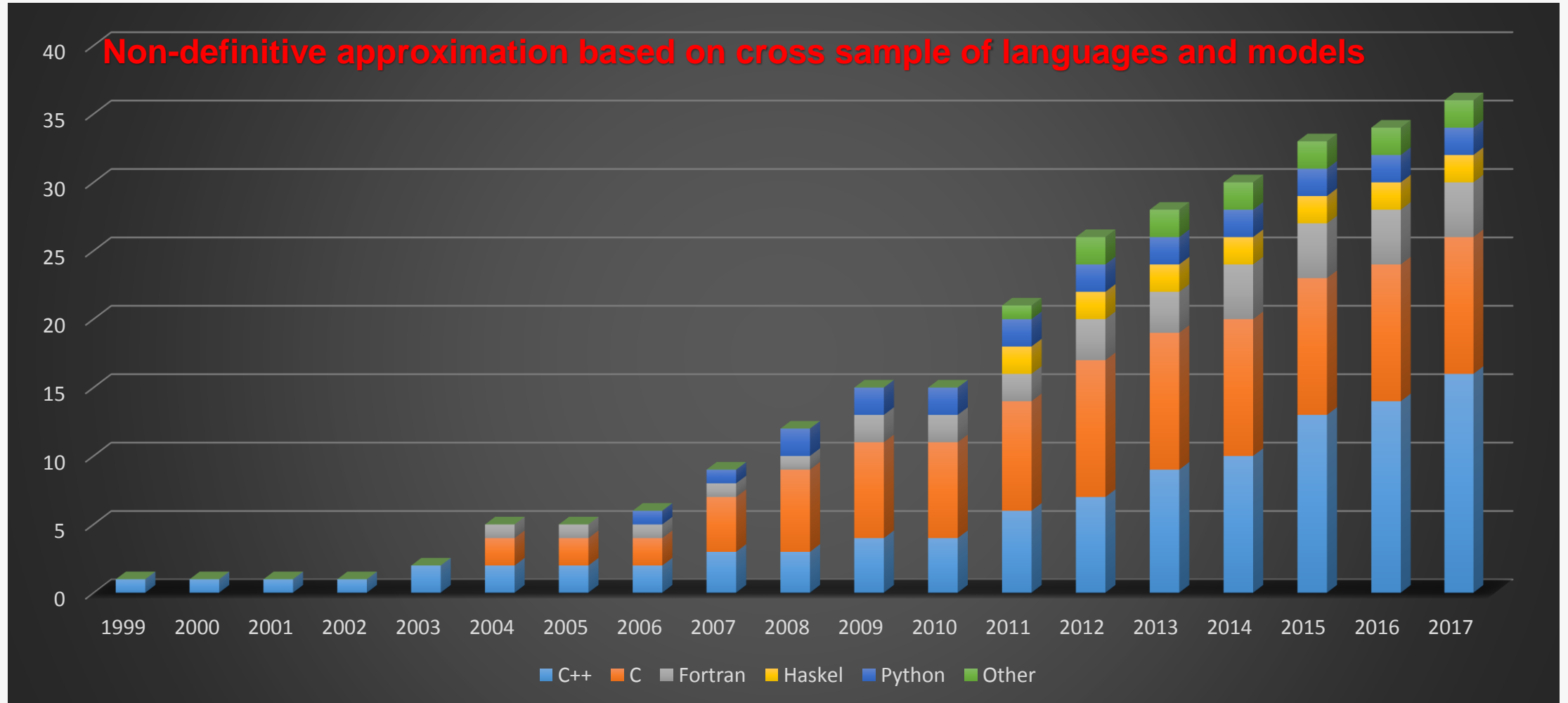


The Rise of C++ in Heterogeneous Programming

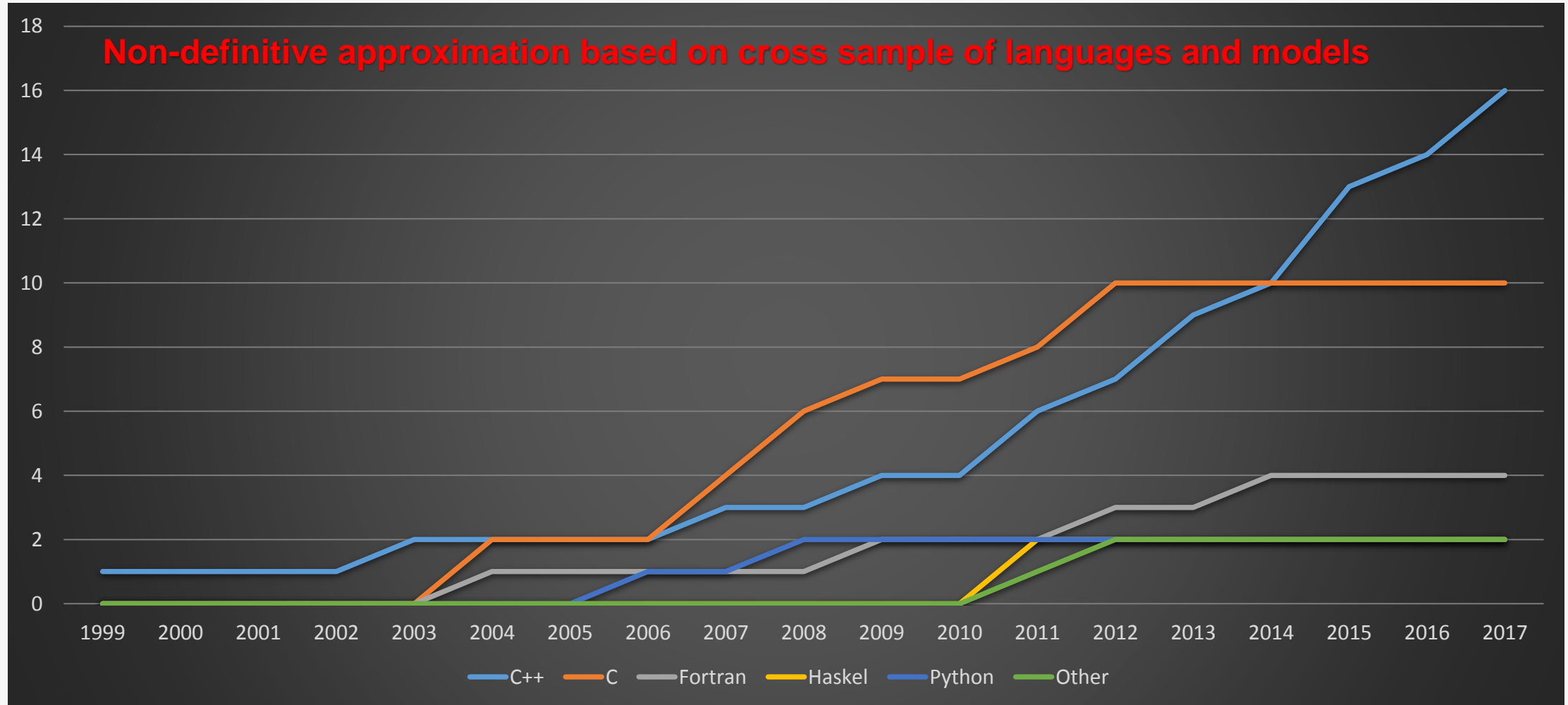
What is the most popular language for heterogeneous computing?

- I have attempted to answer this question

Heterogeneous Programming Languages/Models



Heterogeneous Programming Languages/Models



Understanding the Challenges of the Heterogeneous Era

So what are the biggest challenges for heterogeneous computing?

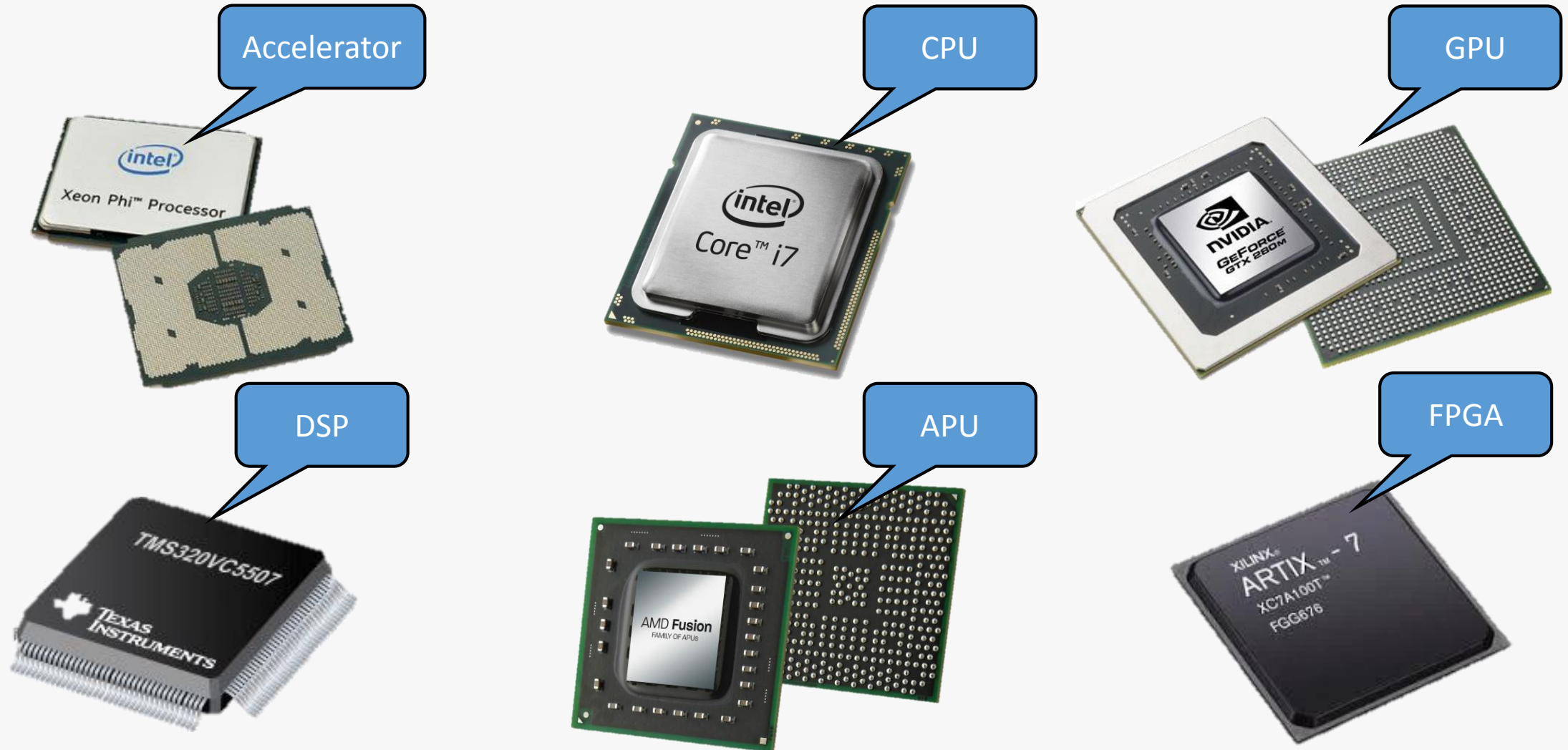
- Performance portability
- Heterogeneous offloading
- Expressing parallelism
- Data locality & movement

Performance Portability

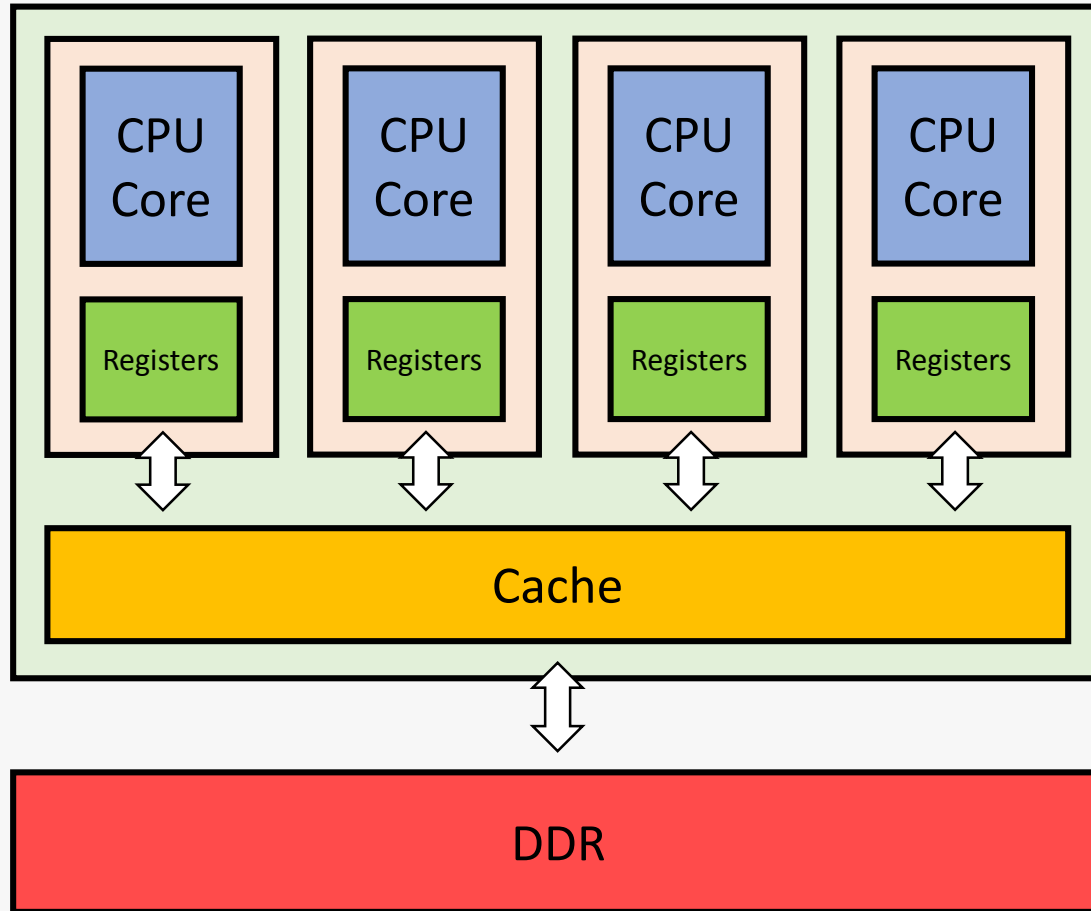
Is performance portability across heterogeneous devices really possible?

- This depends on the way you look at it

Heterogeneous Devices



Typical CPU Architecture



Simplified model of a typical system

Execution:

- Small number of large cores
- Separate instructions on each core independently

Memory:

- Lower bandwidth memory
- Random access

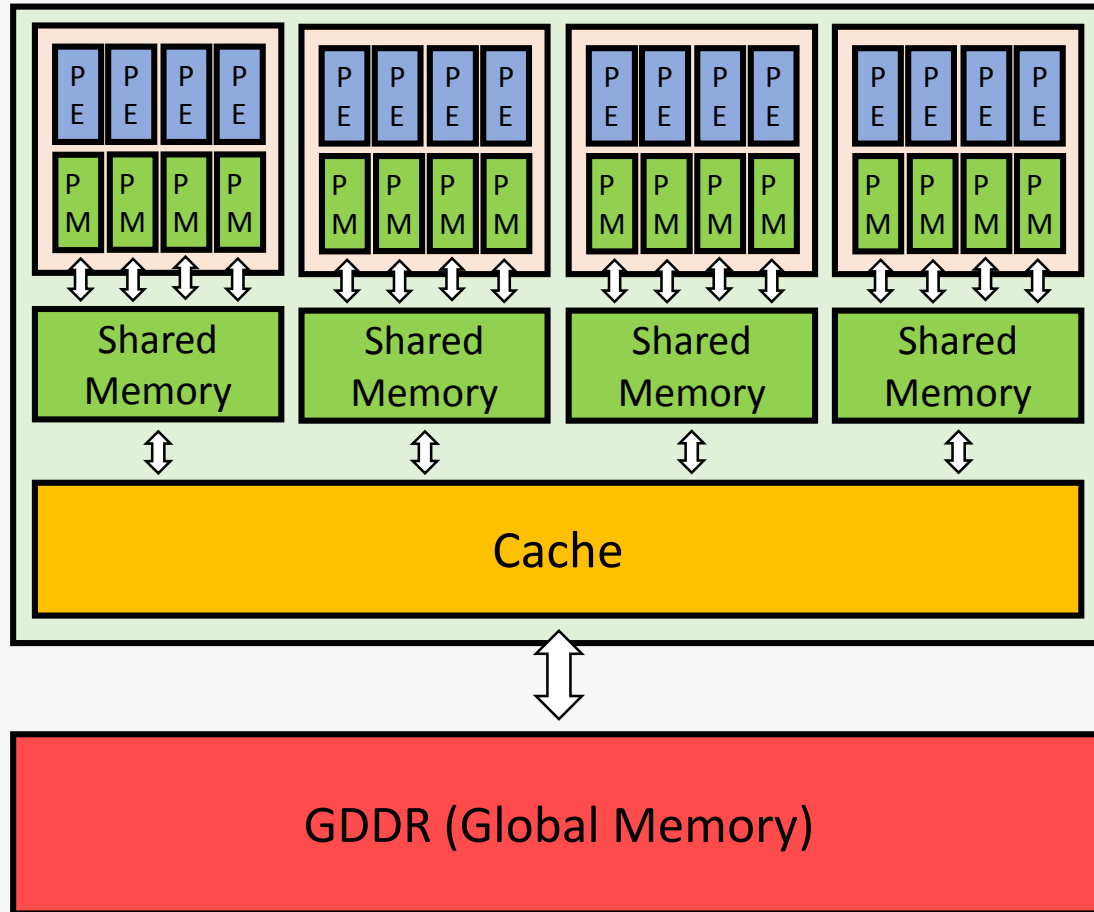
Power Consumption:

- High power consumption

Suggested for:

- Task parallelism

Typical GPU Architecture



Simplified model of a typical system

Execution:

- Large number of small execution units
- Single instruction on multiple execution units (can be in lock-step)

Memory:

- Hierarchical memory structure
- Higher memory bandwidth
- Predictable memory access

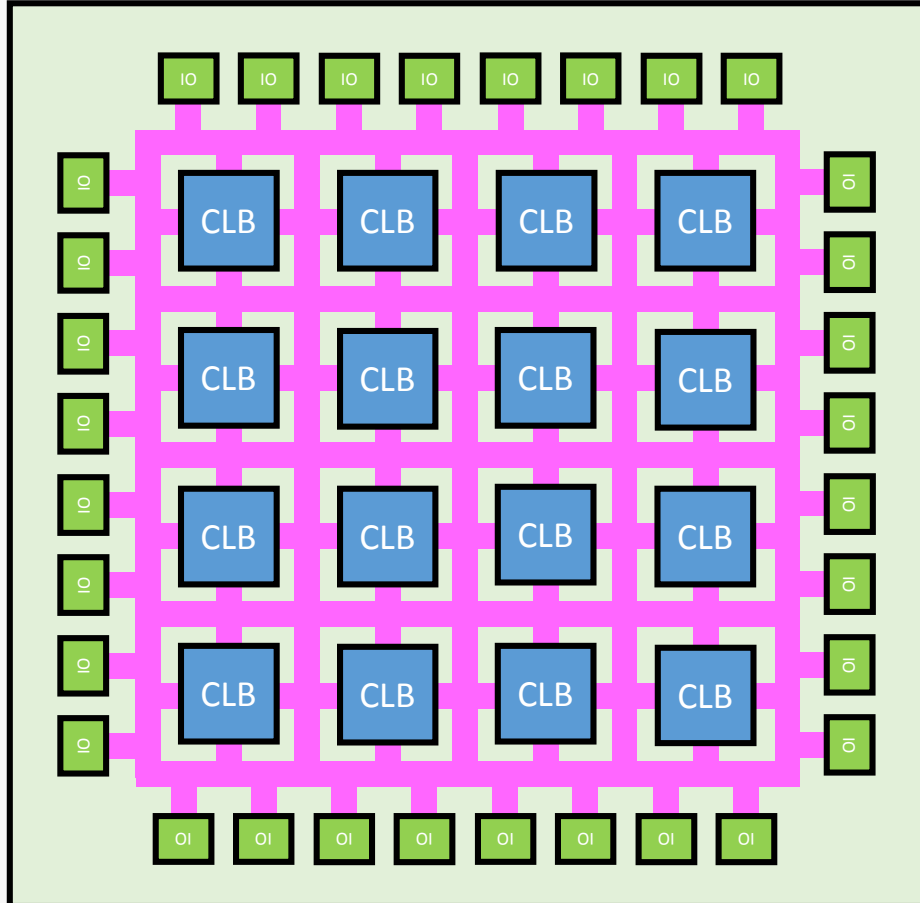
Power Consumption:

- Low power consumption

Suggested for:

- Data parallelism

Typical FPGA Architecture



Simplified model of a typical system

Execution:

- Configurable logic blocks re-programmable via synthetisation
- Each block executes separate instruction on different data from a stream in parallel

Memory:

- Configurable logic blocks with configurable routing for varied bandwidths

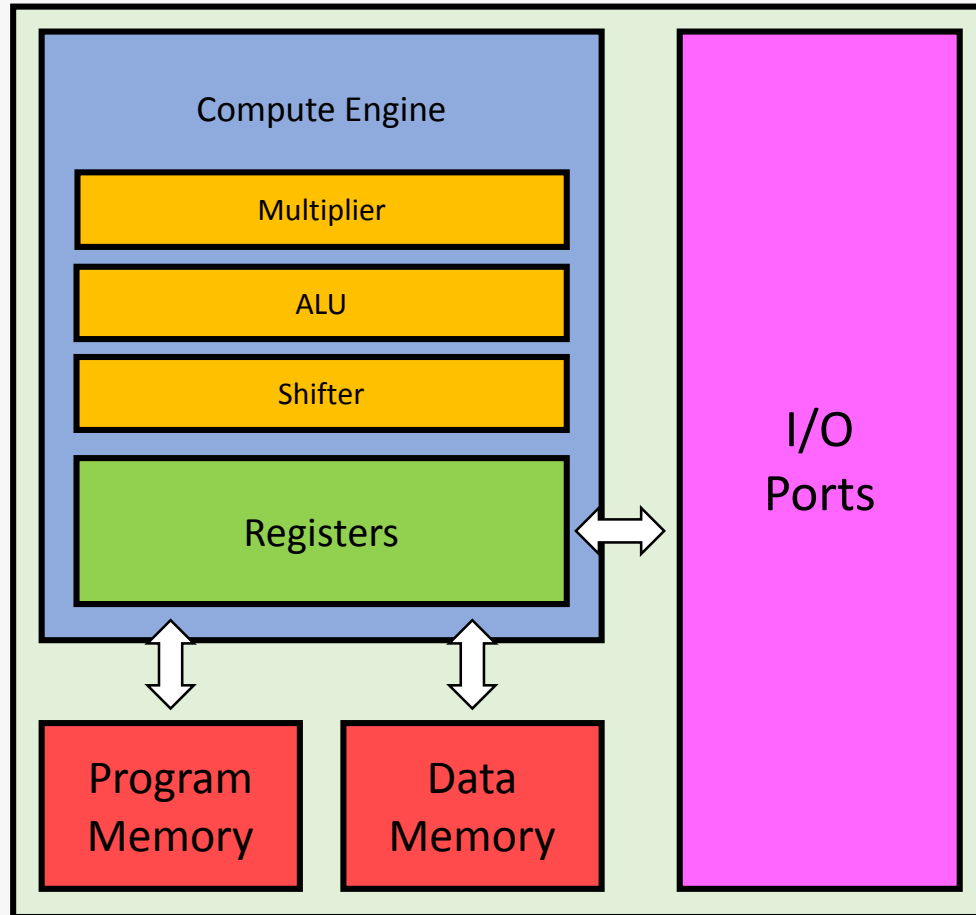
Power Consumption

- Low power consumption

Suggested for:

- Data parallel streaming, image processing

Typical DSP Architecture



Simplified model of a typical system

Execution:

- Purpose built compute engine for performing add, subtract, divide and multiply very quickly
- Compute engine executes single instruction on data from a stream

Memory:

- Low latency on chip memory

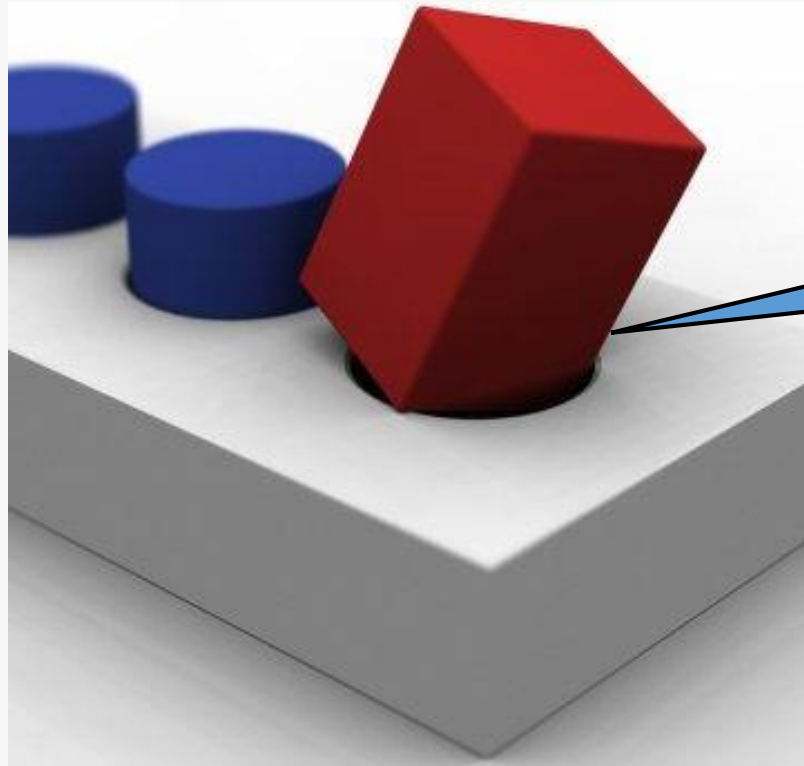
Power Consumption

- Low power consumption

Suggested for:

- Digital/analogue audio data stream processing

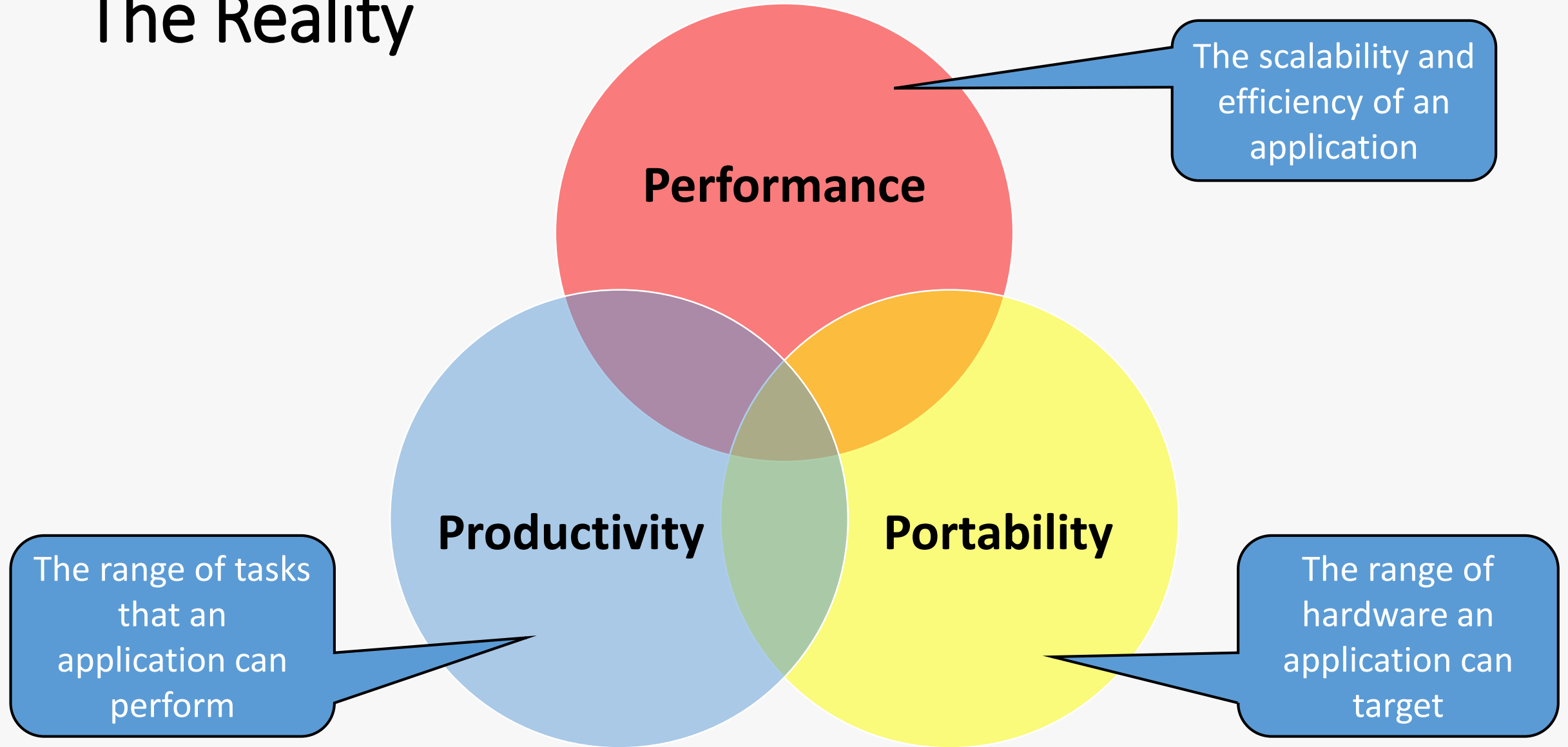
Is Performance Portability Possible?



Short
answer
is no

- Architectures by design are so different that there is no one solution fits all

The Reality



Is Performance Portability Possible?



Long
answer
is yes

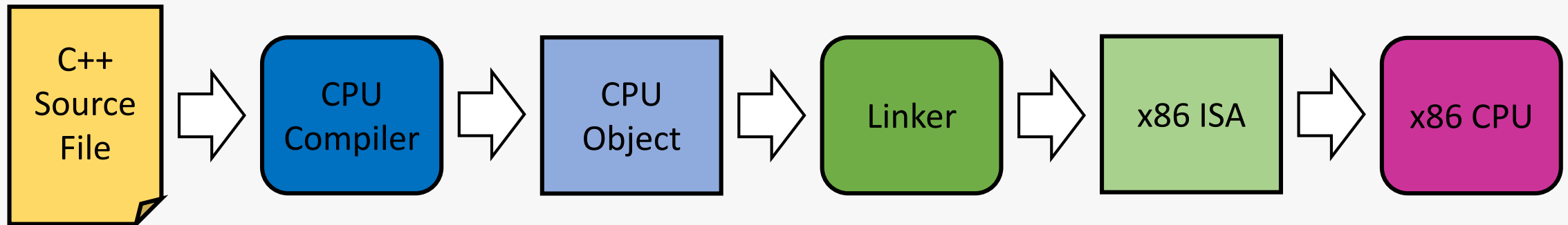
- The right programming model can allow you to express a problem in a way which adapts to different architectures

Heterogeneous Offloading

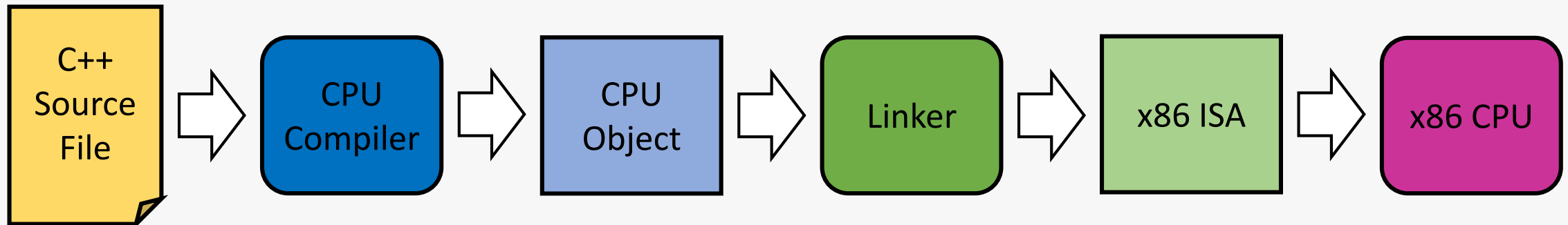
How do we offload code to a heterogeneous device?

- This can be answered by looking at the C++ compilation model

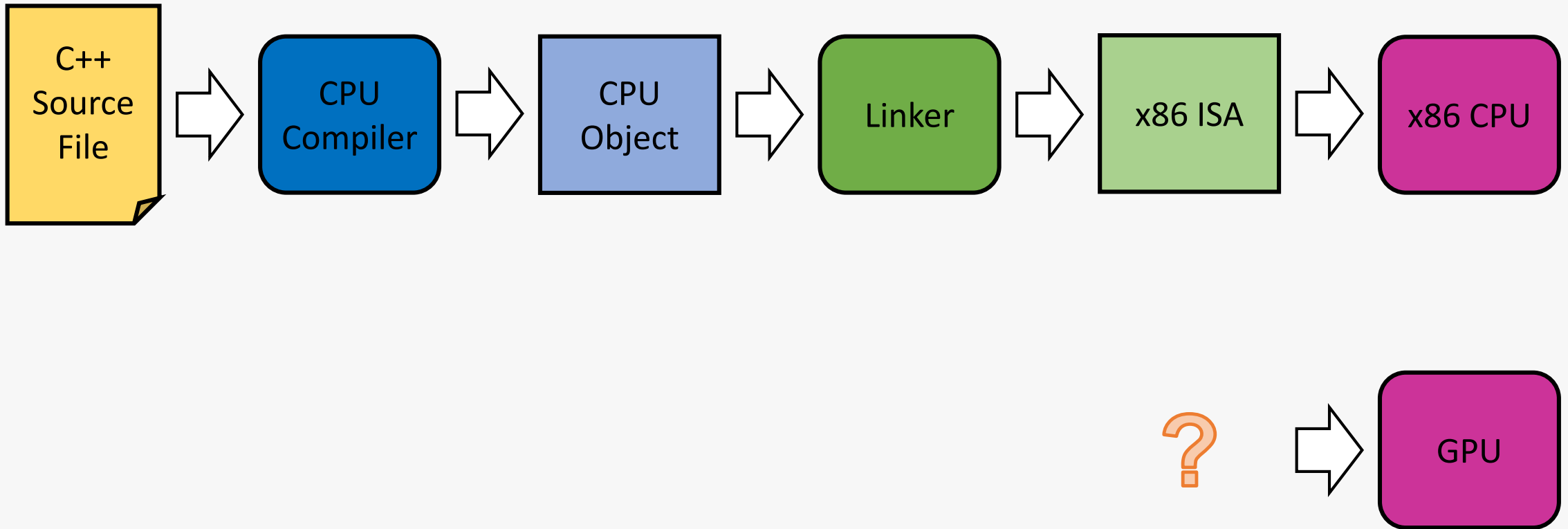
C++ Compilation Model



C++ Compilation Model



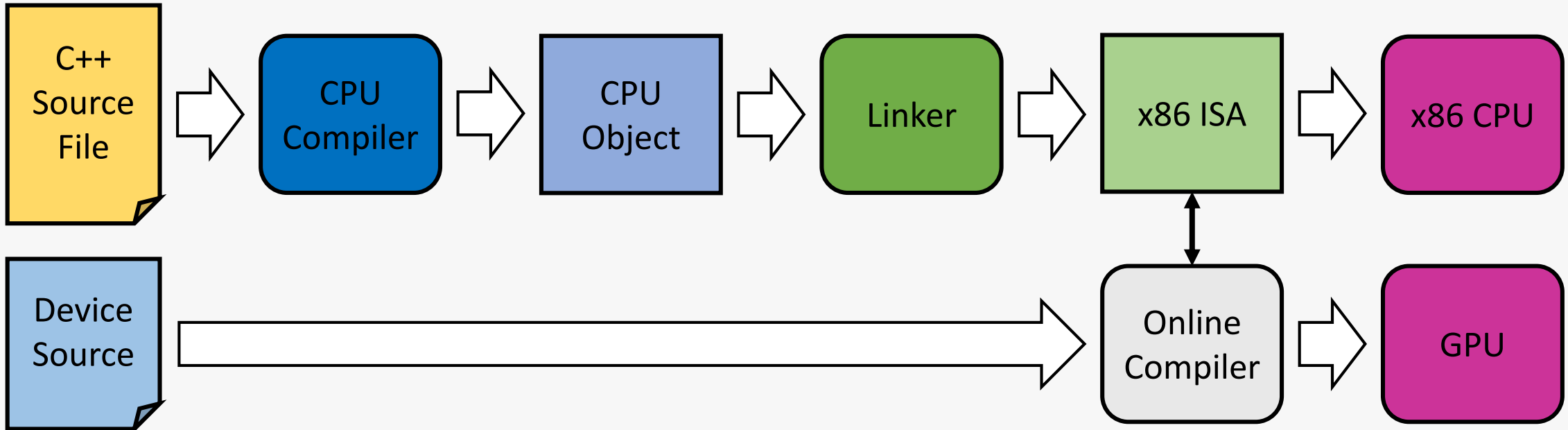
C++ Compilation Model



How can we compile source code for a sub architectures?

- Separate source
- Single source

Separate Source Compilation Model

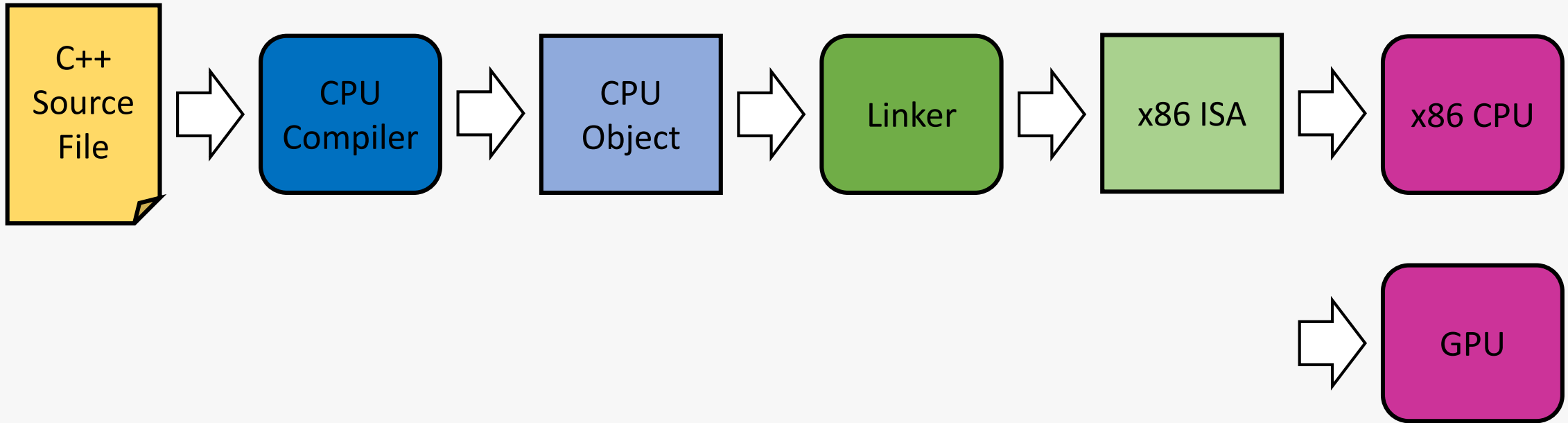


```
float *a, *b, *c;
...
kernel k = clCreateKernel(..., "my_kernel", ...)
clEnqueueWriteBuffer(..., size, a, ...);
clEnqueueWriteBuffer(..., size, a, ...);
clEnqueueNDRange(..., k, 1, {size, 1, 1}, ...);
clEnqueueWriteBuffer(..., size, c, ...);
```

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];
}
```

Single Source Compilation Model

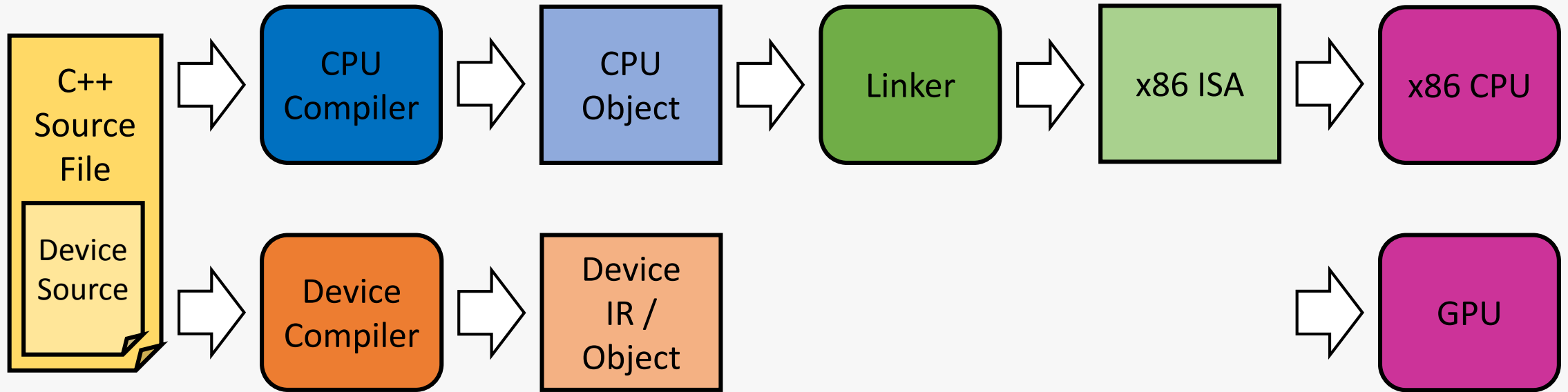


```
array_view<float> a, b, c;  
extent<2> e(64, 64);
```

```
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

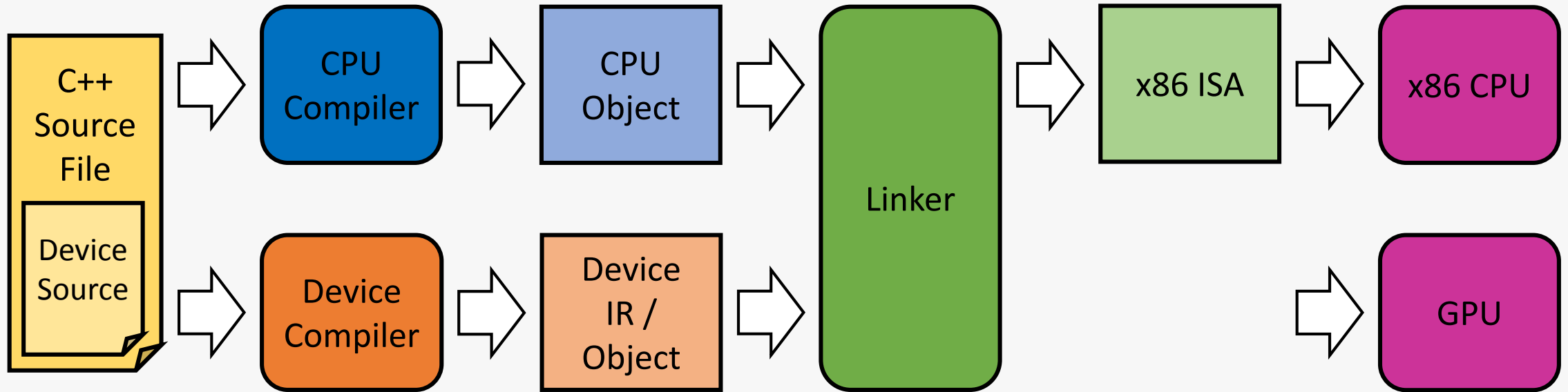


```
array_view<float> a, b, c;  
extent<2> e(64, 64);
```

```
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

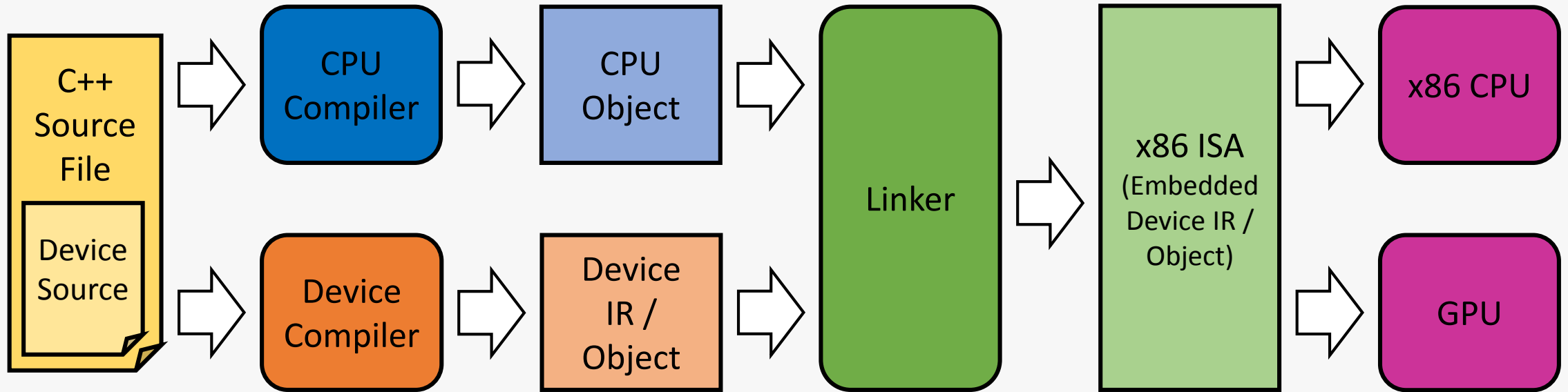


```
array_view<float> a, b, c;  
extent<2> e(64, 64);
```

```
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 C++ 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 enqueue 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

Examples:

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

Implementation:

- The same task is performed across a large data set

Here we're using TBB as an example

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

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 enqueueer

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

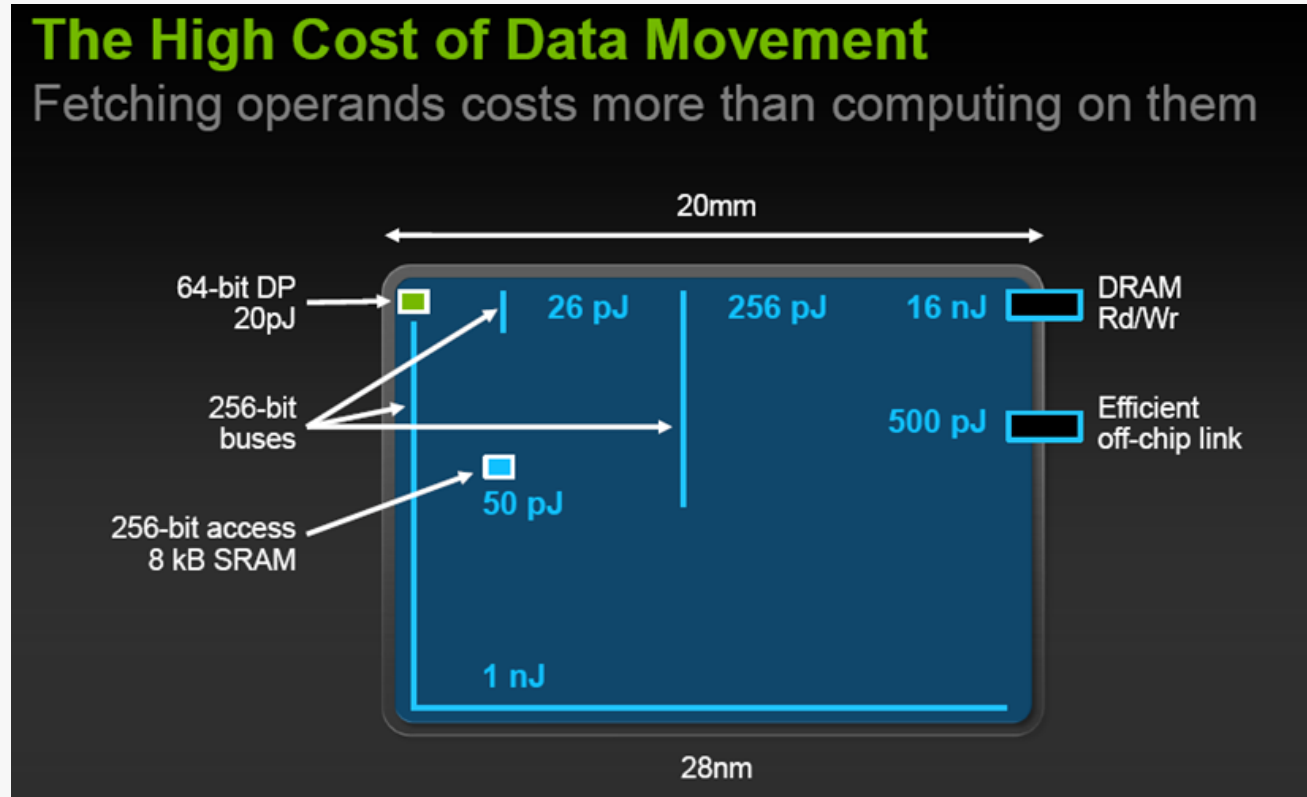
One of the biggest limiting factor in heterogeneous computing

- 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);
```

How do you address memory between host CPU and device?

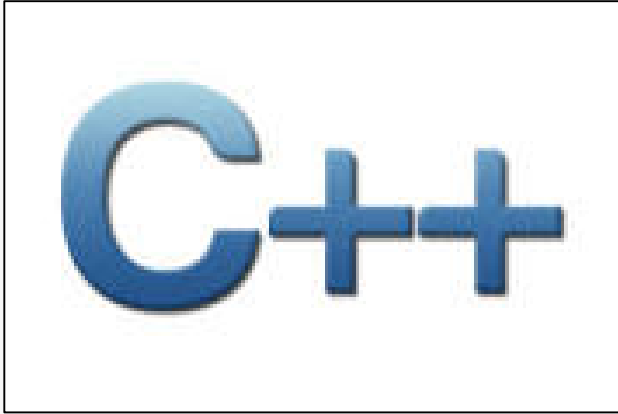
- Multiple address space
- Non-coherent single address space
- Cache coherent single address space

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
 - 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
 - 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

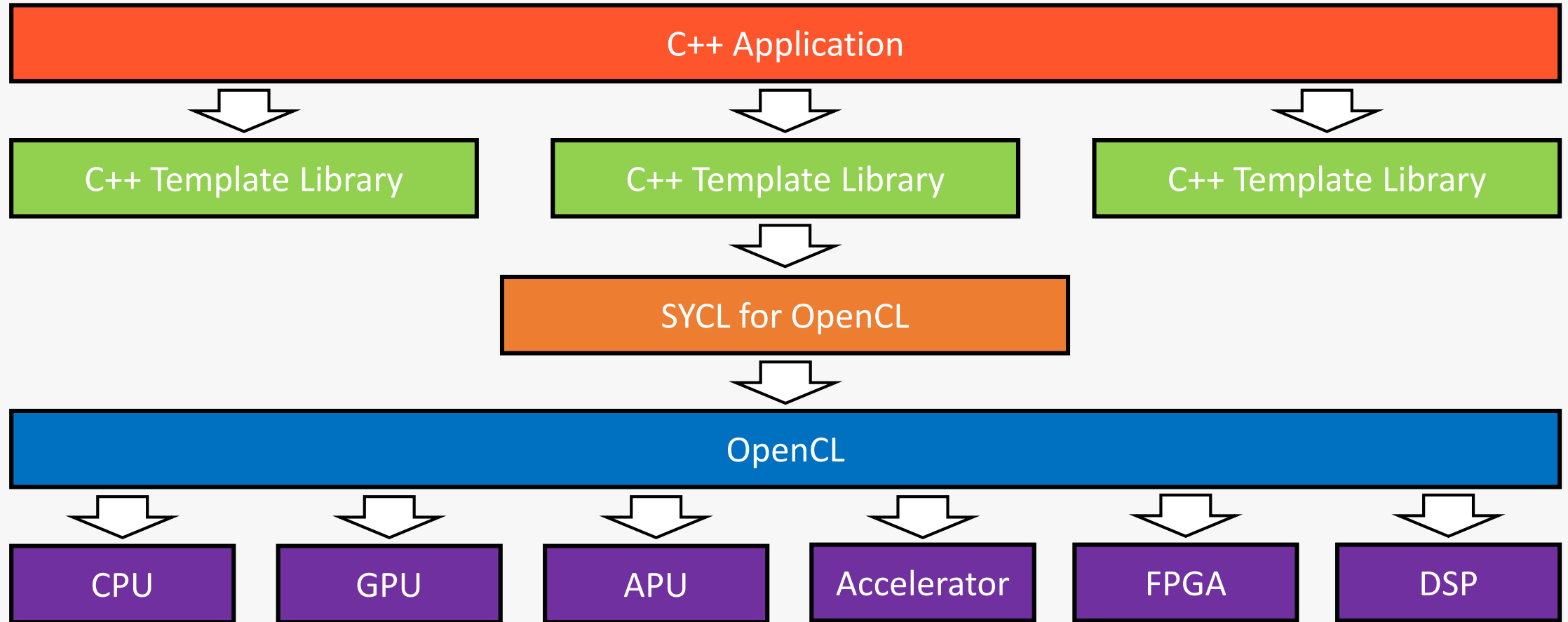
SYCL: A New Approach to Heterogeneous Programming in C++

SYCL for OpenCL



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

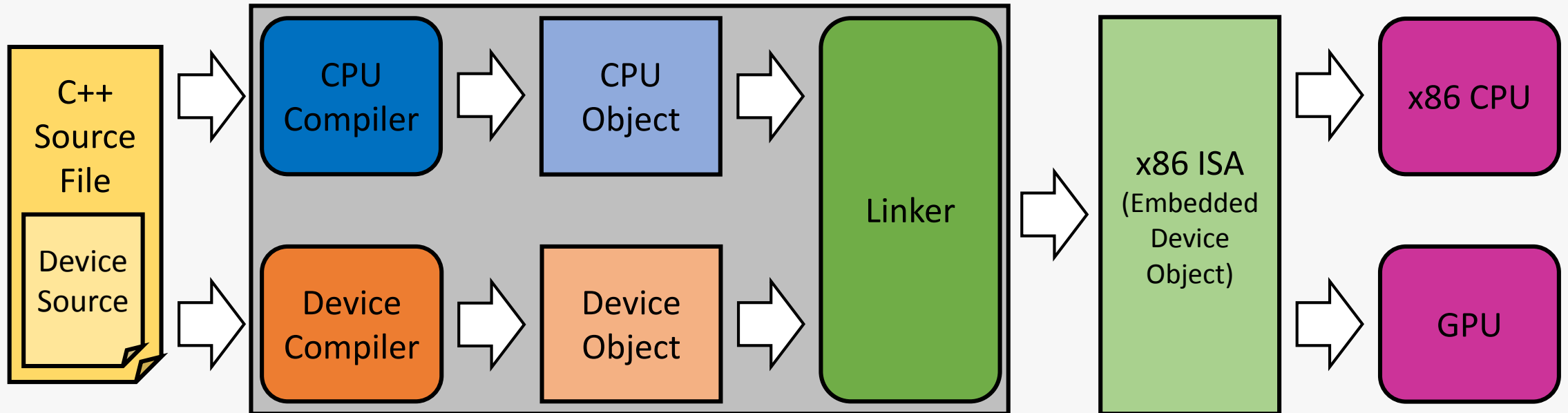
The SYCL Ecosystem



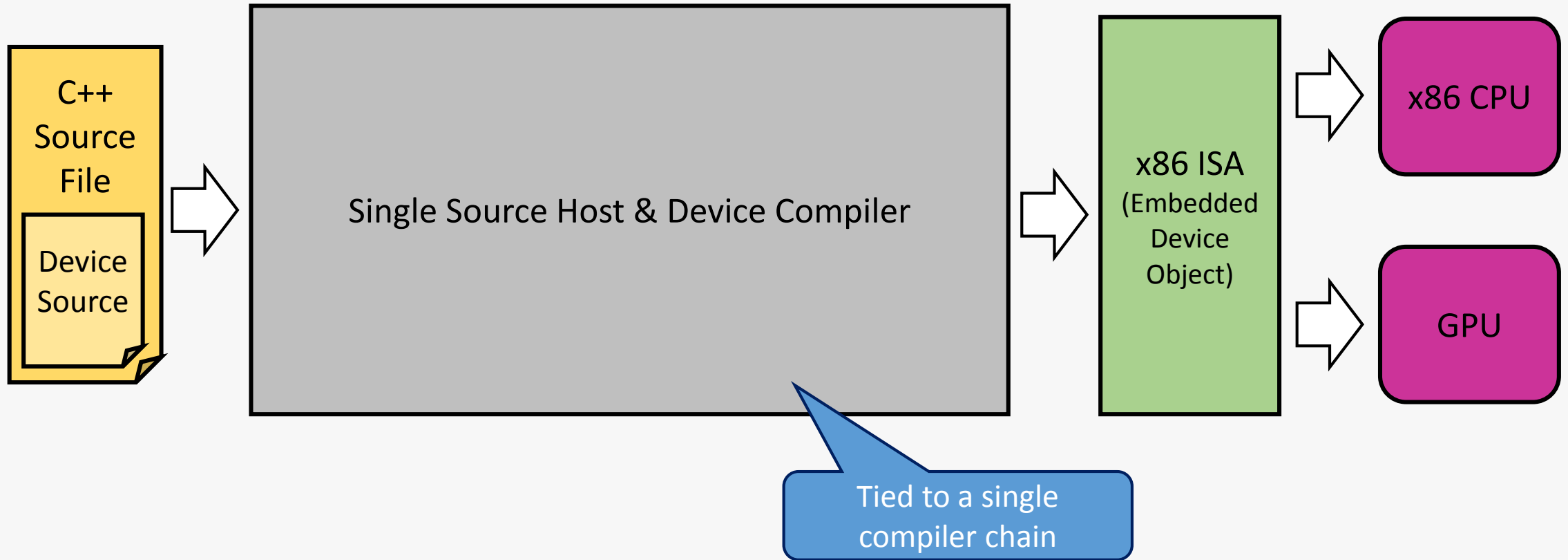
How does SYCL improve heterogeneous offload and performance portability?

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

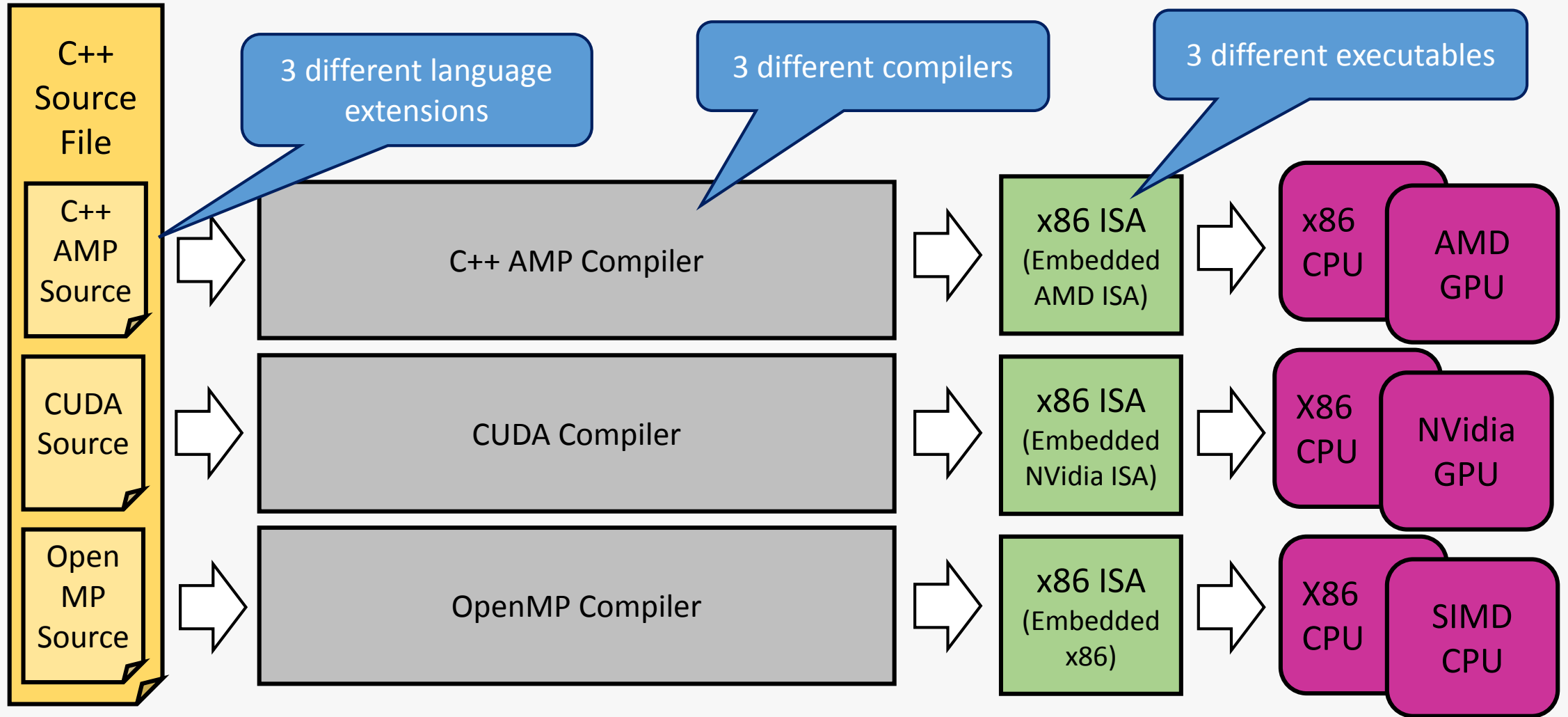
Single Compilation Model



Single Compilation Model



Single Compilation Model



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

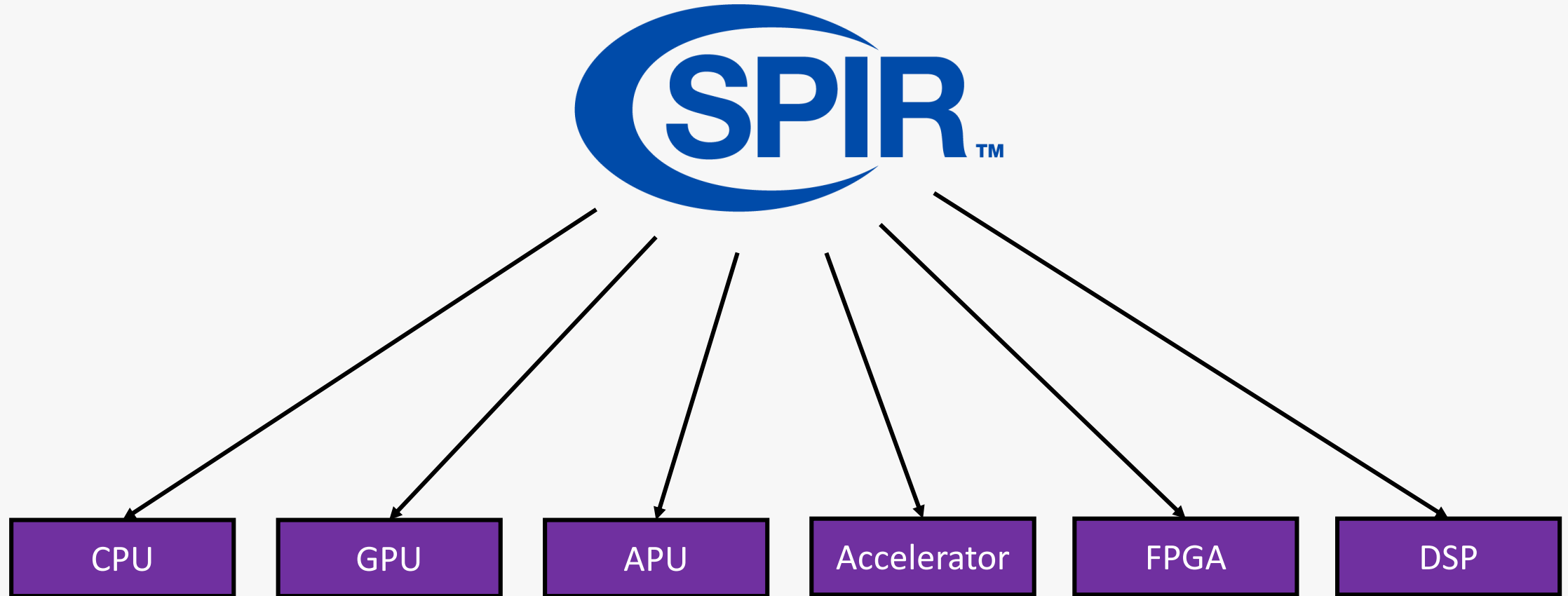
```
parallel_for_each(e, [=](index<2> idx) restrict(amp) {  
    c[idx] = a[idx] + b[idx];  
});
```

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

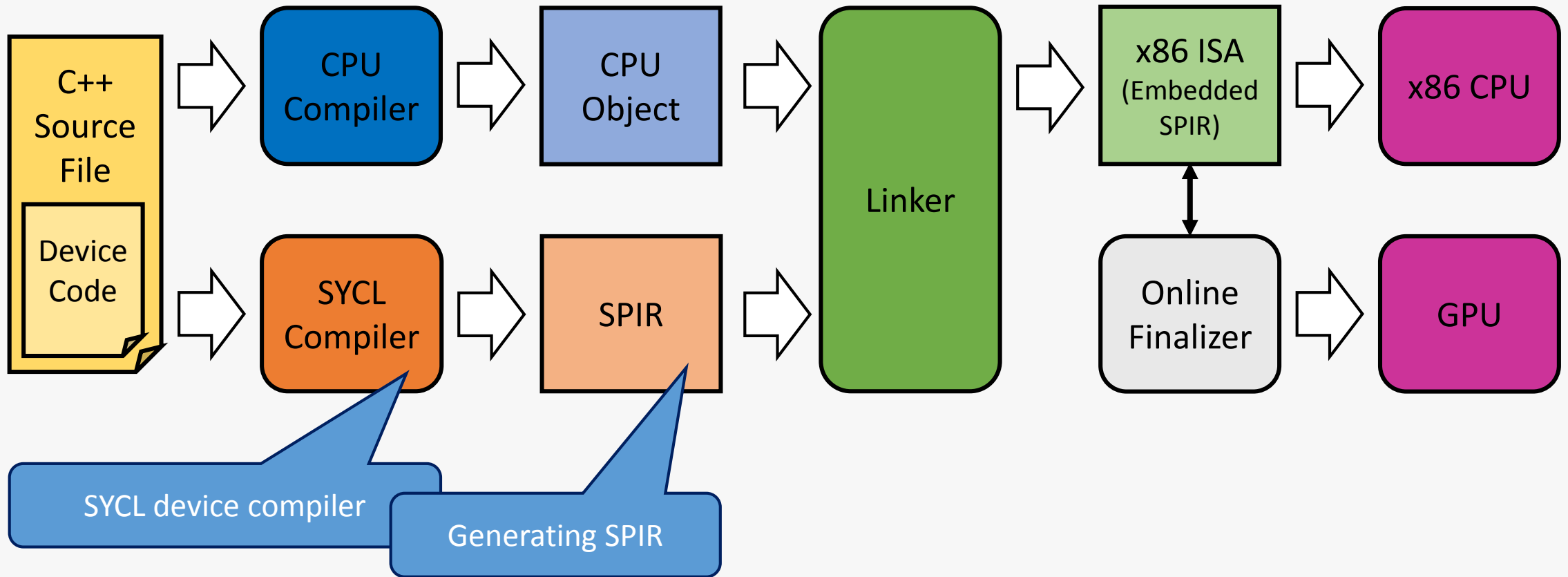
```
#pragma parallel_for  
for (int i = 0; i < a.size(); i++) {
```

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

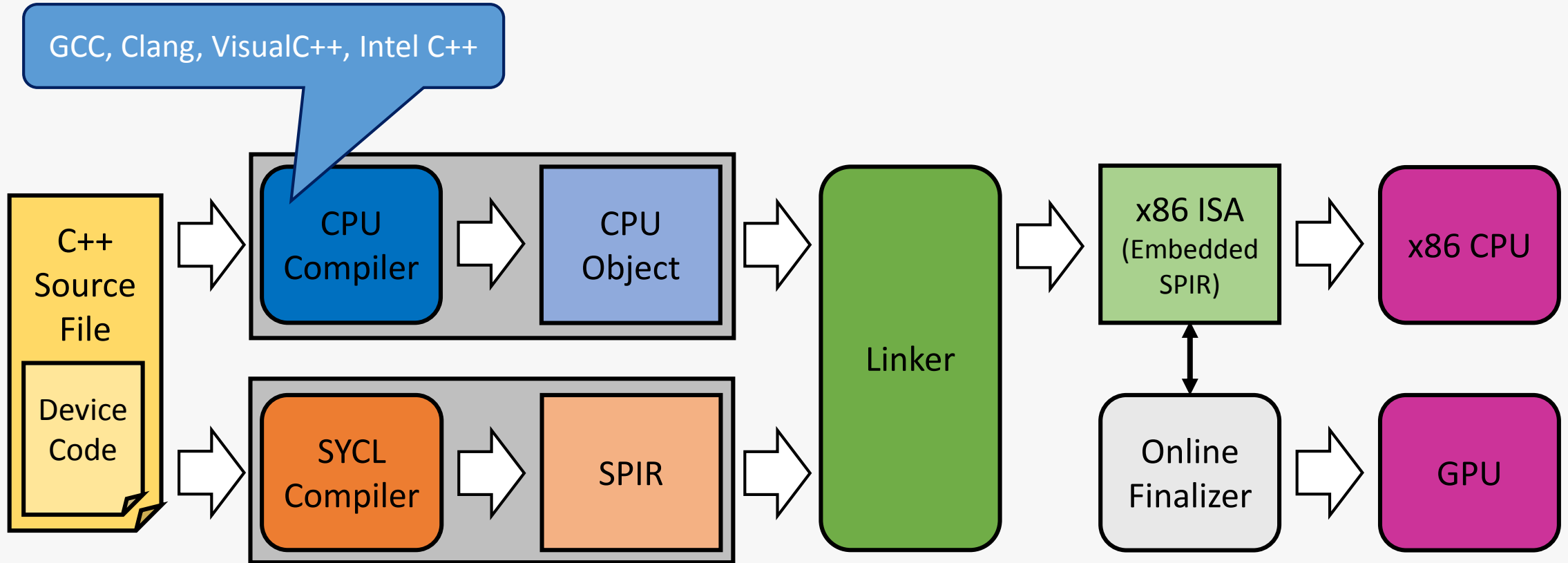
SYCL Targets a Wide Range of Devices with SPIR



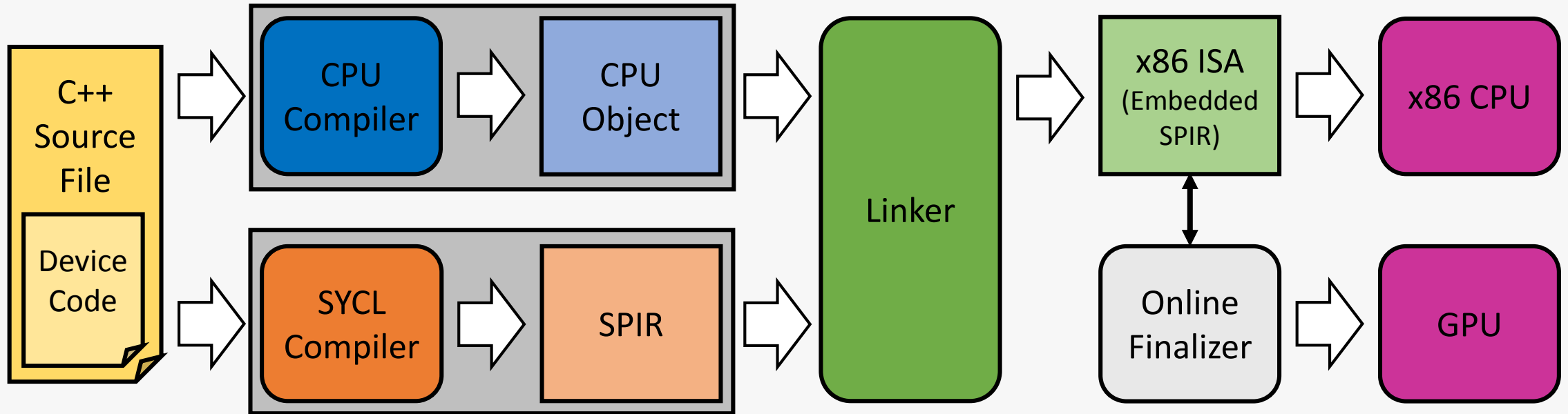
Multi Compilation Model



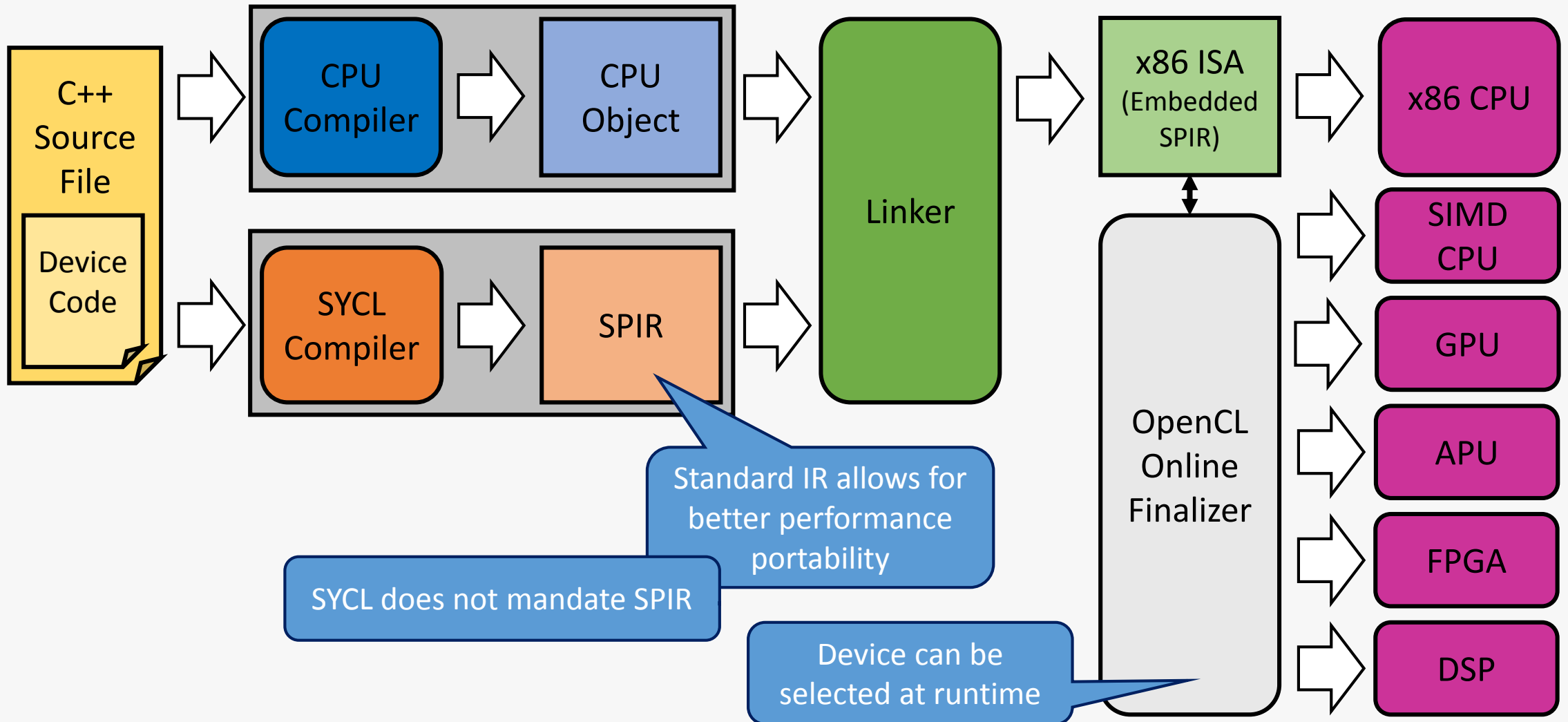
Multi Compilation Model



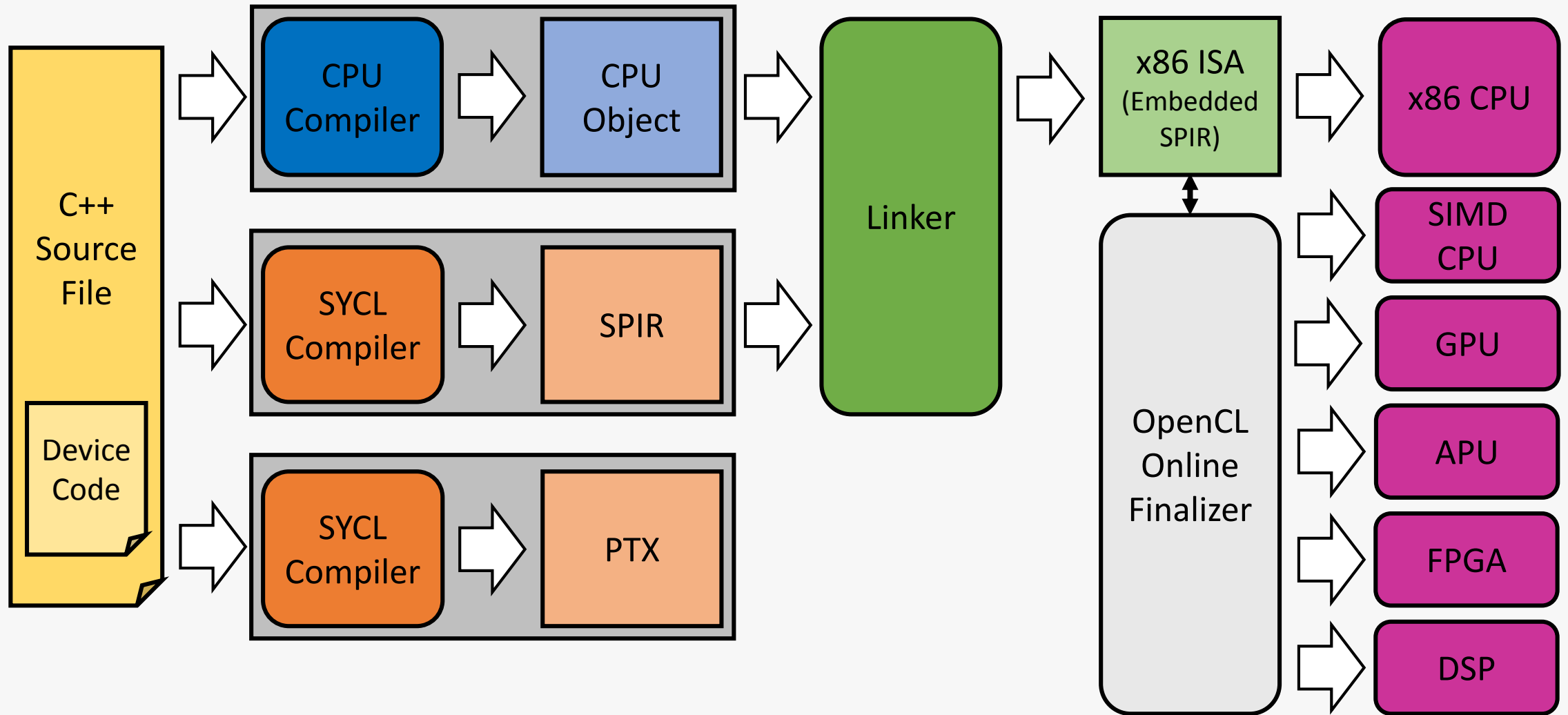
Multi Compilation Model



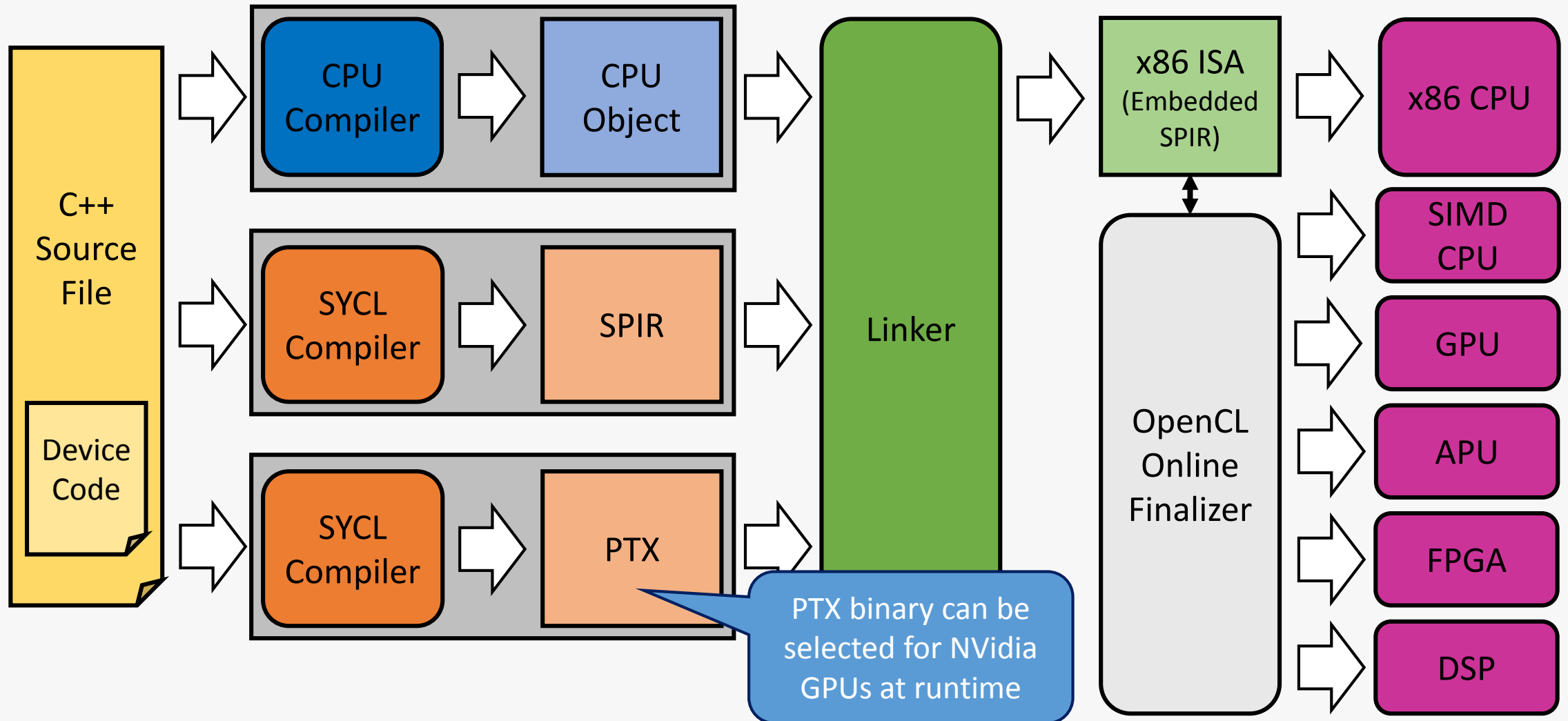
Multi Compilation Model



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

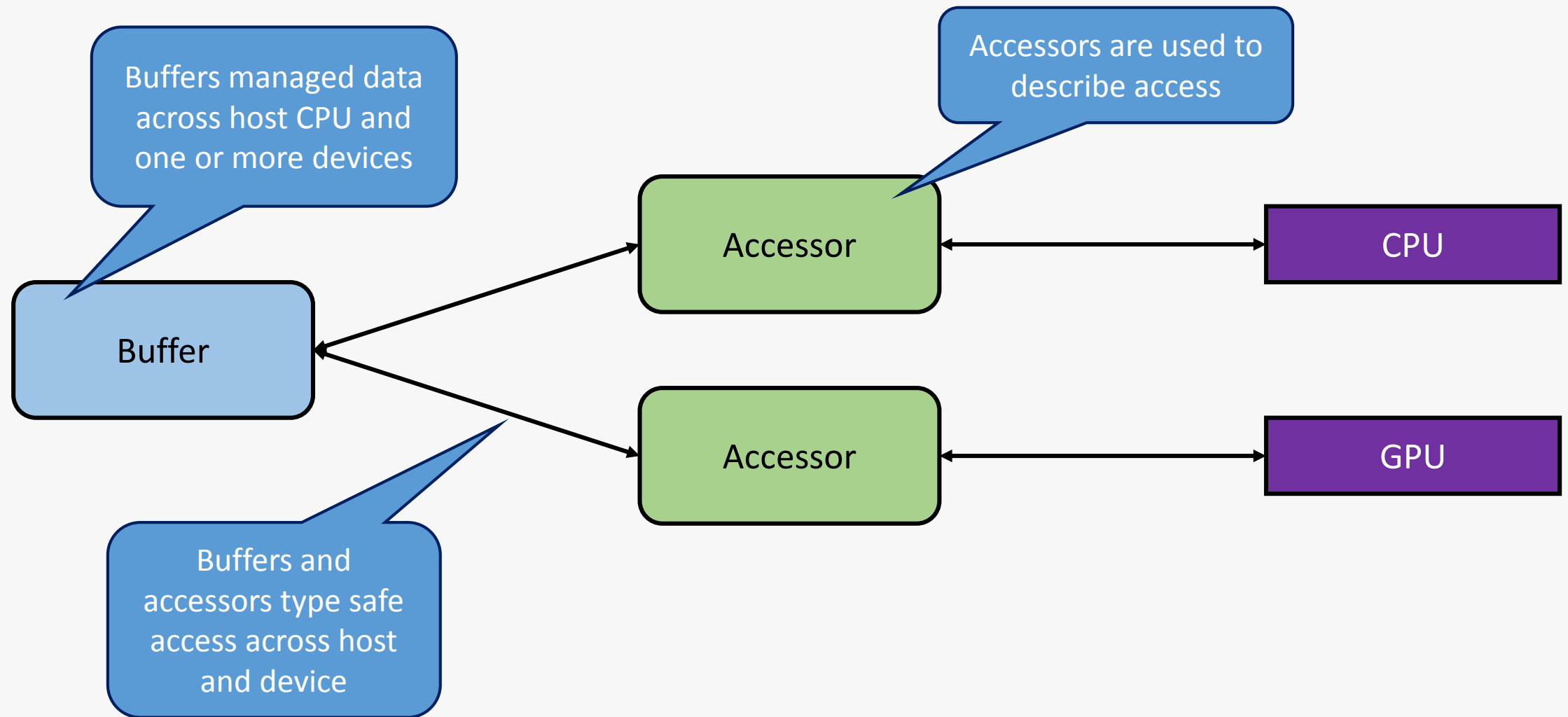
```
cgh.single_task([=]()) {  
    /* task parallel task executed once*/  
});
```

```
cgh.parallel_for(range<2>(64, 64), [=](id<2> idx) {  
    /* data parallel task executed across a range */  
});
```

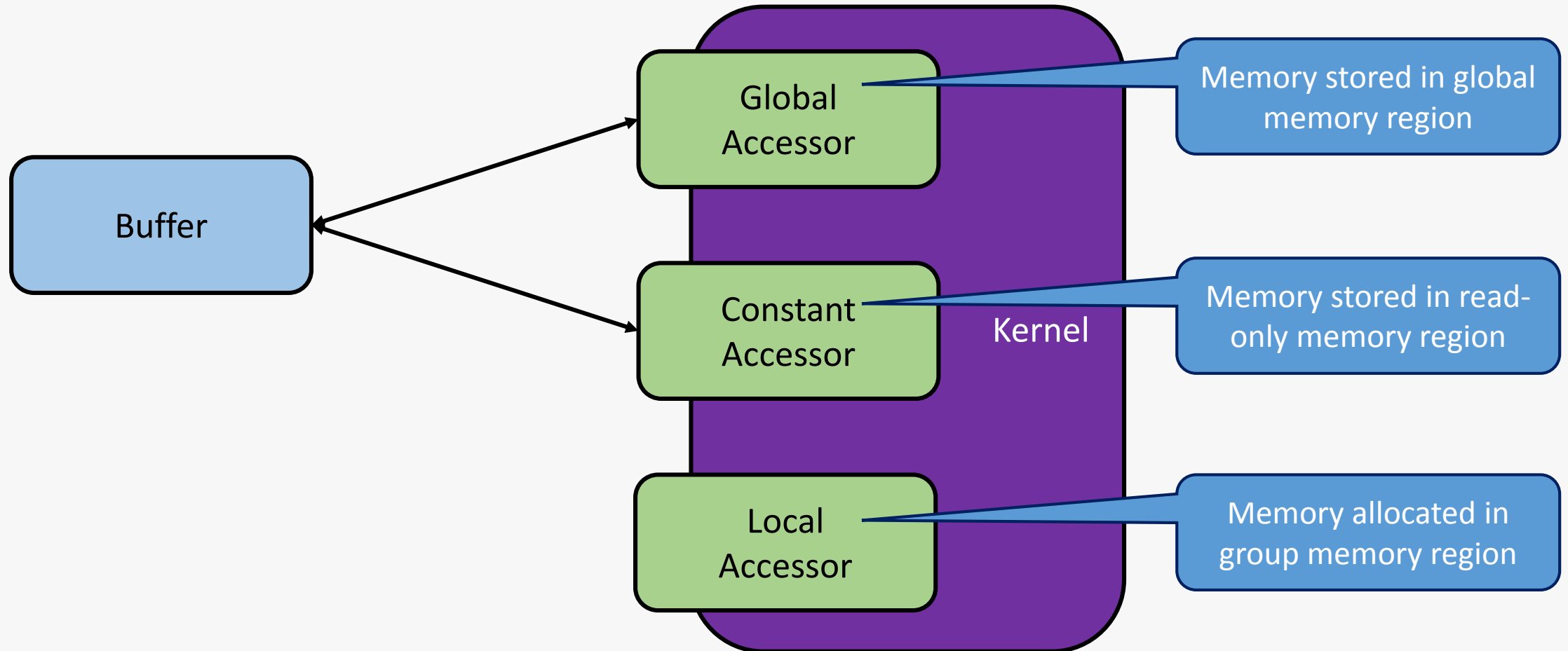
How does SYCL make data movement more efficient?

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

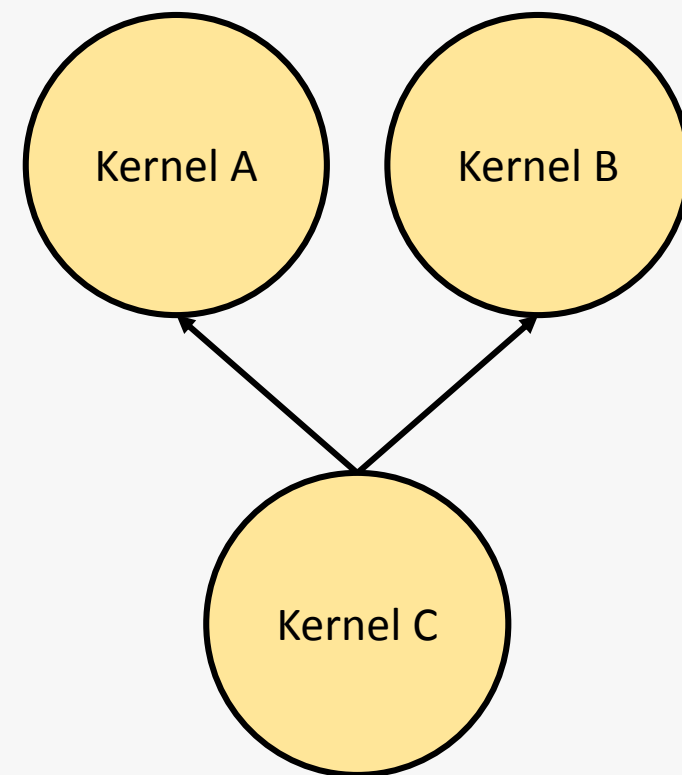
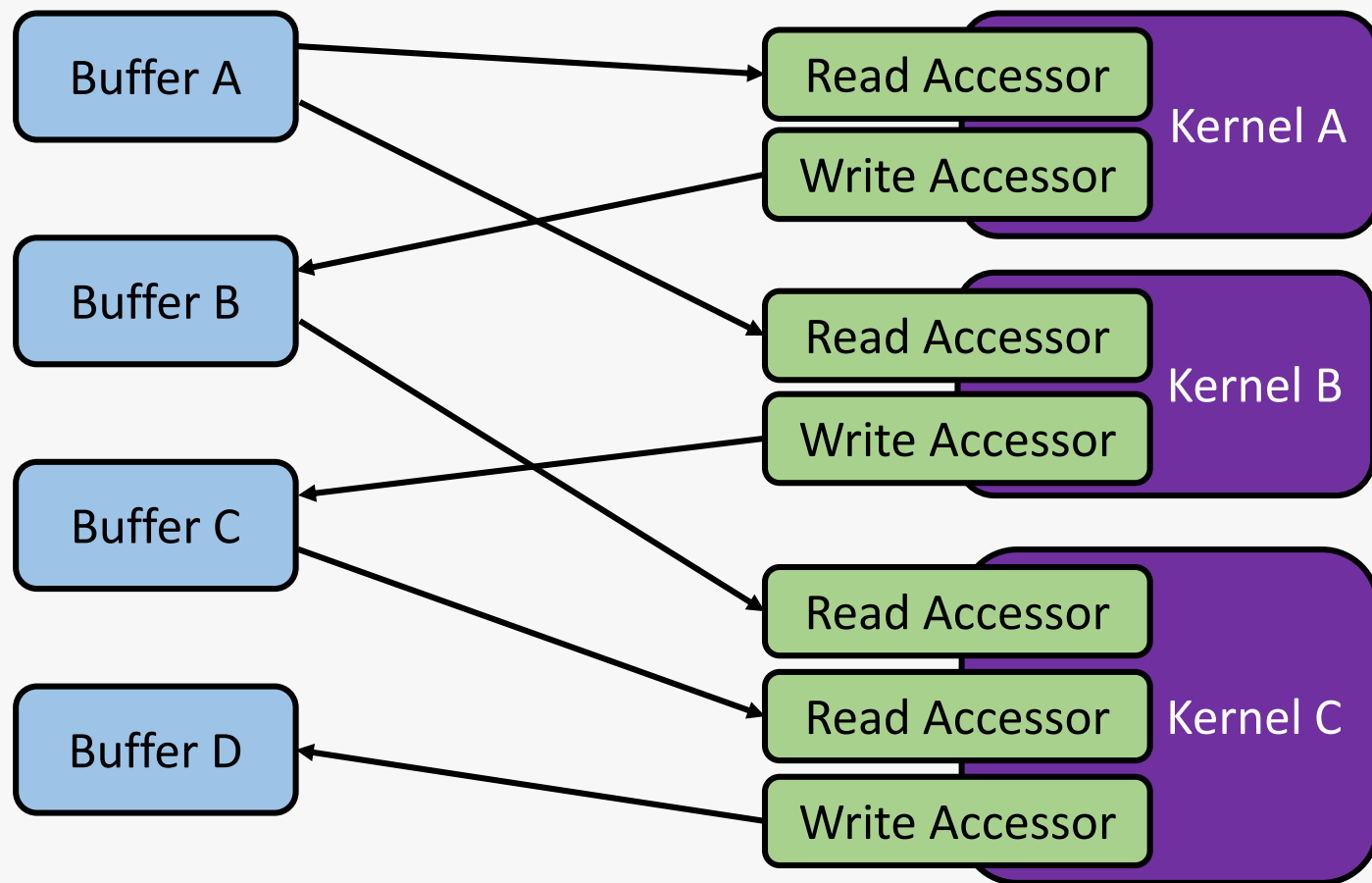
Separating Storage & Access



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) {

}
```

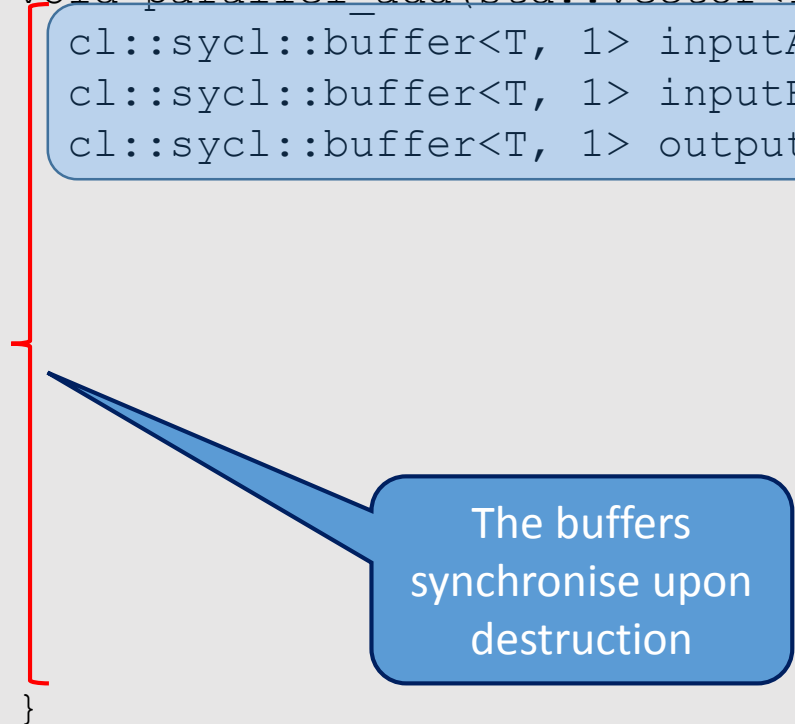
Example: Vector Add

```
#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());

    // ... (parallel execution code) ...

}
```



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);
}
```

The future of heterogeneous programming in C++

So what does the future of heterogeneous programming in C++ look like?

- This is still being discussed within the C++ standards committee

C++1Y(1Y=17/20/22) SG1/SG5/SG14 Plan

red=C++17, blue=C++20? black=future?

Parallelism:

- **Parallel Algorithms:**
- Data-based parallelism.
(vector, SIMD, ...)
- Task-based parallelism
(cilk, OpenMP, fork-join)
- Execution agents
- **Progress guarantees**
- MapReduce
- Pipelines

Concurrency:

- Future++ (then, wait_any, wait_all)
- Executors
- Resumable functions, await
(with futures)
- Lock free
techniques/transactions
- Synchronics
- Atomic Views
- Co-routines
- Counters/queues
- Concurrent vector/unordered
Associative containers
- Latches and barriers
- upgrade_lock
- Atomic smart pointers

Futures & Continuations

- Extensions to C++11 futures
- MS-style .then continuations
 - then()
- Sequential and parallel composition
 - when_all() - join
 - when_any() - choice
- Useful utilities:
 - make_ready_future()
 - is_ready()
 - unwrap()

Heterogeneous Computing

- SYCL for Heterogeneous Computing in C++ for Deep Learning, Machine Vision, Self Driving Cars
- LSU's STELLAR HPX for Distributed Computing
- Nvidia's Agency for Bulk dispatch
- Asynchronous and Distributed Parallelism APIs in HPX
- Heterogeneous Computing Compiler from HSA that supports ultra-low-latency
- Next big frontier

Heterogeneous Computing

- Massive parallelism
- On Multiple Distributed nodes of CPUs or GPUs or integrated devices or FPGAs
- Head start in games...
- Graphics cards

Heterogeneous Computing

- Direct3D
- Nvidia GeForce
- ATI Radeon

Executors

- Executors are to function execution what allocators are to memory allocation
- If a control structure such as `std::async()` or the parallel algorithms describe work that is to be executed
- An executor describes where and when that work is to be executed

The Idea Behind Executors

Diverse
Control
Structures

```
async(...)      for_each(...)
define_task_block(...)  defer(...)
your_favorite_control_structure(...)
```

Unified Interface for Execution

Diverse
Execution
Resources

Operating
System Threads

SIMD vector
units

Thread pool
schedulers

GPU
runtime

OpenMP
runtime

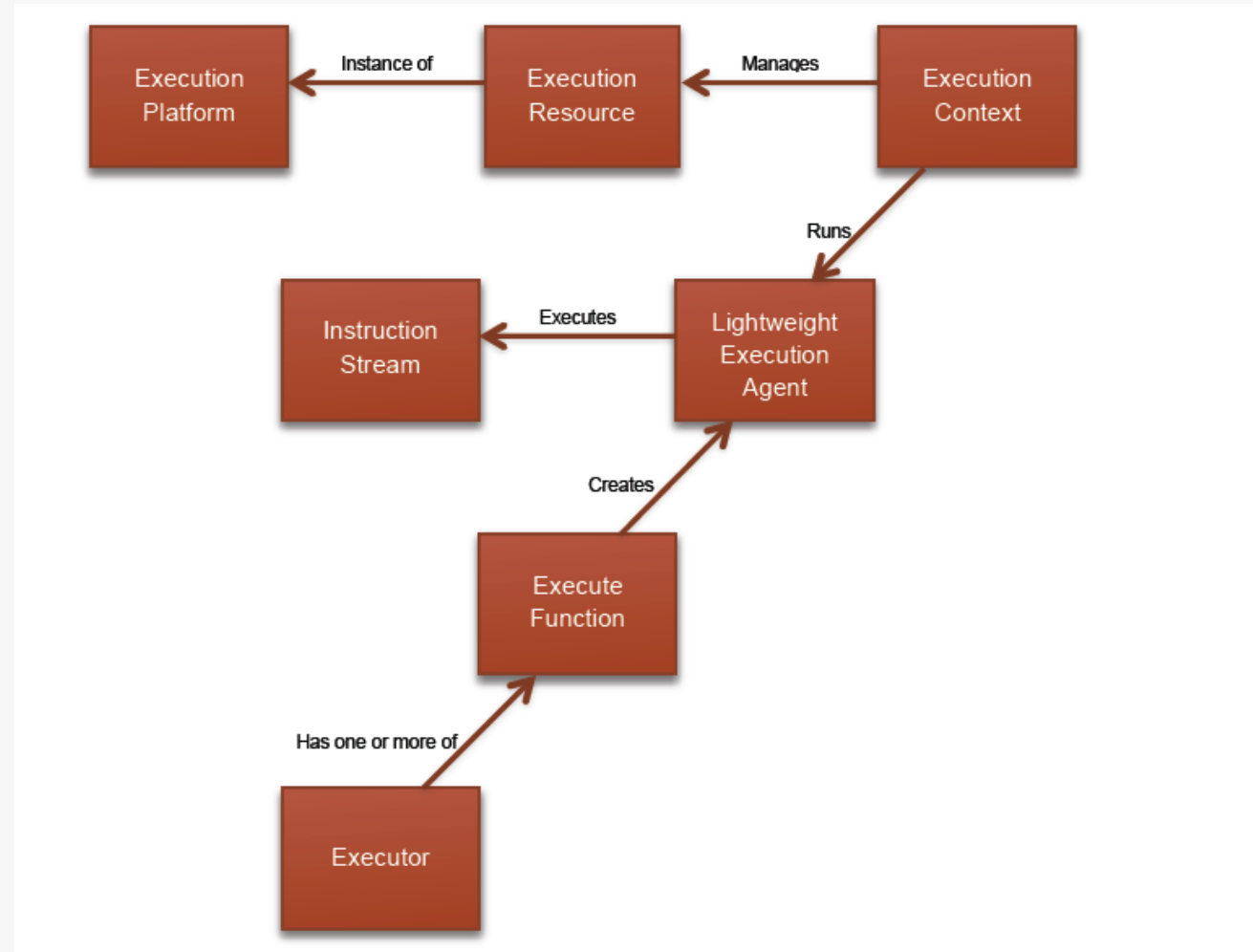
Fibers

Several Competing Proposals

- P0008r0 (Mysen): Minimal interface for fire-and-forget execution
- P0058r1 (Hoberock *et al.*): *Functionality needed for foundations of Parallelism TS*
- P0113r0 (Kohlhoff): Functionality needed for foundations of Networking TS
- P0285r0 (Kohlhoff): Executor categories & customization points

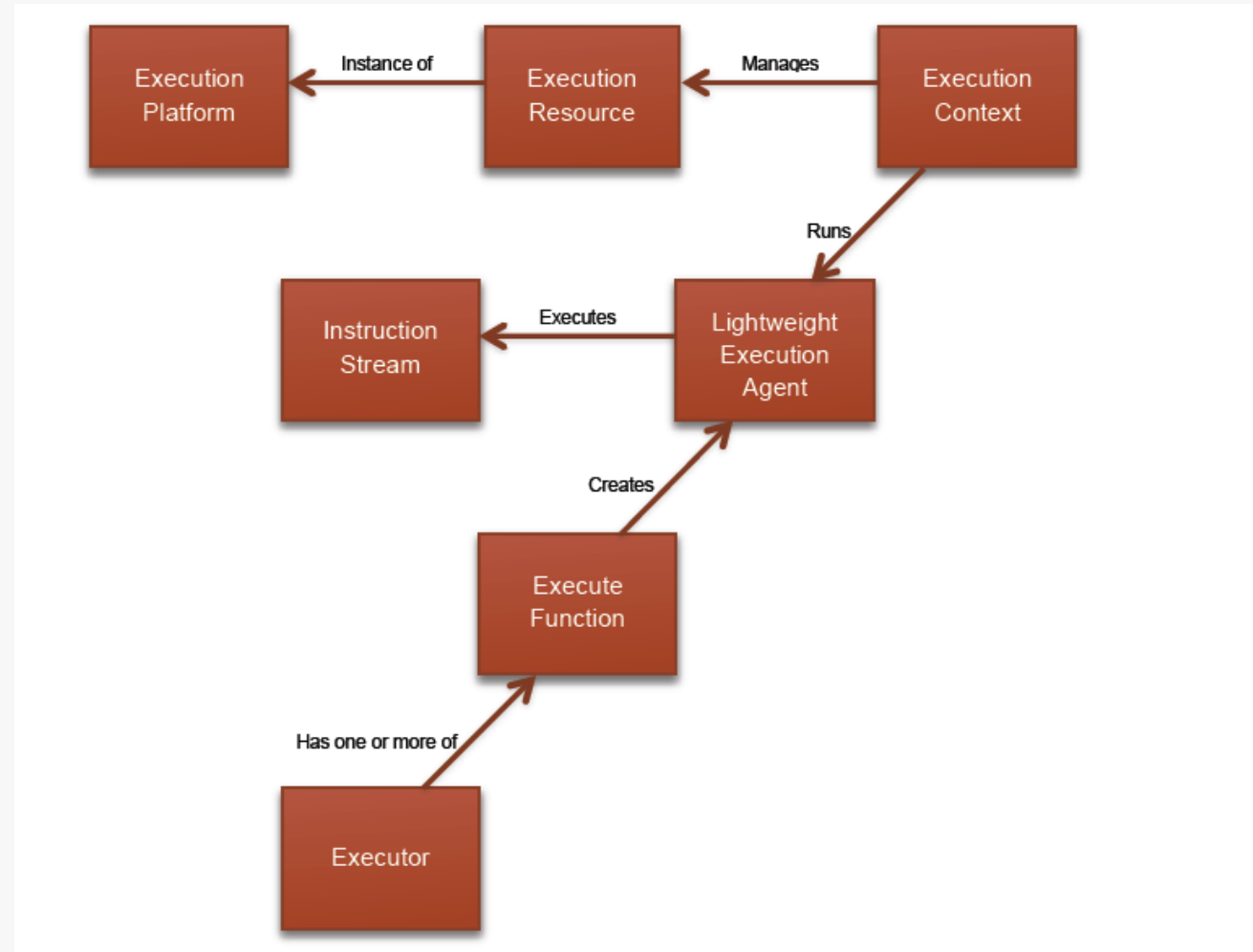
Current Progress of Executors

- Closing in on minimal proposal
- A foundation for later proposals (for heterogeneous computing)
- Still work in progress



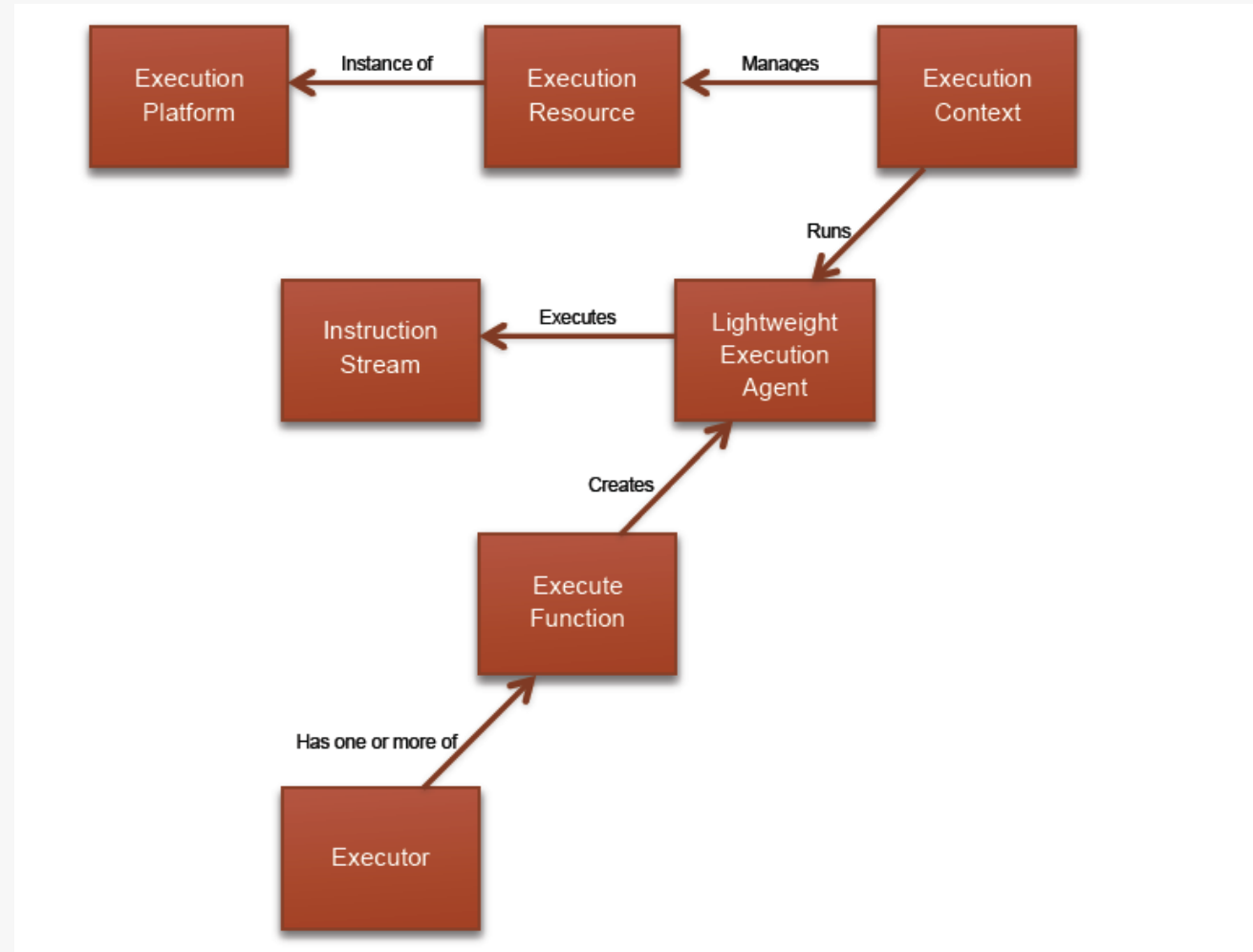
Current Progress of Executors

- An *instruction stream* is the function you want to execute
- An *executor* is an interface that describes where and when to run an *instruction stream*
- An *executor* has one or more *execute functions*
- An *execute function* executes an *instruction stream* on light weight *execution agents* such as threads, SIMD units or GPU threads



Current Progress of Executors

- An *execution platform* is a target architecture such as linux x86
- An *execution resource* is the hardware abstraction that is executing the work such as a thread pool
- An *execution context* manages the light weight *execution agents* of an *execution resource* during the execution



Vector SIMD Parallelism for Parallelism TS2

- No standard!
- Boost.SIMD
- Proposal N3571 by Mathias Gaunard et. al., based on the Boost.SIMD library.
- Proposal N4184 by Matthias Kretz, based on Vc library.
- Unifying efforts and expertise to provide an API to use SIMD portably
- Within C++ (P0203, P0214)
- P0193 status report
- P0203 design considerations
- Please see Pablo Halpern, Nicolas Guillemot's and Joel Falcou's talks on Vector SPMD, and SIMD.

SIMD from Matthias Kretz and Mathias Gaunard

- `std::datapar<T, N, Abi>`
 - `datapar<T, N>` SIMD register holding N elements of type T
 - `datapar<T>` same with optimal N for the currently targeted architecture
 - Abi Defaulted ABI marker to make types with incompatible ABI different
 - Behaves like a value of type T but applying each operation on the N values it contains, possibly in parallel.
- Constraints
 - T must be an integral or floating-point type (tuples/struct of those once we get reflection)
 - N parameter under discussion, probably will need to be power of 2.

Operations on datapar (from Mathias Gaunard)

- Built-in operators
- All usual binary operators are available, for all:
 - $\text{datapar}\langle T, N \rangle \text{ } \text{datapar}\langle U, N \rangle$
 - $\text{datapar}\langle T, N \rangle \text{ } U, U \text{ } \text{datapar}\langle T, N \rangle$
- Compound binary operators and unary operators as well
 - $\text{datapar}\langle T, N \rangle$ convertible to $\text{datapar}\langle U, N \rangle$
 - $\text{datapar}\langle T, N \rangle (U)$ broadcasts the value
- No promotion:
 - $\text{datapar}\langle \text{uint8_t} \rangle (255) + \text{datapar}\langle \text{uint8_t} \rangle (1) == \text{datapar}\langle \text{uint8_t} \rangle (0)$
- Comparisons and conditionals:
 - $==, !=, <, <=, >$ and $>=$ perform element-wise comparison return $\text{mask}\langle T, N, \text{Abi} \rangle$
 - $\text{if}(\text{cond}) x = y$ is written as $\text{where}(\text{cond}, x) = y$
 - $\text{cond} ? x : y$ is written as $\text{if_else}(\text{cond}, x, y)$



Community Edition

Available now for free!

Visit:

compute.cpp.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>

Conclusion

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

We're
Hiring!

codeplay.com/careers/



Thanks



[@codeplaysoft](https://twitter.com/codeplaysoft)



info@codeplay.com



codeplay.com