Towards Composable GPU Programming:

Programming GPUs with Eager Actions and Lazy Views

Michael Haidl · **Michel Steuwer** · Hendrik Dirks
Tim Humernbrum · Sergei Gorlatch

Westfälische Wilhelms-Universität Münster

The University of Edinburgh

The State of GPU Programming

- Low-Level GPU programming with CUDA / OpenCL is widely considered too difficult

- Higher level approaches improve programmability

- Thrust and others allow programmers to write programs by customising and composing patterns

thrust / **thrust**

skelcl / **skelcl**

HSA-Libraries / **Bolt**

AccelerateHS / **accelerate**

Dot Product Example in Thrust

```
float dotProduct(const vector<float>& a,
                 const vector<float>& b) {
 thrust::device_vector<float> d_a = a;
 thrust::device_vector<float> d_b = b;
 return thrust::inner_product(
  d_a.begin(), d_a.end(), d_b.begin(), 0.0f); }
```

Specialized Pattern

Dot Product expressed as special case
No *composition* of universal patterns

Composed Dot Product in Thrust

Intermediate vector required

```
float dotProduct(const vector<float>& a,
                 const vector<float>& b) {
 thrust::device_vector<float> d_a = a;
 thrust::device_vector<float> d_b = b;
 thrust::device_vector<float> tmp(a.size());
 thrust::transform(d_a.begin(), d_a.end(),
                   d_b.begin(), tmp.begin(),
                   thrust::multiplies<floal>());
 return thurst::reduce(tmp.begin(), tmp.end());}
```

Universal patterns

Iterators prevent *composable* programming style

In Thrust:

Two Patterns == Two Kernels → Bad Performance

Composability in the Range-based STL*

- Replacing pairs of *Iterators* with *Ranges* allows for a composable style:

```
float dotProduct(const vector<float>& a,
                 const vector<float>& b) {
 auto mult = [](auto p){
    return get<0>(p) * get<1>(p); };

 return
  accumulate(
   view::transform(view::zip(a,b),mult),0.0f); }
```

Patterns operate on ranges

Patterns are *composable*

- We can even write:

```
view::zip(a,b) | view::transform(mult) | accumulate(0.0f)
```


* https://github.com/ericniebler/range-v3

GPU-enabled container and algorithms

- We extended the `range-v3` library with:
  - GPU-enabled *container*

    ```
    gpu::vector<T>
    ```

  - GPU-enabled *algorithms*

    ```
         void gpu::for_each(InRange, Fun);
    OutRange& gpu::transform(InRange, OutRange, Fun);
            T gpu::reduce(InRange, Fun, T);
    ```

GPU-enabled Dot Product using extended `range-v3`

```
float dotProduct(const vector<float>& a,
                 const vector<float>& b) {
 auto mult = [](auto p){
    return get<0>(p) * get<1>(p); };

 return view::zip(gpu::copy(a), gpu::copy(b))
      | view::transform(mult)
      | gpu::reduce(0.0f);  }
```

1. Copy a and b to gpu :: vectors
2. Combine vectors
3. Multiply vectors pairwise
4. Sum up result

- Executes as fast as `thurst :: inner_product`
- Many Patterns $\neq$ Many Kernels $\rightarrow$ Good Performance

Lazy Views == Kernel Fusion

- *Views* describe non-mutating operations on ranges

```
float dotProduct(const vector<float>& a,
                 const vector<float>& b) {
 auto mult = [](auto p){
    return get<0>(p) * get<1>(p); };

 return view::zip(gpu::copy(a), gpu::copy(b))
      | view::transform(mult)
      | gpu::reduce(0.0f);  }
```

- The implementation of views *guarantees* fusion with the following operation

- Fused with GPU-enabled pattern ⇒ Kernel Fusion

Eager Actions ≠ Kernel Fusion

- *Actions* perform in-place operations on ranges

```
float dotProduct(const vector<float>& a,
                 const vector<float>& b) {
  auto mult = [](auto p) {
    return get<0>(p) * get<1>(p); };

  gpu::vector<pair<float, float>> pairs(a.size());
  for (auto i = 0; i < a.size(); ++i)
    pairs[i] = make_pair(a[i], b[i]);

  return pairs | gpu::action::transform(mult)
               | gpu::reduce(0.0f);
}
```

- Actions are (usually) mutating

- Action implementations use GPU-enabled algorithms

Choice of Kernel Fusion

- Choice between **`views`** and **`actions/algorithms`** is choice for or against kernel fusion
- Simple cost model:
  Every action/algorithm results in a Kernel
- Programmer is in control! Fusion is *guaranteed*.

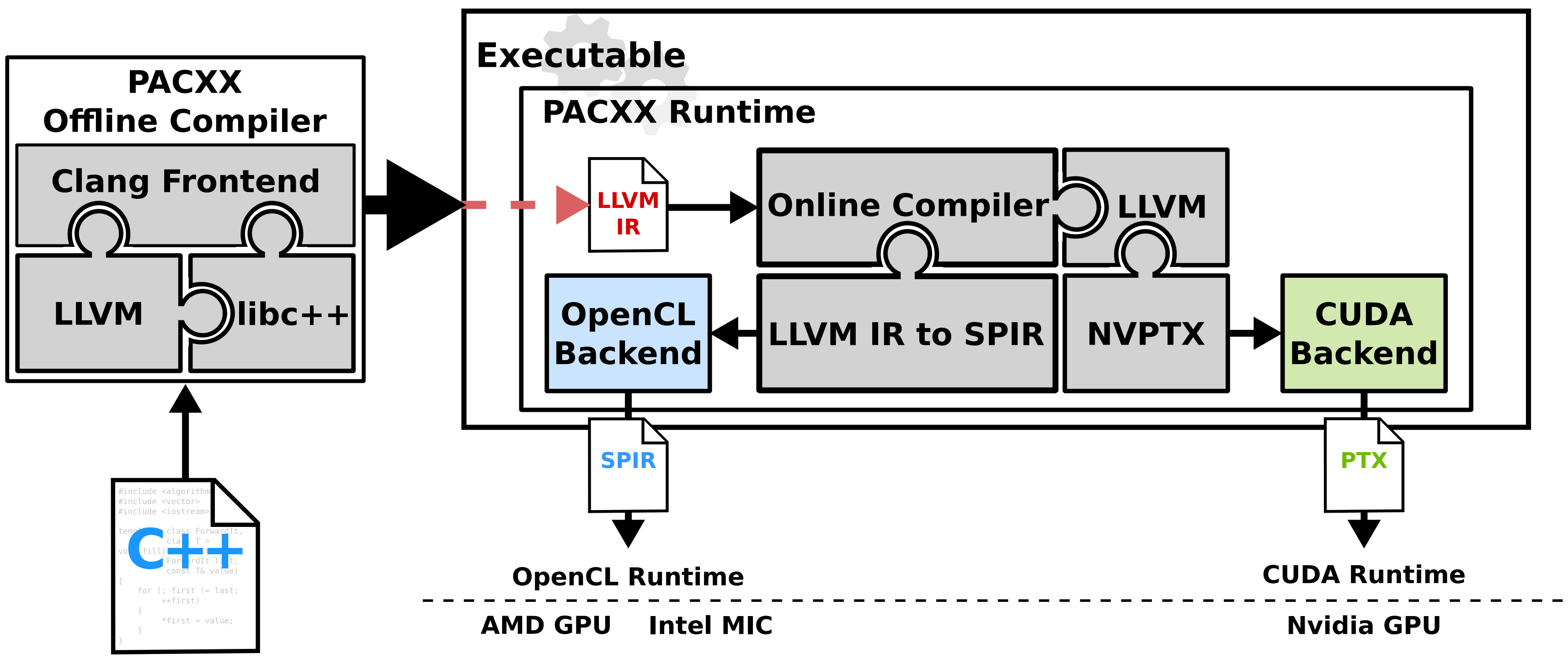
Available for free: Views provided by `range-v3`

- `adjacent_filter`
- `adjacent_remove_if`
- `all`
- `bounded`
- `chunk`
- `concat`
- `const_`
- `counted`
- `delimit`
- `drop`
- `drop_exactly`
- `drop_while`
- `empty`
- **`generate`**
- `generate_n`
- `group_by`
- `indirect`
- `intersperse`
- `ints`
- `iota`
- `join`
- `keys`
- `move`
- `partial_sum`
- `remove_if`
- `repeat`
- `repeat_n`
- `replace`
- `replace_if`
- **`reverse`**
- `single`
- `slice`
- `split`
- `stride`
- `tail`
- **`take`**
- `take_exactly`
- `take_while`
- `tokenize`
- `transform`
- `unbounded`
- `unique`
- `values`
- `zip`
- `zip_with`

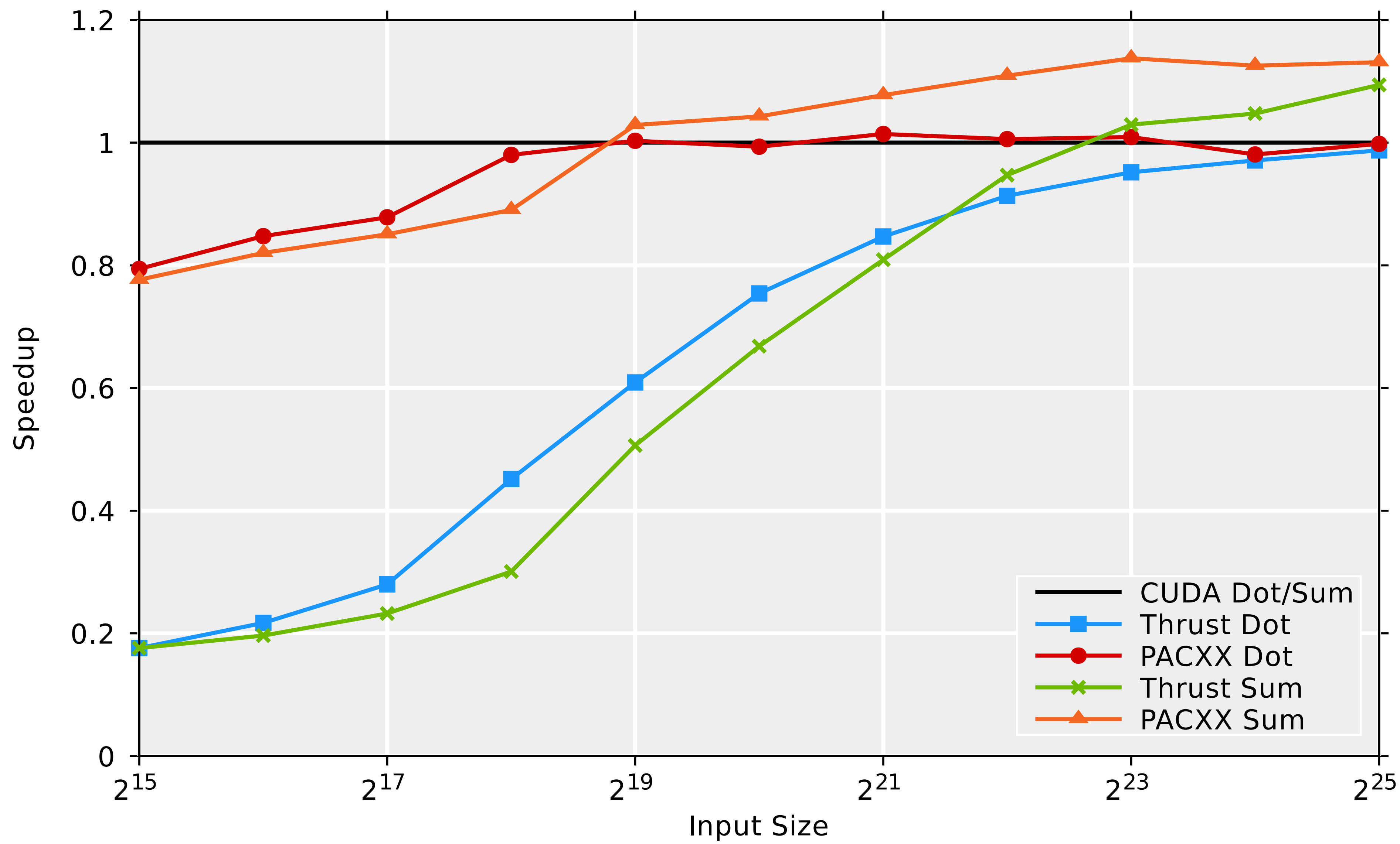
https://ericniebler.github.io/range-v3/index.html#range-views

Code Generation via PACXX

- We use PACXX to compile the extended C++ `range-v3` library implementation to GPU code
- Similar implementation possible with SYCL



Evaluation Sum and Dot Product



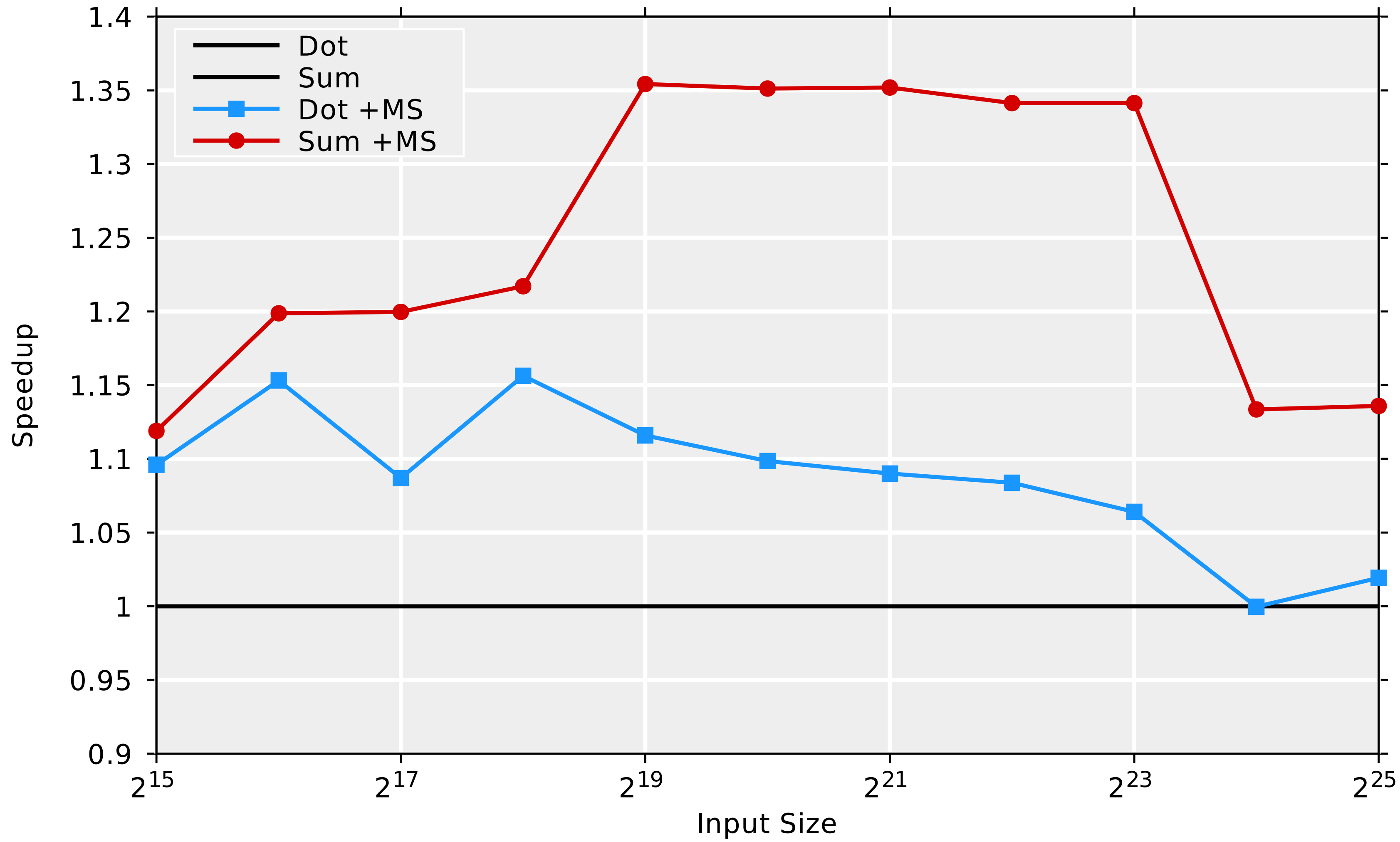
Performance comparable to Thrust and CUDA code

Multi-Staging in PACXX

- PACXX specializes GPU code at CPU runtime
- Implementation of `gpu::reduce` $\Rightarrow$
- Loop bound known at GPU compiler time

```
template <class InRng, class T, class Fun>
auto reduce(InRng&& in, T init, Fun&& fun) {
  // 1. preparation of kernel call
  ...
  // 2. create GPU kernel
  auto kernel = pacxx::kernel(
   [fun](auto&& in, auto&& out,
         int size, auto init) {
    // 2a. stage elements per thread
    int ept = stage(size / glbSize);
    // 2b. start reduction computation
    auto sum = init;
    for (int x = 0; x < ept; ++x) {
      sum = fun(sum, *(in + gid));
      gid += glbSize; }
    // 2c. perform reduction in shared memory
    ...
    // 2d. write result back
    if (lid = 0) *(out + bid) = shared[0];
   }, glbSize, lclSize);
  // 3. execute kernel
  kernel(in, out, distance(in), init);
  // 4. finish reduction on the CPU
  return std::accumulate(out, init, fun); }
```

Performance Impact of Multi-Staging



Up to 1.35x performance improvement

Summary: Towards Composable GPU Programming

- GPU Programming with universal *composable* patterns
- Views vs. Actions/Algorithms determine kernel fusion
- Kernel fusion for views guaranteed ⇒ Programmer in control
- Competitive performance vs. CUDA and specialized Thrust code
- Multi-Staging optimization gives up to 1.35 improvement

Questions?

Towards Composable GPU Programming:

Programming GPUs with Eager Actions and Lazy Views

Michael Haidl · **Michel Steuwer** · Hendrik Dirks
Tim Humernbrum · Sergei Gorlatch

Westfälische Wilhelms-Universität Münster

The University of Edinburgh