codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

Introducing Parallelism to the Ranges TS

Gordon Brown, Christopher Di Bella, Michael Haidl, **Toomas Remmelg**,
Ruyman Reyes, Michel Steuwer

Distributed & Heterogeneous Programming in C/C++, Oxford, 14/05/2018

Introduction

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

Example: SAXPY

Level 1 BLAS primitive

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

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

Problems with the STL interface

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

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

{
  cl::sycl::queue q;

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

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

Copy to device

Copy from device

Copy to device

Copy from device

Problems with the STL interface

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

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

{
  cl::sycl::queue q;

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

  cl::sycl::buffer<float> tmp_buff(x.size());
  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(out.data(), out.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
```

2 Kernels

Explicitly create SYCL buffers

Pass iterators to buffers

Copy to device

Still needs temporary storage

Copy from device

Problems with the STL interface

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

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

{
  cl::sycl::queue q;

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

  cl::sycl::buffer<float> out_buff(out.data(), out.size());
  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”

Ranges, Views and Actions

- Ranges
- Views
  - Lazily perform actions on Ranges
- Actions
  - Eagerly perform actions on Ranges

Example with Views

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

auto plus = [](auto pair) { return std::get<0>(pair) + std::get<1>(pair); };
auto mult = [](auto pair) { return std::get<0>(pair) * std::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::transform(plus)
         | ranges::to_vector;
```

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](git.io/vA5H9)

Example with SYCL and Views

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

auto plus = [](auto pair) { return std::get<0>(pair) + std::get<1>(pair); };
auto mult = [](auto pair) { return std::get<0>(pair) * std::get<1>(pair); };
auto identity = [](auto x) { return x; };

std::vector<float> out(x.size());
{
 // saxpy using sycl & range-v3
 gstorm::sycl_exec exec;

 using std::experimental::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);
 std::experimental::transform(exec, z, copy(exec, out), identity);
}
```

Create SYCL compatible ranges

Materialise the result

Views don't perform computation

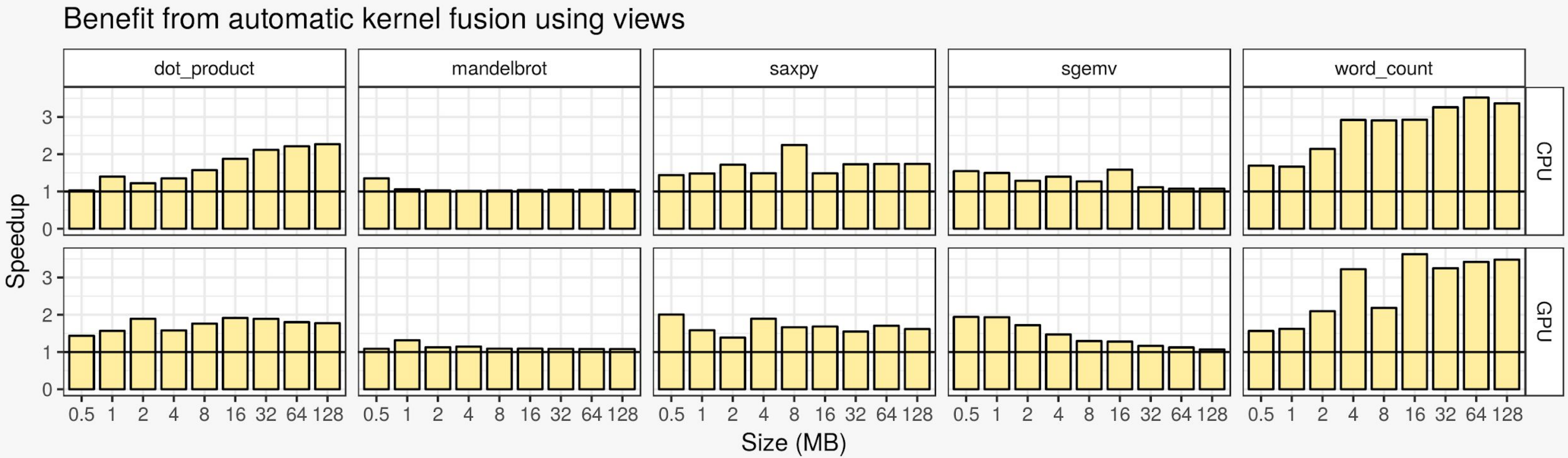
No policy needed

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                                  <- load a
  %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   <- load x[i]
  %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          <- load y[i]
  %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           <- store out[i]
  br label %24
; <label>:24                                      ; preds = %11, %8
  ret void }
```

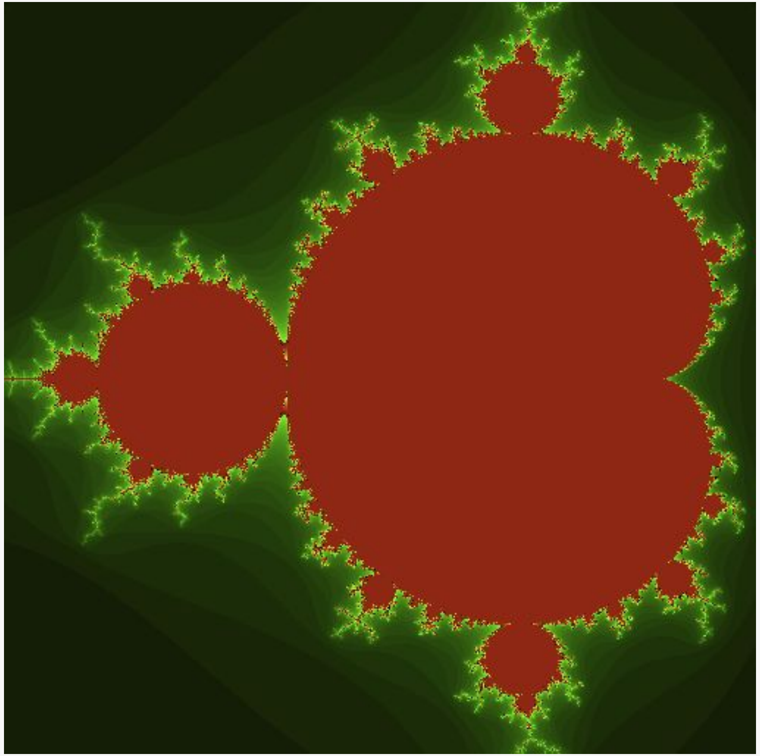
Example with SYCL and Views



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

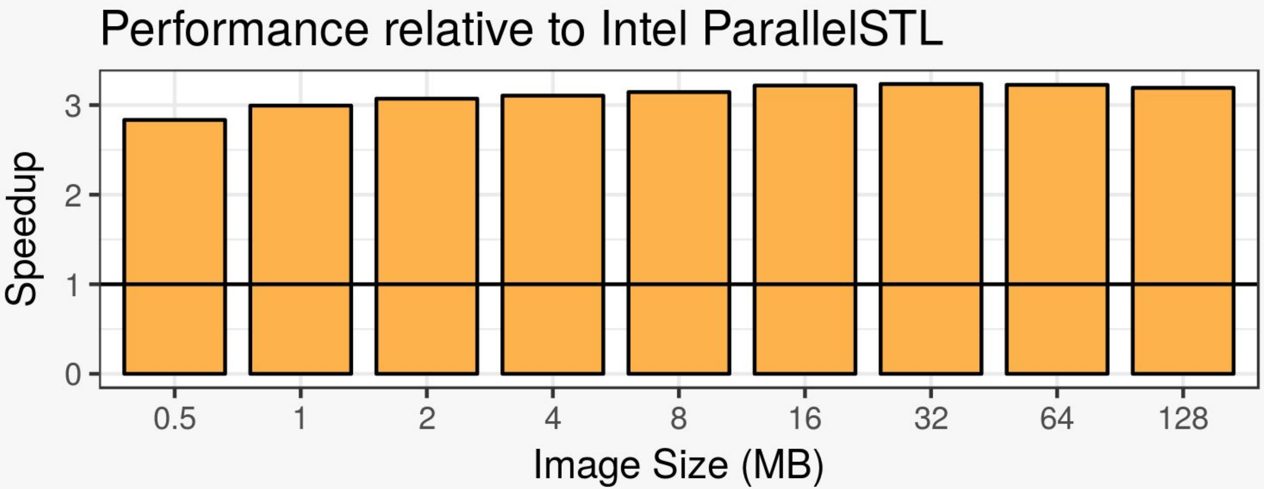
What if there is no predefined function?

```
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 = std::experimental::copy(exec, image);

 auto indices = ranges::view::iota(0)
              | ranges::view::take(width * height);
 std::experimental::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.

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](git.io/vA5H9)

Paper: wg21.link/P0836R0