CPP-Summit 2019

全球C++软件技术大会

C++ Development Technology Summit

Boolan

高端IT互联网教育平台

关注“博览Boolan”服务号

发现更多 会议·课程·活动

C++ 20 state of parallelism, concurrency and heterogeneous programming

Michael Wong

Codeplay Software

VP of Research and Development

http:://wongmichael.com/about

michael@codeplay.com

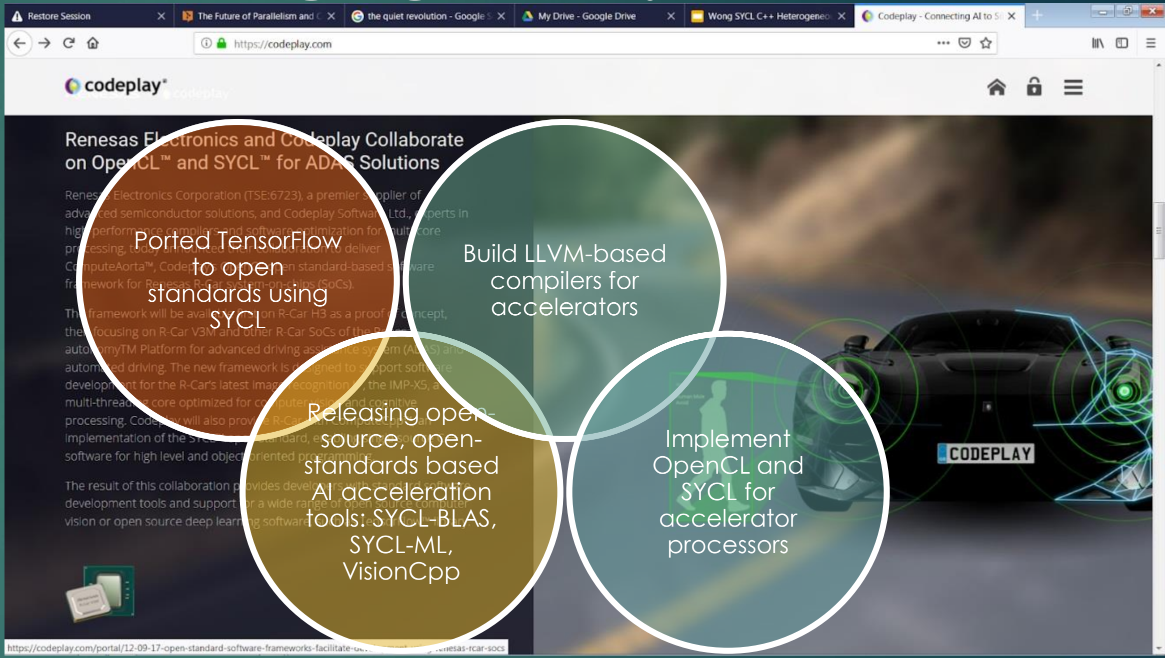
Who am I?

VP of R&D of Codeplay

Chair of SYCL Heterogeneous Programming Language
C++ Directions Group
ISOCPP.org Director, VP
http://isocpp.org/wiki/faq/wg21#michael-wong

Head of Delegation for C++ Standard for Canada
Chair of Programming Languages for Standards Council of Canada
Chair of WG21 SG19 Machine Learning
Chair of WG21 SG14 Games Dev/Low Latency/Financial Trading/Embedded
Editor: C++ SG5 Transactional Memory Technical Specification
Editor: C++ SG1 Concurrency Technical Specification
MISRA C++ and AUTOSAR
wongmichael.com/about
We build GPU compilers for semiconductor companies

- Now working to make AI/Ml heteroegneous acceleration safe for autonomous vehicle



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.

Specifically, Paul Mckenney, Joe Hummel, Bjarne Stroustru, Botond Ballo for some of the slides.

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™

Addressable Markets

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

Technologies: Vision Processing
Machine Learning
Artificial Intelligence
Big Data Compute

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

Customers

BROADCOM
RENESAS
Imagination
QUALCOMM
Movidius

Partners
intel
AMD

3 Act Play

- Is ISO C++ going heterogeneous?
- What is C++20 parallelism & concurrency?
- What is coming for C++23?

- What gets me up every morning?

C++11,14,17"No more Raw Food"

Don't use	Don't use raw numbers, do type-rich programming with UDL
Don't declare	Don't declare, use auto whenever possible
Don't use	Don't use raw NULL or (void *) 0, use nullptr
Don't use	Don't use raw new and delete, use unique_ptr/shared_ptr
Don't use	Don't use heap-allocated arrays, use std::vector and std::string, or the new VLA, then dynarray<>
Don't use	Don't use functors, use lambdas
Don't use	Don't use raw loops; use STL algorithms, ranged-based for loops, and lambdas
Rule	Rule of Three? Rule of Zero or Rule of Five.

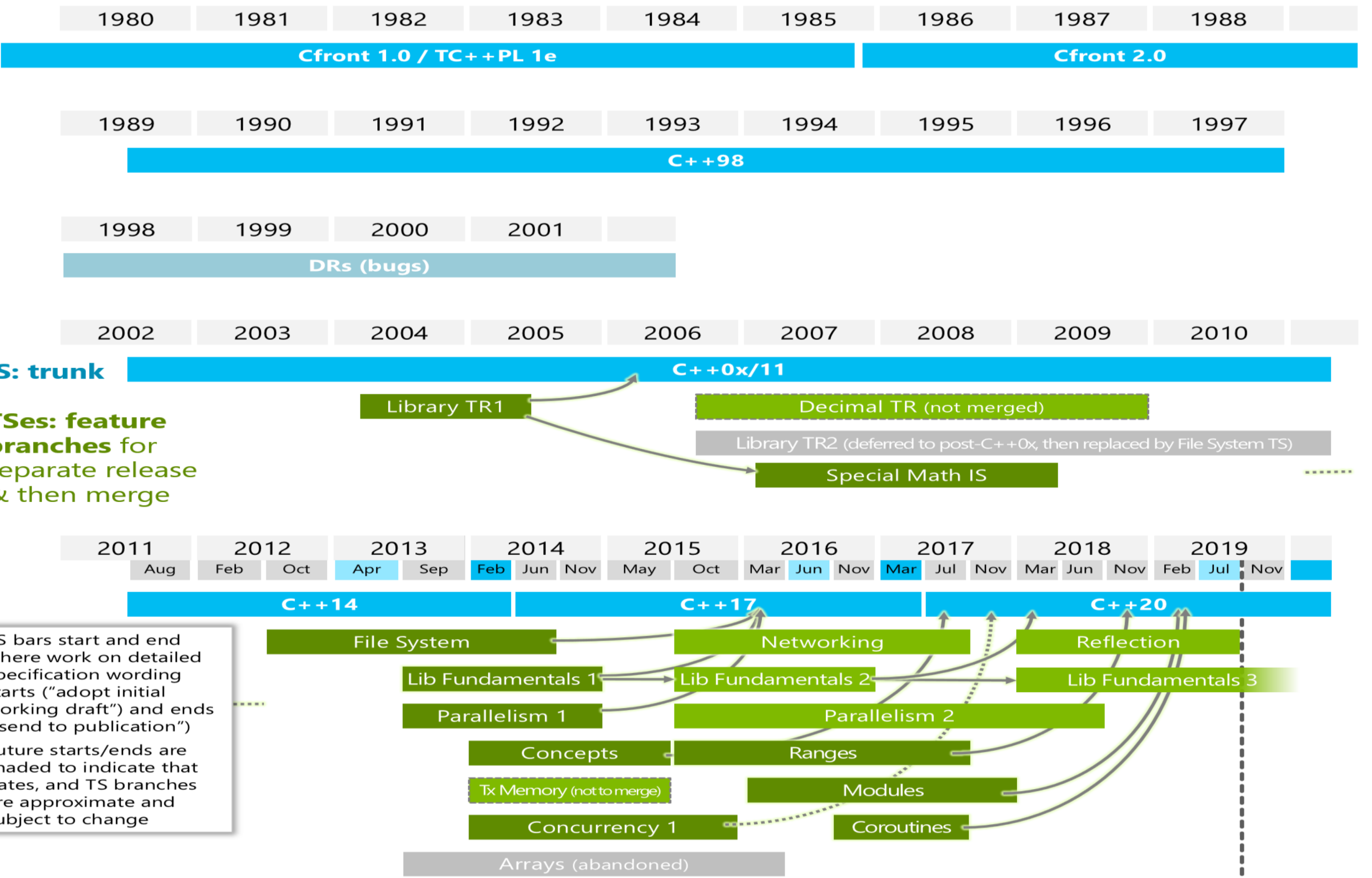
Parallelism "Use the right

Abstraction	How is it supported
Cores	C++11/14/17 threads, async
HW threads	C++11/14/17 threads, async
Vectors	Parallelism TS2
Atomic, Fences, lockfree, futures, counters, transactions	C++11/14/17 atomics, Concurrency TS1, Transactional Memory TS1
Parallel Loops	Async, TBB:parallel_invoke, C++17 parallel algorithms, for_each
Heterogeneous offload, fpga	OpenCL, SYCL, HSA, OpenMP/ACC, Kokkos, Raja
Distributed	HPX, MPI, UPC++
Caches	C++17 false sharing support
Numa	Executors, Execution Context, Affinity
TLS	EALS
[illegible]	[illegible]

Act 1

- Is ISO C++ going heterogeneous?

C++ Std Timeline/status



Parallel/concurrency before C++11 (C++98)

	Asynchronus Agents	Concurrent collections	Mutable shared state	Heterogeneous (GPUs, accelerators, FPGA, embedded AI processors)
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory	Dispatch/offload to other nodes (including distributed)
examples	GUI,background printing, disk/net access	trees, quicksorts, compilation	locked data(99%), lock-free libraries (wizards), atomics (experts)	Pipelines, reactive programming, offload,, target, dispatch
key metrics	responsiveness	throughput, many core scalability	race free, lock free	Independent forward progress,, load-shared
requirement	isolation, messages	low overhead	composability	Distributed, heterogeneous
today's abstractions	POSIX threads, win32 threads, OpenCL, vendor intrinsic	openmp, TBB, PPL, OpenCL, vendor intrinsic	locks, lock hierarchies, vendor atomic instructions, vendor intrinsic	OpenCL, CUDA

Parallel/concurrency after C++11

	Asynchronus Agents	Concurrent collections	Mutable shared state	Heterogeneous (GPUs, accelerators, FPGA, embedded AI processors)
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory	Dispatch/offload to other nodes (including distributed)
examples	GUI,background printing, disk/net access	trees, quicksorts, compilation	locked data(99%), lock-free libraries (wizards), atomics (experts)	Pipelines, reactive programming, offload,, target, dispatch
key metrics	responsiveness	throughput, many core scalability	race free, lock free	Independent forward progress,, load-shared
requirement	isolation, messages	low overhead	composability	Distributed, heterogeneous
today's abstractions	C++11: thread,lambda function, TLS	C++11: Async, packaged tasks, promises, futures, atomics	C++11: locks, memory model, mutex, condition variable, atomics, static init/term	C++11: lambda

Parallel/concurrency after C++14

	Asynchronus Agents	Concurrent collections	Mutable shared state	Heterogeneous
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory	Dispatch/offload to other nodes (including distributed)
examples	GUI,background printing, disk/net access	trees, quicksorts, compilation	locked data(99%), lock-free libraries (wizards), atomics (experts)	Pipelines, reactive programming, offload,, target, dispatch
key metrics	responsiveness	throughput, many core scalability	race free, lock free	Independent forward progress,, load-shared
requirement	isolation, messages	low overhead	composability	Distributed, heterogeneous
today's abstractions	C++11: thread,lambda function, TLS, async C++14: generic lambda	C++11: Async, packaged tasks, promises, futures, atomics,	C++11: locks, memory model, mutex, condition variable, atomics, static init/term, C++ 14: shared_lock/shared_timed_mutex, OOTA, atomic_signal_fence,	C++11: lambda C++14: none

Parallel/concurrency after C++17

	Asynchronus Agents	Concurrent collections	Mutable shared state	Heterogeneous (GPUs, accelerators, FPGA, embedded AI processors)
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory	Dispatch/offload to other nodes (including distributed)
today's abstractions	C++11: thread,lambda function, TLS, async C++14: generic lambda	C++11: Async, packaged tasks, promises, futures, atomics, C++ 17: ParallelSTL, control false sharing	C++11: locks, memory model, mutex, condition variable, atomics, static init/term, C++ 14: shared_lock/shared_timed_mutex, OOTA, atomic_signal_fence, C++ 17: scoped _lock, shared_mutex, ordering of memory models, progress guarantees, TOE, execution policies	C++11: lambda C++14: generic lambda C++17: progress guarantees, TOE, execution policies

Act 1

- What is C++ 20 parallelism and concurrency?

Concurrency vs Parallelism

What makes parallel or concurrent programming harder than serial programming? What’s the difference? How much of this is simply a new mindset one has to adopt?

Parallel/concurrency aiming for C++20

	Asynchronus Agents	Concurrent collections	Mutable shared state	Heterogeneous/Distributed
today's abstractions	C++11: thread,lambda function, TLS, async C++ 20: Jthreads +interrupt _token, coroutines	C++11: Async, packaged tasks, promises, futures, atomics, C++ 17: ParallelSTL, control false sharing C++ 20: Is_ready(), make_ready_future() , simd<T>, Vec execution policy, Algorithm un-sequenced policy, span	C++11: locks, memory model, mutex, condition variable, atomics, static init/term, C++ 14: shared_lock/shared_timed_mutex, OOTA, atomic_signal_fence, C++ 17: scoped _lock, shared_mutex, ordering of memory models, progress guarantees, TOE, execution policies C++20: atomic_ref, Latches and barriers, atomic<shared_ptr> Atomics & padding bits Simplified atomic init Atomic C/C++ compatibility Semaphores and waiting Fixed gaps in memory model , Improved atomic flags, Repair memory model	C++11: lambda C++14: generic lambda C++17: , progress guarantees, TOE, execution policies C++20: atomic_ref, simd<T>, span

C++20 asynchronous, concurrency, parallelism, heterogeneous programming

- SIMD<T>: vector library type
- Futures: is_ready, make_ready _futures
- coroutines
- Jthreads: cooperative cancellation of threads
- Latches and Barriers: new synchronization facilities
- Atomics<shared_ptr<T>>: updates to atomics

Power of Computing

- 1998, when C++ 98 was released
  - Intel Pentium II: 0.45 GFLOPS
  - No SIMD: SSE came in Pentium III
  - No GPUs: GPU came out a year later
- 2011: when C++11 was released
  - Intel Core-i7: 80 GFLOPS
  - AVX: 8 DP flops/HZ*4 cores *4.4 GHz= 140 GFlops
  - GTX 670: 2500 GFLOPS
- Computers have gotten so much faster, how come software have not?
  - Data structures and algorithms
  - latency

In 1998, a typical machine had the following flops

- .45 GFLOP, 1 core

- Single threaded C++98/C99/Fortran dominated this picture

In 2011, a typical machine had the following flops

- 2500 GFLOP GPU

- To program the GPU, you have to use CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP

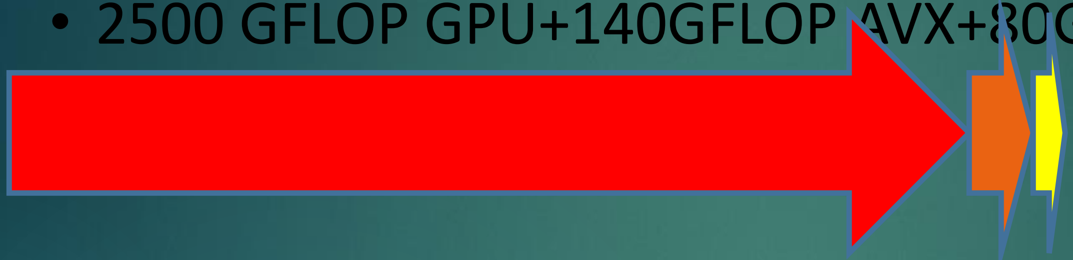
In 2011, a typical machine had the following flops

- 2500 GFLOP GPU+140GFLOP AVX

- To program the GPU, you have to use CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP
- To program the vector unit, you have to use Intrinsics, OpenCL, or auto-vectorization

In 2011, a typical machine had the following flops

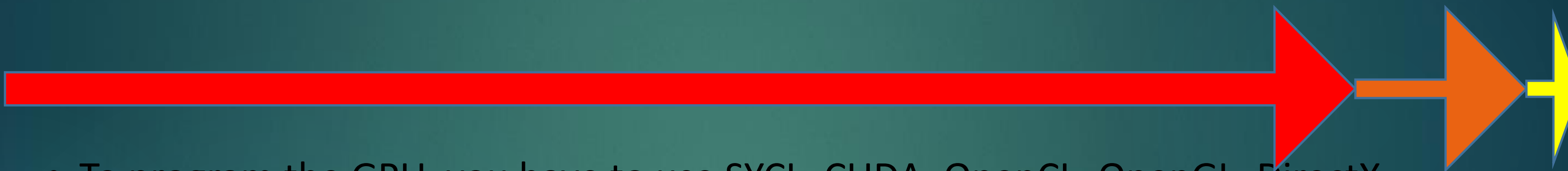
- 2500 GFLOP GPU+140GFLOP AVX+80GFLOP 4 cores



- To program the GPU, you have to use CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP
- To program the vector unit, you have to use Intrinsics, OpenCL, or auto-vectorization
- To program the CPU, you might use C/C++11, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors, OpenCL

In 2017, a typical machine had the following flops

- 4600 GFLOP GPU+560 GFLOP AVX+140 GFLOP



- To program the GPU, you have to use SYCL, CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP, OpenMP
- To program the vector unit, you have to use SYCL, Intrinsics, OpenCL, or auto-vectorization, OpenMP
- To program the CPU, you might use C/C++11/14/17, SYCL, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors, OpenMP, parallelism TS, Concurrency TS, OpenCL

SIMD Language Extensions

- IBM currently has 7 SIMD architectures
  - VMX, VMX128, VSX, SPE, BGL, BGQ, QPX
  - Each has its own proprietary language extension
  - Code written for one language extension can't be moved without a rewrite
  - We don't even have compatibility within our own company
- Intel has 7 SIMD architectures
  - MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX-512/MIC
  - MMX, SSEx, AVX each have a different language extension
  - Code written for one language extension can't be moved without a rewrite

“The Great Hope”- Auto-Vectorizing compilers

- SIMD floating point entered the market 15 years ago
  - (Intel Pentium III, Motorola G4, AMD K6-2)
- Software industry held its breath waiting for a “magic” auto-vectorizing compiler (including Microsoft)
- Despite 15 years of research and development the industry still doesn’t have a good auto-vectorizing compiler
- Industry instead ended up with primitive language support
  - Multiple non-compatible language extensions
  - Compiler intrinsics
- Using intrinsics humans still produce superior vector code but at great pain

Why autovectorization is hard?

- SIMD register width has increased from 128-256-512, 1024 soon
- Instructions are more powerful and complex
  - Hard for compiler to select proper instruction
  - Code pattern needs to be recognized by the compiler

What sort of loops can be vectorized?

- Countable
- Single entry, single exit
- Straight-line code
- Innermost loop of a nest
- No function calls
- Certain non-contiguous memory access
- Some Data dependencies
- Efficient Alignment
- Mixed data types
- Non-unit stride between elements
- Loop body too complex (register pressure)

Industry needs better language support for SIMD

- 80% of Cell programmers time spent vectorizing code
- Need to reduce programming effort
  - Fewer code modifications to vectorize
  - Rapid conversion of scalar to vector code
- Code portability
  - Don't rewrite for every SIMD architecture
- Less code maintenance
  - Intrinsics impossible to maintain
  - Easier to rewrite then figuring out what the code is doing
- Support required vendor-specific extensions

C++ Vector parallelism

- 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++
- P0193 status report
- P0203 design considerations
- P0214 latest SIMD paper

SIMD from Matthias Kretz

- std::simd<T, N, Abi>
  - simd<T, N> SIMD register holding N elements of type T
  - simd<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 SIMD

- Built-in operators
- All usual binary operators are available, for all:
  - simd<T, N> simd<U, N>
  - simd<T, N> U, U simd<T, N>
- Compound binary operators and unary operators as well
  - simd<T, N> convertible to simd<U, N>
  - simd<T, N>(U) broadcasts the value
- No promotion:
  - simd<uint8_t>(255) + simd<uint8_t>(1) == simd<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)

Violence is NEVER the answer

	1. Kill	2. Tell, don't take no for an answer	3. Ask politely, and accept rejection	4. Set flag politely, let it poll if it wants
Tagline	Shoot first, check invariants later	Fire him, but let him clean out his desk	Tap him on the shoulder	Send him an email
Summary	A time-honored way to randomly corrupt your state and achieve undefined behavior	Interrupt at well-defined points and allow a handler chain (but target can't refuse or stop)	Interrupt at well-defined points and allow a handler chain, but request can be ignored	Target actively checks a flag – can be manual, or provided as part of #2 or #3
Pthreads	pthread_kill pthread_cancel (async)	pthread_cancel (deferred mode)	*n/a*	Manual
Java	Thread.destroy Thread.stop	*n/a*	Thread.interrupt	Manual, or Thread.interrupted
.NET	Thread.Abort	*n/a*	Thread.Interrupt	Manual, or Sleep(0)
C++0x	*n/a*	*n/a*	*n/a*	Manual
Guidance	**Avoid,** almost certain to corrupt transaction(s)	OK for languages without exceptions and unwinding…	Good, conveniently automated	Good, but requires more cooperative effort (can be a plus)

Interrupt politely

- Cooperative thread cancellation

4. Set flag politely, let it poll if it wants
Send him an email
Target actively checks a flag – can be manual, or provided as part of #2 or #3
Manual
Manual, or Thread.interrupted
Manual, or Sleep(0)
Manual
Good, but requires more cooperative effort (can be a plus!)

C++ Cooperative cancellation

- std::stop_source and
- std::stop_token to handle cooperative cancellation.
- the target task needs to check
- std::condition_variable_any so the wait can be interrupted by a stop request
- use std::stop_callback to provide your own cancellation mechanism.

```
Data read_file(
  std::stop_token st,
  std::filesystem::path filename ){
   auto
handle=open_file(filename);
   std::stop_callback cb(
   st,[=]{ cancel_io(handle);});
   return read_data(handle); //
blocking
}
```

Work flow

1. Create a std::stop_source
2. In your background task, Obtain a std::stop_token from the std::stop_source
3. Pass the std::stop_token to a new thread or task
4. When you want the background operation to stop call source.request_stop()
5. Periodically Background task call token.stop_requested() to check

jthread is an RAII Style thread

- std::jthread integrates with std::stop_token to support cooperative cancellation.
- Destroying a std::jthread calls source.request_stop() and thread.join().
- The thread needs to check the stop token passed in to the thread function.
- Stop_token is passed as a first parameter
- Backwards compatible: If you don't support having a stop_token, it would not stop.

```
void thread_func(
  std::stop_token st,
  std::string arg1,int arg2){
while(!st.stop_requested()){
  do_stuff(arg1,arg2);
  }
}
void foo(std::string s){
  std::jthread t(thread_func,s,42);
  do_stuff();
} // destructor requests stop and joins
```

make_ready_future

```
future<int> compute(int x) {
    if (x < 0) return make_ready_future<int>(-1);
    if (x == 0) return make_ready_future<int>(0);
    future<int> f1 = async([]() { return do_work(x); });
    return f1;
}
```

is_ready

```
future<int> f1 = async([]() { return
possibly_long_computation(); });
if(!f1.is_ready()) {
    //if not ready, attach a continuation and avoid a
blocking wait
    f1.then([] (future<int> f2) {
        int v = f2.get();
        process_value(v);
    });
}
// if ready, no need to add continuation, process value right
away
else {
    int v = f1.get();
    process_value(v);
}
```

New Synchronization

- Latches
- Barriers
- Semaphores

Latches are for single use

- std::latch is a single-use counter that allows threads to
- wait for the count to reach zero.
- Any waiting threads become unblocked and carry on

1 Create the latch with a non-zero count

2 One or more threads decrease the count

3 Other threads may wait for the latch to be signalled.

4 When the count reaches zero it is permanently signalled and all waiting threads are woken.

```
void foo(){
  unsigned const thread_count=...;
  std::latch done(thread_count);
  my_data data[thread_count];
  std::vector<std::jthread> threads;
  for(unsigned i=0;i<thread_count;++i)
threads.push_back(std::jthread([&,i]{
  data[i]=make_data(i);
  done.count_down();
  do_more_stuff();
  }));
  done.wait();
  process_data();
}
```

Using a latch is great for multithreaded tests

1. Set up the test data
2. Create a latch
3. Create the test threads: The first thing each thread does is test_latch.arrive_and_wait()
4. When all threads have reached the latch they are unblocked to run their code

Barriers is reusable for loop synchronization between parallel tasks

- std::barrier<> is a reusable barrier.
- Barriers are great for loop synchronization between parallel tasks.
- The **completion function** allows you to do something between loops: pass the result on to another step, write to a file, etc.
- Synchronization is done in **phases**:

1. Construct a barrier, with a non-zero count and a **completion function**
2. One or more threads arrive at the barrier
3. These or other threads wait for the barrier to be signalled
4. When the count reaches zero, the barrier is signalled, the **completion function** is called on one of the thread and the count is reset

```
unsigned const num_threads=...;
void finish_task();
std::barrier<std::function<void()>>
b(
num_threads,finish_task);
void worker_thread(
  std::stop_token st,unsigned i){
while(!st.stop_requested()){
  do_stuff(i);
  b.arrive_and_wait();
  }
}
```

Semaphore is a very low level machine to build anything

- A semaphore represents a number of available "slots". If you **acquire** a slot on the semaphore then the count is decreased until you **release** the slot.
- Acquire decrease count, release increase count
- Attempting to acquire a slot when the count is zero will either block or fail.
- A thread may release a slot without acquiring one and vice versa.
- Semaphores can be used to build just about any synchronization mechanism, including latches, barriers and mutexes.
- A **binary semaphore** has 2 states: 1 slot free or no slots free.
- It can be used as a mutex.

- C++20 has std::counting_semaphore<max_count>
- std::binary_semaphore is an alias for
- std::counting_semaphore<1>.
- As well as **blocking** sem.acquire(), there are also sem.try_acquire(), sem.try_acquire_for() and sem.try_acquire_until() functions that fail instead of blocking

```
std::counting_semaphore<5> slots(5);

void func(){

 slots.acquire();

 do_stuff(); // at most 5 threads can be here

 slots.release();

}
```

Atomics

- Low-level waiting for atomics
- Atomic Smart Pointers
- std::atomic_ref

Waiting for atomics

- std::atomic<T> now provides a var.wait() member function to wait for it to change.
- var.notify_one() and var.notify_all() wake one or all threads blocked in wait().
- Like a low level std::condition_variable.

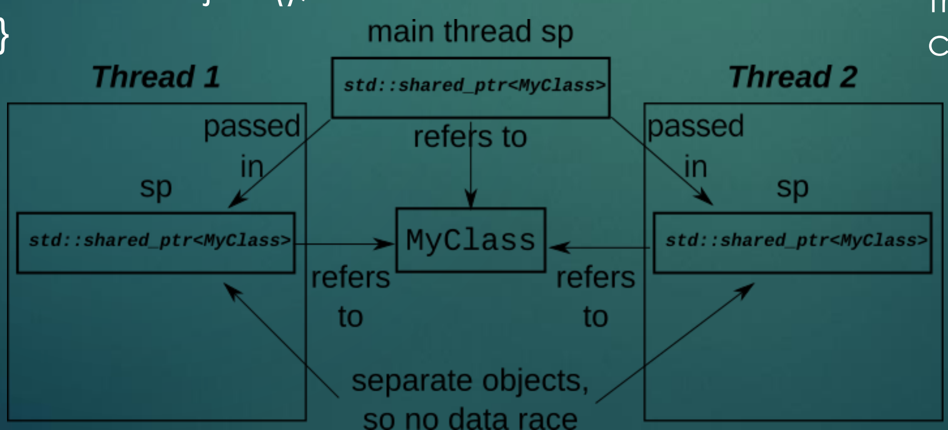
C++11 Smart Pointers

```
class MyList{
    shared_ptr<Node> head;
    ...
    void pop_front(){
    std::shared_ptr<Node>
    p=head;
    while(p &&
    !atomic_compare_exchange
    _strong(&head, &p, p));
    }
};
```

- Error-prone: all access to *head* must go through *atomic_xxx*.
- Inefficient: *atomic_compare_exchange_strong* is a free function taking regular *shared_ptr*, we don't want extra synchronization in *shared_ptr*!

Std::shared_ptr with multiple threads

```
class MyClass;
    void thread_func(std::shared_ptr<MyClass> sp){
        sp->do_stuff();
        std::shared_ptr<MyClass> sp2=sp;
        do_stuff_with(sp2);
    }
int main(){
        std::shared_ptr<MyClass> sp(new MyClass);
        std::thread thread1(thread_func,sp);
        std::thread thread2(thread_func,sp);
        thread2.join();
        thread1.join();
    }
```



- std::shared_ptr works great in multiple threads, *provided each thread has its own copy or copies.*
  - changes to the reference count are synchronized,
  - everything just works, if your shared data is correctly synchronized.
- you need to ensure that it is safe to call MyClass::do_stuff() and do_stuff_with() from multiple threads concurrently on the same instance, but the reference counts are handled OK.

Sharing a std::shared_ptr instance between threads

If we're going to access this from 2 threads, then we have a choice:

1. We could wrap the whole object with a mutex, so only one thread is accessing the list at a time, or
2. **We could try and allow concurrent accesses. But there are problems:**
   a. removing from the front of the list
   b. Race condition on head
   c. Multiple threads calling pop_front

Atomic<shared_ptr<T>>

```
class MyList{
    atomic<shared_ptr<Node>>
    head;
    ...
    void pop_front(){
    std::shared_ptr<Node>
    p=head;
    while(p &&
    !head.compare_exchange_st
    rong(p,p->next));
    }
};
```

- Guaranteed atomic access
- Can be implemented more efficiently

Atomic<shared_ptr<T>>:Just Threads, Anthony Williams

- implementations *may* use a mutex to provide the synchronization in atomic_shared_ptr
- may also manage to make it lock-free
- can be tested using the is_lock_free() member function
- with a lock-free atomic<shared_ptr<t.> using a split reference count for atomic<shared_ptr<T>>
  - double -word compare and swap operation
  - the shared_ptr control block holds a count of "external counters" in addition to the normal reference count, and each atomic_shared_ptr instance that holds a reference has a local count of threads accessing it concurrently.

Atomic smart pointer

- C++20 provides std::atomic<std::shared_ptr<T>> and std::atomic<std::weak_ptr<T>> specializations.
- May or may not be **lock-free**
- If lock-free, can simplify lock-free algorithms.
- If not lock-free, a better replacement for std::shared_ptr<T> and a mutex.
- Can be slow under high contention.

```
template<typename T> class stack{
struct node{
  T value;
  shared_ptr<node> next;
  node(){} node(T&&
nv):value(std::move(nv)){}
};
  std::atomic<shared_ptr<node>> head;
public:
  stack():head(nullptr){}
  ~stack(){ while(head.load()) pop(); }
  void push(T);
  T pop();
};
```

Beyond performance, you also need to choose from other properties of lock-free programming	Reference Counting	Reference Counting with DCAS	RCU	Hazard Pointers
Unreclaimed objects	**Bounded**	**Bounded**	Unbounded	**Bounded**
Non-blocking traversal	Either blocking or lock free with limited reclamation	**Lock free**	**Bounded population oblivious wait free**	**Lock free.**
Non-blocking reclamation (no memory allocator)	Either blocking or lock free with limited reclamation	**Lock free**	Blocking	**Bounded wait free**
Traversal speed	Atomic RMW updates	Atomic RMW updates	**No or low overhead**	Store-load fence
Reference acquisition	**Unconditional**	**Unconditional**	**Unconditional**	Conditional
Contention among readers	Can be very high	Can be very high	**No contention**	**No contention**
Automatic reclamation	**Yes**	**Yes**	No	No
Domain meaning	N/A		Isolate long-latency readers	Limit contention, reduce space bounds, etc.

Atomic_ref

- std::atomic_ref allows you to perform atomic operations on non-atomic objects.
- This can be important when sharing headers with C code, or where a struct needs to match a specific binary layout so you can't use std::atomic.
- **If you use std::atomic_ref to access an object, all accesses to that objec must use std::atomic_ref**.

```
struct my_c_struct{
  int count;
  data* ptr;
};
void    do_stuff(my_c_struct* p){
  std::atomic_ref<int>
count_ref(p->count);
  ++count_ref;
// ...
}
```

coroutines

- A **coroutine** is a function that can be **suspended** mid execution and **resumed** at a later time.
- Resuming a coroutine continues from the suspension point;
- local variables have their values from the original call
- C++20 provides **stackless coroutines**
- Only the locals for the current function are saved
- Everything is localized
- Minimal memory allocation — can have millions of in-flight coroutines
- Whole coroutine overhead can be eliminated by the compiler — Gor's "disappearing coroutines"

```
future<remote_data>
async_get_data(key_type key);
future<data> retrieve_data(
key_type key){
  auto rem_data=
co_await     async_get_data(key);
  co_return process(rem_data);
}
```

Take away

- C++ is pushing towards increasing concurrency facilities and
- Further Heterogeneous device programming
- Adding Study Groups for Machine Learning, Graphics, Education, Linear Algebra, Low Latency
- C++ is good for AI and ML and still works for Legacy code
- C++20 will be MAJOR MAJOR release

What is in C++ 20

	Depends on	Current target (estimated, could slip)
Concepts		C++20 (adopted, including convenience syntax)
Contracts		~~C++20 (adopted)~~
Ranges		C++20 (adopted)
Coroutines		C++20
Modules		C++20
Reflection		TS in C++20 timeframe, IS in C++23
Executors		~~Lite in C++20~~ timeframe, Full in C++23
Networking	Executors, and possibly Coroutines	C++23
future.then, async2	Executors	

Act 1

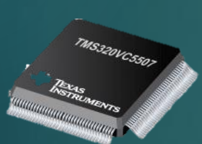
- What is coming for C++ 23

invoke   async   parallel algorithms   future::then   post

defer   define_task_block   dispatch   asynchronous operations   strand<>

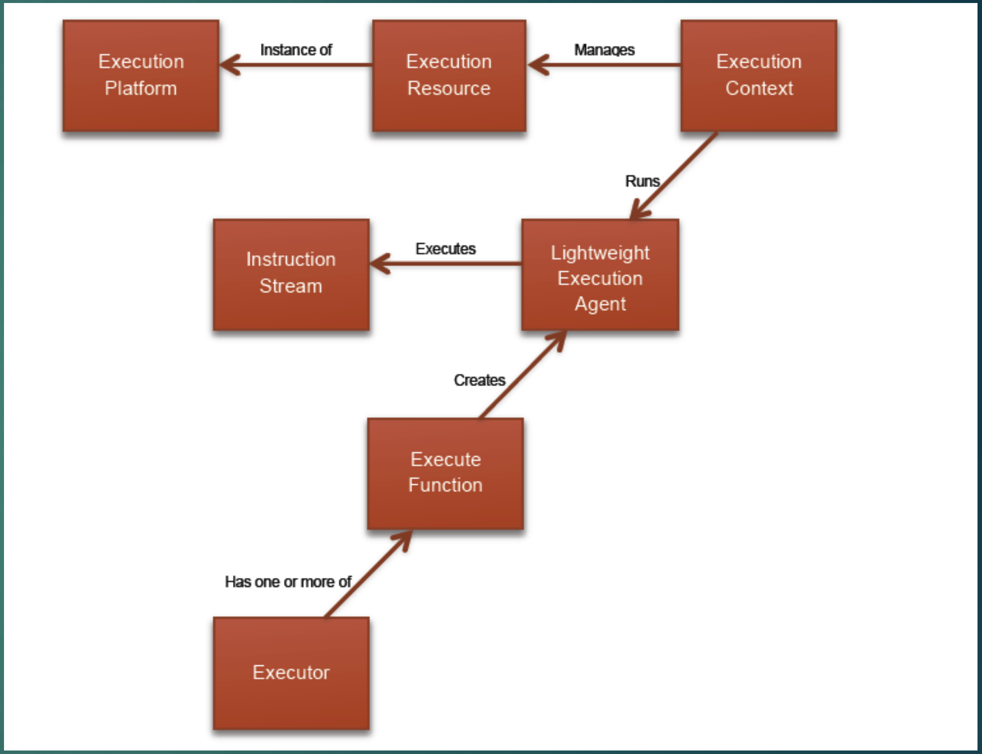
Unified interface for execution

SYCL / OpenCL / CUDA / HCC	OpenMP / MPI	C++ Thread Pool	Boost.Asio / Networking TS



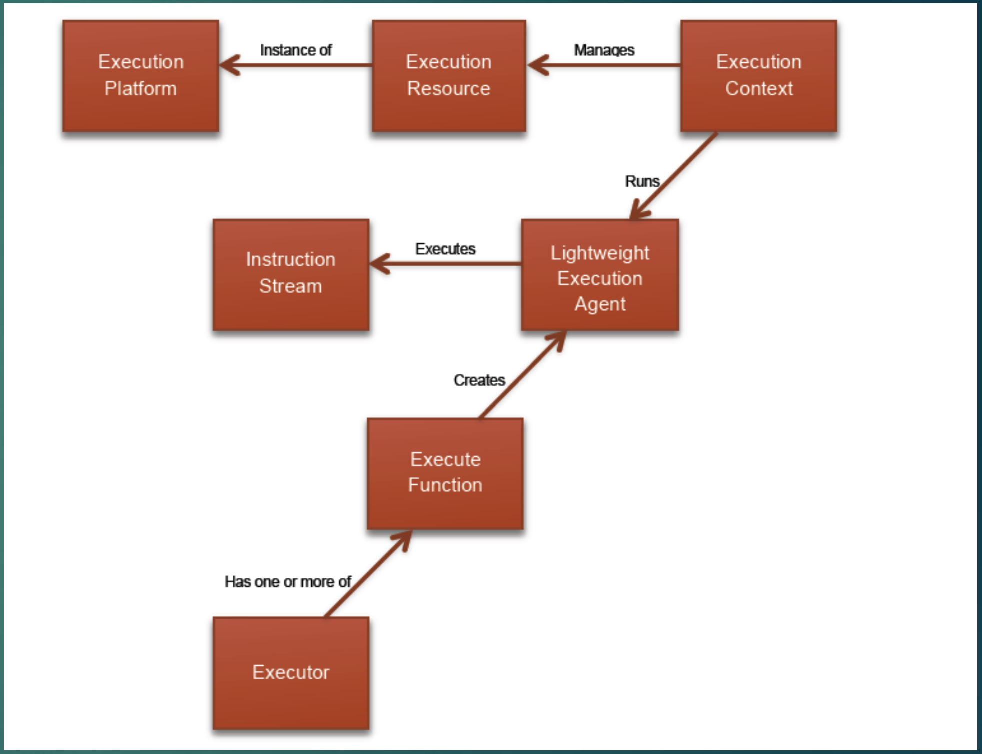
Current Progress of Executors

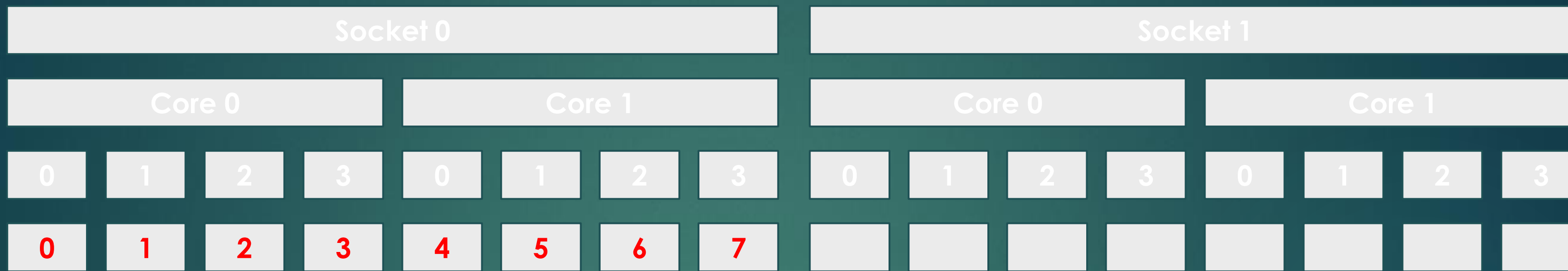
- An ***instruction stream*** is the function you want to execute
- An ***executor*** is an interface that describes where and when to run an ***instruction stream***
- An ***executor*** has one or more ***execute functions***
- An ***execute function*** executes an ***instruction stream*** on light 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






```
{
  auto exec = execution::execution_context{execRes}.executor();

  auto affExec = execution::require(exec, execution::bulk,
    execution::bulk_execution_affinity.compact);

  affExec.bulk_execute([](std::size_t i, shared s) {
    func(i);
  }, 8, sharedFactory);
}
```

Parallel/Concurrency beyond C++20: C++23

	Asynchronus Agents	Concurrent collections	Mutable shared state	Heterogeneous/DIstributed
today's abstractions	C++11: thread,lambda function, TLS, async C++14: generic lambda C++ 20: Jthreads +interrupt _token C++23: networking, asynchronous algorithm, reactive programming, EALS, async2, executors	C++11: Async, packaged tasks, promises, futures, atomics, C++ 17: ParallelSTL, control false sharing C++ 20: Is_ready(), make_ready_future(), simd<T>, Vec execution policy, Algorithm un-sequenced policy span C++23: new futures, concurrent vector,task blocks, unordered associative containers, two-way executors with lazy sender-receiver	C++11: … C++ 14: … C++ 17: … C++20: atomic_ref, Latches and barriers atomic<shared_ptr> Atomics & padding bits Simplified atomic init Atomic C/C++ compatibility Semaphores and waiting Fixed gaps in memory model , Improved atomic flags , Repair memory model C++23: hazard_pointers, rcu/snapshot, concurrent queues, counters, upgrade lock, TM lite, more	C++17: , progress guarantees, TOE, execution policies C++20: atomic_ref, mdspan, C++23: affinity, pipelines, EALS, freestanding/embedded support well specified, mapreduce, ML/AI, reactive programming executors, mdspan

C++23

- Library support for coroutines
- Executors
- Networking
- A modular standard library

After C++20

- Much more libraries
  - Audio
  - Linear Algebra
  - Graph data structure
  - Tree Data structures
  - Task Graphs
  - Differentiation
  - Reflection
    - IPR paper
    - https://github.com/GabrielDosReis/ipr

A Principled, Complete, and Efficient Representation of C++

Gabriel Dos Reis and Bjarne Stroustrup

Abstract. We present a systematic representation of C++, called IPR, for complete semantic analysis and semantics-based program transformations. We describe the ideas and design principles that shaped the IPR. In particular, we describe how general type-based unification is key to minimal compact representation, fast type-safe traversal, and scalability. For example, the representation of a fairly typical non-trivial C++ program in GCC 3.4.2 was 32 times larger than its IPR representation; this led to significant improvements to GCC. IPR is general enough to handle real-world programs involving many translation units, archaic programming styles, and generic programming using C++0x extensions that affect the type system. The difficult issue of how to represent irregular (ad hoc) features in a systematic (non ad hoc) manner is among the key contributions of this paper. The IPR data structure can represent all of C++ with just 157 simple node types; to compare the ISO C++ grammar has over 700 productions. The IPR is used for a variety of program analysis and transformation tasks, such as visualization, loop simplification, and concept extraction. Finally, we report impacts of this work on existing C++ compilers.

1. Introduction

The C++ programming language [16] is a general-purpose programming language, with bias toward system programming. It has, for the last two decades, been widely used in diverse application areas [32, 33, 31]. Besides traditional applications of general-purpose programming languages, it is being used in high-performance com-

After C++23

- Reflection
- Pattern matching
- C++ ecosystem

Use the Proper Abstraction with C++

Abstraction	How is it supported
Cores	C++11/14/17 threads, async
HW threads	C++11/14/17 threads, async
Vectors	Parallelism TS2->C++20
Atomic, Fences, lockfree, futures, counters, transactions	C++11/14/17 atomics, Concurrency TS1->C++20, Transactional Memory TS1
Parallel Loops	Async, TBB:parallel_invoke, C++17 parallel algorithms, for_each
Heterogeneous offload, fpga	OpenCL, SYCL, HSA, OpenMP/ACC, Kokkos, Raja P0796 on affinity
Distributed	HPX, MPI, UPC++ P0796 on affinity
Caches	C++17 false sharing support
Numa	Executors, Execution Context, Affinity, P0443->Executor TS
TLS	EALS, P0772

If you have to remember 3 things

1

Expose more parallelism

2

Increase Locality of reference

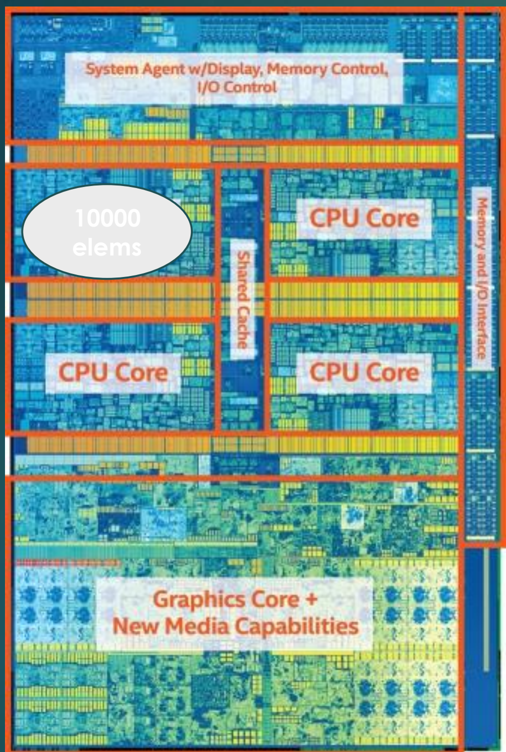
3

Use Heterogeneous C++ today

THE HETEROGENEOUS SYSTEMS EXPERTS

Oh one more thing!
SYCL Parallel STL today

What can I do with a Parallel For Each?

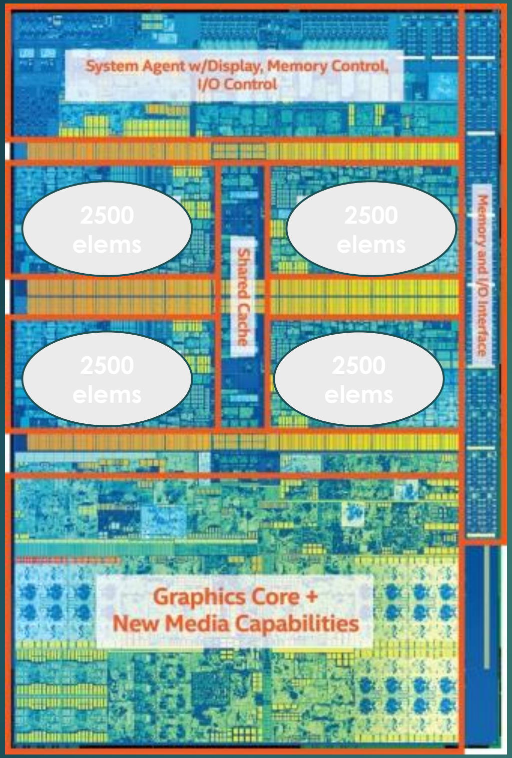


Intel Core i7 7th generation

```
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?



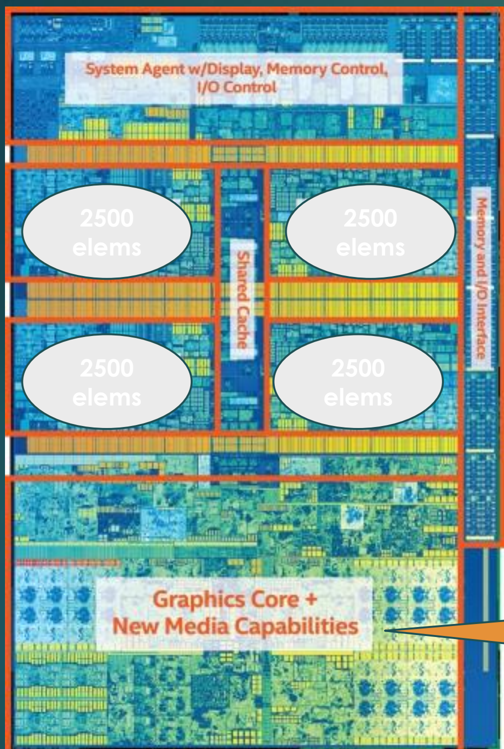
Intel Core i7 7th generation

```
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

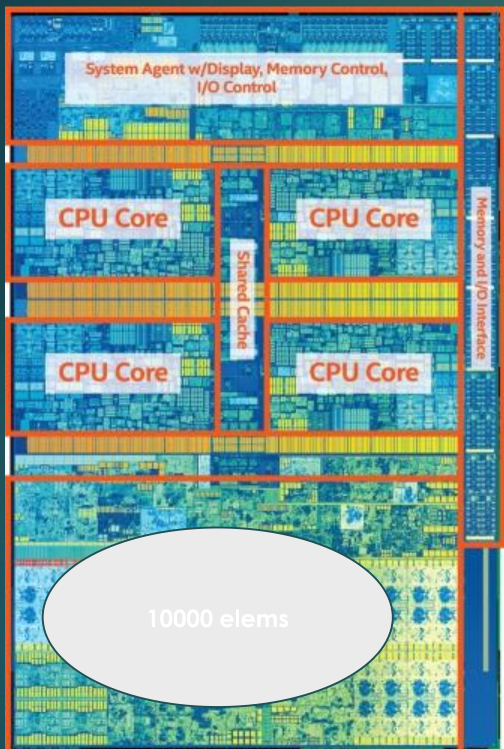
```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

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)

What can I do with a Parallel For Each?



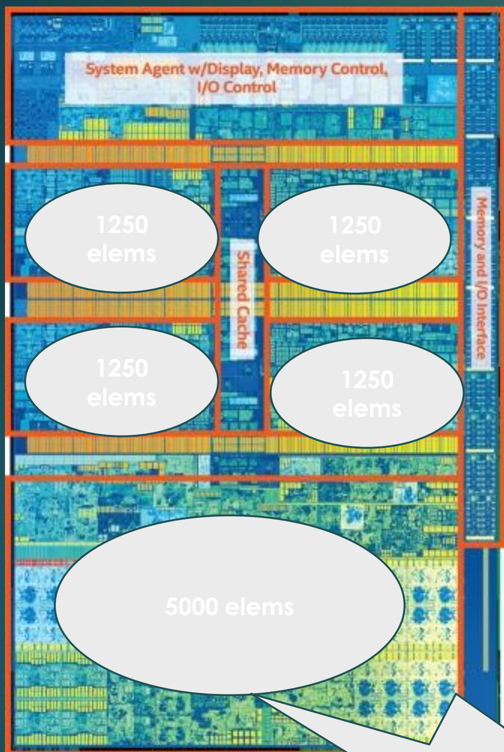
Intel Core i7 7th generation

```
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?



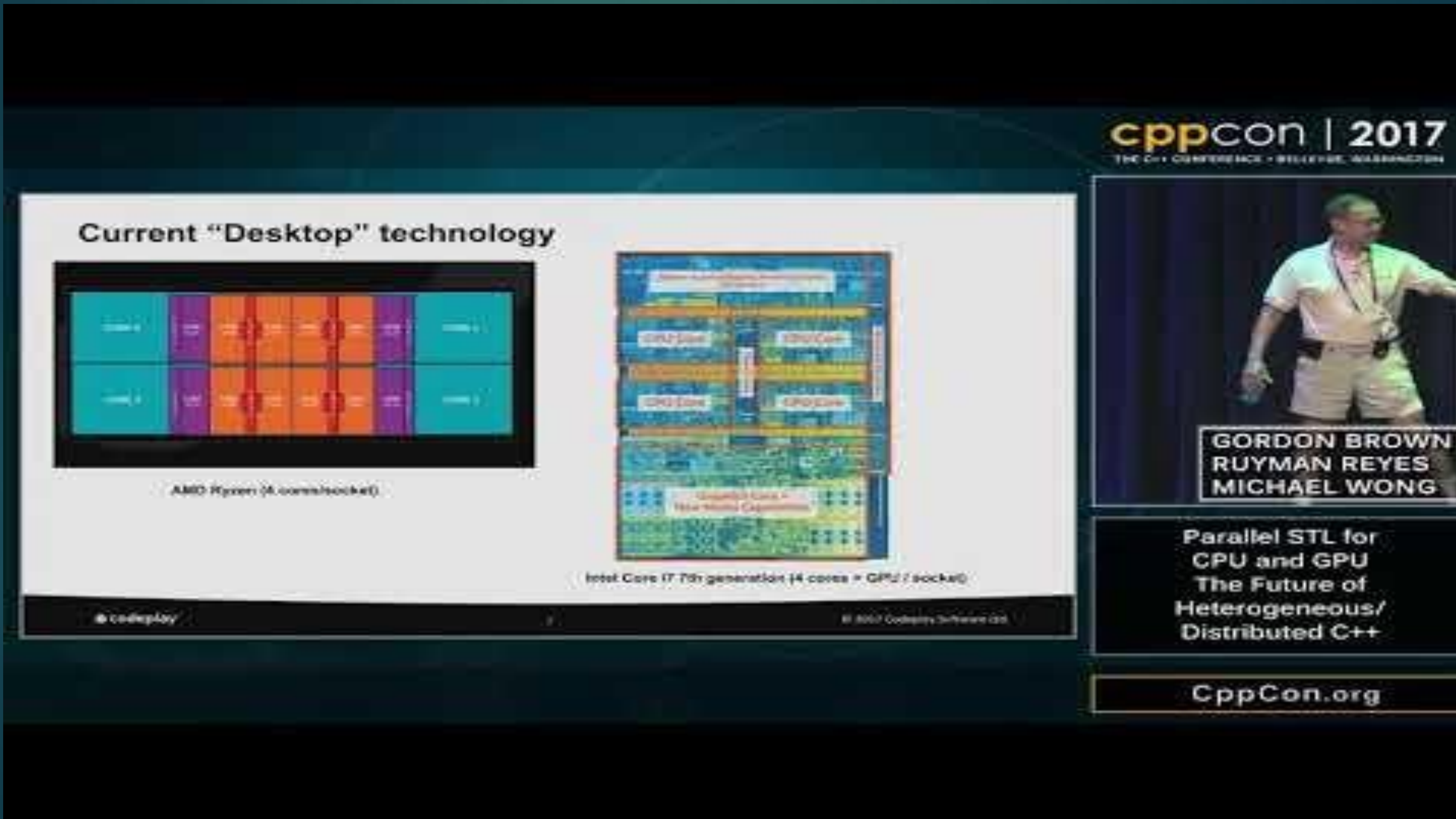
Intel Core i7 7th

```
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!

Experimental!

(mileage may vary, implementation-specific behaviour)



Current “Desktop” technology
AMD Ryzen (4 cores/socket)
Intel Core i7 7th generation (4 cores + GPU / socket)
cppcon | 2017
GORDON BROWN
RUYMAN REYES
MICHAEL WONG
Parallel STL for
CPU and GPU
The Future of
Heterogeneous/
Distributed C++
CppCon.org

Demo Results - Running std::sort (Running on Intel i7 6600 CPU & Intel HD Graphics 520)

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

SYCL Ecosystem

- ComputeCpp - https://codeplay.com/products/computesuite/computecpp
- triSYCL - https://github.com/triSYCL/triSYCL
- SYCL - http://sycl.tech
- SYCL ParallelSTL - https://github.com/KhronosGroup/SyclParallelSTL
- VisionCpp - https://github.com/codeplaysoftware/visioncpp
- SYCL-BLAS - https://github.com/codeplaysoftware/sycl-blas
- TensorFlow-SYCL - https://github.com/codeplaysoftware/tensorflow
- Eigen http://eigen.tuxfamily.org

Eigen Linear Algebra Library

- SYCL backend in mainline
- Focused on Tensor support, providing support for machine learning/CNNs
- Equivalent coverage to CUDA
- Working on optimization for various hardware architectures (CPU, desktop and mobile GPUs)
- https://bitbucket.org/eigen/eigen/

TensorFlow

- SYCL backend support for all major CNN operations
- Complete coverage for major image recognition networks
  - GoogLeNet, Inception-v2, Inception-v3, ResNet, ….
- Ongoing work to reach 100% operator coverage and optimization for various hardware architectures (CPU, desktop and mobile GPUs)
- https://github.com/tensorflow/tensorflow

SYCL Ecosystem

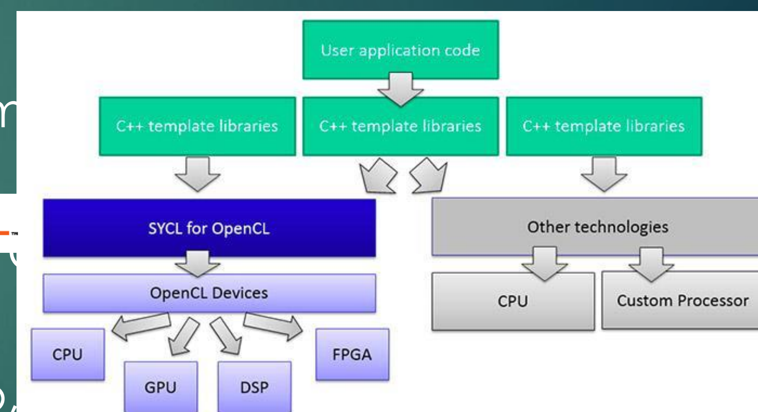
- Single-source heterogeneous programming using STANDARD C++
  - Use C++ templates and lambda functions for host & device code
  - Layered over OpenCL
- Fast and powerful path for bring C++ apps and libraries to OpenCL
  - C++ Kernel Fusion - better perform
    hand-coding
  - Halide, Eigen, Boost.Compute, S
    TensorFlow, SYCL GTX
  - CL, ComputeCpp, VisionCpp,
- M nformation at http://sycl.tech

Developer Choice

The development of the two specifications are aligned so code can be easily shared between the two approaches

C++ Kernel Language
Low Level Control
'GPGPU'-style separation of device-side kernel source code and host code

Single-source C++
Programmer Familiarity
Approach also taken by C++ AMP and OpenMP



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

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

SYCL™

ComputeCpp™

Community Edition

Available now for free!

Visit:

computecpp.codeplay.com

SYCL™

ComputeCpp™

- 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 ?