12th CENTRAL & EASTERN EUROPEAN
SOFTWARE ENGINEERING CONFERENCE IN RUSSIA

2016 CEE-SEC(R)

October 28 - 29, Moscow

Heterogeneous Computing in C++ for Self-driving cars

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

- How do we get to programming self-driving cars?
- 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

How do we get from here...

0
Level 0
- Warnings

1
Level 1
- Adaptive
- Assist

2
Level 2
- Execute
- Automated manoeuvres

3
Level 3
- Limited overall control

4
Level 4
- Deep self control
- All conditions

5
Level 5
- Autonomous
- Stages from very local to extensive journeys

... to here ?

These are the *SAE levels* for autonomous vehicles. Similar challenges apply in other embedded intelligence industries

We have a mountain to climb

- How do we get to the top?
- When we don’t know what the top looks like...
- … and we want to get there in safe, manageable, affordable steps…
- … without getting lost on our own…
- … or climbing the wrong mountain

This presentation will focus on:

- The hardware and software platforms that will be able to deliver the results
- The software tools to build up the solutions for those platforms
- The open standards that will enable solutions to interoperate
- How Codeplay can help deliver embedded intelligence

Codeplay

Standards bodies

- HSA Foundation: Chair of software group, spec editor of runtime and debugging
- Khronos: chair & spec editor of SYCL. Contributors to OpenCL, Safety Critical, Vulkan
- ISO C++: Chair of Low Latency, Embedded WG; Editor of SG1 Concurrency TS
- EEMBC: members

Research

- Members of EU research consortiums: PEPPHER, LPGPU, LPGPU2, CARP
- Sponsorship of PhDs and EngDs for heterogeneous programming: HSA, FPGAs, ray-tracing
- Collaborations with academics
- Members of HiPEAC

Open source

- HSA LLDB Debugger
- SPIR-V tools
- RenderScript debugger in AOSP
- LLDB for Qualcomm Hexagon
- TensorFlow for OpenCL
- C++ 17 Parallel STL for SYCL
- VisionCpp: C++ performance-portable programming model for vision

Presentations

- Building an LLVM back-end
- Creating an SPMD Vectorizer for OpenCL with LLVM
- Challenges of Mixed-Width Vector Code Gen & Scheduling in LLVM
- C++ on Accelerators: Supporting Single-Source SYCL and HSA
- LLDB Tutorial: Adding debugger support for your target

Company

- Based in Edinburgh, Scotland
- 57 staff, mostly engineering
- License and customize technologies for semiconductor companies
- ComputeAorta and ComputeCpp: implementations of OpenCL, Vulkan and SYCL
- 15+ years of experience in heterogeneous systems tools

2001 - 2003

VectorC for x86
Our VectorC technology was chosen and actively used for Computer Vision

First showing of VectorC{VU}

Delivered VectorC{VU} to the National Center for Supercomputing

VectorC{EE} released
An optimising C/C++ compiler for PlayStation®2 Emotion Engine (MIPS)

2005 - 2006

Ageia chooses Codeplay for PhysX
Codeplay is chosen by Ageia to provide a compiler for the PhysX processor.

Codeplay joins the Khronos Group

2007 - 2011

Sieve C++ Programming System released
Aimed at helping developers to parallelise C++ code, evaluated by numerous researchers

Offload released for Sony PlayStation®3

OffloadCL technology developed

Codeplay joins the PEPPHER project

2013

New R&D Division
Codeplay forms a new R&D division to develop innovative new standards and products

Becomes specification editor of the SYCL standard

2014

LLDB Machine Interface Driver released

Codeplay joins the CARP project

Codeplay shows technology to accelerate Renderscript on OpenCL using SPIR

2015

Chair of HSA System Runtime working group

Development of tools supporting the Vulkan API

2016

Open-Source HSA Debugger release

Releases partial OpenCL support (via SYCL) for Eigen Tensors to power TensorFlow

ComputeAorta 1.0 release

ComputeCpp Community Edition beta release
First public edition of Codeplay's SYCL technology

Codeplay build the software platforms that deliver massive performance

What our ComputeCpp users say about us

Benoit Steiner – Google TensorFlow engineer

“We at Google have been working closely with Luke and his Codeplay colleagues on this project for almost 12 months now. Codeplay's contribution to this effort has been tremendous, so we felt that we should let them take the lead when it comes down to communicating updates related to OpenCL. … we are planning to merge the work that has been done so far… we want to put together a comprehensive test infrastructure”

ONERA

“We work with royalty-free SYCL because it is hardware vendor agnostic, single-source C++ programming model without platform specific keywords. This will allow us to easily work with any heterogeneous processor solutions using OpenCL to develop our complex algorithms and ensure future compatibility”

Hartmut Kaiser - HPX

“My team and I are working with Codeplay's ComputeCpp for almost a year now and they have resolved every issue in a timely manner, while demonstrating that this technology can work with the most complex C++ template code. I am happy to say that the combination of Codeplay's SYCL implementation with our HPX runtime system has turned out to be a very capable basis for Building a Heterogeneous Computing Model for the C++ Standard using high-level abstractions.”

WIGNER Research Centre for Physics

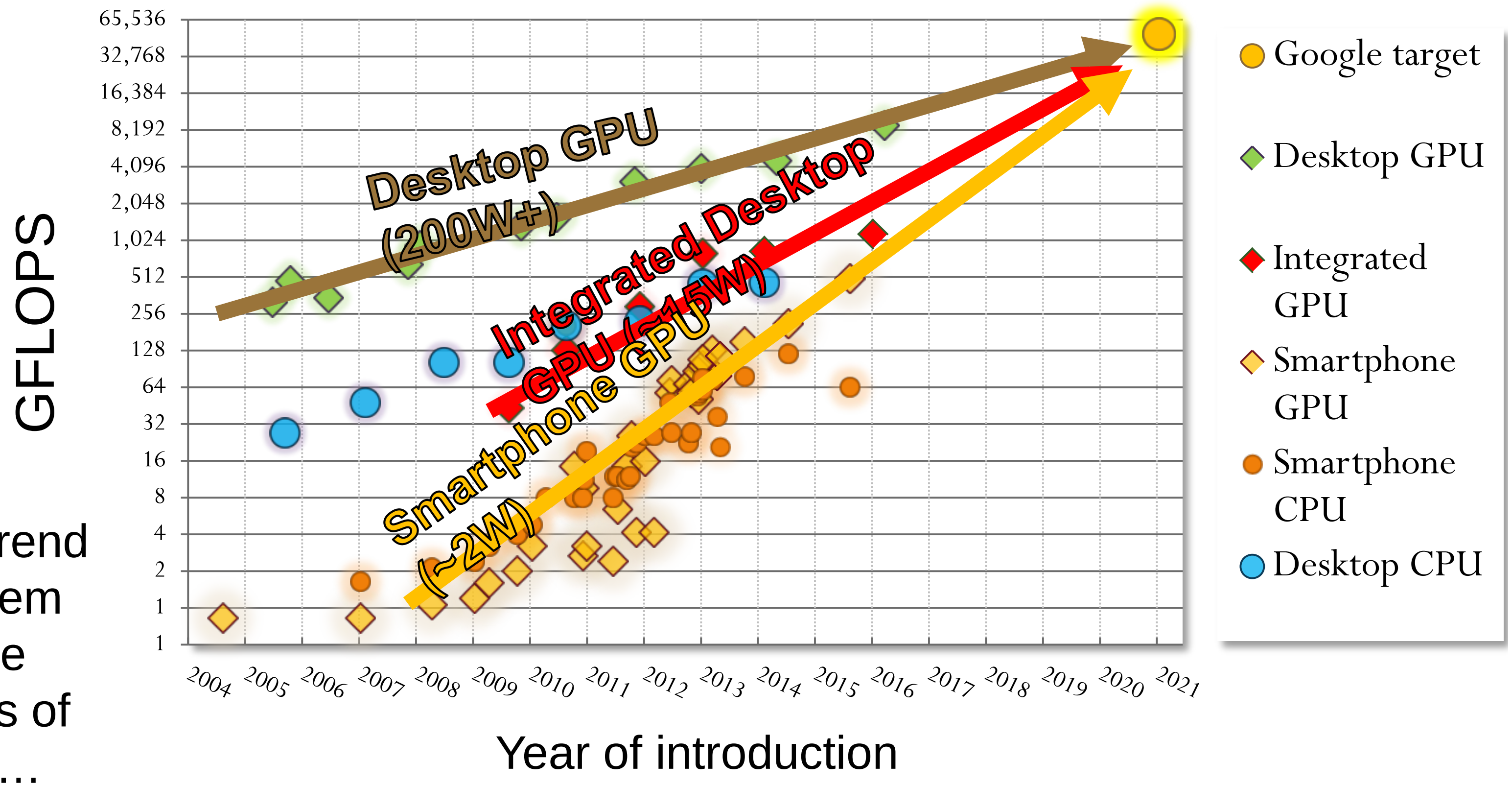
It was a great pleasure this week for us, that Codeplay released the ComputeCpp project for the wider audience. We've been waiting for this moment and keeping our colleagues and students in constant rally and excitement. We'd like to build on this opportunity to increase the awareness of this technology by providing sample codes and talks to potential users. We're going to give a lecture series on modern scientific programming providing field specific examples.“

Where do we need to go?

"On a 100 millimetre-squared chip, Google needs something like 50 teraflops of performance"

- Daniel Rosenband (Google's self-driving car project) at HotChips 2016

Performance trends



These trend lines seem to violate the rules of physics…

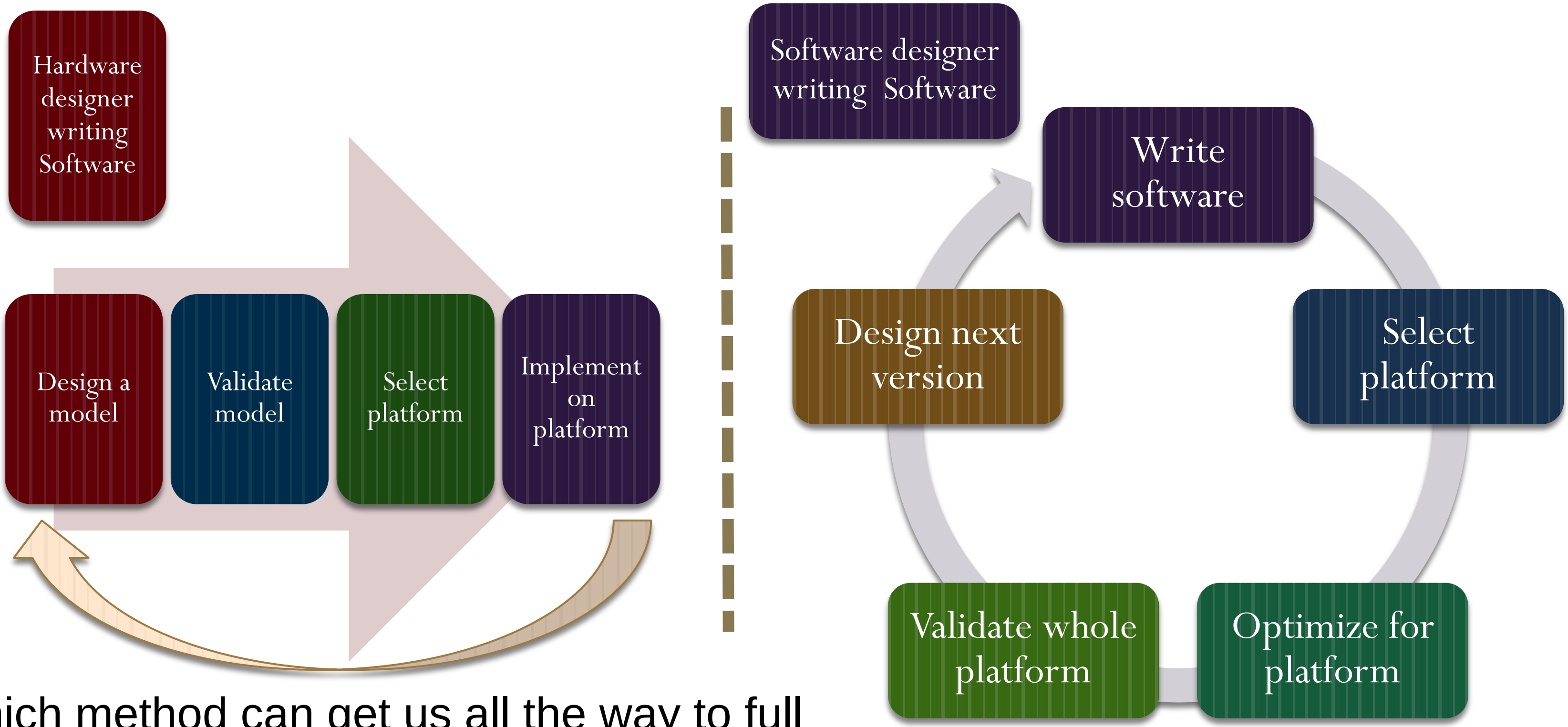
How do we get there from here?

1. We need to write software today for platforms that cannot be built yet

We need to start with simpler systems that are not fully autonomous

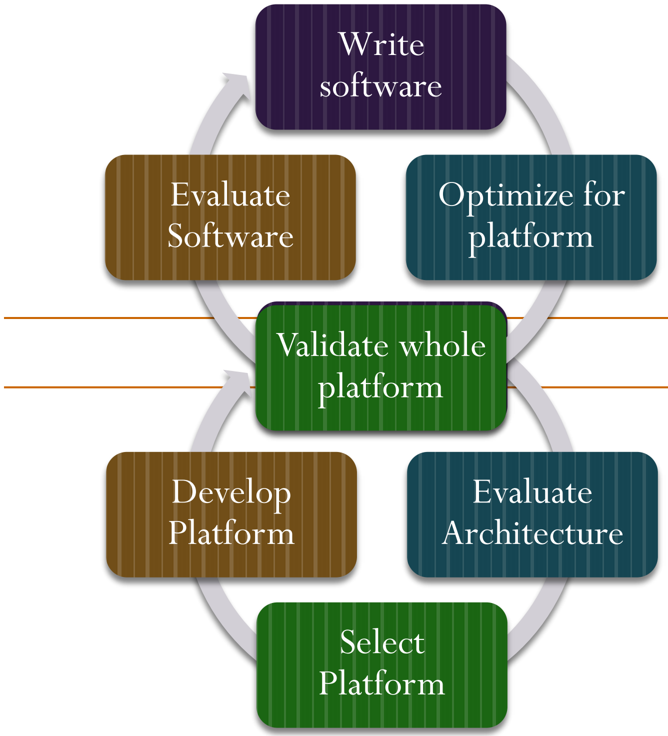
We need to validate the systems as safe

Two models of software development



Which method can get us all the way to full autonomy?

Desirable Development

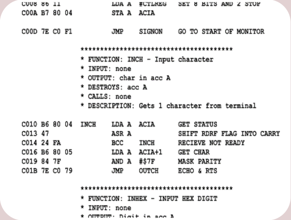


Software Application

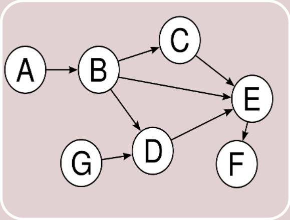
Well Defined Middleware

Hardware & Low-Level Software

The different levels of programming model



SPIR



Device-specific programming
- Assembly language
- VHDL
- Device-specific C-like programming models

Higher-level language enabler
- NVIDIA PTX
- HSA
- OpenCL SPIR
- SPIR-V

C-level programming
- OpenCL C
- DSP C
- MCAPI/MTAPI

C++-level programming
- SYCL
- CUDA
- HCC
- C++ AMP

Graph programming
- OpenCV
- OpenVX
- Halide
- VisionCpp
- TensorFlow
- Caffe

Device-specific programming

Cannot …
develop software
today for future
platforms

Can deliver
quick results
today

Can…
hand-optimize
directly for the
device

- Not a route to full autonomy
- Does not allow software developers to invest today

The route to full autonomy

- Graph programming
  - This is the most widely-adopted approach to machine vision and machine learning
- Open standards
  - This lets you develop today for future architectures

Why graph programming?

When you scale the number of cores:

- You don't scale the number of memory ports
- Your compute performance increases
- But your off-chip memory bandwidth does not increase

Therefore:

- You need to reduce off-chip memory bandwidth by processing everything on-chip
- This is achieved by *tiling*

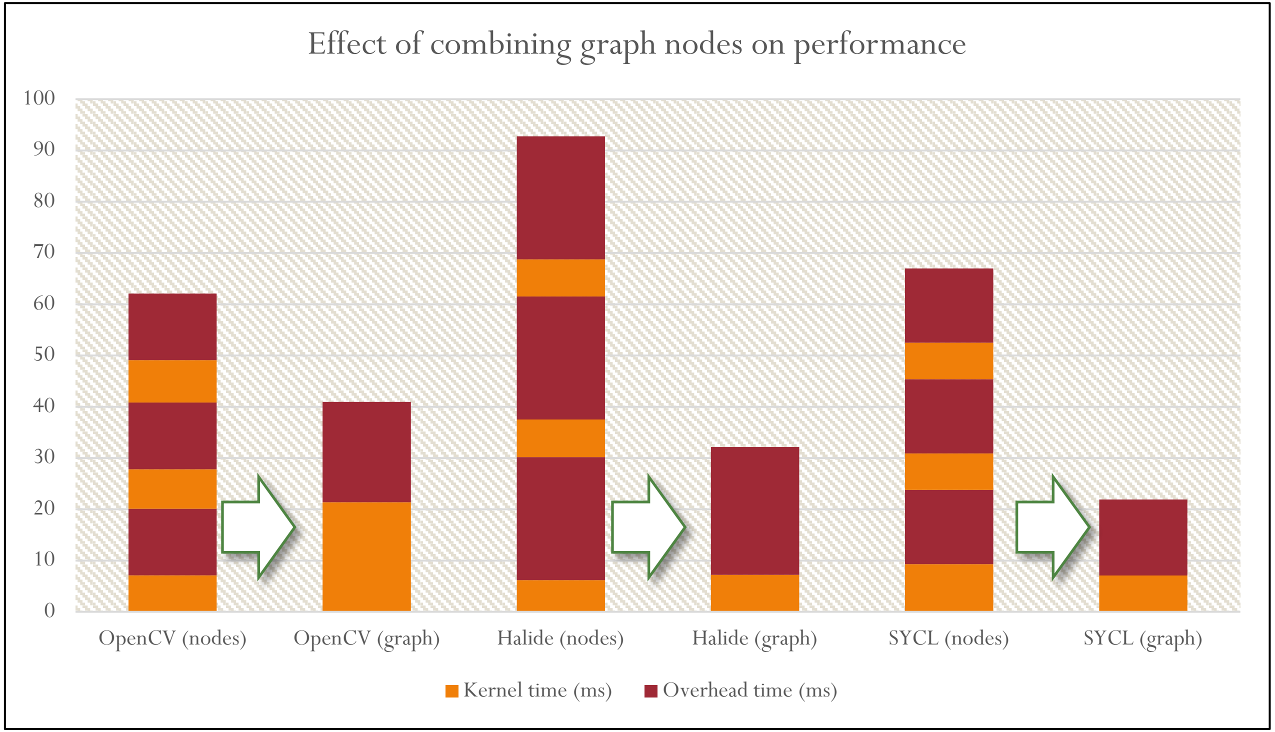
However, writing tiled image pipelines is hard

Ilif we build up a graph of operations (e.g. convolutions) and then have a runtime system split into fused tiled operations across an entire system-on-chip, we get great performance

Graph programming: some numbers

In this example, we perform 3 image processing operations on an accelerator and compare 3 systems when executing individual nodes, or a whole graph

The system is an AMD APU and the operations are: RGB->HSV, channel masking, HSV->RGB



Halide and SYCL use kernel fusion, whereas OpenCV does not. For all 3 systems, the performance of the whole graph is significantly better than individual nodes executed on their own

Graph programming

- For both machine vision algorithms and machine learning, graph programming is the most widely-adopted approach
- Two styles of graph programming that we commonly see:

C-style graph programming	C++-style graph programming
• OpenVX • OpenCV	• Halide • RapidMind • Eigen (also in TensorFlow) • VisionCpp

C-style graph programming

OpenVX: open standard

- Can be implemented by vendors
- Create a graph with C API, then map to an entire SoC

OpenCV: open source

- Implemented on OpenCL
- Implemented on device-specific accelerators
- Create a graph with C API, then execute

OpenCV & OpenVX Vision Acceleration

Device-Specific Programming

What happens if we invent our own graph nodes?

Runtime systems can automatically optimize the graphs

Can … develop software today for future platforms

How do we adapt it for all the graph nodes we need?

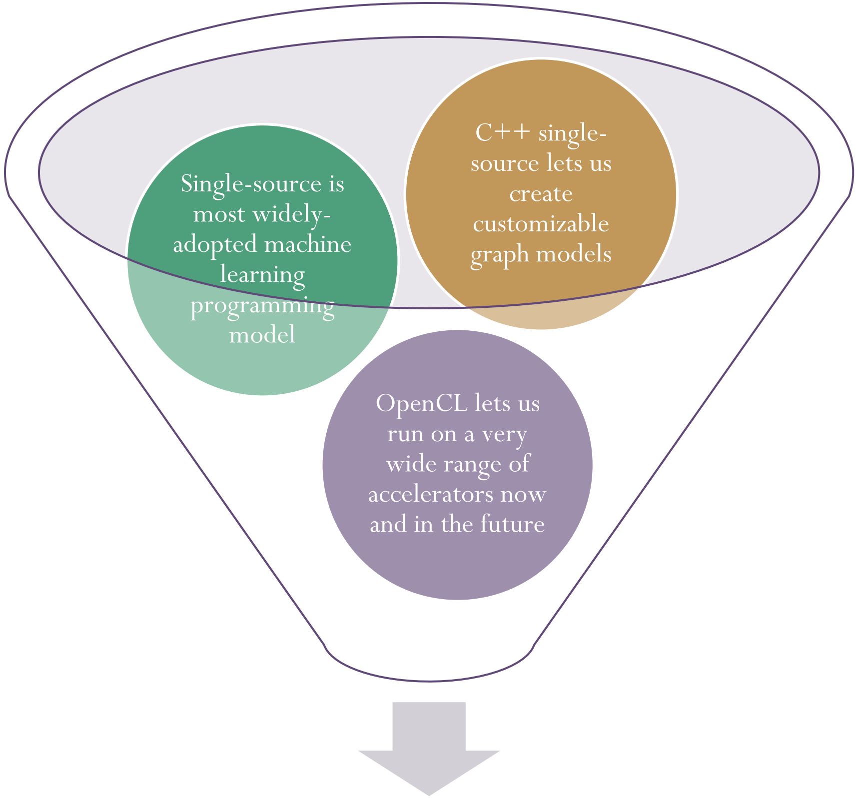
C++-style graph programming

Examples in machine vision/machine learning	C++ compilers that support this style
• Halide	• CUDA
• RapidMind	• C++ OpenMP
• Eigen (also in TensorFlow)	• C++ 17 Parallel STL
• VisionCpp	• SYCL

C++ single-source programming

- C++ lets us build up graphs at compile-time
  - This means we can map a graph to the processors offline
- C++ lets us write custom nodes ourselves
- This approach is called a *C++ Embedded Domain-Specific Language*
- Very widely used, eg Eigen, Boost, TensorFlow, RapidMind, Halide

Combining open standards, C++ and graph programming



SYCL combines C++ single-source with OpenCL acceleration

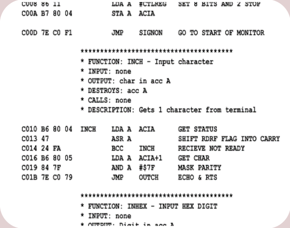
Putting it all together: building it

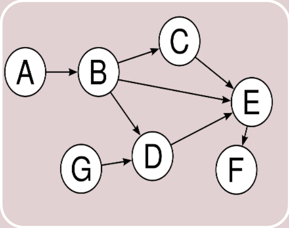
Higher-level programming enablers

NVIDIA PTX	HSA	OpenCL SPIR	SPIR-V
• NVIDIA CUDA-only	• Royalty-free open standard • HSAIL is the IR • Provides a single address space, with virtual memory • Low-latency communication	• Defined for OpenCL v1.2 • Based on Clang/LLVM (the open-source compiler)	• Open standard • Defined by Khronos • Supports compute and graphics (OpenCL, Vulkan and OpenGL) • Not tied to any compiler

Open standard intermediate representations enable tools to be built on top and support a wide range of platforms

Which model should we choose?





Device-specific programming
- Assembly language
- VHDL
- Device-specific C-like programming models

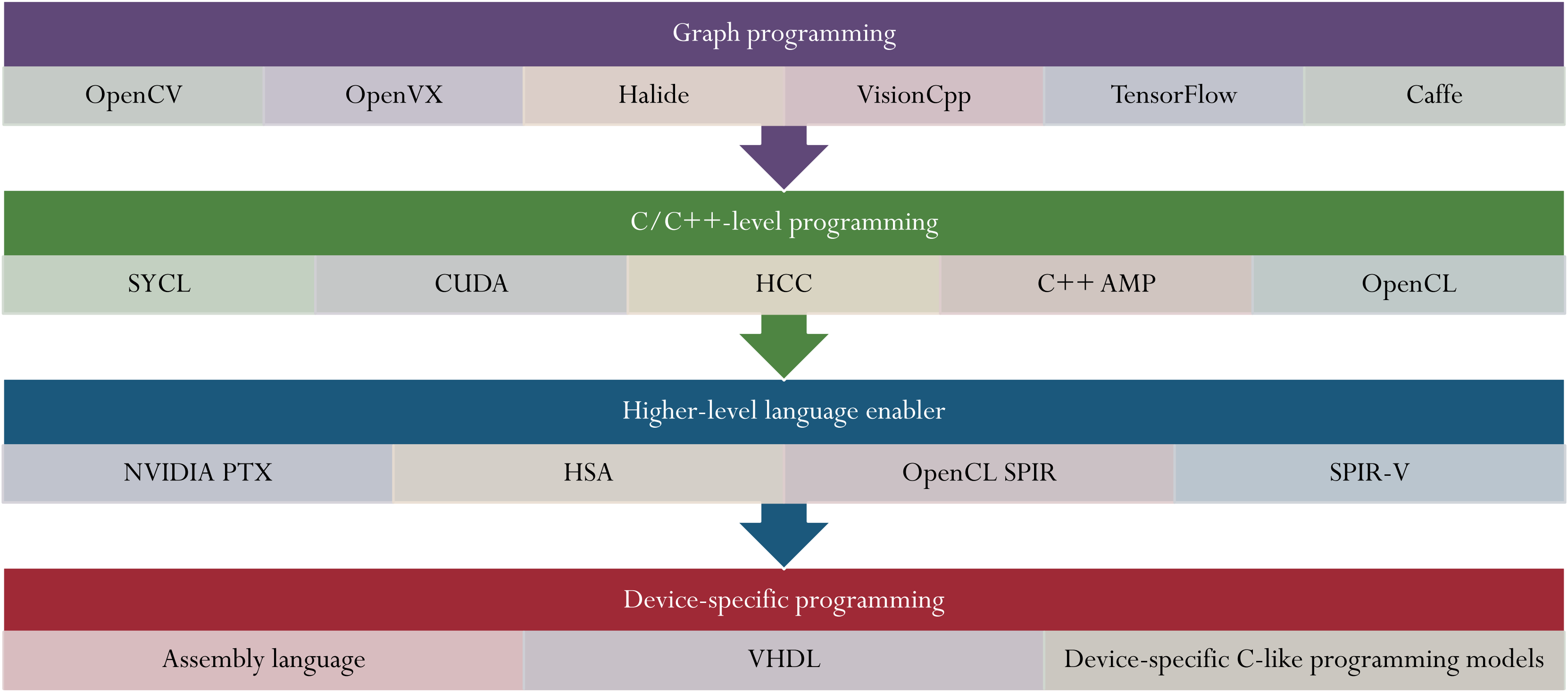
Higher-level language enabler
- NVIDIA PTX
- HSA
- OpenCL SPIR
- SPIR-V

C-level programming
- OpenCL C
- DSP C
- MCAPI/MTAPI

C++-level programming
- SYCL
- CUDA
- HCC
- C++ AMP

Graph programming
- OpenCV
- OpenVX
- Halide
- VisionCpp
- TensorFlow
- Caffe

They are not *alternatives*, they are *layers*



Can specify, test and validate each layer

- **Graph programming**
  - Validate graph models
  - Validate the code using standard tools
- **C/C++-level programming**
  - OpenCL/SYCL specs
  - Clsmith testsuite
  - Conformance testsuites
  - Wide range of other testsuites
- **Higher-level language enabler**
  - SPIR/SPIR-V/HSAIL specs
  - Conformance testsuites
- **Device-specific programming**
  - Device-specific specification
  - Device-specific testing and validation

For Codeplay, these are our layer choices

We have chosen a layer of standards, based on current market adoption

- TensorFlow and OpenCV
- SYCL
- OpenCL (with SPIR)
- LLVM as the standard compiler back-end

Device-specific programming
- LLVM

Higher-level language enabler
- OpenCL SPIR

C/C++-level programming
- SYCL

Graph programming
- TensorFlow
- OpenCV

The actual choice of standards may change based on market dynamics, but by choosing widely adopted standards and a layering approach, it is easy to adapt

For Codeplay, these are our products

ComputeAorta™

ComputeCpp™

VisionCpp™

Device-specific programming

- LLVM

Higher-level language enabler

- OpenCL SPIR

C/C++-level programming

- SYCL

Graph programming

- TensorFlow
- OpenCV

Further information

- OpenCL https://www.khronos.org/opencl/
- OpenVX https://www.khronos.org/openvx/
- HSA http://www.hsafoundation.com/
- SYCL http://sycl.tech
- OpenCV http://opencv.org/
- Halide http://halide-lang.org/
- VisionCpp https://github.com/codeplaysoftware/visioncpp

Agenda

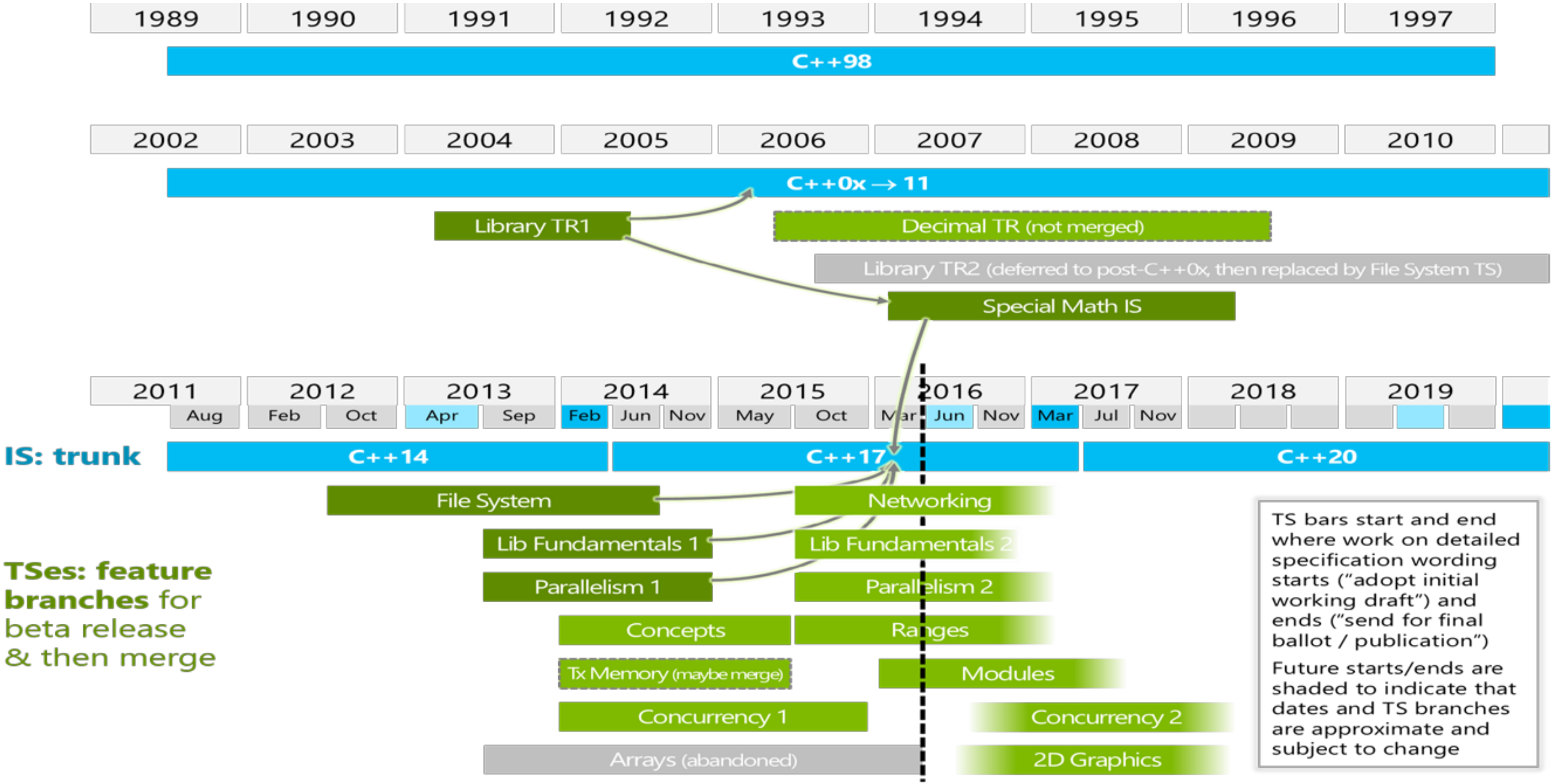
- How do we get to programming self-driving cars?
- SYCL: The open Khronos standard
  - A comparison of Heterogeneous Programming Models
  - SYCL Design Philosophy: C++ end to end model for HPC and consumers
- The ecosystem:
  - VisionCpp
  - Parallel STL
  - TensorFlow, Machine Vision, Neural Networks,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

C++ support for massive parallel heterogeneous devices

- Memory allocation (near, far memory)
- Better affinity for cpu and memory
- Templates (static, compile time)
- Exceptions
- Polymorphism
- Tasks blocks
- Execution Agents/Context
- Progress Guarantees
- Current Technical Specifications
  - Concepts, Parallelism, Concurrency, TM

C++ Std Timeline/status

https://wongmichael.com/2016/06/29/c17-all-final-features-from-oulu-in-a-few-slides/



Pre-C++11 projects

ISO number	Name	Status	What is it?	C++17?
ISO/IEC TR 18015:2006	Technical Report on C++ Performance	Published 2006 (ISO store) Draft: TR18015 (2006-02-15)	C++ Performance report	No
ISO/IEC TR 19768:2007	Technical Report on C++ Library Extensions	Published 2007-11-15 (ISO store) Draft: n1745 (2005-01-17) TR 29124 split off, the rest merged into C++11	Has 14 Boost libraries, 13 of which was added to C++11.	N/A (mostly already included into C++11)
ISO/IEC TR 29124:2010	Extensions to the C++ Library to support mathematical special functions	Published 2010-09-03 (ISO Store) Final draft: n3060 (2010-03-06). Under consideration to merge into C++17 by p0226 (2016-02-10)	Really, ORDINARY math today with a Boost and Dinkumware Implementation	YES
ISO/IEC TR 24733:2011	Extensions for the programming language C++ to support decimal floating-point arithmetic	Published 2011-10-25 (ISO Store) Draft: n2849 (2009-03-06) May be superseded by a future Decimal TS or merged into C++ by n3871	Decimal Floating Point decimal32 decimal64 decimal128	No. Ongoing work in SG6

Status after June Oulu C++ Meeting

ISO number	Name	Status	links	C++17?
ISO/IEC TS 18822:2015	C++ File System Technical Specification	Published 2015-06-18. (ISO store). Final draft: n4100 (2014-07-04)	Standardize Linux and Windows file system interface	YES
ISO/IEC TS 19570:2015	C++ Extensions for Parallelism	Published 2015-06-24. (ISO Store). Final draft: n4507 (2015-05-05)	Parallel STL algorithms.	YES but removed dynamic execution policy, exception_lists, changed some names
ISO/IEC TS 19841:2015	Transactional Memory TS	Published 2015-09-16, (ISO Store). Final draft: n4514 (2015-05-08)	Composable lock-free programming that scales	No. Already in GCC 6 release and waiting for subsequent usage experience.
ISO/IEC TS 19568:2015	C++ Extensions for Library Fundamentals	Published 2015-09-30, (ISO Store). Final draft: n4480 (2015-04-07)	optional, any, string_view and more	YES but moved Invocation Traits and Polymorphic allocators into LF TS2
ISO/IEC TS 19217:2015	C++ Extensions for Concepts	Published 2015-11-13. (ISO Store). Final draft: n4553 (2015-10-02)	Constrained templates	No. Already in GCC 6 release and and waiting for subsequent usage experience.

Status after June Oulu C++ Meeting

ISO number	Name	Status	What is it?	C++17?
ISO/IEC TS 19571:2016	C++ Extensions for Concurrency	Published 2016-01-19. (ISO Store) Final draft: p0159r0 (2015-10-22)	improvements to future, latches and barriers, atomic smart pointers	No. Already in Visual Studio release and waiting for subsequent usage experience.
ISO/IEC DTS 19568:xxxx	C++ Extensions for Library Fundamentals, Version 2	DTS. Draft: n4564 (2015-11-05)	source code information capture and various utilities	No. Resolution of comments from national standards bodies in progress
ISO/IEC DTS 21425:xxxx	Ranges TS	In development, Draft n4569 (2016-02-15)	Range-based algorithms and views	No. Wording review of the spec in progress
ISO/IEC DTS 19216:xxxx	Networking TS	In development, Draft n4575 (2016-02-15)	Sockets library based on Boost.ASIO	No. Wording review of the spec in progress.
	Modules	In development, Draft p0142r0 (2016-02-15) and p0143r1 (2016-02-15)	A component system to supersede the textual header file inclusion model	No. Initial TS wording reflects Microsoft's design; changes proposed by Clang implementers expected.

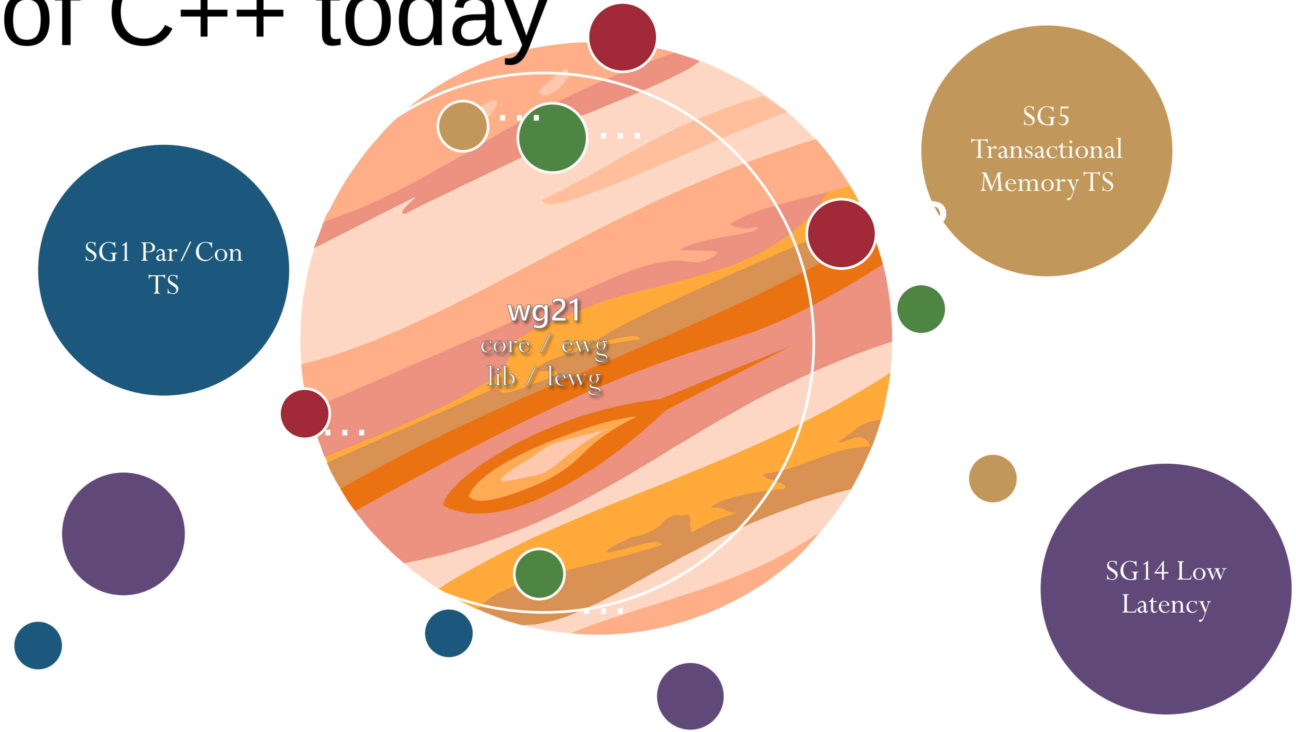
Status after June Oulu C++ Meeting

ISO number	Name	Status	What is it?	C++17?
	Numerics TS	Early development. Draft p0101 (2015-09-27)	Various numerical facilities	No. Under active development
ISO/IEC DTS 19571:xxxx	Concurrency TS 2	Early development	Exploring executors, synchronic types, lock-free, atomic views, concurrent data structures	No. Under active development
ISO/IEC DTS 19570:xxxx	Parallelism TS 2	Early development. Draft n4578 (2016-02-22)	Exploring task blocks, progress guarantees, SIMD.	No. Under active development
ISO/IEC DTS 19841:xxxx	Transactional Memory TS 2	Early development	Exploring on_commit, in_transaction.	No. Under active development.
	Graphics TS	Early development. Draft p0267r0 (2016-02-12)	2D drawing API	No. Wording review of the spec in progress
ISO/IEC DTS 19569:xxxx	Array Extensions TS	Under overhaul. Abandoned draft: n3820 (2013-10-10)	Stack arrays whose size is not known at compile time	No. Withdrawn; any future proposals will target a different vehicle

Status after June Oulu C++ Meeting

ISO number	Name	Status	What is it?	C++17?
	Coroutine TS	Initial TS wording will reflect Microsoft's await design; changes proposed by others expected.	Resumable functions	No. Under active development
	Reflection TS	Design direction for introspection chosen; likely to target a future TS	Code introspection and (later) reification mechanisms	No. Under active development
	Contracts TS	Unified proposal reviewed favourably.)	Preconditions, postconditions, etc.	No. Under active development
	Massive Parallelism TS	Early development	Massive parallelism dispatch	No. Under active development.
	Heterogeneous Device TS	Early development.	Support Hetereogeneous Devices	No. Under active development.
	C++17	On track for 2017	Filesystem TS, Parallelism TS, Library Fundamentals TS I, if constexpr, and various other enhancements are in. See slide 44-47 for details.	YES

The Parallel and concurrency planets of C++ today



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/channels

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

Part 1: Parallel C++ Library

In C++17

Execution Policies Published 2015

```
using namespace std::experimental::parallelism;
std::vector<int> vec = ...

// previous standard sequential sort
std::sort(vec.begin(), vec.end());

// explicitly sequential sort
std::sort(std::seq, vec.begin(), vec.end());

// permitting parallel execution
std::sort(std::par, vec.begin(), vec.end());

// permitting vectorization as well
std::sort(std::par_unseq, vec.begin(), vec.end());
```

Part 2: Forward Progress guarantees in C++17

ParallelSTL

- C++17 execution policies require concurrent or parallel forward progress guarantees
  - This means GPUs are not support by the standard execution policies
- Executors intend to interface with execution policies

```
parallel_for_each(par.on(exec), vec.begin(), vec.end(),
  [=](int&e){ /* … */ });
```

Forward Progress Guarantees

- C++17 forward progress guarantees are:
  - Concurrent forward progress guarantees
    - a thread of execution is required to make forward progress regardless of the forward progress of any other thread of execution.
  - Parallel forward progress guarantees
    - a thread of execution is not required to make forward progress until an execution step has occurred and from that point onward a thread of execution is required to make forward progress regardless of the forward progress of any other thread of execution.
  - Weakly parallel forward progress guarantees
    - a thread of execution is not required to make progress.
- These are not specific guarantees for GPUs

Part 3: Futures++ (.then, wait_any, wait_all) in future C++ 20

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()

Summary Of Proposed Extensions (1)

```
template<typename F>
auto then(F&& func) -> future<decltype(func(*this))>;

template<typename F>
auto then(executor &ex, F&& func) -> future<decltype(func(*this))>;

template<typename F>
auto then(launch policy, F&& func) -> future<decltype(func(*this))>;
```

Summary Of Proposed Extensions (2)

```
template <typename T>
future<typename decay<T>::type> make_ready_future(T&& value);

future<void> make_ready_future();

bool is_ready() const;

template<typename R2>
future<R> future<R>::unwrap();
// R is a future<R2> or shared_future<R2>
```

Summary Of Proposed Extensions (3)

```
template <class InputIterator>
<...> when_all(InputIterator first, InputIterator last);

template <typename... T>
<...> when_all(T&&... futures);

template <class InputIterator>
<...> when_any(InputIterator first, InputIterator last);

template <typename... T>
<...> when_any(T&&... futures);


template <class InputIterator>
<see below> when_any_swapped(InputIterator first, InputIterator last);
```

Part 4: Executors in future

C++20

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
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0443r0.html

The Idea Behind Executors

Diverse Control Structures	`async(...)` `for_each(...)` `define_task_block(...)` `defer(...)` `your_favorite_control_structure(...)`

Unified Interface for Execution

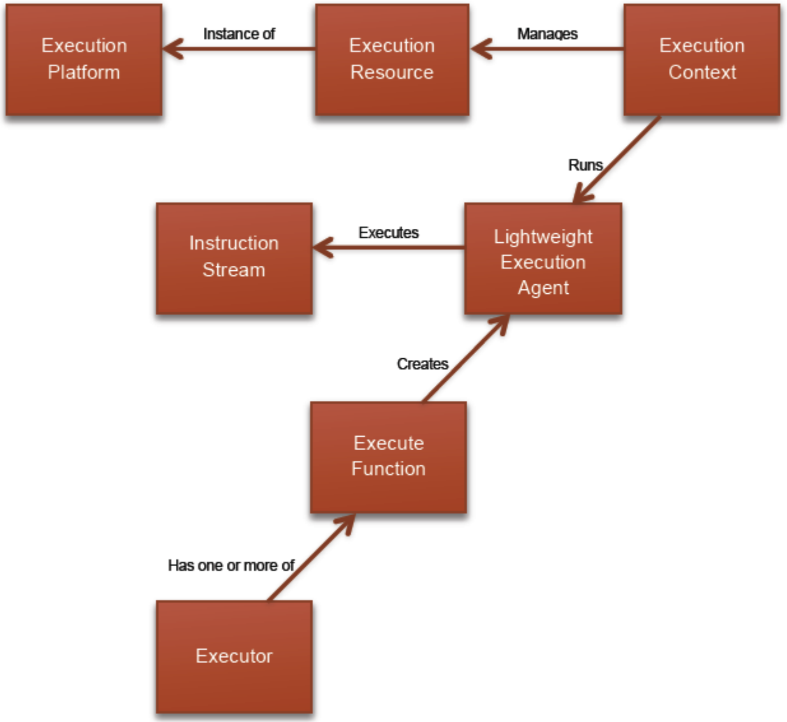
Diverse Execution Resources	Operating System Threads	Thread pool schedulers	OpenMP runtime
	SIMD vector units	GPU runtime	Fibers

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

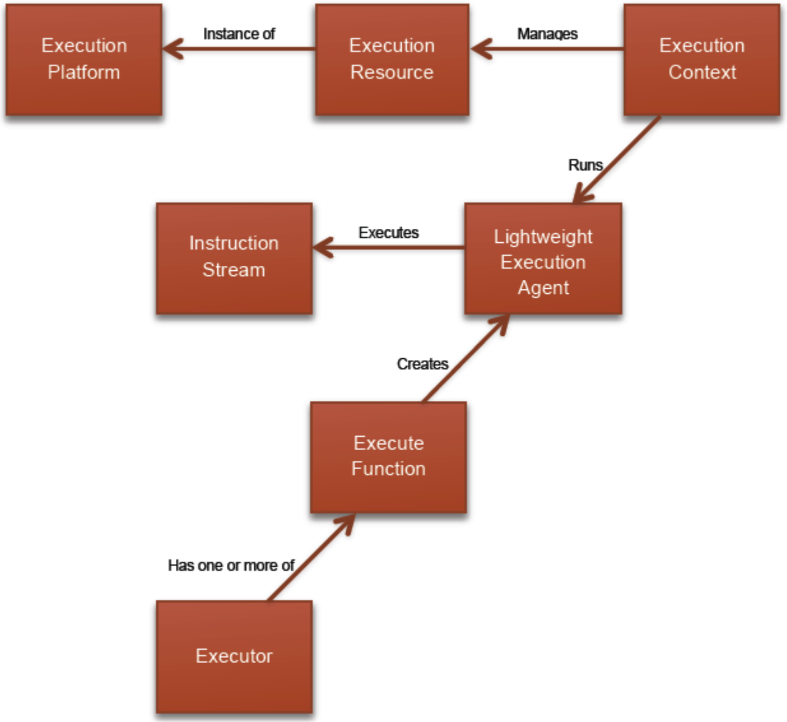
Current Progress of Executors

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



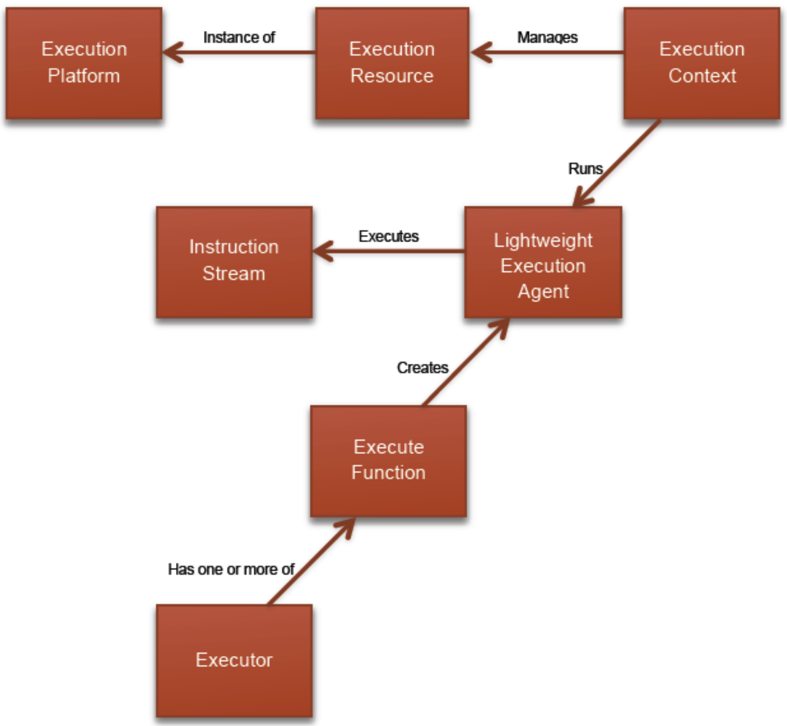
Current Progress of Executors

- 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



Executors: Bifurcation

- Bifurcation of one-way vs two-way
  - One-way –does not return anything
  - Two-way –returns a future type
- Bifurcation of blocking vs non-blocking (WIP)
  - May block –the calling thread may block forward progress until the execution is complete
  - Always block –the calling thread always blocks forward progress until the execution is complete
  - Never block –the calling thread never blocks forward progress.
- Bifurcation of hosted vs remote
  - Hosted –Execution is performed within threads of the device which the execution is launched from, minimum of parallel forward progress guarantee between threads
  - Remote –Execution is performed within threads of another remote device, minimum

Features of C++ Executors

- One-way non-blocking single execute executors
- One-way non-blocking bulk execute executors
- Remote executors with weakly parallel forward progress guarantees
- Top down relationship between execution context and executor
- Reference counting semantics in executors
- A minimal execution resource which supports bulk execute
- Nested execution contexts and executors
- Executors block on destruction

Executor Framework: Abstract Platform details of execution.

Create execution agents

Manage data they share

Advertise semantics

Mediate dependencies

```
class sample_executor
{
public:
  using execution_category = ...;
  using shape_type = tuple<size_t,size_t>;
  template<class T> using future = ...;
  template<class T> future<T>
    make_ready_future(T&& value);
 template<class Function, class Factory1,
    class Factory2>  future<...>
    bulk_async_execute(Function f,
    shape_type shape, Factory1
    result_factory, Factory2
    shared_factory);...
}
```

Purpose of executors:where/how execution

- Placement is, by default, at discretion of the system.

```
for_each(par, I.begin(), I.end(), [](int i) { y[i] += a*x[i]; });
```

- If the Programmer want to control placement:

```
auto exec1 = choose_some_executor();
auto exec2 = choose_another_executor();

for_each(par.on(exec1), I.begin(), I.end(), ...);
for_each(par.on(exec2), I.begin(), I.end(), ...);
```

Control relationship with Calling threads

- async (launch_flag, function)
- async (executor, function)

Executor Interface:semantic types exposed by executors

Type	Meaning
execution_category	Scheduling semantics amongst agents in a task. (sequenced, vector-parallel, parallel, concurrent)
shape_type	Type for indexing bulk launch of agents. (typically n-dimensional integer indices)
future<T>	Type for synchronizing asynchronous activities. (follows interface of **std::future)**

Executor Interface:core constructs for launching work

Type of agent tasks	Constructs

Single-agent tasks	result sync_execute(Function f);
	future<result> async_execute(Function f);
	future<result> then_execute(Function f, Future& predecessor);

Multi-agent tasks	result bulk_sync_execute(Function f, shape_type shape, Factory result_factory, Factory shared_factory); future<result> bulk_async_execute(Function f, shape_type shape, Factory result_factory, Factory shared_factory); future<result> bulk_then_execute(Function f, Future& predecessor, shape_type shape, Factory result_factory, Factory shared_factory);

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

- Built-in operators
- All usual binary operators are available, for all:
  - datapar<T, N> datapar<U, N>
  - datapar<T, N> U, U datapar<T, N>
- Compound binary operators and unary operators as well
  - datapar<T, N> convertible to datapar<U, N>
  - datapar<T, N>(U) broadcasts the value

- No promotion:
  - datapar<uint8_t>(255) + datapar<uint8_t>(1) == datapar<uint8_t>(0)
- Comparisons and conditionals:
  - ==, !=, <, <=,> and >= perform element-wise comparison return mask<T, N, Abi>
  - if(cond) x = y is written as where(cond, x) = y
  - cond ? x : y is written as if_else(cond, x, y)

SYCL v1.2 release

BOARD OF PROMOTERS

KHRONOS™ GROUP

Over 100 members worldwide

any company is welcome to join

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 …*

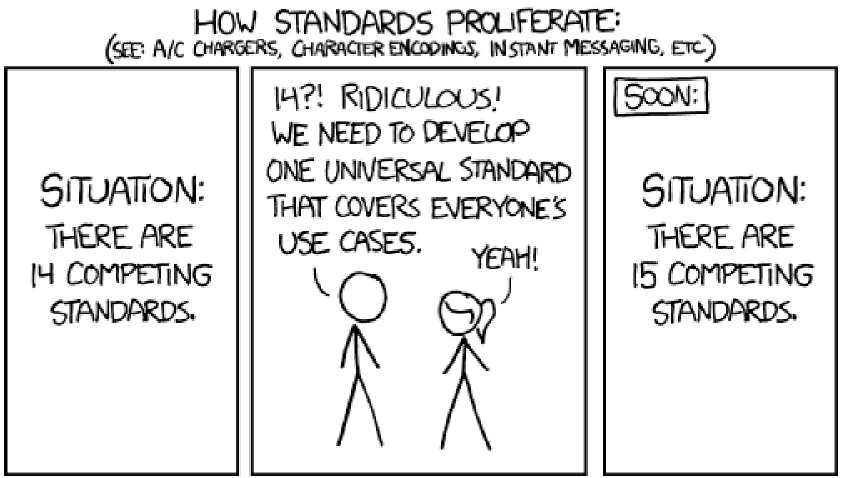
  - And enable the **how** to be: *run fast on an OpenCL device*

- 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…

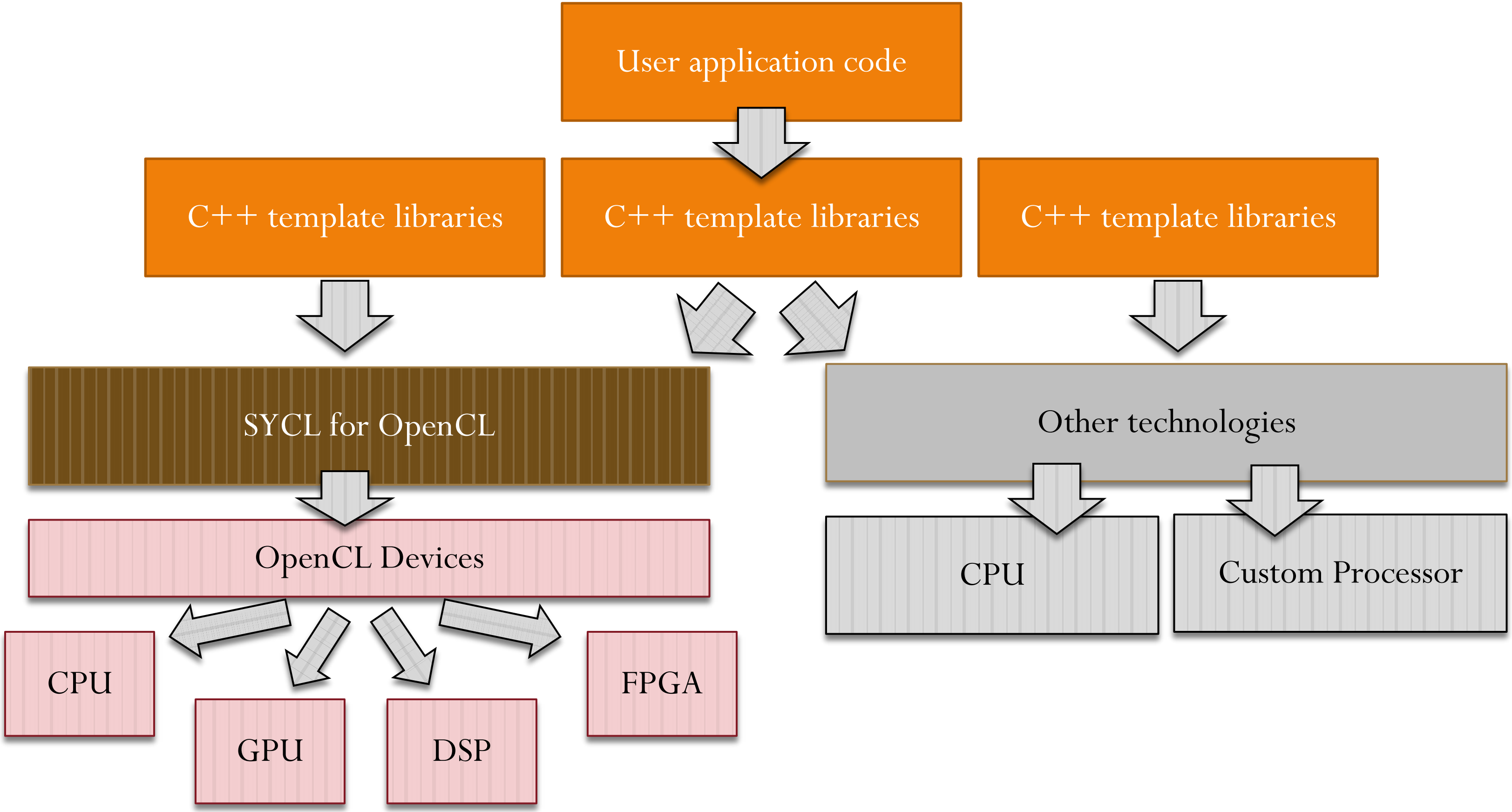
- We want to make it easy to write high-performance OpenCL code in C++
  - SYCL code in C++ must use memory and execute kernels efficiently
  - We must provide developers with all the optimization options they have in OpenCL

- We want to enable OpenCL C code to interoperate with C++ SYCL code
  - Sharing of contexts, memory objects etc

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];
                  })
              );
          });
….
}
```

```cpp
#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];
        });
    });
}
```

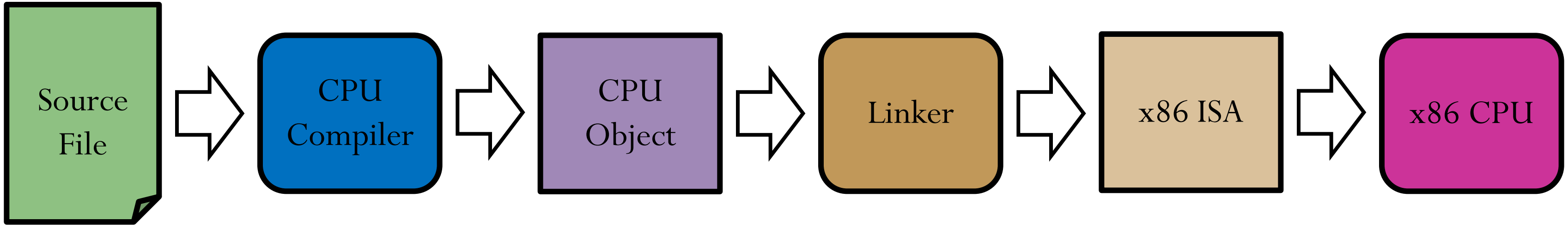
- #include the SYCL header file
- Encapsulate data in SYCL *buffers* which be mapped or copied to or from OpenCL devices
- Create a *queue*, preferably on a GPU, which can execute *kernels*
- Submit to the queue all the work described in the handler lambda that follows
- Create *accessors* which encapsulate 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?

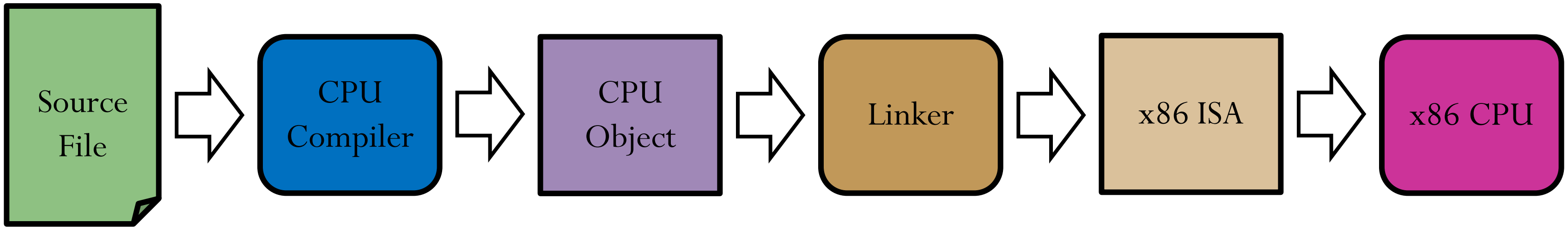
What was our thinking?

How do we offload code to a heterogeneous device?

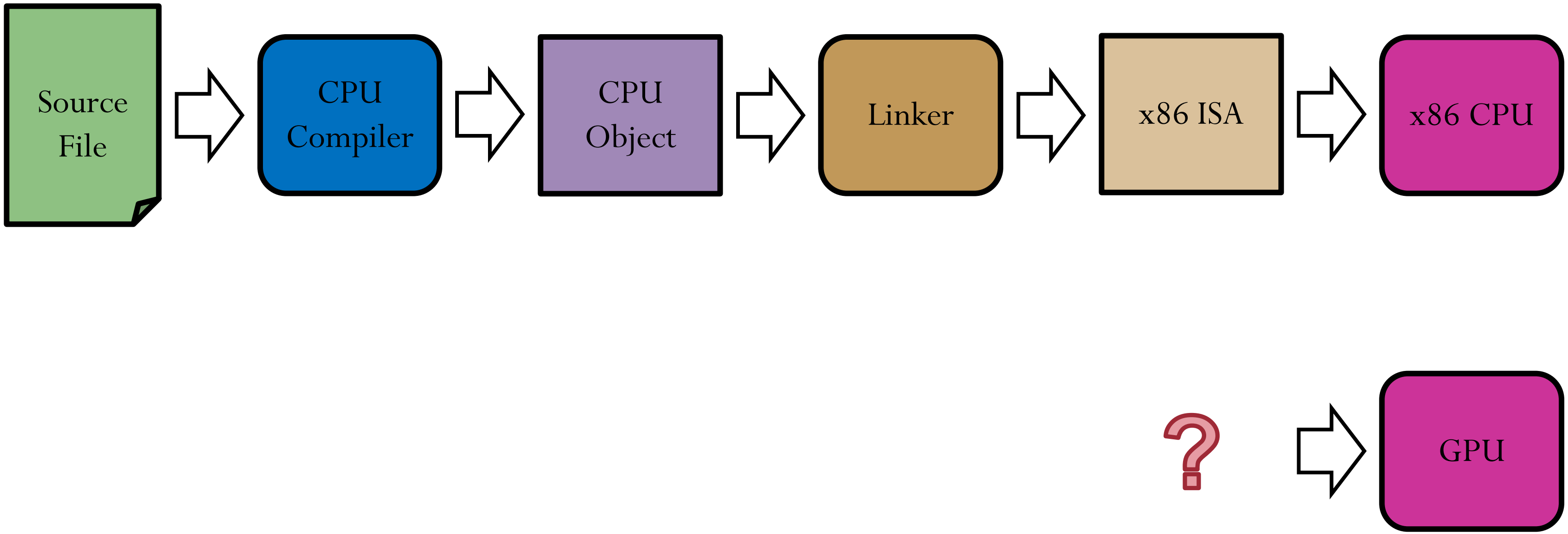
Compilation Model



Compilation Model



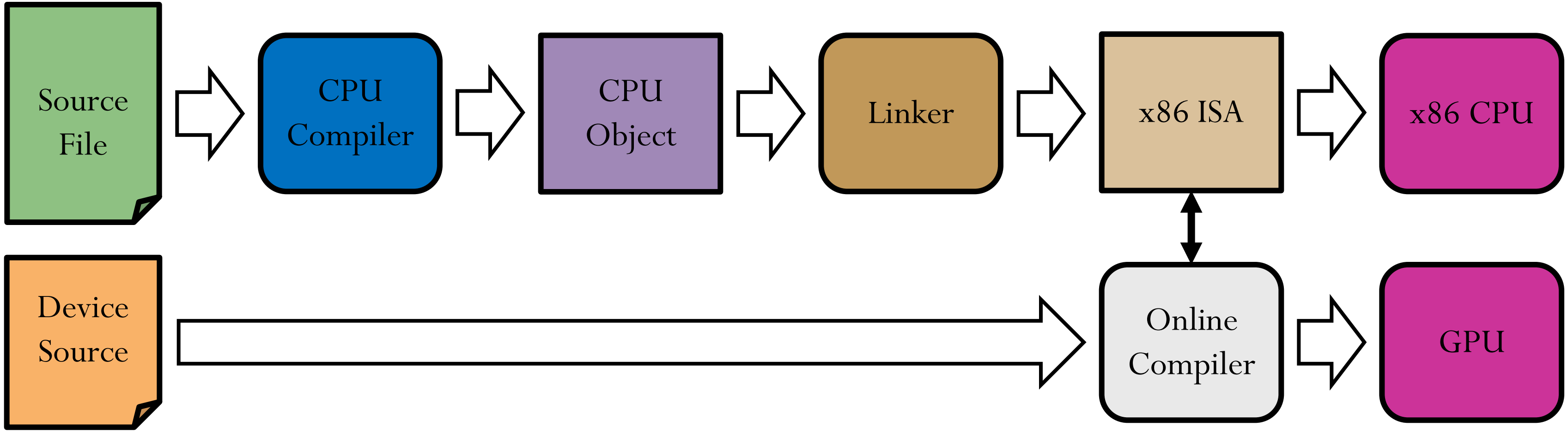
Compilation Model



How can we compile source code for a sub architectures?

- Separate source
- Single source

Separate Source Compilation Model

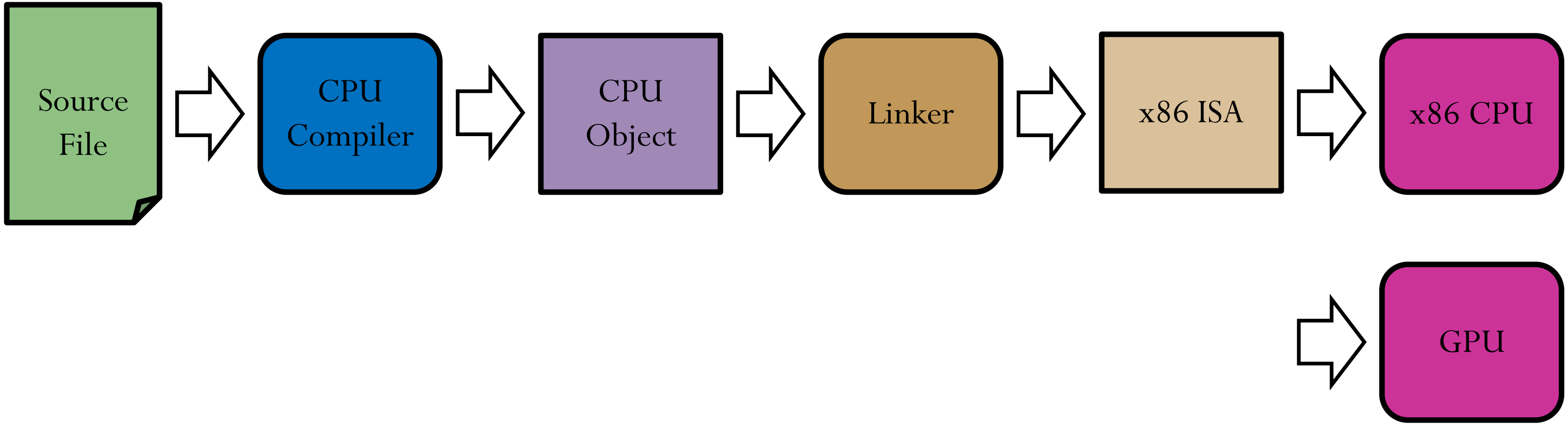



```
float *a, *b, *c;
…
kernel k = clCreateKernel(…, “my_kernel”,
clEnqueueWriteBuffer(…, size, a, …);
clEnqueueWriteBuffer(…, size, a, …);
clEnqueueNDRange(…, k, 1, {size, 1, 1}, …)
clEnqueueWriteBuffer(…, size, c, …);
```

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];
}
```

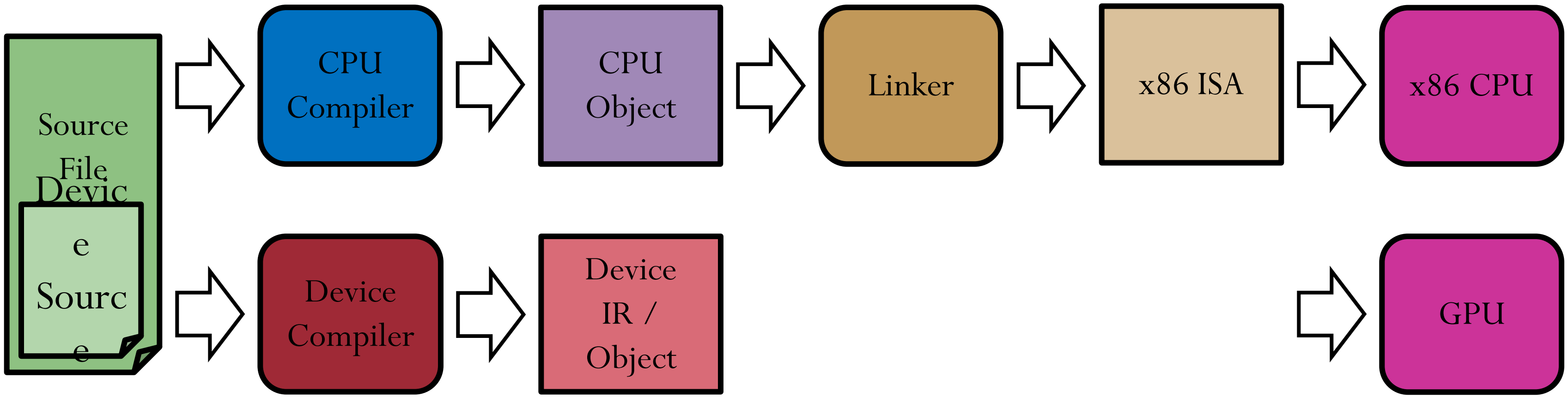
Single Source Compilation Model




```
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

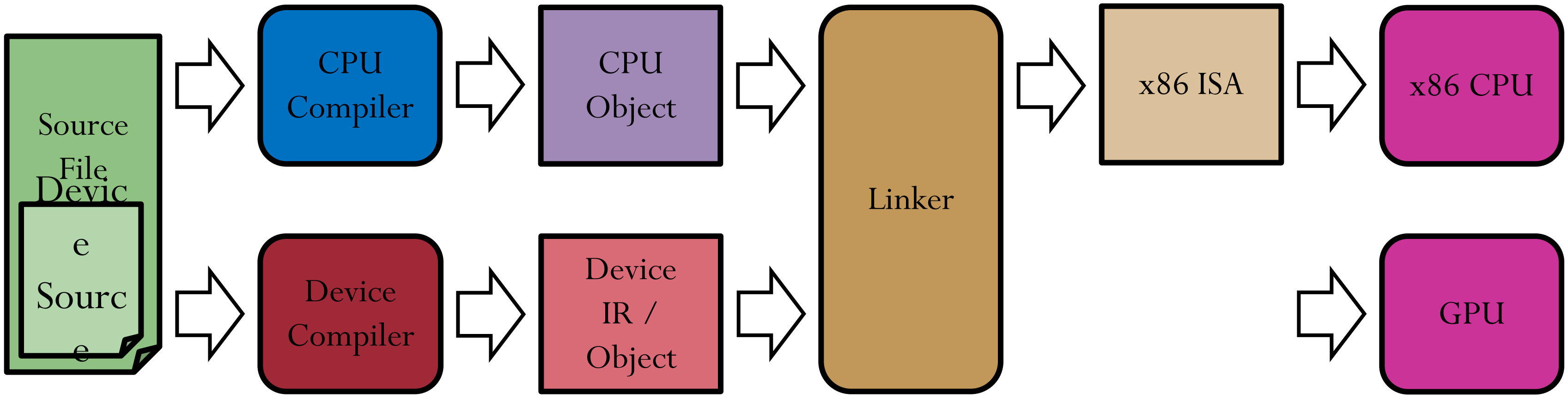
Single Source Compilation Model




```
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

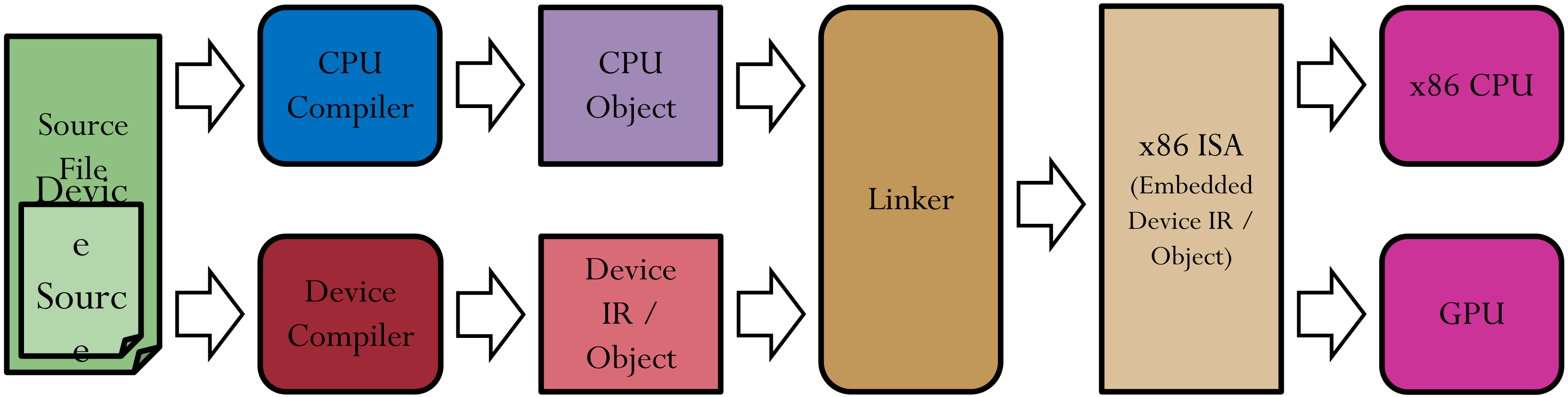
Single Source Compilation Model




```
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

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];
}
```

Examples:

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

Implementation:

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

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

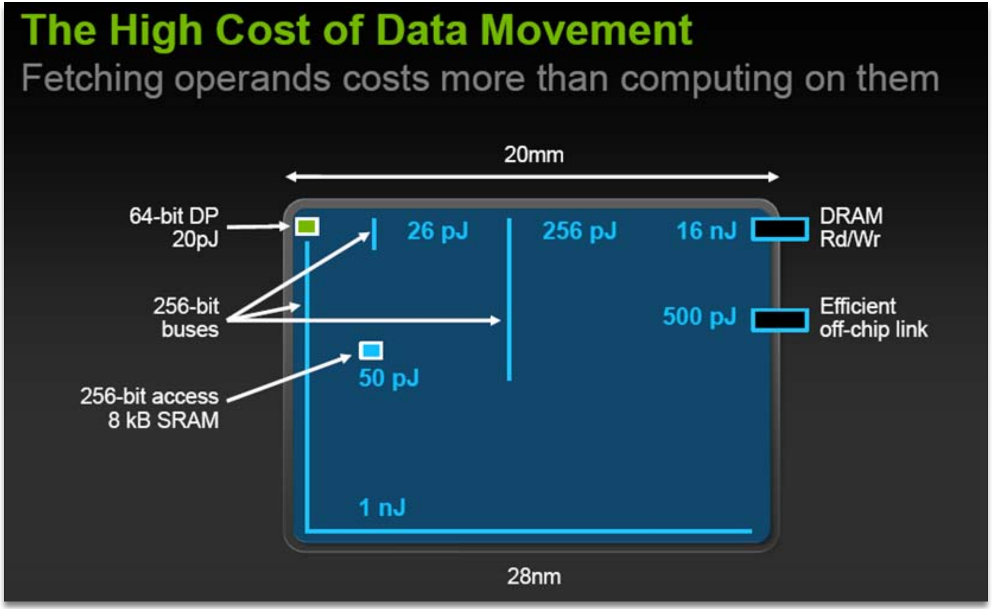
One of the biggest limiting factor in heterogeneous computing

- 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

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;
});
```

Examples:

- OpenCL, CUDA, OpenMP

Implementation:

- Data is moved to the device via explicit copy APIs

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

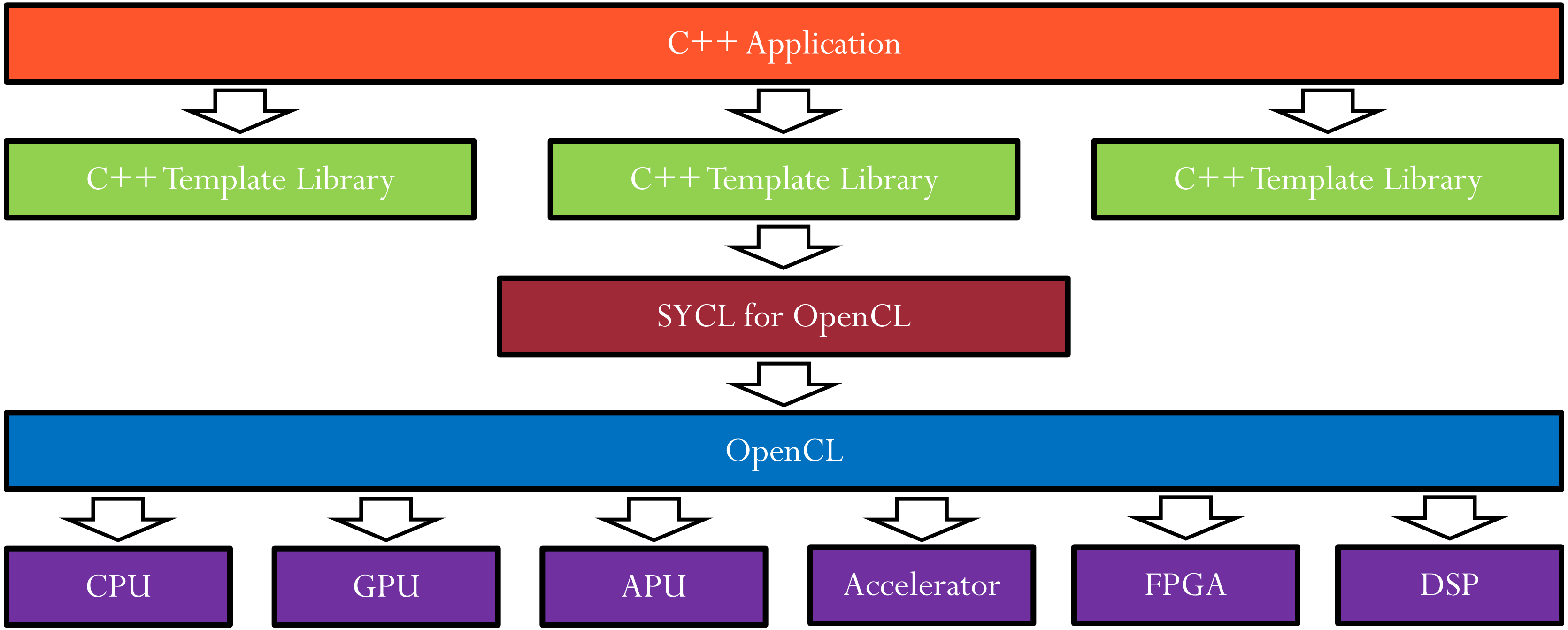
- Multiple address space
  - SYCL 1.2, C++AMP, OpenCL 1.x, CUDA
  - Pointers have keywords or structures for representing different address spaces
  - Allows finer control over where data is stored, but needs to be defined explicitly
- Non-coherent single address space
  - SYCL 2.2, HSA, OpenCL 2.x , CUDA 4, 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

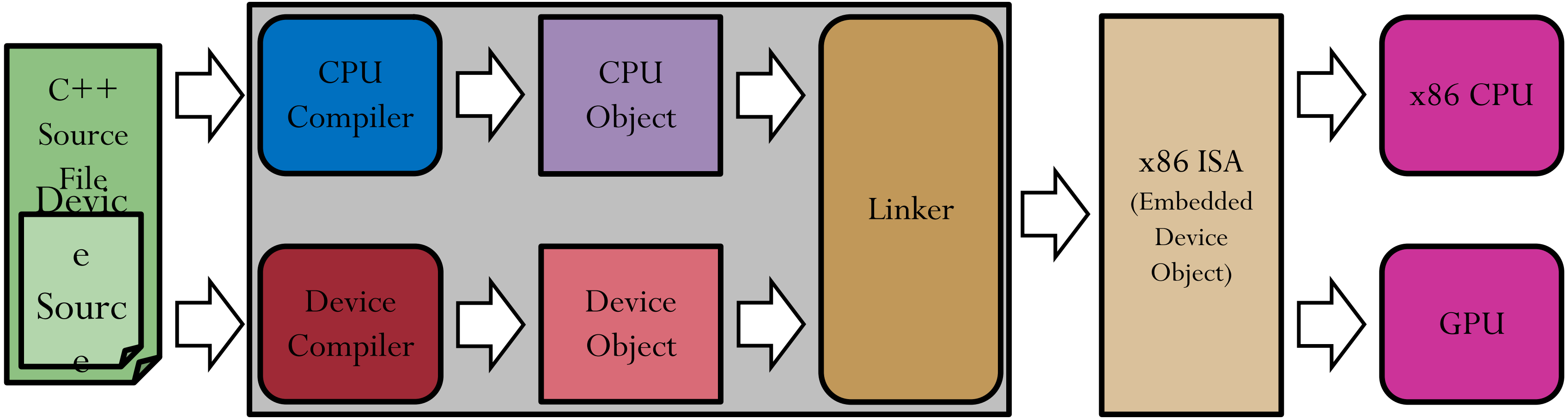
The SYCL Ecosystem



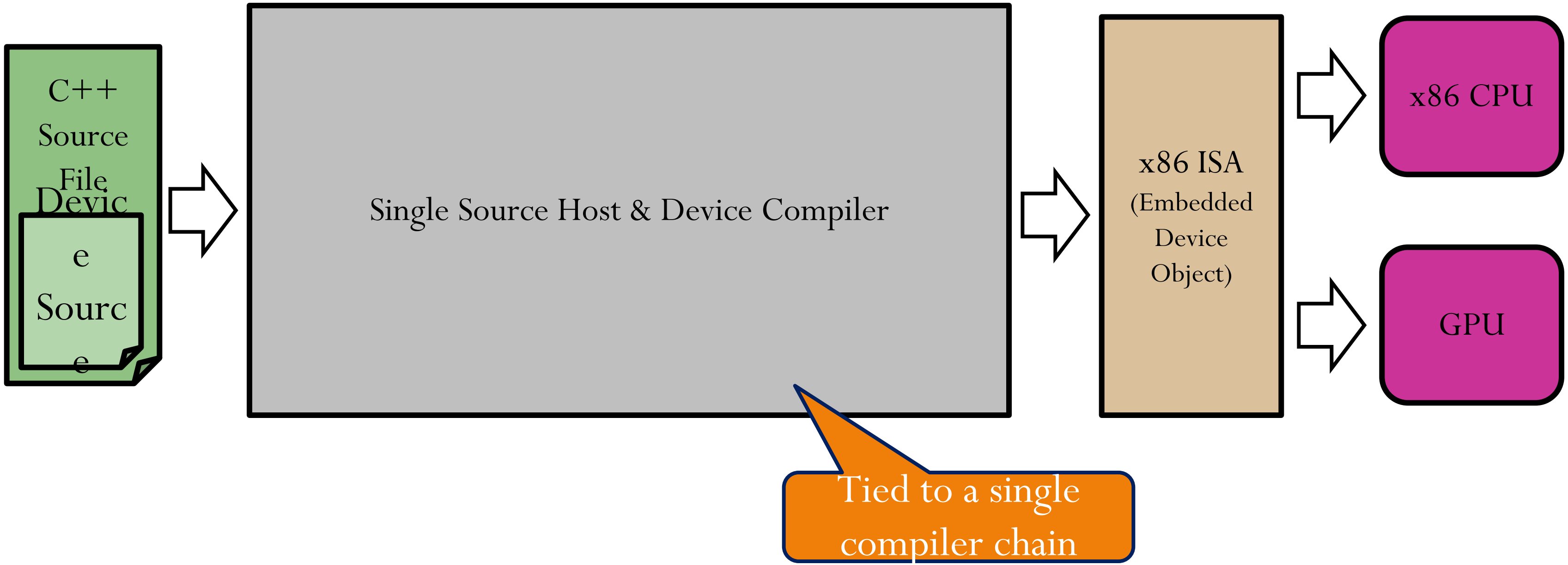
How does SYCL improve heterogeneous offload and performance portability?

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

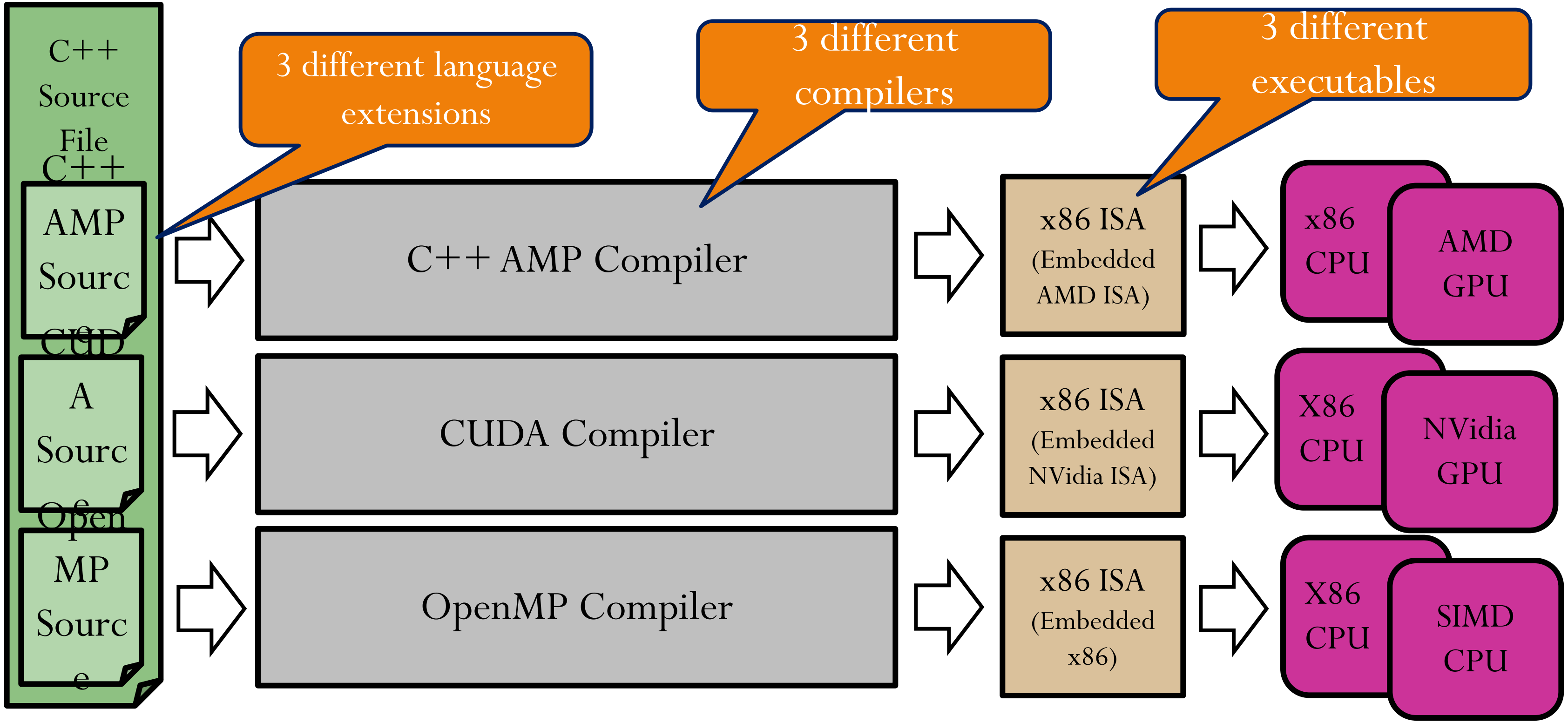
Single Compilation Model



Single Compilation Model



Single Compilation Model



SYCL is Entirely Standard C++

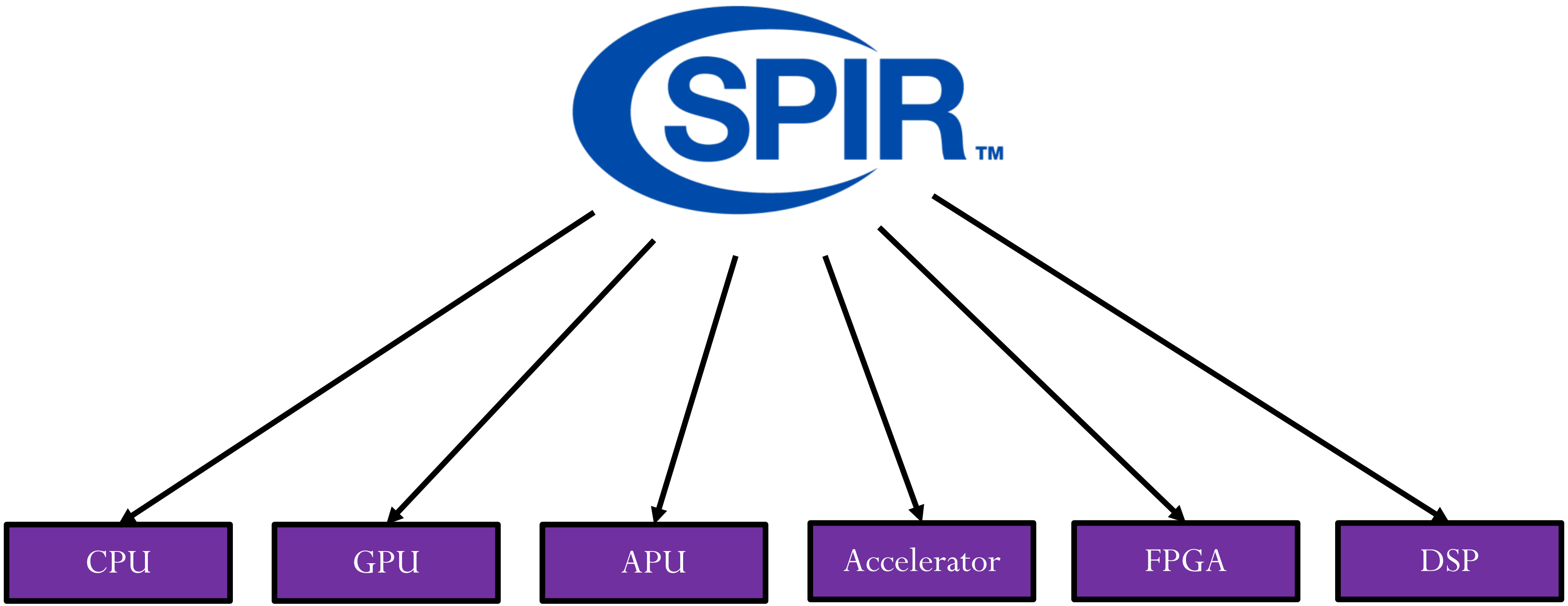
```
__global__ vec_add(float *a, float *b, float *c) {
  return c[i] = a[i] + b[i];
}

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

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

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

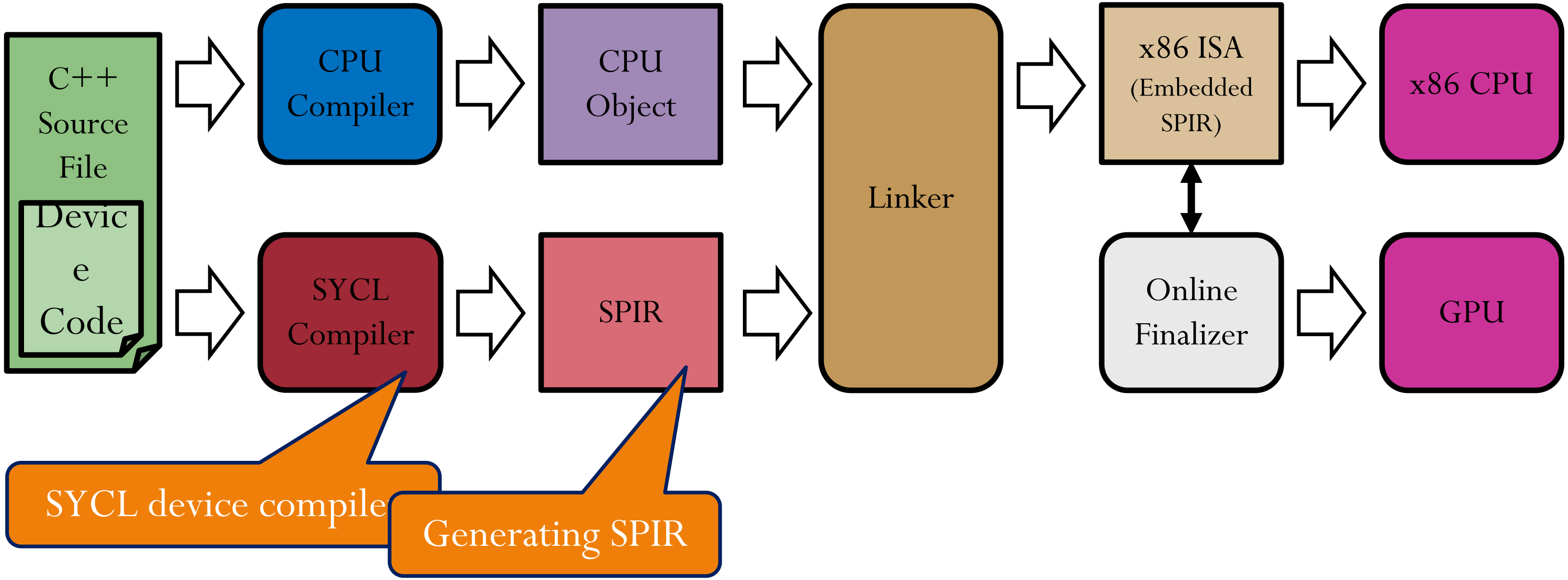
```
array_view<float> a, b, c;
extent<2> e(64, 64);

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];
}));
```

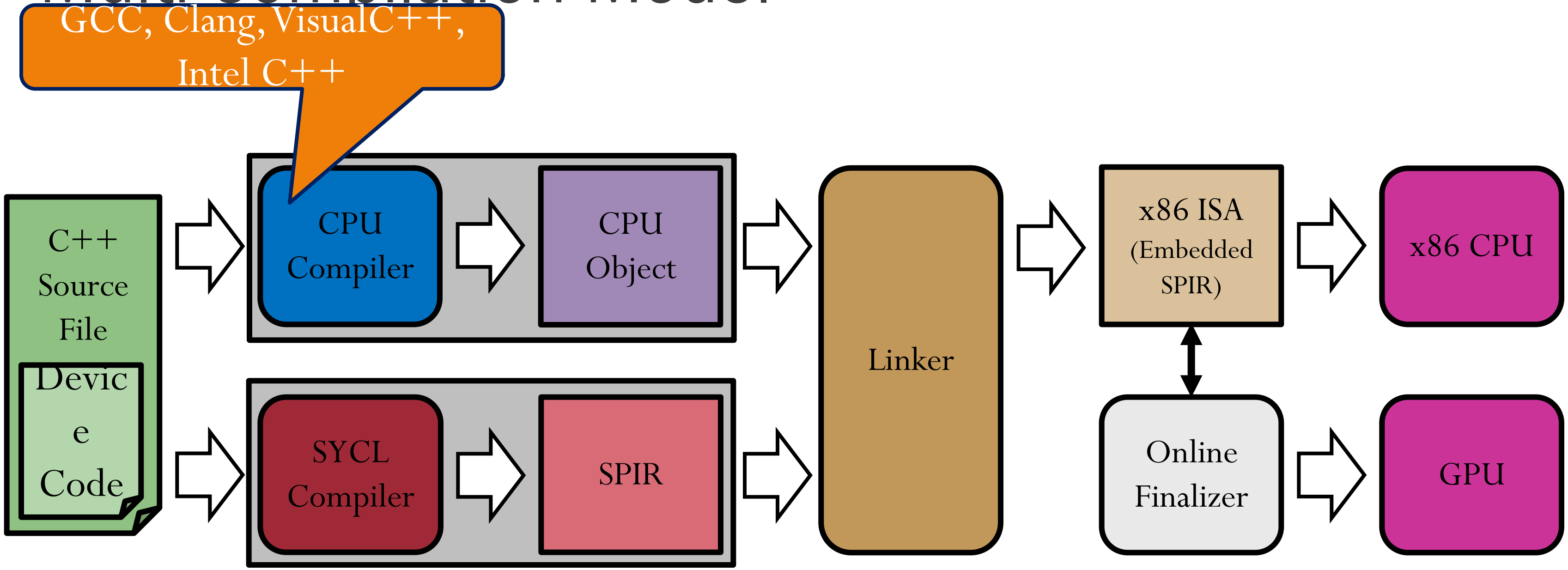
SYCL Targets a Wide Range of Devices with SPIR



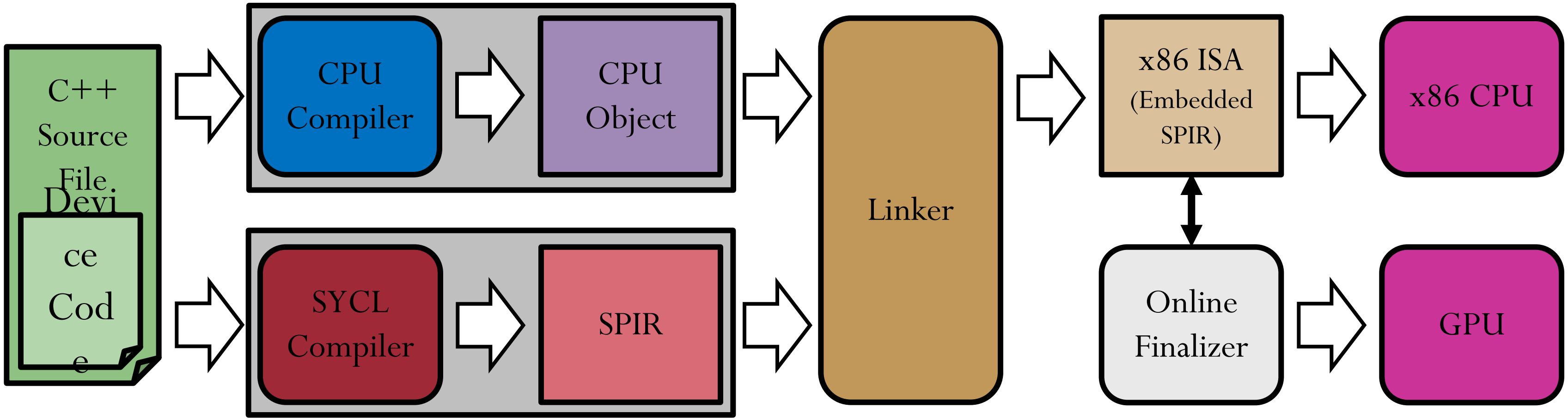
Multi Compilation Model



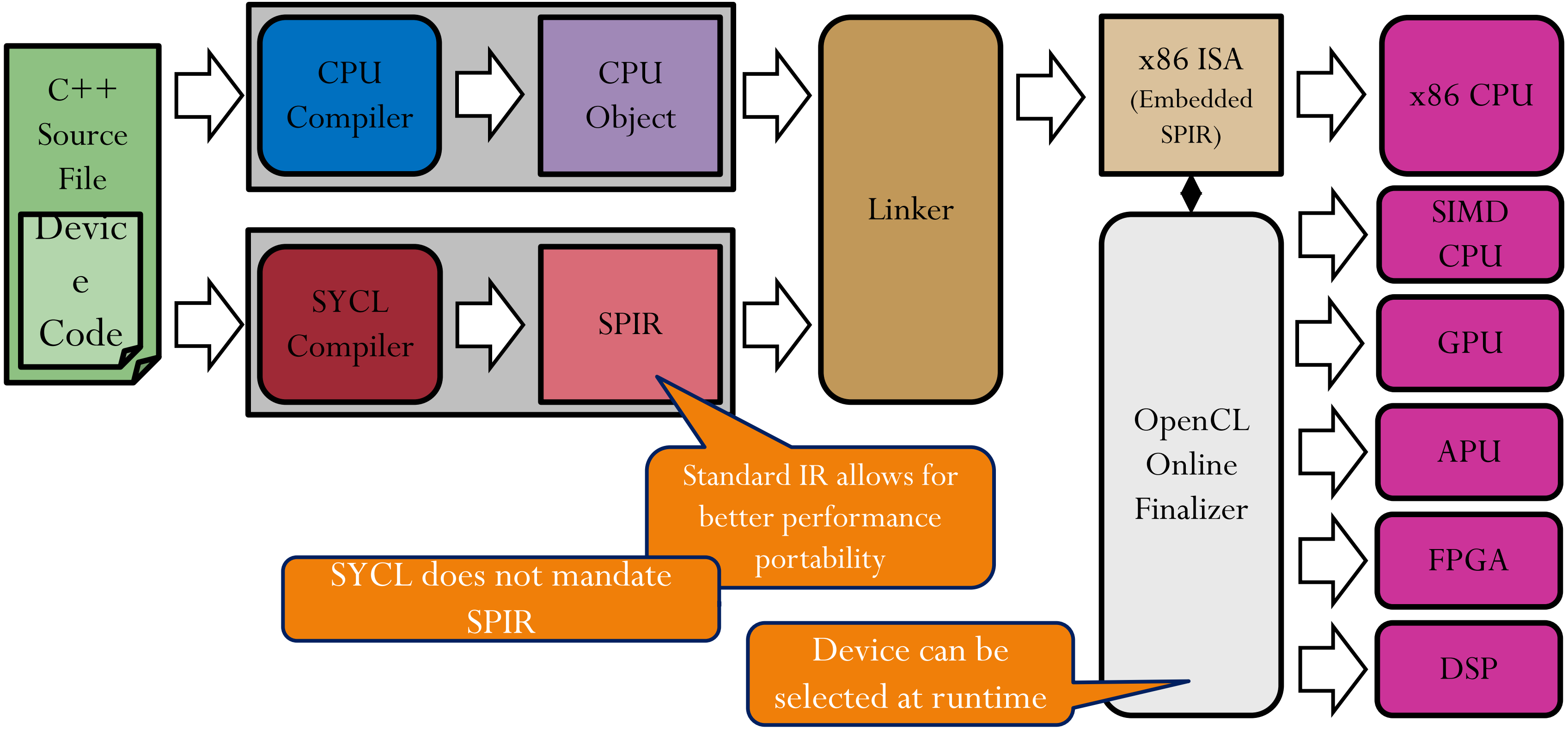
Multi Compilation Model



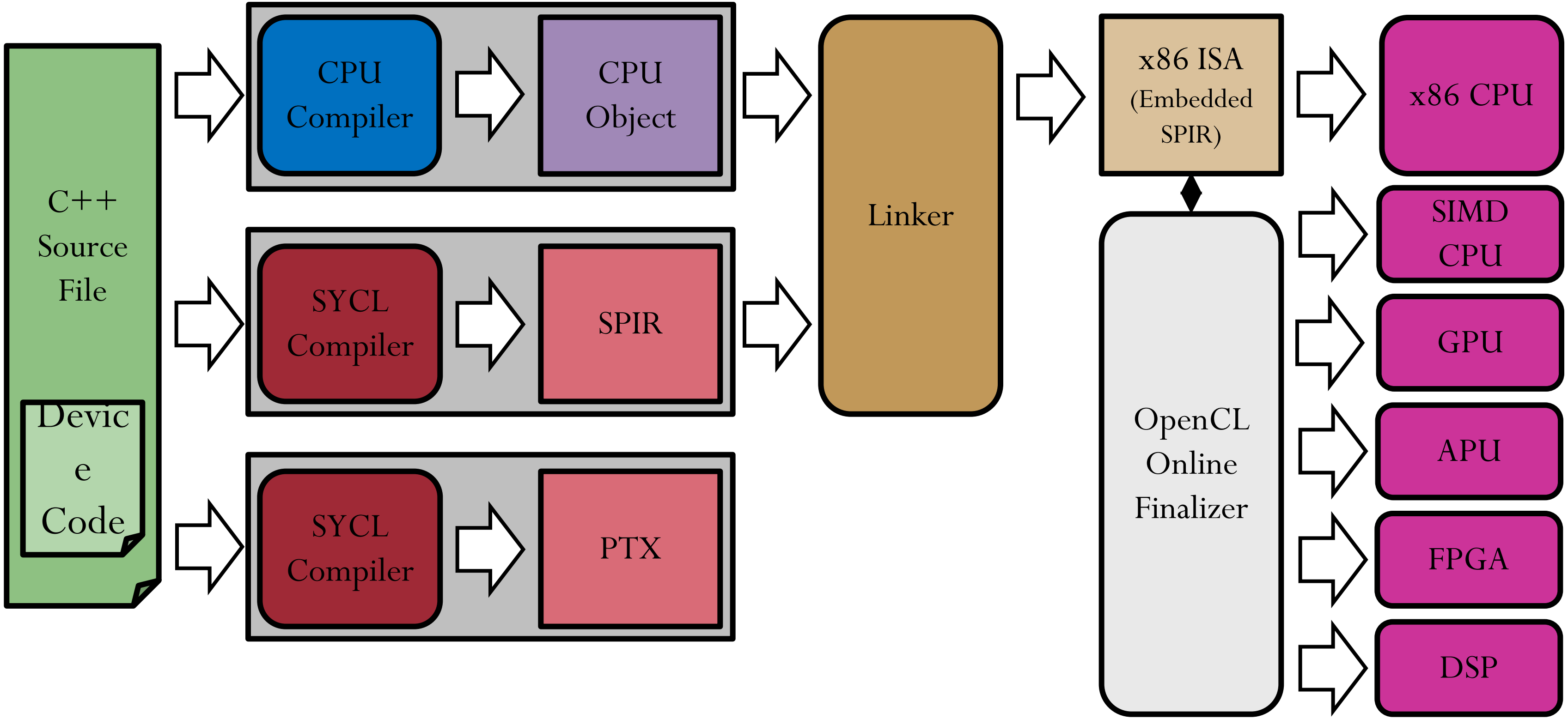
Multi Compilation Model



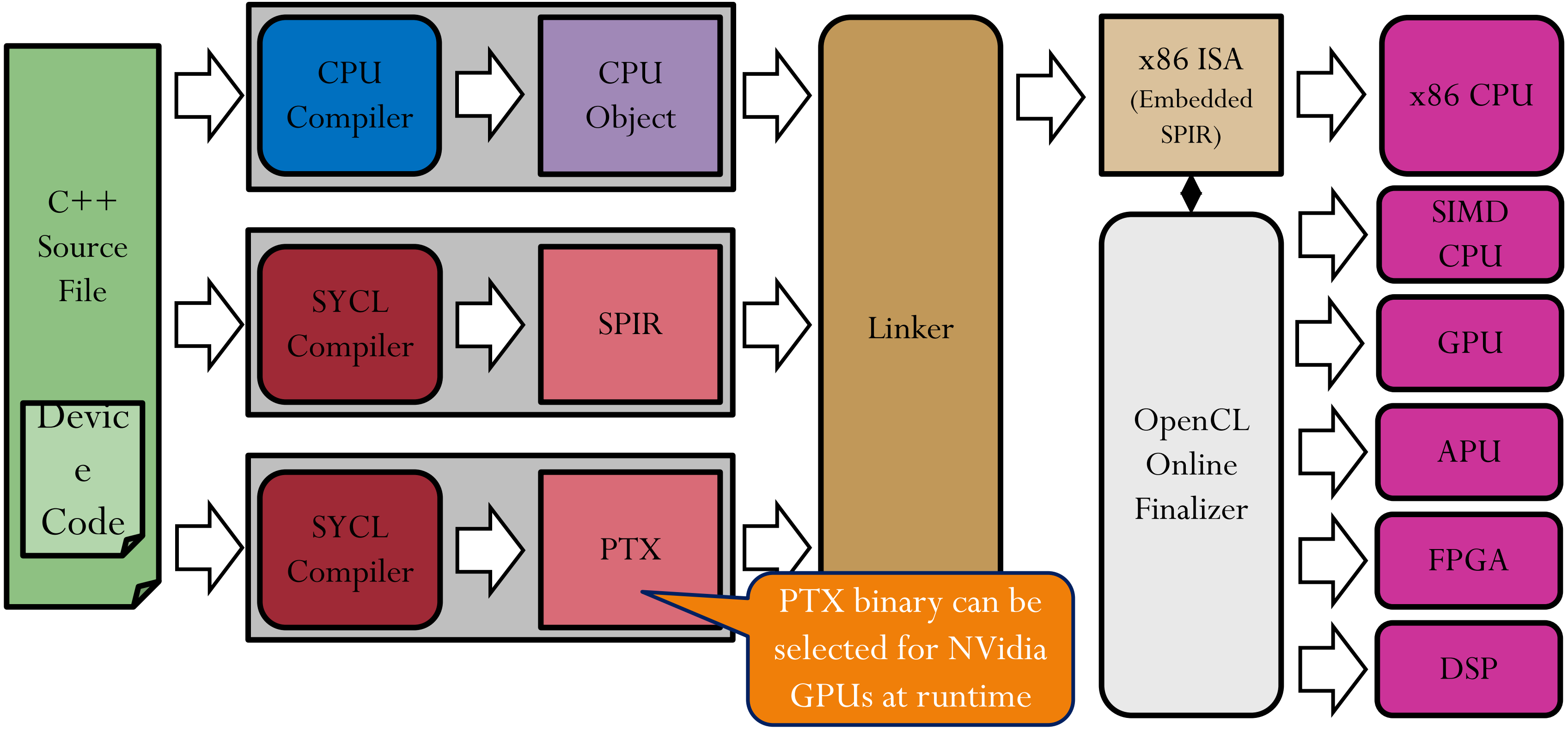
Multi Compilation Model



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

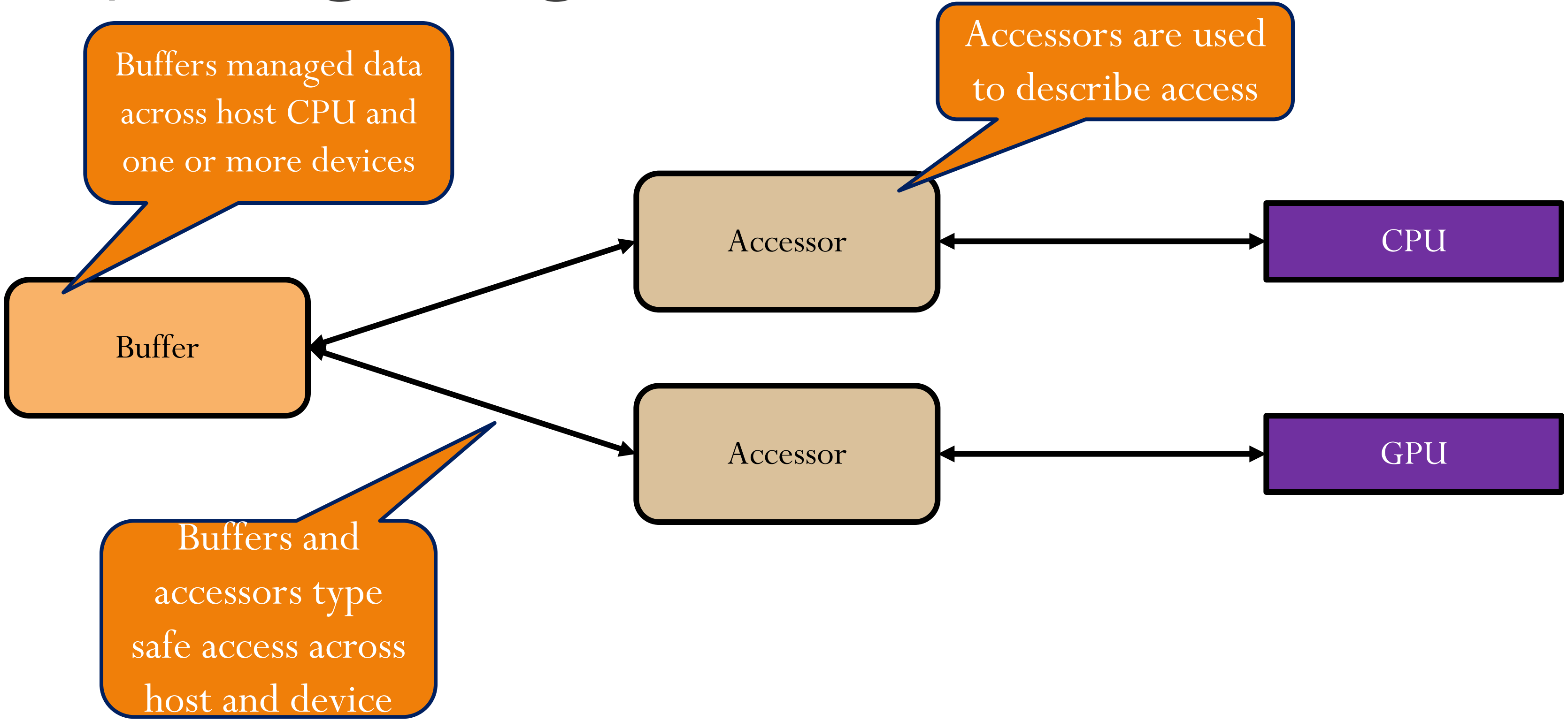
```
cgh.single_task([=](){

  /* task parallel task executed once*/

});
```

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

  /* data parallel task executed across a range */

});
```

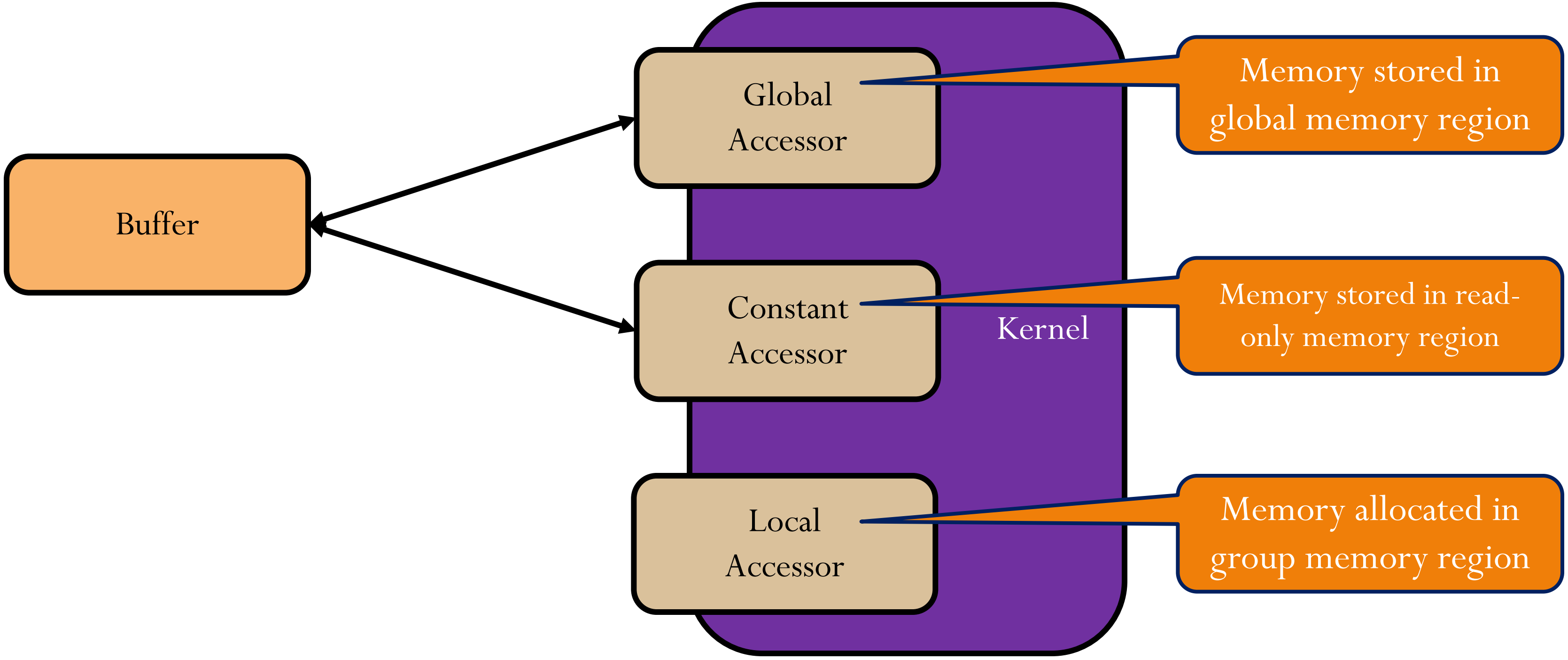
How does SYCL make data movement more efficient?

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

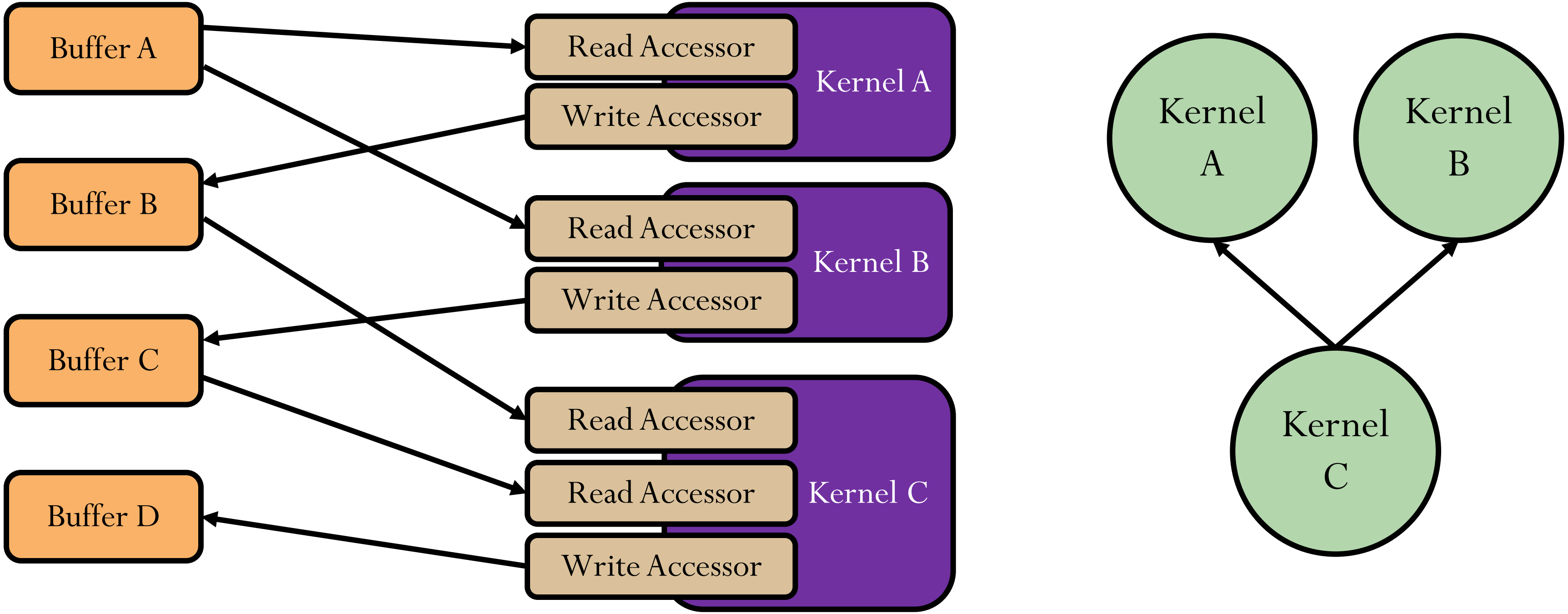
Separating Storage & Access



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

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

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

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

- How do we get to programming self-driving cars?
- 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:

Ease of use

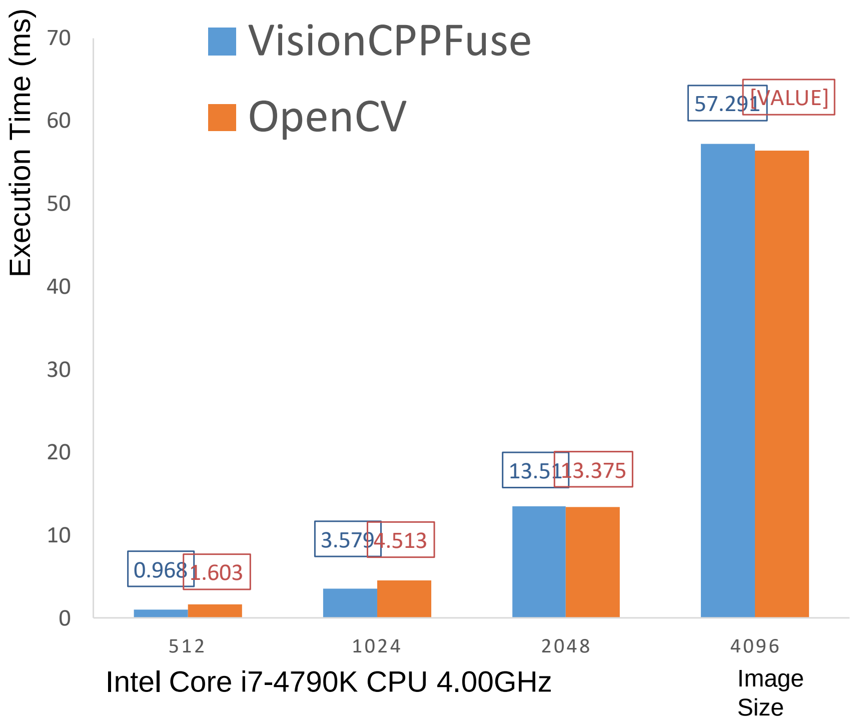
- Applications
  - Easy to write code
  - Unified front end to client code (API)
- Easily able to add customisable operations
- Compile time graph validation
- Predictable memory usage

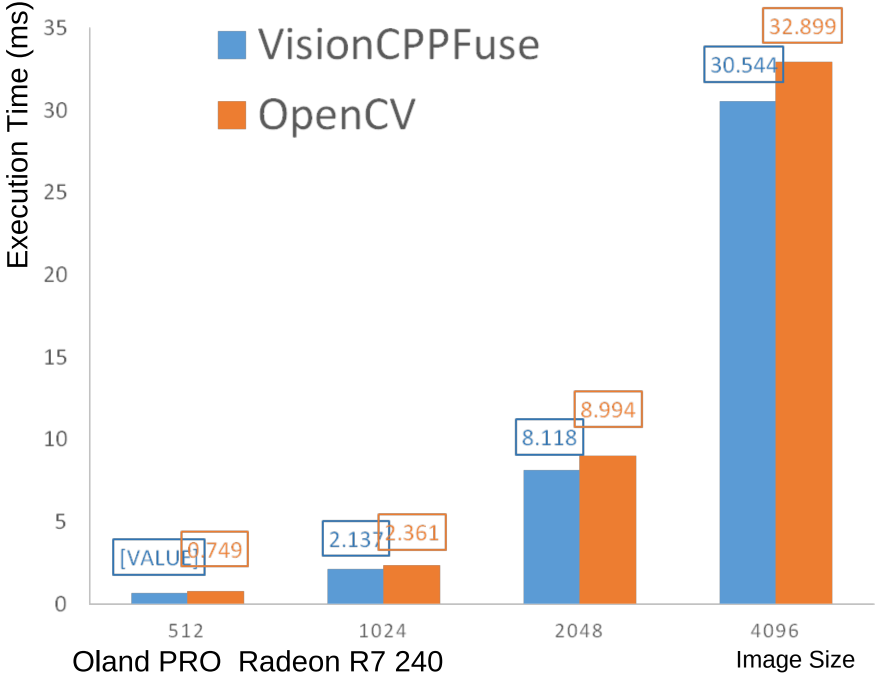
Cross-platform portability

- Separation of concern
  - Portable between different programming models and architectures
- No modification in application computation
  - OpenCL enabled devices
  - CPU

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 ;
}
```

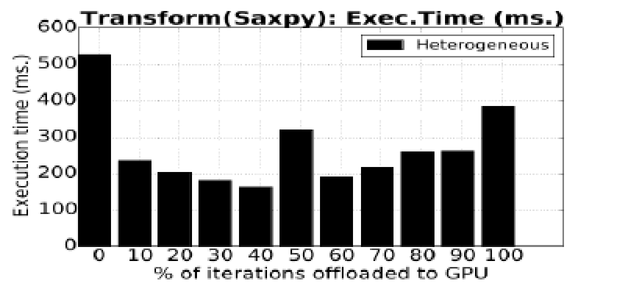
```
template <typename KernelName = DefaultKernelName >
class sycl_execution_policy {
public :
    using kernelName = KernelName ;
    sycl_execution_policy () = default ;
    sycl_execution_policy (cl :: sycl :: queue q);
    cl :: sycl :: queue get_queue () const ;
};
```

- 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 workload
  - → 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 programming has been supported through OpenCL for years
- C++ is a prominent language for doing this but currently is only CPU-based
- Graph programming languages enables Neural network, machine vision
- SYCL allows you to program heterogeneous devices with standard C++ today
- ComputeCpp is available now for you to download and experiment
  - For engineers/companies/consortiums producing embedded devices for automotive ADAS, machine vision, or neural network
  - Who want to deliver artificial intelligent devices that are also low power for e.g. self-driving cars, smart homes
  - But who are dissatisfied with the current single vendor locked in heterogeneous solution or design/code with no reuse solution
  - We provide performance-portable open-standard software across multiple platforms with long-term support
  - Unlike vertical locked in solutions such as CUDA, C++AMP, or HCC
  - We have assembled a whole ecosystem of software accelerated by your parallel hardware enabling reuse with open standards

Agenda

- How do we get to programming self-driving cars?
- 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

- 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

Recommendations for the authors

- We encourage you to use this template, although it is not mandatory
- If you use your own template, please insert the CEE-SECR logo anywhere on the title page. Just copy it from here:

2016
CEE-SEC(R)

- Consider the “Death by PowerPoint” recommendations: http://www.slideshare.net/thecroaker/death-by-powerpoint