Comparing SYCL with HPX, Kokkos, Raja and C++ Executors
The future of ISO C++ Heterogeneous Computing

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

SC2016

Agenda

- Heterogensous Computing for ISO C++
- SYCL
- HPX (slides thanks to Hartmut Kaiser)
- Kokkos (slides thanks to Carter Edwards, Christian Trott)
- Raja (Slides thanks to David Beckingsale)
- State of C++ Concurrency and Parallelism
- C++ Executors coming in C++20
- C++ simd/vector coming in C++20

The goal for C++

- Great support for cpu latency computations through concurrency TS-
- Great support for cpu throughput through parallelism TS
- Great support for Heterogeneous throughput computation in future

Many alternatives for Massive dispatch/heterogeneous

- Programming Languages
- OpenGL
- DirectX
- CUDA
- OpenCL
- OpenMP
- OpenACC
- C++ AMP
- HPX
- HSA
- SYCL
- Vulkan

Usage experience

- OpenMP/OpenACC
- HPC
- SYCL
- OpenCL
- CUDA

Not that far away from a Grand Unified Theory

- GUT is achievable
- What we have is only missing 20% of where we want to be
- It is just not designed with an integrated view in mind … Yet
- Need more focus direction on each proposal for GUT, whatever that is, and add a few elements

What we want for Massive dispatch/Heterogeneous computing by 2020

- Integrated approach for 2020 for C++
  - Marries concurrency/parallelism TS/co-routines
- Heterogeneous Devices and/or just Massive Parallelism
- Works for both HPC, consumer, games, embedded, fpga
- Make asynchrony the core concept
- Supports integrated (APU), but also discrete memory models
- Supports High bandwidth memory
- Support distributed architecture

Better candidates

- Goal: Use standard C++ to express all intra-node parallelism

1. Khronos' OpenCL SYCL
2. Agency extends Parallelism TS
3. HCC
4. HPX extends parallelism and concurrency TS
5. C++ AMP
6. KoKKos
7. Raja

- All exclusively C++
- All use C++11 lambda
- All use some form of execution policy to separate concerns
- All have some form of dimension shape for range of data
- All are aiming to be subsumed/integrated into future C++ Standard, but want to continue future exploratory research

Agenda

- Heterogensous Computing for ISO C++
- SYCL
- HPX
- Kokkos
- Raja
- State of C++ Concurrency and Parallelism
- C++ Executors coming in C++20
- C++ simd/vector coming in C++20

SYCL

Explicit parallelism

Queuing model

Task and data-based parallelism

Uses OpenCL/SPIRV to dispatch to multiple devices

Designed to dispatch to all accelerators

Implements Parallelism TS

Rich Ecosystem supporting Eigen/Tensorflow, Neural network, Machine Vision

- Implicit Data Movement using Dependency Call Graph enables efficient synchronization and data transfer

Most interesting contribution:

Separate Memory Storage and Data access model using Accessors and Storage Buffers using Depedency graph

Multiple Compilation Single Source model

Example SYCL Code

```
#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

Agenda

- Heterogensous Computing for ISO C++
- SYCL
- HPX (slides thanks to Hartmut Kaiser)
- Kokkos (slides thanks to Carter Edwards, Christian Trott)
- Raja (Slides thanks to David Beckingsale)
- State of C++ Concurrency and Parallelism
- C++ Executors coming in C++20
- C++ simd/vector coming in C++20

HPX

C++ Standard Library based closed on C++ Parallelism and Concurrency TS

Explicit Parallelism, work-queuing model, message driven computation using tasked based, fork-join, parallel algorithms and Asynchronous Continuation

Has Execution Policies that describe Where, and Sequencing, as well as Grainsize

Extends with Dataflow dependencies

Designed for both Distributed and now Accelerators

- Data Movement is Explicit but can handle implicit with Dataflow Dependencies
- Data Placement using
  - hpx::vector<T, Alloc>
  - hpx::partitioned_vector<T>
  - std::allocator_traits

Most Interesting Contribution:
Distributed Computing nodes with asynchroous algorithm execvution
Execution Policy with executors (Par.on executors) and Grainsize

HPX Examples

Parallel Algorithms

```
// rebind execution policy
// .on(): executor object, ‘where and when’
// .with(): parameter object(s), possibly executor specific parameters
std::vector<double> d(1000);
parallel::fill(
   par.on(exec).with(par1, par2, ...),
   begin(d), end(d), 0.0);
```

Dot-product: Vectorization

```
std::vector<float> data1 = {...};
std::vector<float> data2 = {...};
inner_product(
   datapar,// parallel and vectorizedexecution
   std::begin(data1), std::end(data1),
   std::begin(data2),
   0.0f,
   [](autot1, autot2) { returnt1 + t2; },// std::plus<>()
   [](autot1, autot2) { returnt1 * t2; } // std::multiplies<>()
);
```

Agenda

- Heterogensous Computing for ISO C++
- SYCL
- HPX (slides thanks to Hartmut Kaiser)
- Kokkos (slides thanks to Carter Edwards, Christian Trott)
- Raja (Slides thanks to David Beckingsale)
- State of C++ Concurrency and Parallelism
- C++ Executors coming in C++20
- C++ simd/vector coming in C++20

Kokkos

Implements Parallel Pattern, execution Policy (how), execution Space (where computation), Memory Space (where data), and Layout (how laid out)

Name stands out to identify Parallel Patterns

Manages inter-thread synchronization and reductions of thread local temporaries

Have Atomic views and floating point Atomics

- Explicit Data movement
- Memory Space control
- Layout space control

Most interesting Contribution:
Memory Space tells where user data resides (Host, GPU, HBM)
Layout Space tells how user data is laid-out (Row/column major, AOS, SOA)

Kokkos examples

Pattern composed with policy drives the computational body

```
for ( int i = 0 ; i < N ; ++i )
{ /* body */ }
```

pattern policy body

```
parallel_for ( N , [=]( int i )
{ /* body */ } );
```

Sparse-Matrix-Vector Multiply

```
parallel_for( TeamPolicy<Space>(nrow,AUTO),
KOKKOS_LAMBDA(
TeamPolicy<Space>::member_type member ) {
  const int i = member.league_rank();
  double result = 0 ;
  parallel_reduce(
    TeamThreadRange(member,irow[i],irow[i+1]),
    [&]( int j , double & tmp) { tmp += A[j] * x[jcol[j]];},
    result );
  if ( member.team_rank() == 0 ) y[i] = result ;
});
```

Agenda

- Heterogensous Computing for ISO C++
- SYCL
- HPX (slides thanks to Hartmut Kaiser)
- Kokkos (slides thanks to Carter Edwards, Christian Trott)
- Raja (Slides thanks to David Beckingsale)
- State of C++ Concurrency and Parallelism
- C++ Executors coming in C++20
- C++ simd/vector coming in C++20

Raja

Has execution policy covering CUDA, OpenMP, sequential

IndexSet spacing partitions index space into segments that cover the sapce;

Segemnts is a container of iterators that can be executed in parallel, Serial or depdendency relationships

Focus on Execute loop iterations, handle hardware & Programming model details while application select iteration patterns and execution policies

- Implicit Data Movement
- Handled with extension projects
  - CHAI data movement abstraction layer
  - Nested Loops & loop interchange

Most Interesting contribution:
Execution Policy
IndexSet and Segments

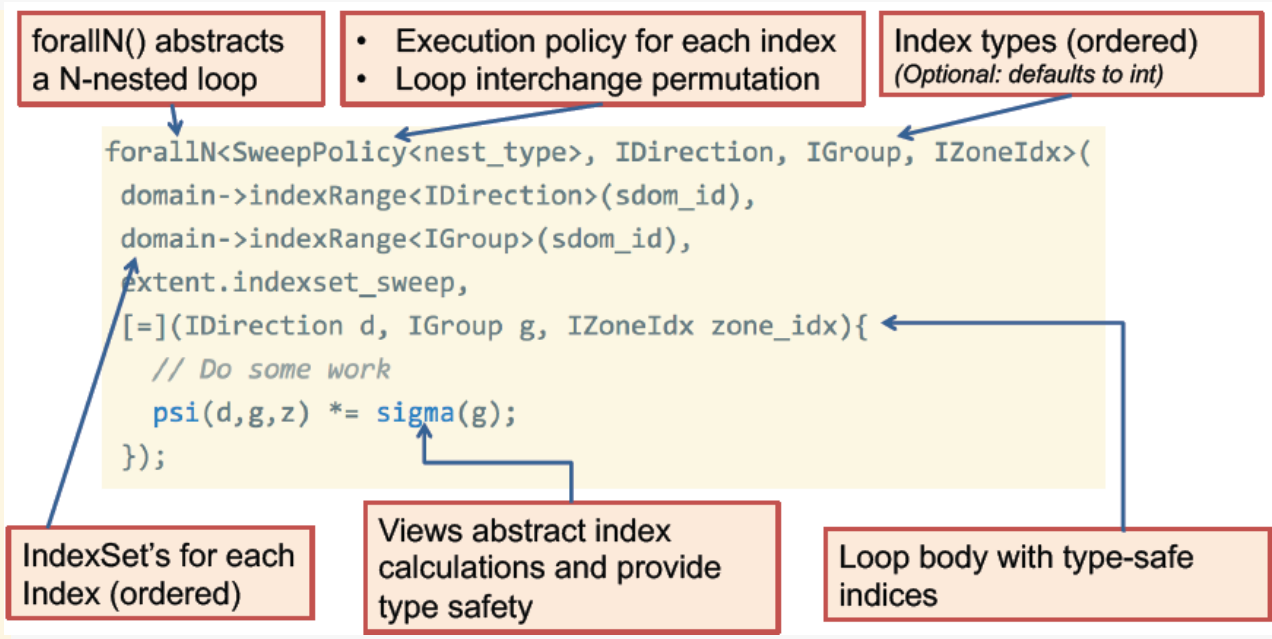
Raja examples

Execution policies control loop scheduling & execution

```
double* x ; double* y ;
double a ;
RAJA::SumReduction<reduce_policy, double> tsum(0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);

RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
     y[i] += a * x[i] ;
     tsum += y[i];
     tmin.min( y[i] );
});
```

forallN is a RAJA extension for nested-loops



Agenda

- Heterogensous Computing for ISO C++
- SYCL
- HPX
- Kokkos
- Raja
- State of C++ Concurrency and Parallelism
- C++ Executors coming in C++20
- C++ simd/vector coming in C++20

C++1Y(1Y=17/20/22) SG1/SG5/SG14 Plan red=C++17, blue=C++20? Black=future?

Parallelism

- Parallel Algorithms:
- Library Vector Types
- Vector loop algorithm/exec policy
- Task-based parallelism (cilk, OpenMP, fork-join)
- Execution Agents
- Progress guarantees
- MapReduce

Concurrency

- TS1
  - Future++ (then, wait_any, wait_all):
  - Latches and Barriers
  - Atomic smart pointers
- TS2
  - osync_stream
  - Atomic Views, fp_atomics,
  - Counters/Queues
  - Lock free techniques
  - Synchronics replacement/atomic flags
- Executors TS 1
- Transactional Memory TS 1
- Co-routines TS 1
- Concurrent Vector/Unordered Associative Containers
- upgrade_lock
- Pipelines/channels

Agenda

- Heterogensous Computing for ISO C++
- SYCL
- HPX
- Kokkos
- Raja
- State of C++ Concurrency and Parallelism
- C++ Executors coming in C++20
- C++ simd/vector coming in 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

Telecon calls between Oulu to Issaquah meeting

Jared Hoberock (thanks for the slides!)

Michael Garland

Chris Kohlhoff

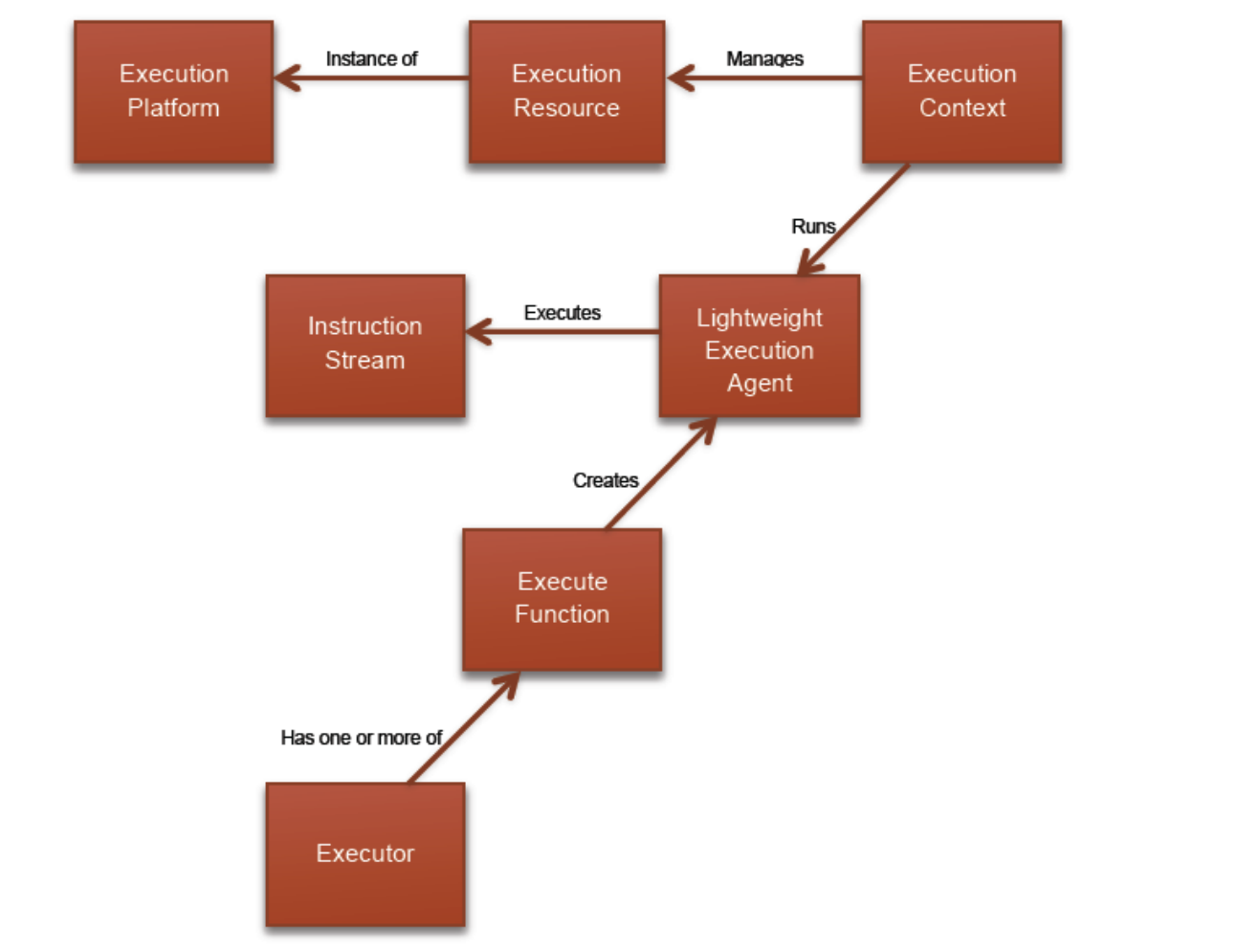
Chris Mysen

Carter Edwards

- Hans Boehm
- Gordon Brown
- Thomas Heller
- Lee Howes
- Hartmut Kaiser
- Bryce Lelbach
- Gor Nishanov
- Thomas Rodgers
- Michael Wong

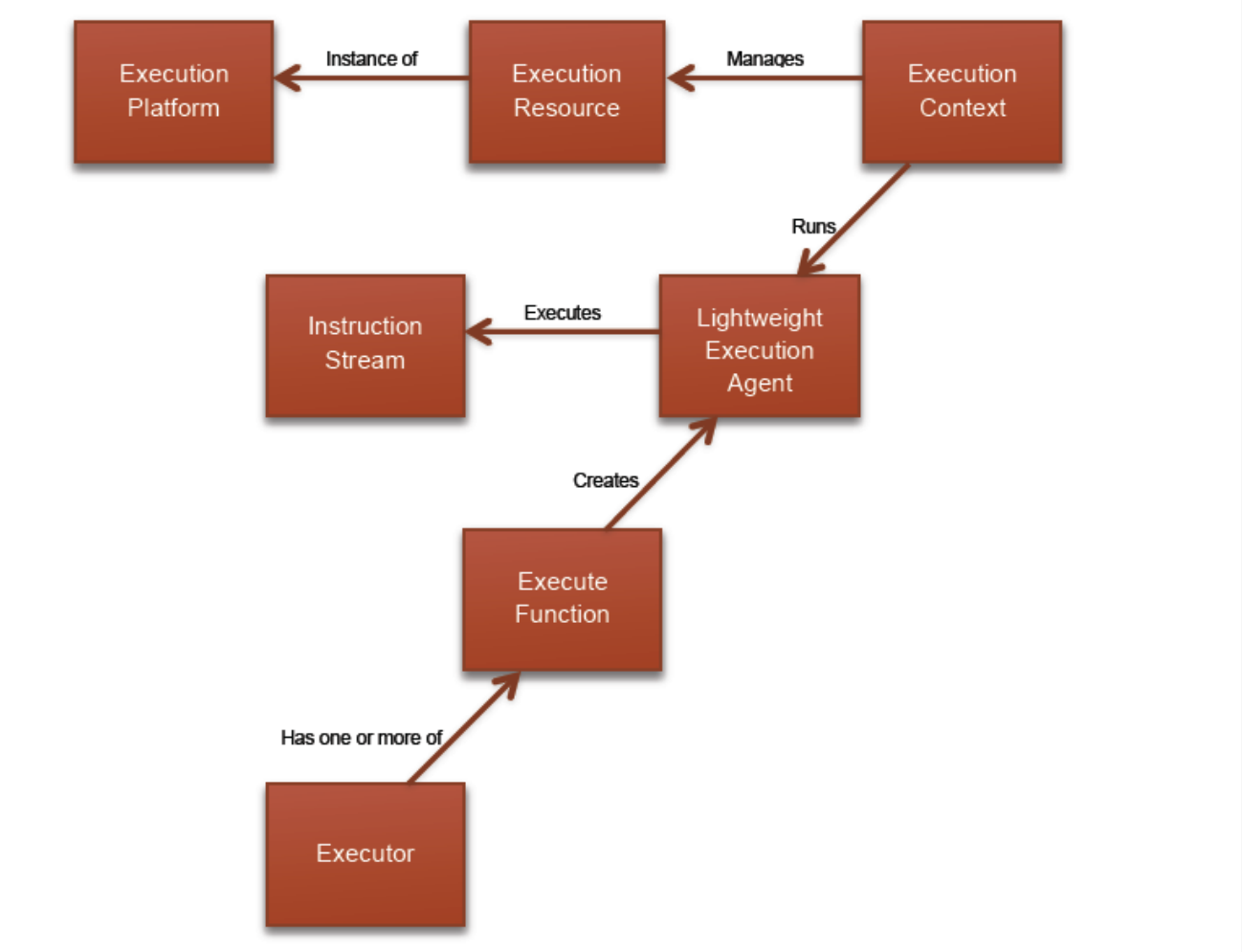
Current Progress of Executors

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



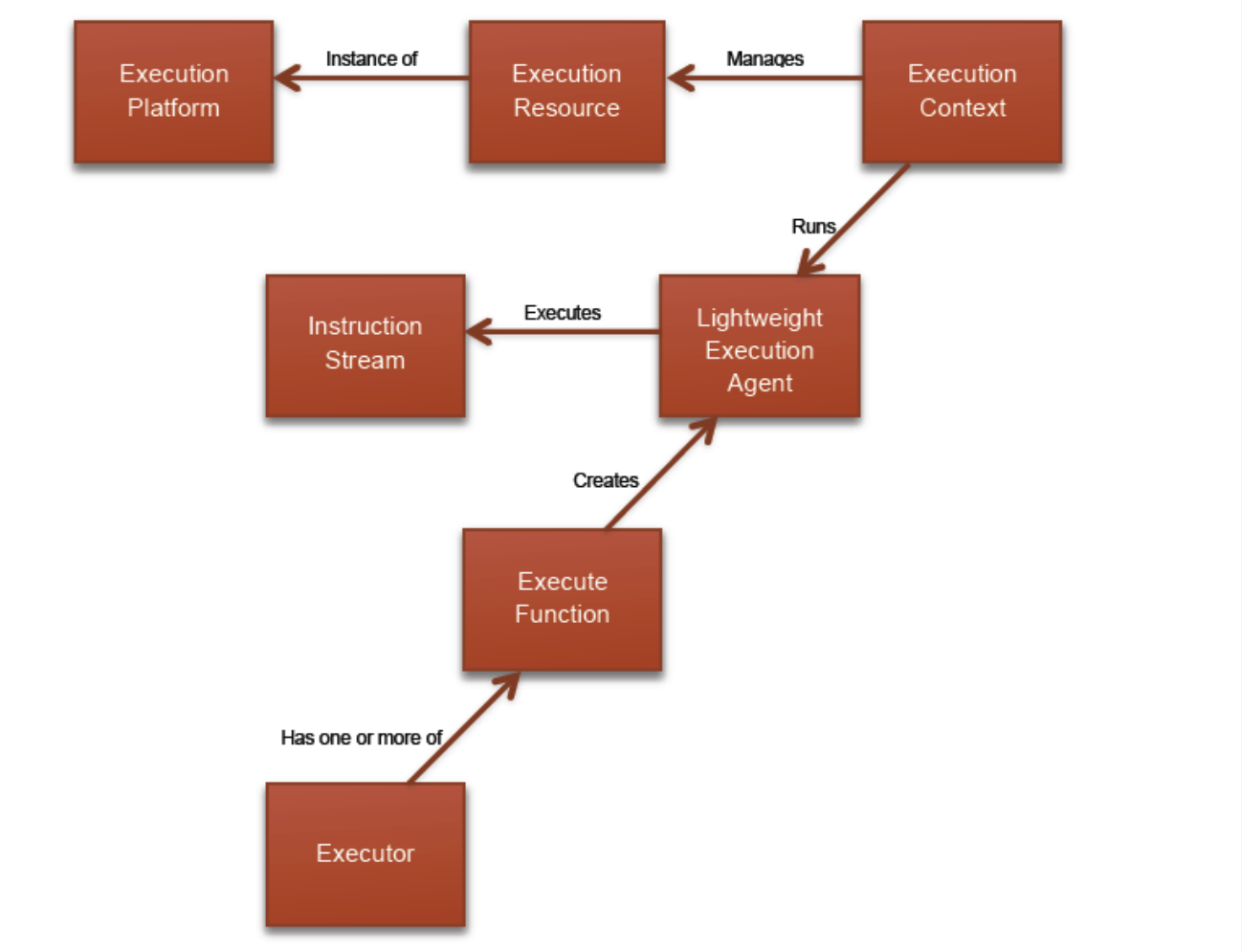
Current Progress of Executors

- 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 1 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(), ...);
```

Purpose 2 of executors

- Control relationship with Calling threads
  - `async(launch_flags, function);`
  - `async(executor, function);`

Purpose 3 of executors

- Uniform interface for scheduling semantics across control structures
  - `for_each(P.on(executor), ...);`
  - `async(executor, ...);`
  - `future.then(executor, ...);`
  - `dispatch(executor, ...);`

SHORT TERM GOALS

- Compose with existing control structures
  - In C++17:
    - `async()`, `invoke()`, `for_each()`, `sort()`, `...`
  - In technical specifications:
    - `define_task_block()`, `future.then()`, Networking TS, asynchronous operations, Transactional memory

UNIFIED DESIGN

- Distinguish **executors** **from** **execution contexts**
- **Categorize** **executors**
- Enable **customization**
- Describe composition with existing control structures

EXECUTORS & CONTEXTS

Light-weight views on long-lived resources

- Distinguish **executors** **from** **execution contexts**
- **Categorize** **executors**
- Enable **customization**
- Describe composition with existing control structures

EXECUTORS & CONTEXTS

Light-weight views on long-lived resources

- **Executors are (potentially short-lived) objects that create** execution agents on execution contexts.
- **Execution contexts are (potentially long-lived) objects** that manage the lifetime of underlying execution resources.

EXECUTORS & CONTEXTS
Example: simple thread pool

```
struct my_thread_pool
{
   template<class Function>
   void submit(Function&& f);
   struct executor_t
   {
      my_thread_pool& ctx;
      template<class Function>
      void execute(Function&& f) const
      {
         // forward the function to the thread pool
         ctx.submit(std::forward<Function>(f));
      }
      my_thread_pool& context() const noexcept {return ctx;}
      bool operator==(const executor_t& rhs) const noexcept {return ctx == rhs.ctx;}
      bool operator!=(const executor_t& rhs) const noexcept {return ctx != rhs.ctx;}
   };
   executor_t executor() { return executor_t{*this}; }
...
};
```

- Context: `my_thread_pool`
- Executor: `my_thread_pool::executor_t`
- `.execute()` submits a task to the thread pool
- The executor is created by the context

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);**

EXECUTOR CATEGORIES

Name collections of use cases

Each executor operation identifies a unique use case

- `execute(f)` : “fire-and-forget `f`”
- `async_execute(f)` : “asynchronously execute `f` and return a future”

Categorize executor types by the uses cases they natively support

- `OneWayExecutor` : executors that natively fire-and-forget
- `TwoWayExecutor` : executors that natively provide a channel to the result

EXECUTOR CATEGORIES

Name collections of use cases

`HostBased*`

- As if the execution agent is running on a std::thread
- Passes the “database test”
- .execute(f,alloc) : “fire-and-forget f, use alloc for allocation”

`Bulk*`

Create multiple execution agents with a single operation

.bulk_execute(f,n,sf) : “fire-and-forget f n times in bulk”

EXECUTOR CATEGORIES

	OneWay	HostBasedOneWay	NonBlockingOneWay	TwoWay	NonBlockingTwoWay	BulkOneWay	BulkTwoWay
.execute(f) -> void	■	■	■			■	■
.execute(f,alloc) -> void		■	■				
.post(f) -> void			■				
.post(f,alloc) -> void			■				
.defer(f) -> void			■				
.defer(f,alloc) -> void			■				
.async_post(f) -> future					■		
.async_defer(f) -> future					■		
.sync_execute(f) -> result				■	■	■	■
.async_execute(f) -> future				■	■	■	■
.then_execute(f,fut) -> future							
.bulk_execute(f,n,sf) -> void						■	
.bulk_sync_execute(f,n,rf,sf) -> result							■
.bulk_async_execute(f,n,rf,sf) -> future							■
.bulk_then_execute(f,n,fut,rf,sf) -> future							■

CUSTOMIZATION POINTS

Enable uniform use

`Free functions in namespace execution::`

- execute(f) — execution::execute(exec, f)
- async_execute(f) — execution::async_execute(exec, f)

`Adapt exec when operation not natively provided`

CUSTOMIZATION POINTS
Enable uniform use

```
struct has_async_execute
{
template<class F>
future<...> async_execute(F&&) const;
...
};
struct hasnt_async_execute { ... };
has_async_execute has;
hasnt_async_execute hasnt;
// calls has.async_execute(f)
auto fut1 = execution::async_execute(has, f);
// adapts hasnt to return a future
auto fut2 = execution::async_execute(hasnt, g);
```

- Need to use executors in a variety of use cases
- Not all executors natively implement every use case
- Customization points enables uniform use of executors across use cases*
- *when semantically possible

CUSTOMIZATION POINTS
Traits enable introspection

```
template<class TwoWayExecutor, class F, class = enable_if_t<is_two_way_executor_v<TwoWayExecutor>>>
auto async_execute(Executor exec, F&& f) -> decltype(exec.async_execute(std::forward<F>(f)))
{
   // use .async_execute() directly
   return exec.async_execute(std::forward<F>(f));
}
template<class OneWayExecutor, class F, class = enable_if_t<!is_two_way_executor_v<OneWayExecutor>>>
executor_future_t<OneWayExecutor,result_of_t<F()>
   async_execute(OneWayExecutor exec, F&& f)
{
   // adapt exec to return a future
   executor_future_t<OneWayExecutor,result_of_t<F()>> result_future = ...
   // call another customization point
   execution::execute(exec, ...);
   return std::move(result_future);
}
```

EXECUTORS & THE STANDARD LIBRARY
Composition with control structures

```
my_executor exec =
get_my_executor(...);

using namespace std;

auto fut1 = async(exec, task);

sort(execution::par.on(exec),
vec.begin(), vec.end());

auto fut2 = fut1.then(exec,
continuation);
```

- Most programmers use higher-level control structures
- Need to compose with user-defined executors

EXECUTORS & THE STANDARD LIBRARY

Possible implementation of std::for_each

```
template<class Policy, class Iterator>
using __enable_if_bulk_sync_executable<Policy,Iterator> =
   enable_if_t<
      is_same_v<
         typename Policy::execution_category,
         parallel_execution_tag
      > &&
      is_convertible_v<
         typename iterator_traits<Iterator>::iterator_category,
         random_access_iterator_tag
      >
>;
```

POSSIBLE EXTENSIONS

Out of scope of minimal proposal

- Error handling
- Requirements on user-defined `Future` types
- Heterogeneity
- Distributed memory
- Additional abstractions for bulk execution
- Higher-level variadic abstractions
- Remote execution
- Additional thread pool functionality
- System resources
- Syntactic sugar for contexts + control structures

Summary Executors

Executors **decouple** **control structures from work creation**

Short-term goal: **compose with existing** **control structures**

P0443 is the **minimal** **proposal to achieve short-term goal**

Provides a foundation for **extensions** **to build on**

Agenda

- Heterogensous Computing for ISO C++
- SYCL
- HPX
- Kokkos
- Raja
- State of C++ Concurrency and Parallelism
- C++ Executors coming in C++20
- C++ simd/vector coming in C++20

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)

Codeplay

- 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

- 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

- 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

- 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

- 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

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."

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

Community Edition

Available now for free!

Visit:

computecpp.codeplay.com

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

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

Questions ?

@codeplaysoft

/codeplaysoft

codeplay.com