



Parallel STL for CPU and GPU The Future of Heterogeneous/Distributed C++

Michael Wong, Gordon Brown, Ruyman Reyes, Christopher DiBella

Acknowledgement Disclaimer

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

But I claim all credit for errors, and stupid mistakes. **These are mine, all mine!**

Legal Disclaimer

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

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

Codeplay - Connecting AI to Silicon

Products

ComputeCpp™

C++ platform via the SYCL™ open standard, enabling vision & machine learning e.g. TensorFlow™

ComputeAorta™

The heart of Codeplay's compute technology enabling OpenCL™, SPIR™, HSA™ and Vulkan™

Company

High-performance software solutions for custom heterogeneous systems

Enabling the toughest processor systems with tools and middleware based on open standards

Established 2002 in Scotland

~70 employees



Addressable Markets

Automotive (ISO 26262)
IoT, Smartphones & Tablets
High Performance Compute (HPC)
Medical & Industrial

Technologies: Vision Processing
Machine Learning
Artificial Intelligence
Big Data Compute

Customers



Agenda

- History of ParallelSTL
- What changed from TS to C++17: the life and times of a TS
- ParallelSTL on a CPU
- One more thing: ParallelSTL on **GPU**
- Live Demo

C++17 Parallel STL: Democratizing Parallelism in C++

What is Parallel STL?

Parallel STL greatly facilitates the usage of parallelism in C++ by exposing a parallel interface for the STL algorithms.

Why do I care?

Hardware architecture is becoming increasingly parallel. You cannot escape. See Herb Sutter [The Free Lunch Is Over](#), which is *now over 10 years old now!* More updated version: [Welcome to the jungle](#)

What does it include?

It adds wording for parallel execution on the C++ standard, and **Execution Policies** to the STL interface that enable selecting the appropriate level of parallelism.

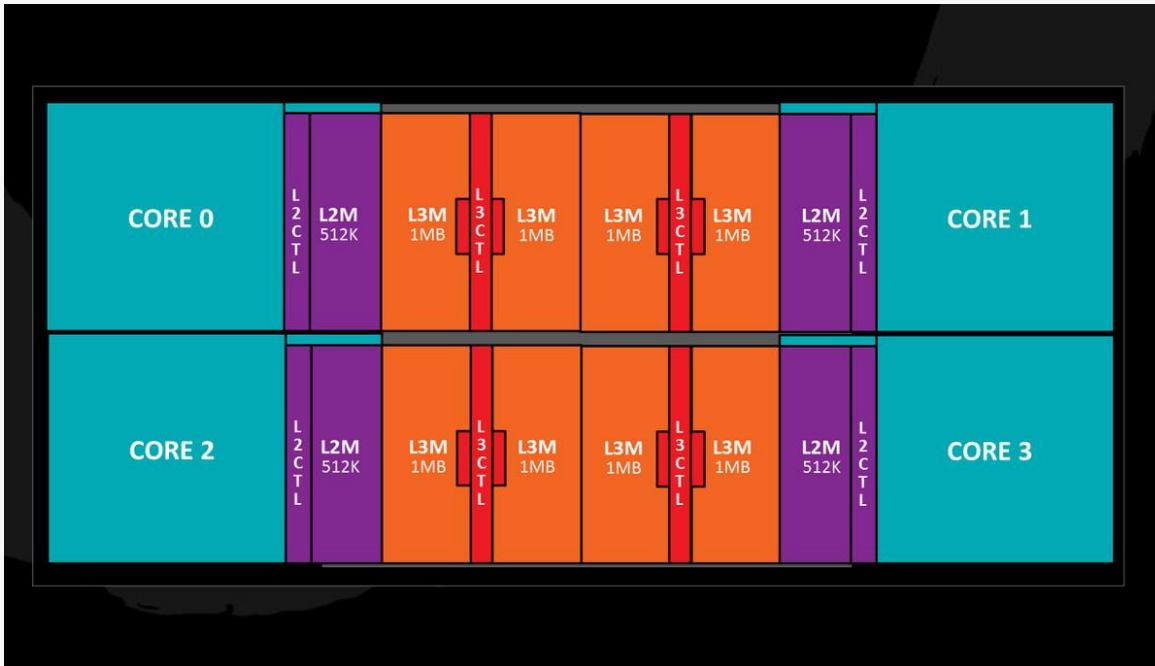
New parallel algorithms are also added to the interface.

What do I take from this talk?

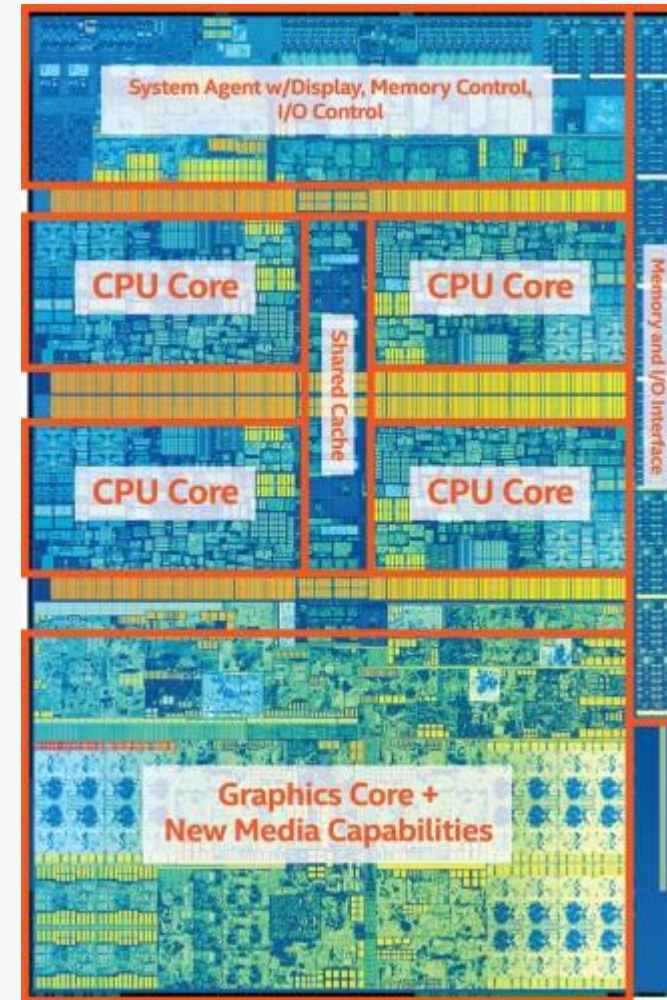
You will understand what Parallel STL is and learn the basic to use them. You'll be ready to use also the SYCL ParallelSTL on your accelerator.

C++ goes parallel!

Current “Desktop” technology



AMD Ryzen (4 cores/socket)



Intel Core i7 7th generation (4 cores + GPU / socket)

History

- Various libraries existed over the years:
 - AMD Bolt, NVIDIA Thrust, Microsoft C++ AMP algorithms...
- In 2012, two separate proposals for parallelism come to C++ standard:
 - NVIDIA (N3408) based on Thrust
 - Microsoft and Intel (N3429), based on Intel TBB and PPL/C++AMP



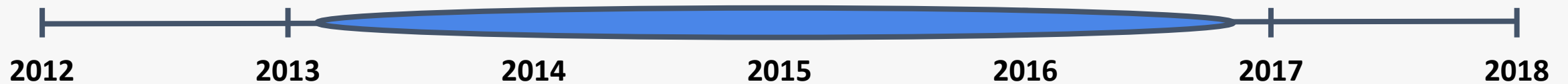
History

- Joint Proposal in 2013:
 - A Parallel Algorithms Library (n3554)



History

- Proposal evolved/matured for a couple of years
 - N3554, N3850, N3960, N4071, N4409...



History

- **Final proposal P0024R2 accepted for C++17 during Jacksonville**
- Many corrections and clarifications before C++17
- This is the life of a TS from birth to ratification



Agenda

- History of ParallelSTL
- What changed from TS to C++17: the life and times of a TS
- ParallelSTL on a CPU
- One more thing: ParallelSTL on GPU
- Live Demo

Road from Parallelism TS to C++17

- Dropped dynamic execution policy - before the TS
- Execution policy Name changed from vec to unseq- Jacksonville Meeting
- Detached exception from attached to algorithm to attached to execution policy
 - <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0394r4.html>
- Removed exception_list and replaced with terminate and don't unwind
 - <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0394r4.html>
- Changes to numeric parts:
 - <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0571r0.html>
 - <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0452r1.html>
- Name change for transform_reduce
 - <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0623r0.html>
- Changed random_access_iterator instead of just forward_access_iterator so it can be invalidated safely
 - <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0523r1.html>
- Allowed cloning to arguments enable SYCL accessors
 - <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0518r0.html>

Agenda

- History of ParallelSTL
- What changed from TS to C++17: the life and times of a TS
- **ParallelSTL on a CPU**
- One more thing: ParallelSTL on **GPU**
- Live Demo

Sorting with the STL

```
std::vector<int> data = { 8, 9, 1, 4 };
```

```
std::sort(std::begin(data), std::end(data));
```

**Normal sequential
sort algorithm**

```
if (std::is_sorted(data)) {  
    Std::cout << " Data is sorted!" << std::endl;  
}
```

```
std::vector<int> data = { 8, 9, 1, 4 };
```

**Extra parameter to STL
algorithms enable
parallelism**

```
std::sort(std::execution_policy::par,  
          std::begin(data), std::end(data));  
  
if (std::is_sorted(data)) {  
    Std::cout << " Data is sorted!" << std::endl;  
}
```


The Execution Policy: Standard policy classes

- Defined in the execution namespace
 - Sequenced policy
 - Never do parallel, sequenced in-order execution
 - `constexpr sequenced_policy sequenced;`
 - Parallel policy
 - Can use caller thread but may span others (`std::thread`)
 - Invocations do not interleave on a single thread
 - `constexpr sequenced_policy par;`
 - Parallel unsequenced
 - Can use caller thread or others (e.g `std::thread`)
 - Multiple invocations may be interleaved on a single thread
 - `constexpr sequenced_policy par_unseq;`

Many different existing implementations

Available today

- 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: https://thrust.github.io/doc/group__execution__policies.html
- Codeplay: <http://github.com/KhronosGroup/SyclParallelSTL>
- Clang: Not yet available

Expect major C++ compilers to implement it soon!

Using execution policies

```
using std::execution_policy;

// May execute in parallel
std::sort(par, std::begin(data), std::end(data))
// May be parallelized and vectorized
std::sort(std::par_unseq, std::begin(data), std::end(data));
// Will not be parallelized/vectorized
std::sort(std::sequenced, std::begin(data), std::end(data));
// Vendor-specific policy, read their documentation!
std::sort(custom_vendor_policy, std::begin(data), std::end(data));
```

Propagating the policy to the end user

```
using std::execution_policy;

template<typename Policy, typename Iterator>
void library_function(Policy p, Iterator begin,
                    Iterator end) {
    std::sort(p, begin, end);
    std::for_each(p, begin, end,
                  [&](Iterator::value_type e&) { e ++;}) ;
    std::for_each(std::sequenced, begin, end,
                  non_parallel_operation) ;
}
```

Parallel overloads available

Table 1 — Table of parallel algorithms

adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
transform_exclusive_scan	transform_inclusive_scan	transform_reduce	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

New algorithms into the STL: Parallel For Each

```
template<class ExecutionPolicy, class InputIterator, class Function>  
void for_each(ExecutionPolicy && exec, InputIterator first,  
InputIterator last, Function f);
```

```
template<class ExecutionPolicy, class InputIterator, class Size, class  
Function>  
InputIterator for_each_n(ExecutionPolicy && exec,  
                          InputIterator first, Size n,  
                          Function f) ;
```

```
template<class InputIterator, class Size, class Function>  
InputIterator for_each_n(InputIterator first, Size n, Function f);
```

- **for_each**: Applies f to elements in range [first, last).
- **for_each_n**: Applies f to elements in [first, first + n)

New algorithms into the STL

Numerical Parallel Algorithms

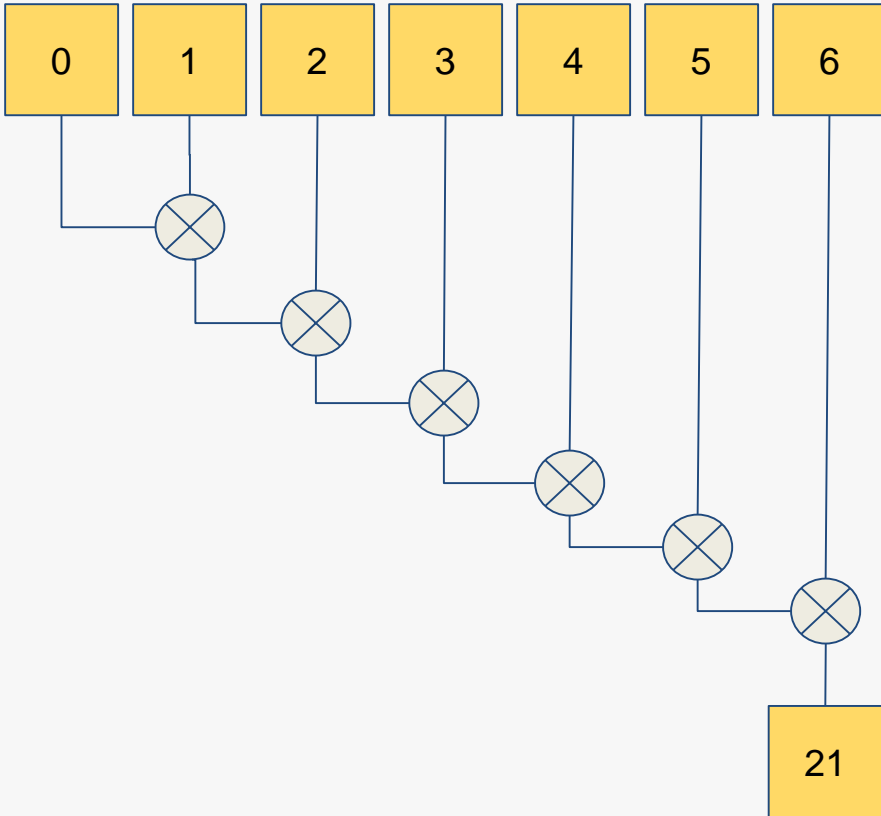
```
template < class InputIterator >
typename iterator_traits < InputIterator >:: value_type
reduce ( InputIterator first , InputIterator last ) ;
```

```
template < class InputIterator , class T >
T reduce ( InputIterator first , InputIterator last , T init ) ;
```

```
template < class InputIterator , class T , class BinaryOperation >
T reduce ( InputIterator first , InputIterator last , T init ,
BinaryOperation binary_op ) ;
```

Implements a reduction operation (the order of the binary_op is not relevant).
The sequential equivalent is accumulate

New algorithms into the STL (Serial Reduction pattern)



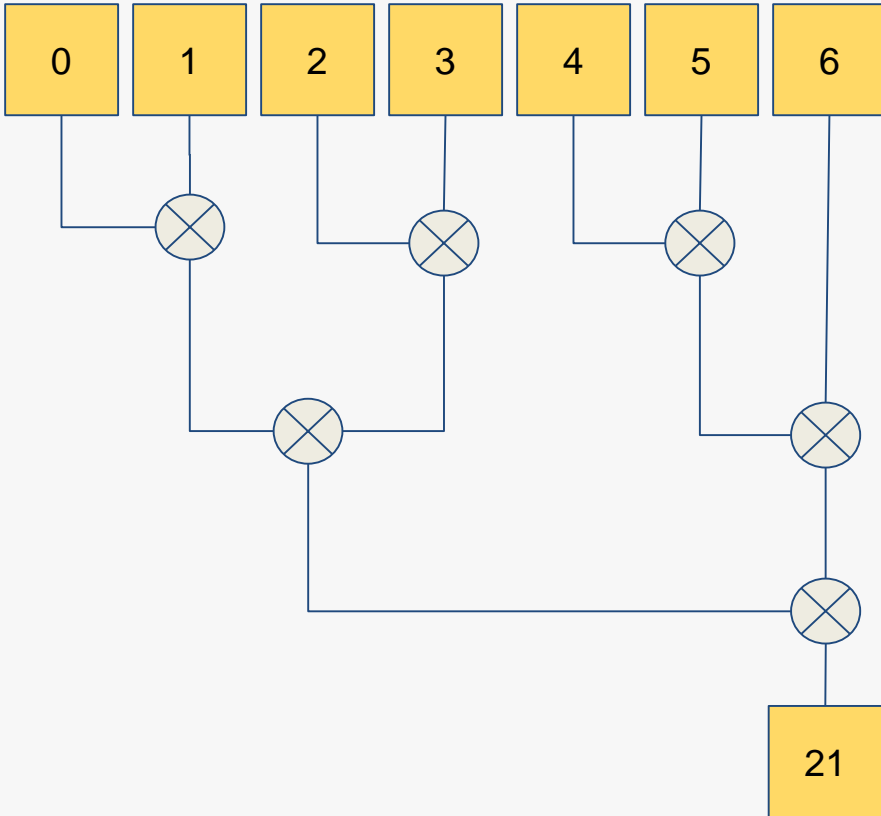
```
size_t nElems = 1000u;
```

```
std::vector<float> nums(nElems);
```

```
std::accumulate(std::begin(v1), nElems, 1);
```

Only one core is used for the different additions.

New algorithms into the STL (Parallel Reduction Pattern)



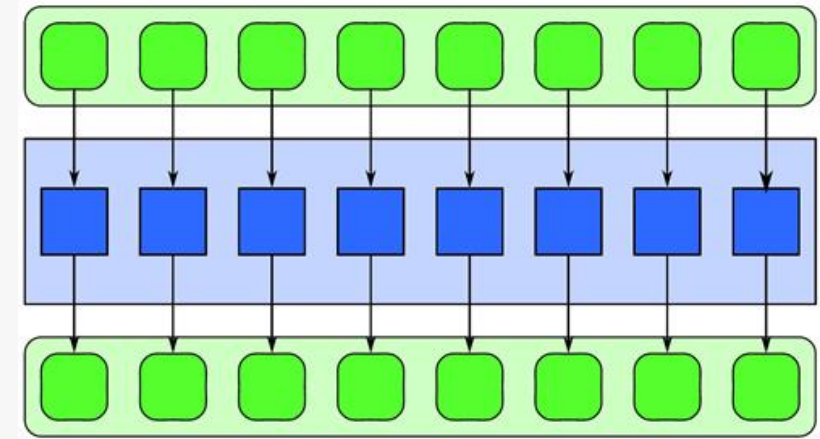
```
size_t nElems = 1000u;  
std::vector<float> nums(nElems);
```

```
std::reduce(std::execution_policy::par,  
           std::begin(v1), nElems, 1);
```

If operation is commutative and associative, can be run in parallel.
Reduction uses all cores!

Transform

```
std::transform(std::execution::par,  
              v1.begin(), v1.end(),  
              v2.begin(), output.begin(),  
              [=](int val1, int val2)  
                { return val1 + val2 + 1; });
```



- transform (a.k.a map) applies a function to an input range and stores the result on the output range. Operation is out of order.

Transform reduce

```
template<class ExecutionPolicy,  
        class ForwardIt1, class ForwardIt2, class T>  
T transform_reduce(ExecutionPolicy&& policy,  
                  ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2, T init);
```

(4) (since C++17)

```
template<class ExecutionPolicy,  
        class ForwardIt1, class ForwardIt2, class T, class BinaryOp1, class BinaryOp2>  
T transform_reduce(ExecutionPolicy&& policy,  
                  ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2,  
                  T init, BinaryOp1 binary_op1, BinaryOp2 binary_op2);
```

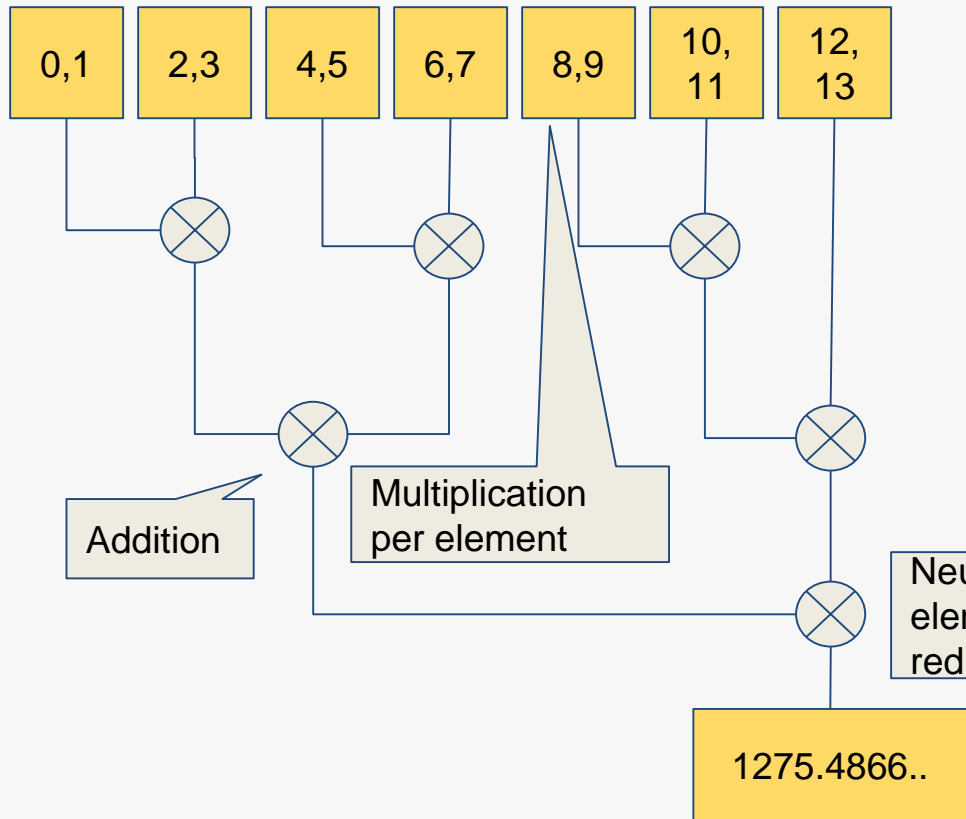
(5) (since C++17)

```
template<class ExecutionPolicy,  
        class ForwardIt, class T, class BinaryOp, class UnaryOp>  
T transform_reduce(ExecutionPolicy&& policy,  
                  ForwardIt first, ForwardIt last,  
                  T init, BinaryOp binary_op, UnaryOp unary_op);
```

(6) (since C++17)

- transform_reduce applies a function to an input range and then applies the binary operation to reduce the values

Transform Reduce example



```
struct elem {  
    float a;  
    float b;  
} elem;
```

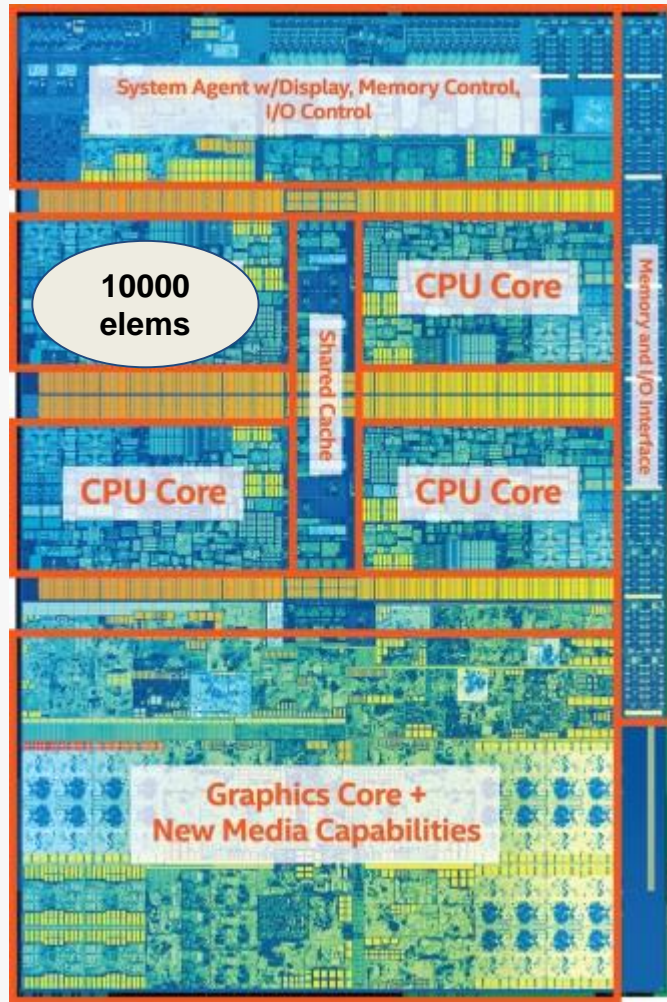
```
auto transformReduce = [&]() {  
    float pi = PI;  
    float res = std::experimental::parallel::transform_reduce(  
        snp, std::begin(v), std::end(v), [=](elem x) { return x.a * x.b * pi; },  
        0, // map multiplication  
        [=](float a, float b) { return a + b; }); // reduce addition  
}
```

Apply to each
element of v

Neutral
element of
reduction

Reduction
function:
addition

What can I do with a Parallel For Each?



Intel Core i7 7th generation

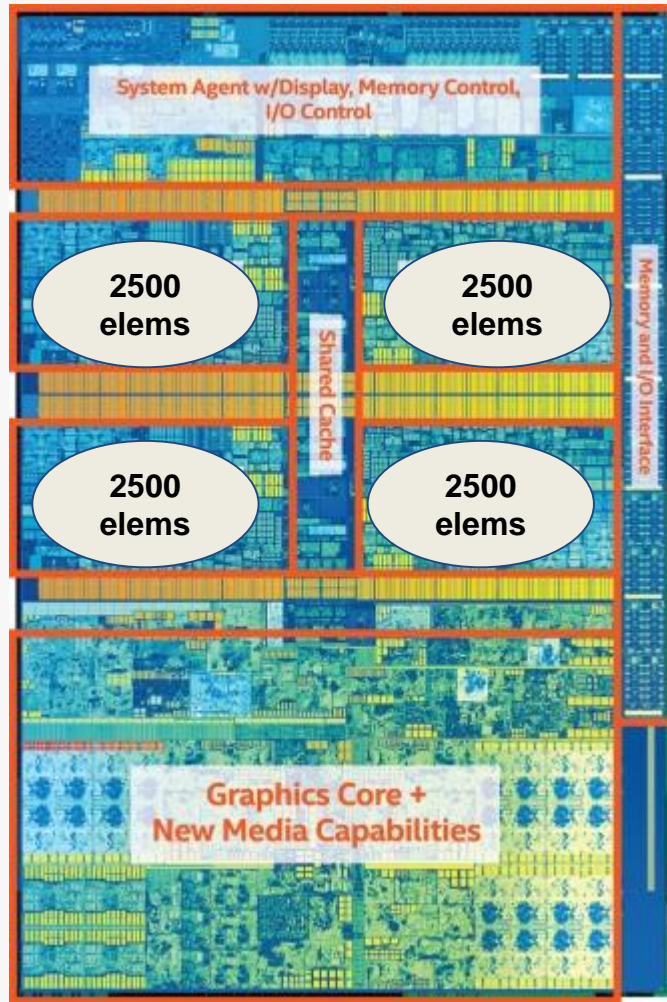
```
size_t nElems = 1000u;  
std::vector<float> nums(nElems);
```

```
std::fill_n(std::begin(v1), nElems, 1);
```

```
std::for_each(std::begin(v), std::end(v),  
              [=](float f) { f * f + f });
```

**Traditional for each uses only one core,
rest of the die is unutilized!**

What can I do with a Parallel For Each?



Intel Core i7 7th generation

```
size_t nElems = 1000u;  
std::vector<float> nums(nElems);
```

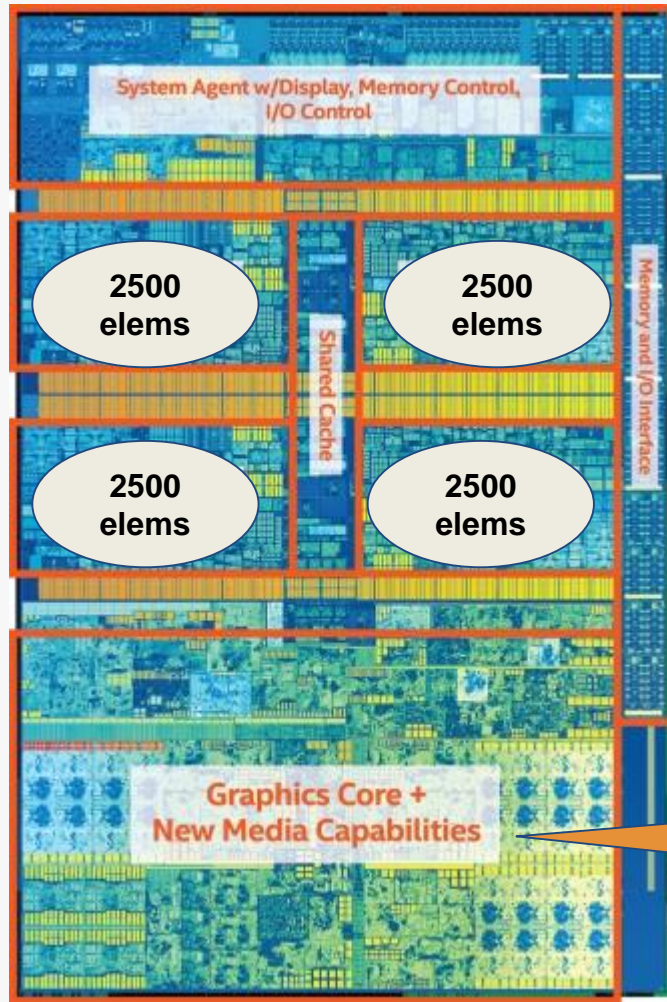
```
std::fill_n(std::execution_policy::par,  
            std::begin(v1), nElems, 1);
```

```
std::for_each(std::execution_policy::par,  
              std::begin(v), std::end(v),  
              [=](float f) { f * f + f });
```

Workload is distributed across cores!

(mileage may vary, implementation-specific behaviour)

What can I do with a Parallel For Each?



Intel Core i7 7th generation

```
size_t nElems = 1000u;  
std::vector<float> nums(nElems);
```

```
std::fill_n(std::execution_policy::par,  
            std::begin(v1), nElems, 1);
```

```
std::for_each(std::execution_policy::par,  
              std::begin(v), std::end(v),  
              [=](float f) { f * f + f });
```

What about this part?

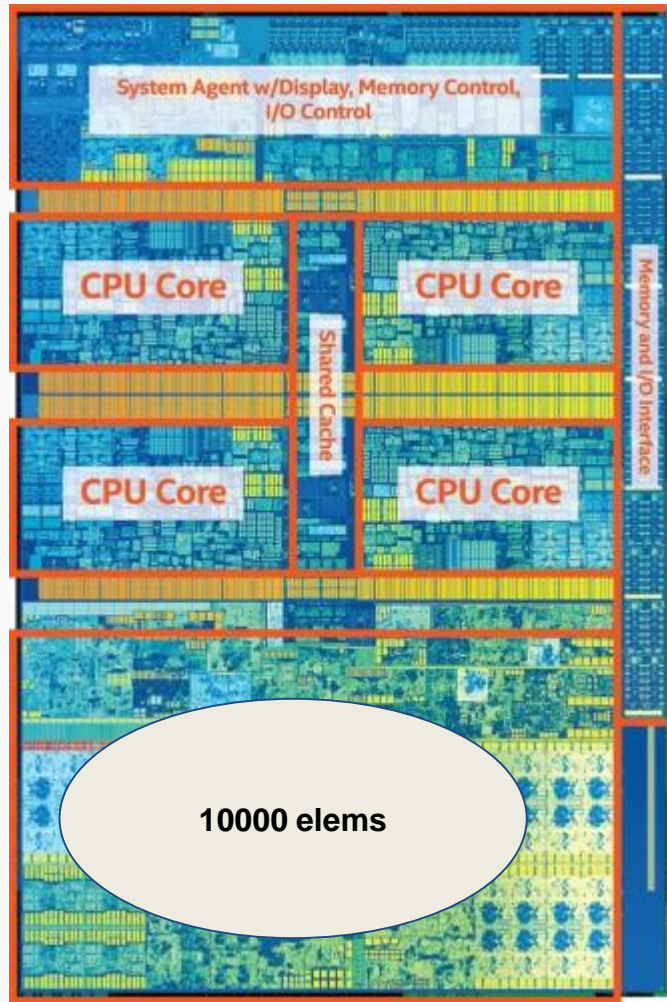
Workload is distributed across cores!

(mileage may vary, implementation-specific behaviour)

Agenda

- History of ParallelSTL
- What changed from TS to C++17: the life and times of a TS
- ParallelSTL on a CPU
- One more thing: ParallelSTL on GPU
- Live Demo

What can I do with a Parallel For Each?



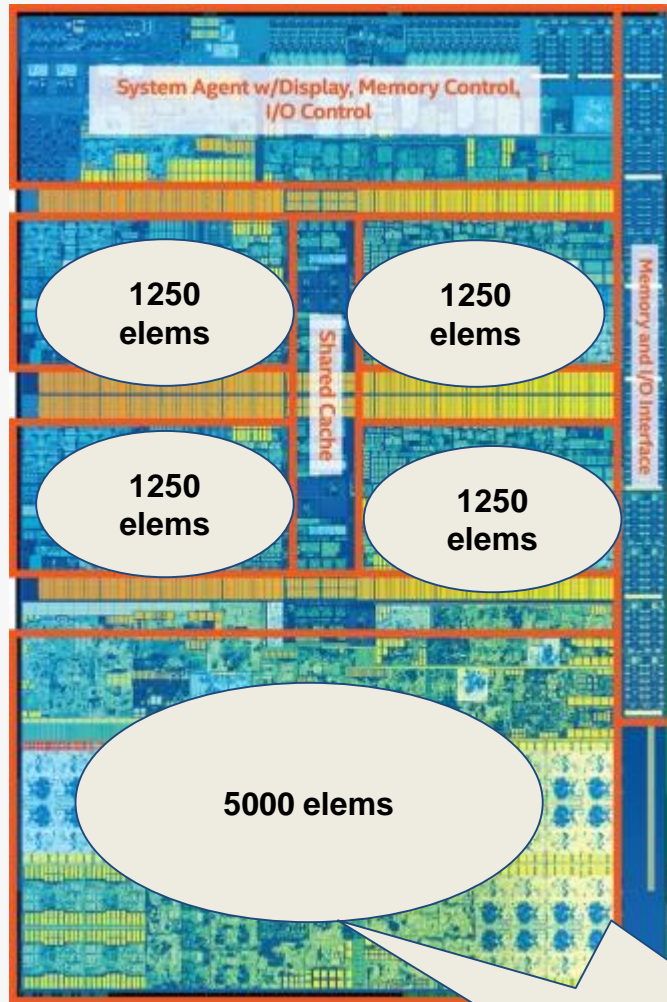
Intel Core i7 7th generation

```
size_t nElems = 1000u;  
std::vector<float> nums(nElems);  
  
std::fill_n(sycl_policy,  
            std::begin(v1), nElems, 1);  
  
std::for_each(sycl_named_policy  
              <class KernelName>,  
              std::begin(v), std::end(v),  
              [=](float f) { f * f + f });
```

Workload is distributed on the GPU cores

(mileage may vary, implementation-specific behaviour)

What can I do with a Parallel For Each?



Intel Core i7-7700K

Experimental!

```
size_t nElems = 1000u;  
std::vector<float> nums(nElems);
```

```
std::fill_n(sycl_heter_policy(cpu, gpu, 0.5),  
            std::begin(v1), nElems, 1);
```

```
std::for_each(sycl_heter_policy<class kName>  
              (cpu, gpu, 0.5),  
              std::begin(v), std::end(v),  
              [=](float f) { f * f + f });
```

Workload is distributed on all cores!

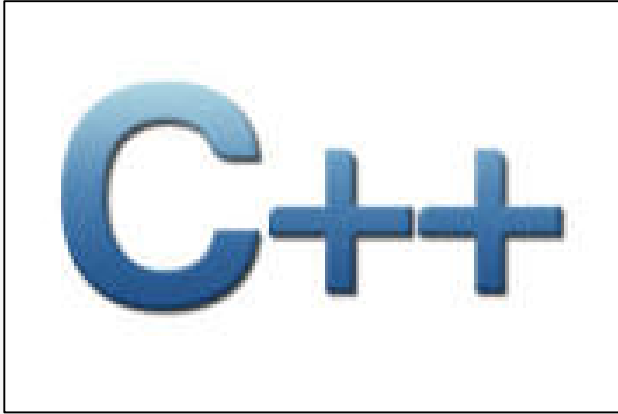
(mileage may vary, implementation-specific behaviour)

Parallel overloads available in SYCL Parallel STL

Table 1 — Table of parallel algorithms

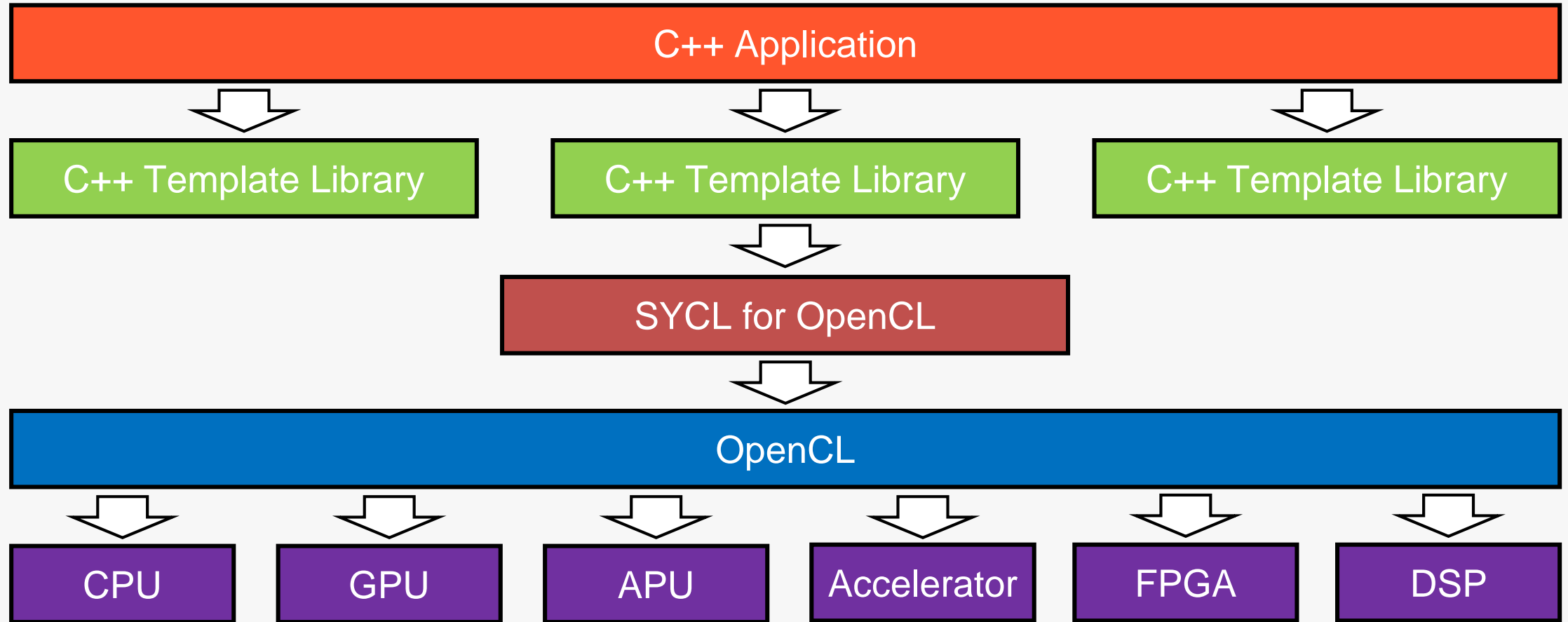
adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if ✓	copy_n	count
count_if ✓	equal	exclusive_scan ✓	fill ✓
fill_n	find ✓	find_end	find_first_of
find_if ✓	find_if_not ✓	for_each ✓	for_each_n ✓
generate	generate_n	includes	inclusive_scan ✓
inner_product ✓	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce ✓	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort ✓
stable_partition	stable_sort	swap_ranges	transform ✓
transform_exclusive_scan	transform_inclusive_scan	transform_reduce ✓	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

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



Example: Vector Add



Example: Vector Add

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

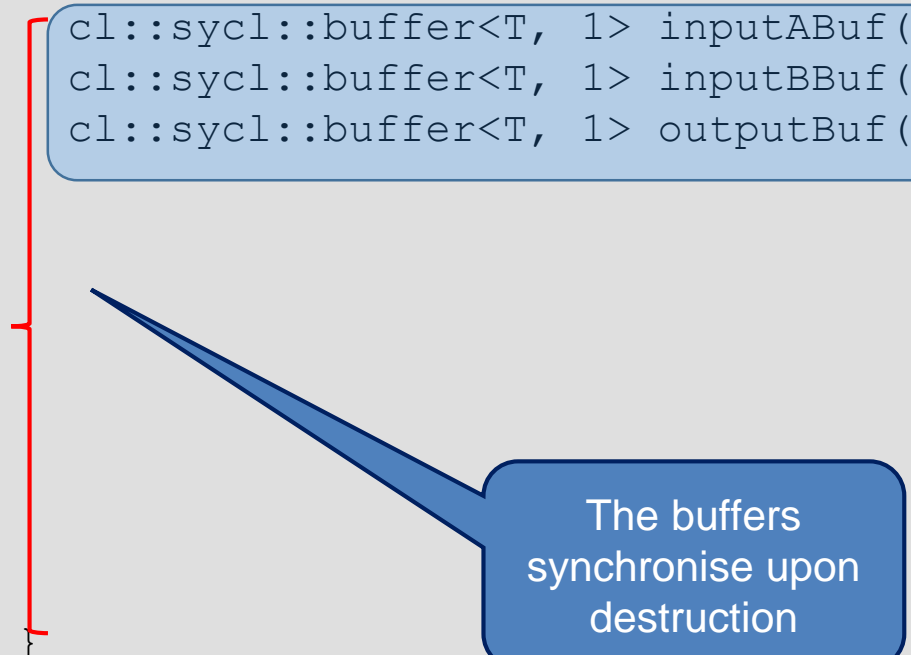
```
template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> &out) {

}
```


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());
}
```

A red vertical line is positioned to the left of the three buffer creation lines in the code block. A blue arrow points from a blue callout box to this red line. The callout box contains the text "The buffers synchronise upon destruction".

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);

    ...
}
```

Implementing Parallel STL with SYCL

```
/* sycl_execution_policy.  
 * The sycl_execution_policy enables algorithms to be executed using  
 * a SYCL implementation.  
 */
```

```
template <class KernelName = DefaultKernelName>
```

```
class sycl_execution_policy {
```

```
    cl::sycl::queue m_q;
```

```
public:
```

```
    // The kernel name when using lambdas
```

```
    using kernelName = KernelName;
```

```
    sycl_execution_policy() = default;
```

```
    sycl_execution_policy(cl::sycl::queue q) : m_q(q){};
```

```
    sycl_execution_policy(const sycl_execution_policy&) = default;
```

```
    // Returns the name of the kernel as a string
```

```
    std::string get_name() const { return typeid(kernelName).name(); };
```

```
    // Returns the queue, if any
```

```
    cl::sycl::queue get_queue() const { return m_q; }
```

Creates a SYCL policy
using an existing queue

Typeid information only valid for
debugging

Implementing Parallel STL with SYCL

```
/* for_each  
*/
```

Iterator can be any
RandomAccess tag

Functions can take C++ iterators
or SYCL-specific iterators

```
template <class Iterator, class UnaryFunction>  
void for_each(Iterator b, Iterator e, UnaryFunction f) {  
    impl::for_each(*this, b, e, f);  
}
```

For_each member function on the policy
forwards to implementation


```
template <class ExecutionPolicy, class Iterator, class UnaryFunction>
void for_each(ExecutionPolicy &sep, Iterator b, Iterator e, UnaryFunction op) {
{
    cl::sycl::queue q(sep.get_queue());
    auto device = q.get_device();
    size_t localRange =
        device.get_info<cl::sycl::info::device::max_work_group_size>();
    auto bufl = sycl::helpers::make_buffer(b, e);
    auto vectorSize = bufl.get_count();
    size_t globalRange = sep.calculateGlobalSize(vectorSize, localRange);
}
```

Obtain the queue from the policy

Obtain device parameters

Prepare allocations on device

Continues...

```

auto f = [vectorSize, localRange, globalRange, &buf1, op](
    cl::sycl::handler &h) mutable {
    cl::sycl::nd_range<1> r{
        cl::sycl::range<1>{std::max(globalRange, localRange)},
        cl::sycl::range<1>{localRange}};
    auto al = buf1.template get_access<cl::sycl::access::mode::read_write>(h);
    h.parallel_for<typename ExecutionPolicy::kernelName>(
        r, [al, op, vectorSize](cl::sycl::nd_item<1> id) {
            if (id.get_global(0) < vectorSize) {
                op(al[id.get_global(0)]);
            }
        });
};
q.submit(f);
}

```

Device Lambda

User functor

Submit for execution on the device

Demo time

Demo Results - Running `std::sort`

(Running on Intel i7 6600 CPU & Intel HD Graphics 520)

size	2 ¹⁶	2 ¹⁷	2 ¹⁸	2 ¹⁹
<code>std::seq</code>	0.27031s	0.620068s	0.669628s	1.48918s
<code>std::par</code>	0.259486s	0.478032s	0.444422s	1.83599s
<code>std::unseq</code>	0.24258s	0.413909s	0.456224s	1.01958s
<code>sycl_execution_policy</code>	0.273724s	0.269804s	0.277747s	0.399634s

Future direction

Future Heterogeneous/Distributed directions in C++

- SG14/SG1 driving towards a future TS in Heterogeneous ISO C++
 - Executors enables multiple resources
 - Asynchronous Algorithm enables latency hiding
 - Context
 - Affinity before inaccessible memory
 - Data movement to access inaccessible memory
 - Exception handling in a concurrent environment

Executors

invoke async parallel algorithms future::then post
defer define_task_block dispatch asynchronous operations strand<>

Unified interface for execution

SYCL / OpenCL /
CUDA / HCC

OpenMP / MPI

C++ Thread Pool

Boost.Asio /
Networking TS



Summary

- Show History of Parallelism TS
- Discussed the path from a TS to C++17, many changes and continued discussions in design points based on feedback
- Showed Parallelism STL running on CPU
- ... and on GPU
- Show future direction in Heterogeneous/Distributed ISO C++

We're
Hiring!
codeplay.com/careers/

Thanks for Listening



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



info@codeplay.com



codeplay.com

What are Thread Execution Steps

- Termination of the thread of execution
- Performing an access through a volatile glvalue
- Completion of a call to a library I/O function
- Synchronization operation (e.g. mutex)
- Atomic operation

A thread of execution makes progress when an execution step occurs

Forward Progress Guarantees

Concurrent forward progress

If a thread offers *concurrent forward progress guarantee*, it will *make progress* (as defined above) in finite amount of time, for as long as it has not terminated, regardless of whether other threads (if any) are making progress.

The standard encourages, but doesn't require that the main thread and the threads started by [std::thread](#) offer concurrent forward progress guarantee.

Forward Progress Guarantees

Parallel forward progress

If a thread offers *parallel forward progress guarantee*, the implementation is not required to ensure that the thread will eventually make progress if it has not yet executed any execution step (I/O, volatile, atomic, or synchronization), but once this thread has executed a step, it provides *concurrent forward progress* guarantees (this rule describes a thread in a thread pool that executes tasks in arbitrary order)

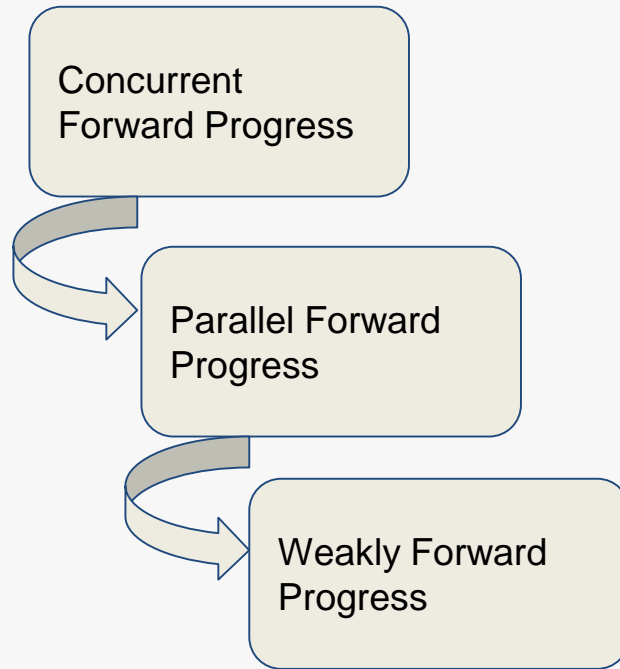
Forward Progress Guarantees

Weakly parallel forward progress

If a thread offers *weakly parallel forward progress guarantee*, it does not guarantee to eventually make progress, regardless of whether other threads make progress or not.

The parallel algorithms from the C++ standard library block with forward progress delegation on the completion of an unspecified set of library-managed threads.

What does that means



- Concurrent Forward Progress
 - all threads can progress completely independent
- Parallel Forward Progress
 - When a thread starts will finish but no guarantees that all threads will start simultaneously
- Weakly Forward Progress
 - No guarantees when/if a thread will re-start