codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

Parallelising the Modern C++ Standard Library

Christopher Di Bella
Staff Software Engineer, ComputeCpp Runtime

GPU Day, Budapest, 2018-06-21

Co-authors

- Gordon Brown
- Michael Haidl
- Toomas Remmelg
- Ruyman Reyes
- Michel Steuwer
- Michael Wong

Agenda

1. STL Crash Course
2. SAXPY
3. Problems with Parallel STL
4. Views
5. Parallel Ranges

Motivation

- Programming parallel and heterogeneous systems is hard
- Traditionally requires the use of low level APIs
- Optimising is even harder
- Parallel STL can simplify programming

STL Crash Course: Containers

- Data structures that own collections of objects
  - `std::vector<T>`
  - `std::list<T>`
  - `std::map<K, V>`

- "Ownership" means controlling allocation and deallocation

STL Crash Course: Algorithms

- Describe operations on containers and other sequences
- Three kinds:
  - Non-modifying (e.g. `std::find`, `std::accumulate`)
  - Modifying (e.g. `std::transform`, `std::iota`)
  - Sorting (e.g. `std::sort`)

STL Crash Course: Iterators

- Containers have different internal layouts
- Iterators are an abstraction for addressing elements
- Operations
- Different containers have different iterator concepts

STL Crash Course: Ranges

- Abstraction of containers
- `begin(rng)` returns an iterator to the first element in `rng`.
- `end(rng)` returns a sentinel to the range `rng`.
- `[begin(rng), end(rng))` denotes a range.

STL Crash Course: Usage

Iterator-based algorithm (from C++98 on)	Range-based algorithm (C++20, proposed)

`auto v = std::vector{/*...*/};`
`if (auto i = find(cbegin(v), cend(v), x);`
`     i != cend(v)) {`
`   std::cout << *i << '\n';`
`}`
`else {`
`   std::cerr << "Not found.\n";`
`}`
	`auto v = std::vector{/*...*/};`
	`if (auto i = find(v, x); i != cend(v)) {`
	`   std::cout << *i << '\n';`
	`}`
	`else {`
	`   std::cerr << "Not found.\n";`
	`}`

Note: both still operate on a range!

Example: SAXPY

Level 1 BLAS primitive

single-precision **a** times **x** **p**lus **y**

$\mathbf{y} = \alpha \mathbf{x} + \mathbf{y}$

STL Crash Course: SAXPY

```cpp
std::vector<float> x = // ...
std::vector<float> y = // ...
float a =              // ...

std::vector<float> out(size(x));

std::transform(cbegin(x), cend(x), begin(out), [a](float x){ return a * x; });
std::transform(cbegin(out), cend(out), cbegin(y), begin(out), std::plus<>{});
```

```cpp
std::ranges::transform(x, begin(out), [a](float x) { return a * x; });
std::ranges::transform(out, y, begin(out), std::plus<>{});
```

STL Crash Course: Parallel Algorithms

- Overloaded algorithms
- Take an extra argument
- Same semantics

```
std::transform(std::execution::par, cbegin(x), cend(x),
    begin(out), [a](float x) { return a * x; });
std::transform(std::execution::par_unseq, cbegin(out), cend(out),
    begin(y), begin(out), std::plus<>{});
```

- We will use `parallel::` to indicate the parallel algorithms.

Problems with the STL interface

```
std::vector<float> x = // ...
std::vector<float> y = // ...
float a =              // ...

std::vector<float> out(size(x));

{
  cl::sycl::queue q;

  std::vector<float> tmp(size(x));
  sycl::sycl_execution_policy<class Scale> exec1(q);
  parallel::transform(exec1, cbegin(x), cend(x), begin(tmp),
                      [a](float x) { return a * x; });

  sycl::sycl_execution_policy<class Add> exec2(q);
  parallel::transform(exec2, cbegin(tmp), cend(tmp), cbegin(y), begin(out),
                      std::plus<>{});
}
```

Copy to device

Copy from device

Copy to device

Copy from device

Problems with the STL interface

```
{
  cl::sycl::queue q;

  cl::sycl::buffer<float> x_buff(data(x), size(x));
  cl::sycl::buffer<float> y_buff(data(y), size(y));

  cl::sycl::buffer<float> tmp_buff(size(x));
  sycl::sycl_execution_policy<class Scale> exec1(q);
  parallel::transform(exec1, begin(x_buff), end(x_buff), begin(tmp_buff),
                      [a](float x) { return a * x; });

  cl::sycl::buffer<float> out_buff(data(out), size(size));
  sycl::sycl_execution_policy<class Add> exec2(q);
  parallel::transform(exec2, begin(tmp_buff), end(tmp_buff), begin(y_buff),
                      begin(out_buff), std::plus<>{});
} // data copied back after exiting the scope
```

Explicitly create SYCL buffers

Pass iterators to buffers

Copy to device

Still needs temporary storage

Copy from device

2 Kernels

Problems with the STL interface

```
{
  cl::sycl::queue q;

  cl::sycl::buffer<float> x_buff(data(x), size(x));
  cl::sycl::buffer<float> y_buff(data(y), size(y));

  cl::sycl::buffer<float> out_buff(data(out), size(out));
  sycl::sycl_execution_policy<class Saxpy> exec(q);
  parallel::transform(exec, begin(x_buff), end(x_buff), begin(y_buff),
                      begin(out_buff),
                      [a](float x, float y) { return a * x + y; });
} // data copied back after exiting the scope
```

Manually fused kernel

Problems with the STL interface

- Not composable
- Need to be aware of all appropriate functions
- Performance hits otherwise
- Not always a predefined function "on hand"

Views

- Special kind of range
- Constant-time operations on a range
  - Copying
  - Moving
  - Assignment
- Typically lazy in nature
  - This talk only cares about the lazy ones

Views

<table>
<tr><th>Without views</th><th>With views</th></tr>
<tr><td>

```
auto v = std::vector{/*...*/};
if (auto i = find(crbegin(v), crend(v), x);
     i != crend(v)) {
   std::cout << *i << '\n';
}
else {
   std::cerr << "Not found.\n";
}
```

</td><td>

```
auto v = std::vector{/*...*/};
if (auto i = find(v | view::reverse, x);
     i != crend(v)) {
   std::cout << *i << '\n';
}
else {
   std::cerr << "Not found.\n";
}
```

</td></tr>
</table>

Example with Views

```
auto plus = [](auto const& pair) {
   return get<0>(pair) + get<1>(pair); };
auto mult = [](auto const& pair) {
   return get<0>(pair) * get<1>(pair); };

// saxpy using range-v3
auto ax = ranges::view::zip(view::repeat(a), x)
        | ranges::view::transform(mult);
auto out = ranges::view::zip(ax, y)
         | ranges::view::transform(plus)
         | ranges::to_vector;
```

Lambdas for describing behaviour

Composition operator

Materialise the result as views are lazy

Prototype of SYCL Parallel STL with Ranges

- Using
  - ComputeCpp SYCL implementation
  - C++11 compatible range-v3

Open Source on GitHub: git.io/vA5H9

Example with SYCL and Views

```
// saxpy using sycl & range-v3
gstorm::sycl_exec exec;

using parallel::copy;
auto ax = ranges::view::zip(ranges::view::repeat(a), copy(exec, x))
        | ranges::view::transform(mult);
auto z = ranges::view::zip(ax, copy(exec, y))
       | ranges::view::transform(plus);
parallel::transform(exec, z, copy(exec, out), identity);
```

Create SYCL compatible ranges

Materialise the result

Views don't perform computation
No policy needed

5 views, but 1 algorithm ⇔ 1 kernel

Example with SYCL and Views

```
define spir_kernel void @SYCL_saxpy(i64, %"..."* byval nocapture, float addrspace(1)*, i64, float
addrspace(1)*, i64, float addrspace(1)*, i64) #1 {
  %9 = tail call spir_func i64 @_Z13get_global_idj(i32 0) #0
  %10 = icmp ult i64 %9, %0
  br i1 %10, label %11, label %24
; <label>:11                                      ; preds = %8
  %12 = getelementptr inbounds %"...", %"..."* %1, i64 0, i32 0, i32 0, i32 0
  %13 = load float, float* %12, align 4
  %14 = add i64 %9, %3
  %15 = add i64 %9, %5
  %16 = getelementptr inbounds float, float addrspace(1)* %2, i64 %14
  %17 = load float, float addrspace(1)* %16, align 4, !tbaa !11, !noalias !15
  %18 = fmul float %13, %17
  %19 = getelementptr inbounds float, float addrspace(1)* %4, i64 %15
  %20 = load float, float addrspace(1)* %19, align 4, !tbaa !11
  %21 = fadd float %18, %20
  %22 = add i64 %9, %7
  %23 = getelementptr inbounds float, float addrspace(1)* %6, i64 %22
  store float %21, float addrspace(1)* %23, align 4, !tbaa !11
  br label %24
; <label>:24                                      ; preds = %11, %8
  ret void }
```

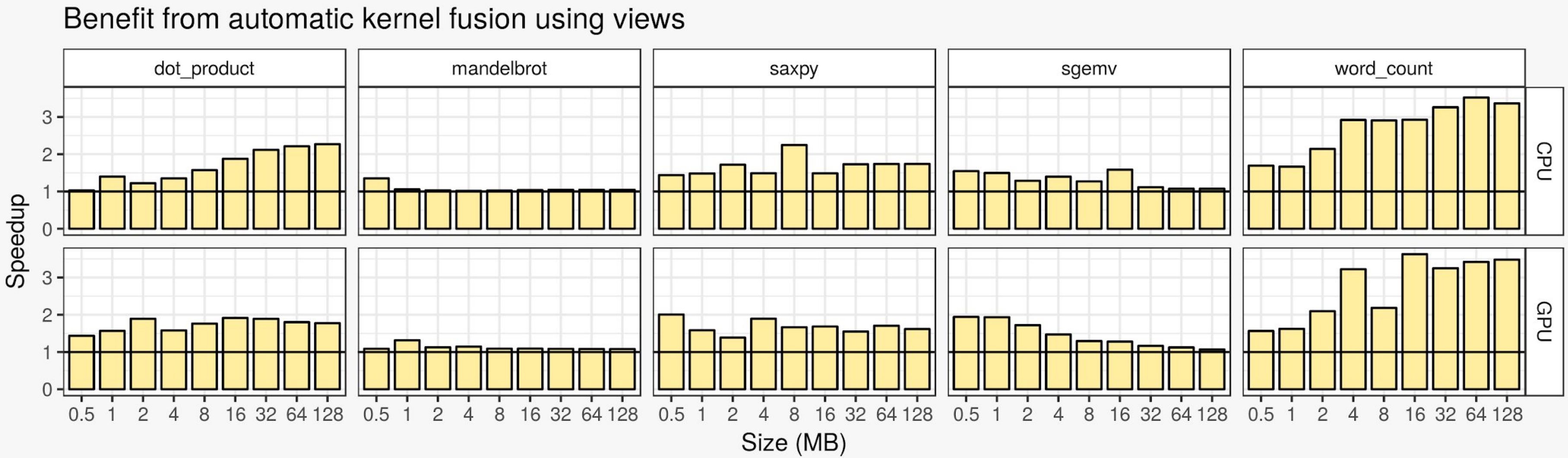
load a

load x[i]

load y[i]

store out[i]

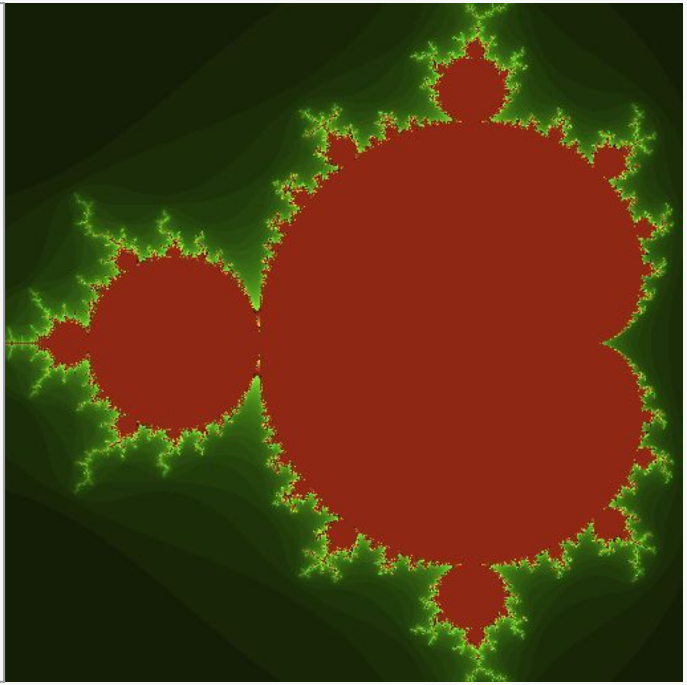
Example with SYCL and Views



Intel i7-6700K CPU and Intel HD Graphics 530 GPU
Using the zero-copy functionality
Execution time includes buffer creation and queuing overheads
Speedups calculated from median execution times of 100 runs per experiment

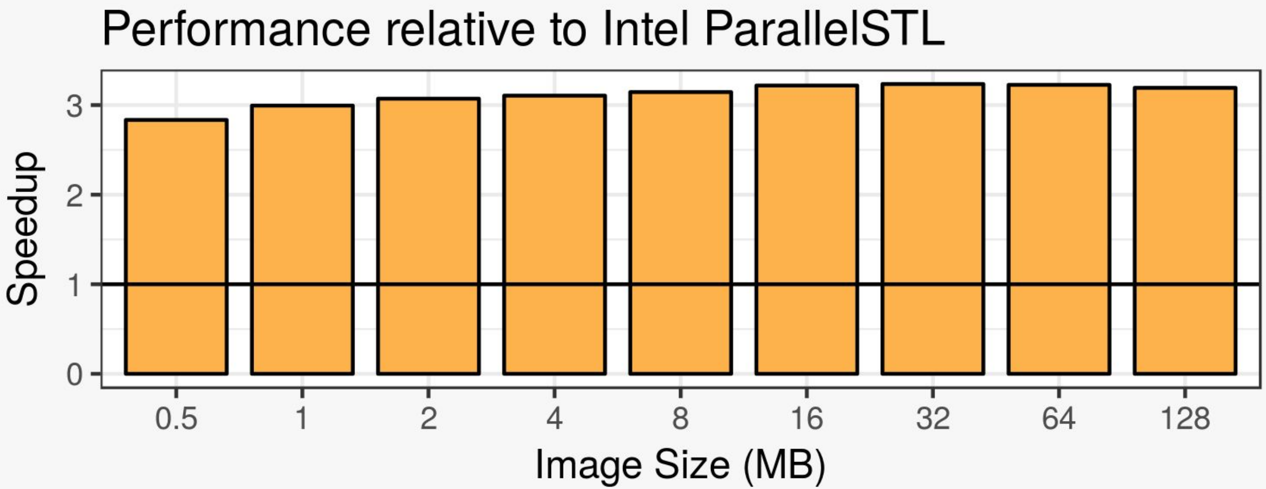
What if there is no predefined function?

```
const auto height = 512;
const auto width = 512;
const auto iterations = 100;
std::vector<pixel> image(height * width);

{
 gstorm::sycl_exec exec;
 auto gpu_image = parallel::copy(exec, image);

 auto indices = ranges::view::iota(0)
              | ranges::view::take(size(image));
 parallel::transform(exec, indices, gpu_image,
            CalculatePixel{height, width, iterations});
}
```



What if there is no predefined function?

- No `std::iota` with `std::transform` & no parallel `std::iota`
- Mandelbrot Intel PSTL vs SYCL Ranges
- Speedup for free by using views!



Future work

- We will continue to explore parallel algorithms with ranges and fusion.
- We would like to explore data layout transformations and concept definitions for parallel algorithms.
- We would like to investigate ways to refine std::tuple as standard-layout for heterogeneous programming.

- [Parallel Programming with Modern C++: From CPU to GPU]
- [Generic Programming 2.0 with Concepts and Ranges]

Conclusion

- Ranges, Views and Actions further simplify exploiting parallel systems
- Write programs in a more composable style
- Potential speedups where not possible before

GitHub: git.io/vA5H9

Paper: wg21.link/P0836R0

We're Hiring!
codeplay.com/careers/

Thanks for Listening

@codeplaysoft

info@codeplay.com

codeplay.com