



# Michel Steuwer

<http://homepages.inf.ed.ac.uk/msteuwer/>



THE UNIVERSITY  
*of* EDINBURGH

# SkelCL: Algorithmic Skeletons for GPUs

$$\sum_i a_i * b_i = \text{reduce}(+) 0 (\text{zip}(\times) A B)$$

```
#include <SkelCL/SkelCL.h> #include <SkelCL/Zip.h>
#include <SkelCL/Reduce.h> #include <SkelCL/Vector.h>

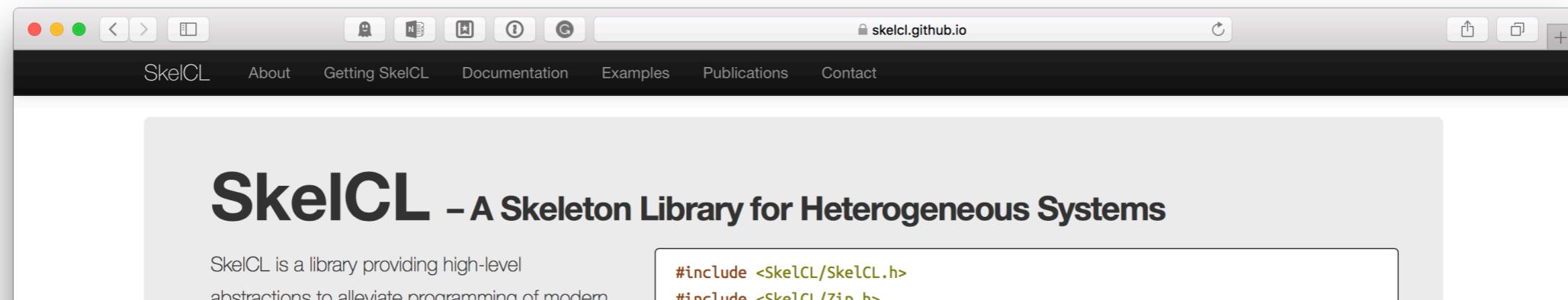
float dotProduct(const float* a, const float* b, int n) {
    using namespace skelcl;
    skelcl::init(1_device.type(deviceType::ANY));

    auto mult = zip([](float x, float y) { return x*y; });
    auto sum = reduce([](float x, float y) { return x+y; }, 0);

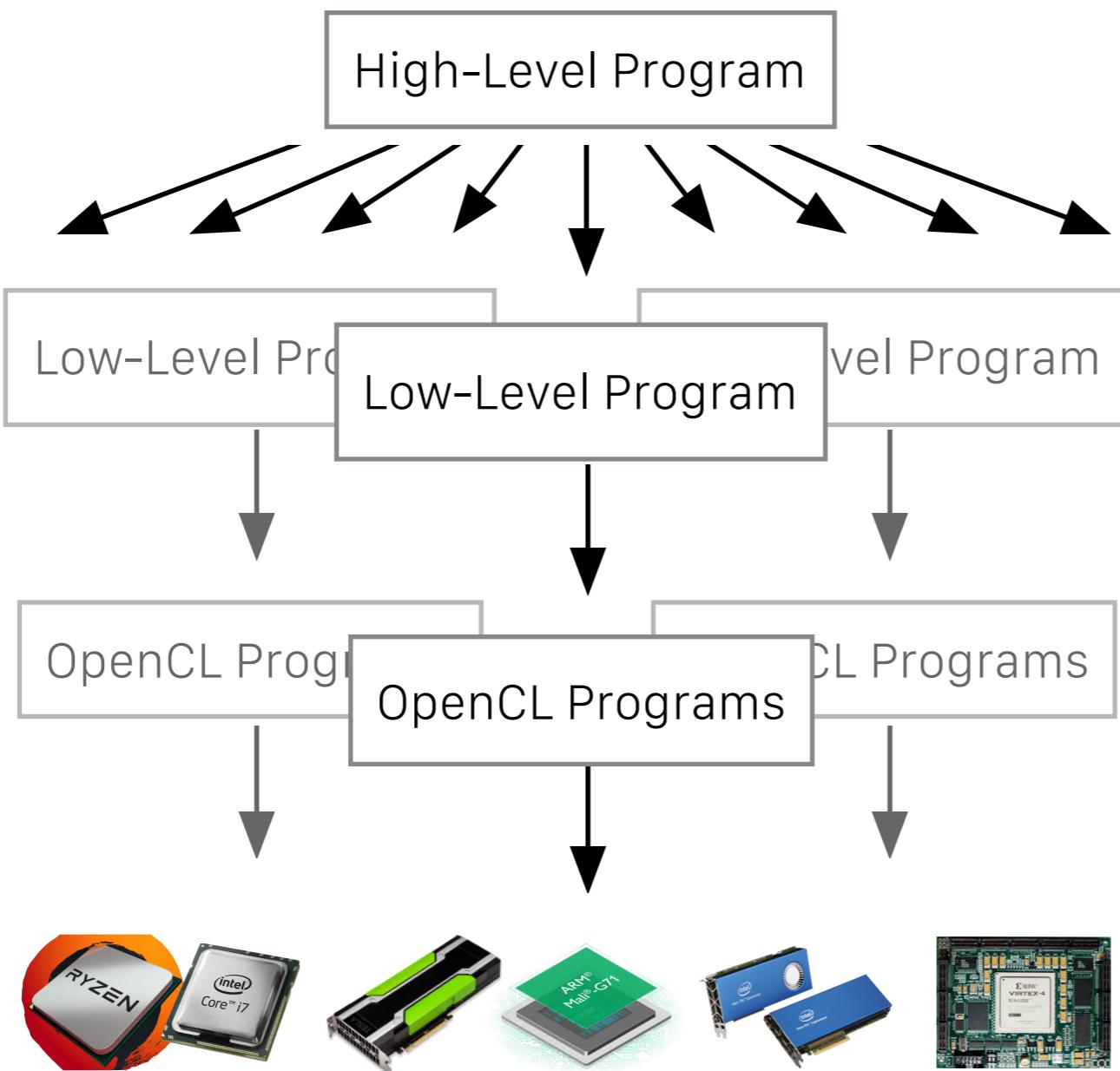
    Vector<float> A(a, a+n); Vector<float> B(b, b+n);

    Vector<float> C = sum(mult(A, B));
    return C.front();
}
```

[skelcl.github.io](https://skelcl.github.io)

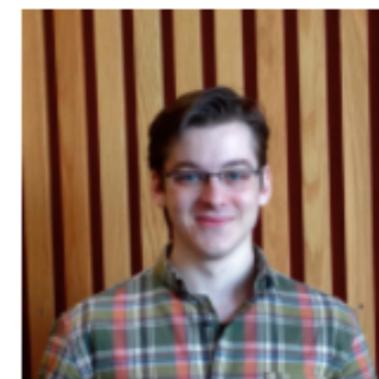
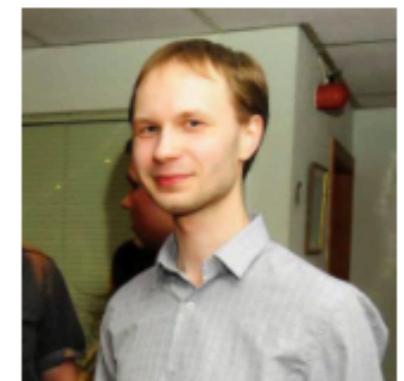
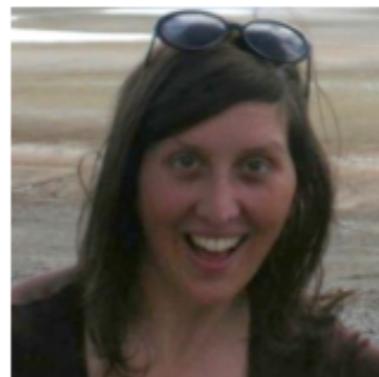


# LIFT: Generating Performance Portable Code using Rewrite Rules



Automatic  
Rewriting

Code  
Generation



# The LIFT Team

# LIFT

Fork me on GitHub

Papers and more infos at: [lift-project.org](http://lift-project.org)

Source code at: [github.com/lift-project/lift](https://github.com/lift-project/lift)

The screenshot shows the GitHub repository page for 'lift-project/lift'. The repository has 1,923 commits, 1 branch, 0 releases, 10 contributors, and is licensed under MIT. The latest commit was made 2 days ago. The repository page includes links for Code, Issues (0), Pull requests (0), Projects (0), Wiki, Pulse, Graphs, and Settings. It also features a 'Clone or download' button.

The Lift programming language <http://www.lift-project.org/> — Edit

1,923 commits 1 branch 0 releases 10 contributors MIT

Branch: master ▾ New pull request Create new file Upload files Find file Clone or download ▾

michel-steuwer committed on GitHub Made LICENSE file parsable for github Latest commit 8b13aac 2 days ago

docker Cleaning up the top folder of the repo and restructuring the docker s... 4 months ago

highLevel refactoring 7 months ago

lib Bump ArithExpr 6 days ago

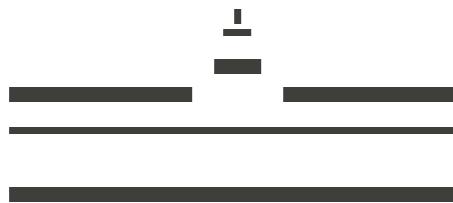
native Add support for querying if the device supports double a year ago

presentations Added power point slides of ICFP, PL Interest and PENCIL meeting. a year ago

# 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

```
1 float dotProduct(const vector<float>& a,
2                     const vector<float>& b) {
3     thrust::device_vector<float> d_a = a;
4     thrust::device_vector<float> d_b = b;
5     return thrust::inner_product(
6         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

The diagram shows a C++ code snippet for a composed dot product function. The code is enclosed in a black-bordered box. Several annotations are present:

- A red diagonal line from the top right corner of the box to the word "tmp" in line 5.
- A green arrow pointing from the text "Universal patterns" to the "return" statement in line 9.
- A red arrow pointing from the text "Iterators prevent *composable* programming style" to the "begin()" and "end()" iterator calls in lines 6-8.

```
1 float dotProduct(const vector<float>& a,
2                      const vector<float>& b) {
3     thrust::device_vector<float> d_a = a;
4     thrust::device_vector<float> d_b = b;
5     thrust::device_vector<float> tmp(a.size());
6     thrust::transform(d_a.begin(), d_a.end(),
7                       d_b.begin(), tmp.begin(),
8                       thrust::multiplies<float>());
9     return thrust::reduce(tmp.begin(), tmp.end());}
```

Universal patterns

Intermediate vector required

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:

```
1 float dotProduct(const vector<float>& a,
2                     const vector<float>& b) {
3     auto mult = [](auto p){
4         return get<0>(p) * get<1>(p); };
5
6     return
7     accumulate(
8         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)
```

# 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

```
1 float dotProduct(const vector<float>& a,
2                     const vector<float>& b) {
3     auto mult = [](auto p){
4         return get<0>(p) * get<1>(p); };
5
6     return view::zip(gpu::copy(a), gpu::copy(b))
7         | view::transform(mult) 1. Copy a and b to gpu::vectors
8         | gpu::reduce(0.0f);   } 2. Combine vectors
```

3. Multiply vectors pairwise

4. Sum up result

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

# Lazy Views == Kernel Fusion

- Views describe non-mutating operations on ranges

```
1 float dotProduct(const vector<float>& a,
2                     const vector<float>& b) {
3     auto mult = [](auto p){
4         return get<0>(p) * get<1>(p); };
5
6     return view::zip(gpu::copy(a), gpu::copy(b))
7         | view::transform(mult)
8         | gpu::reduce(0.0f); }
```

- The implementation of views guarantees fusion with the following operation
- Fused with GPU-enabled pattern ⇒ Kernel Fusion

# Eager Actions $\neq$ Kernel Fusion

- Actions perform in-place operations on ranges

```
float asum(const vector<float>& a) {
    auto abs = [] (auto x) {
        return if (x < 0) { -x; } else { x; } ;
    }

    auto gpuBuffer = gpu::copy(a);

    return gpuBuffer | gpu::action::transform(abs)
                      | 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*.

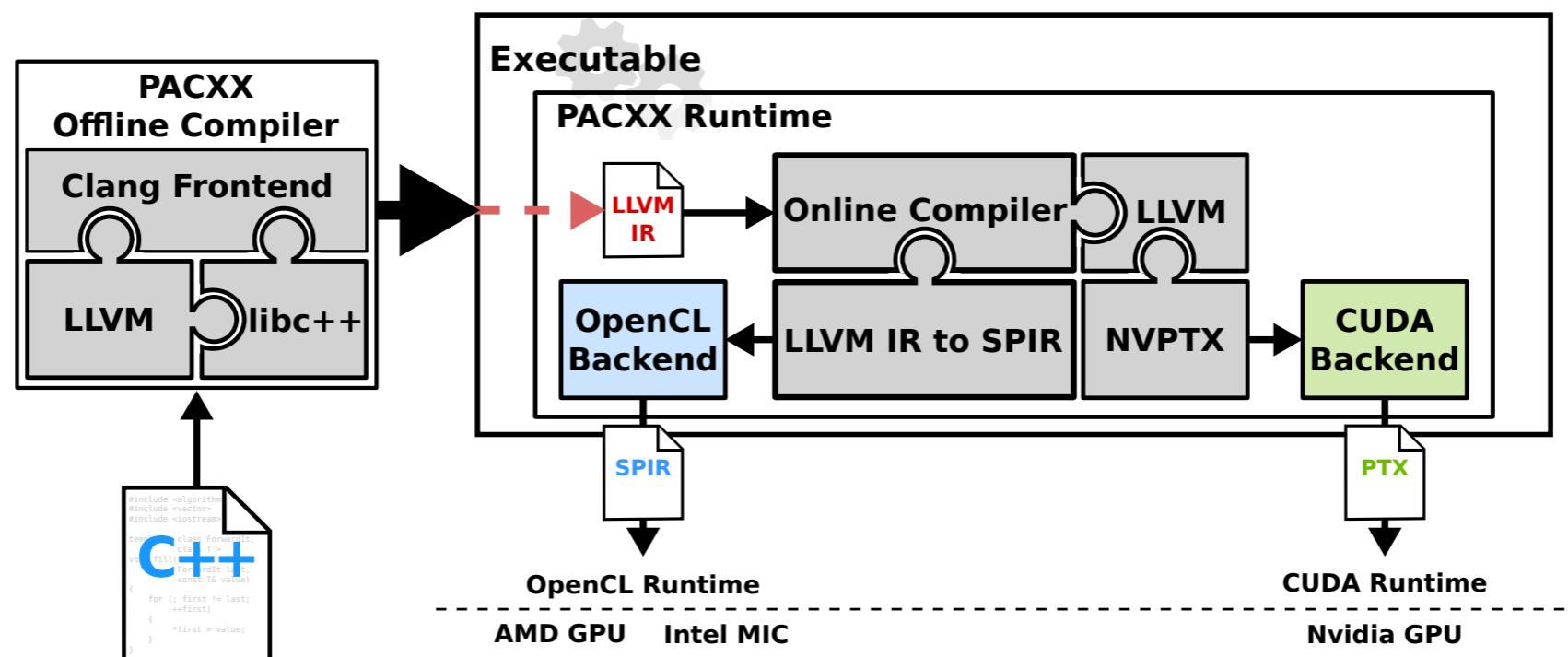
# Available for free: Views provided by range-v3

- 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

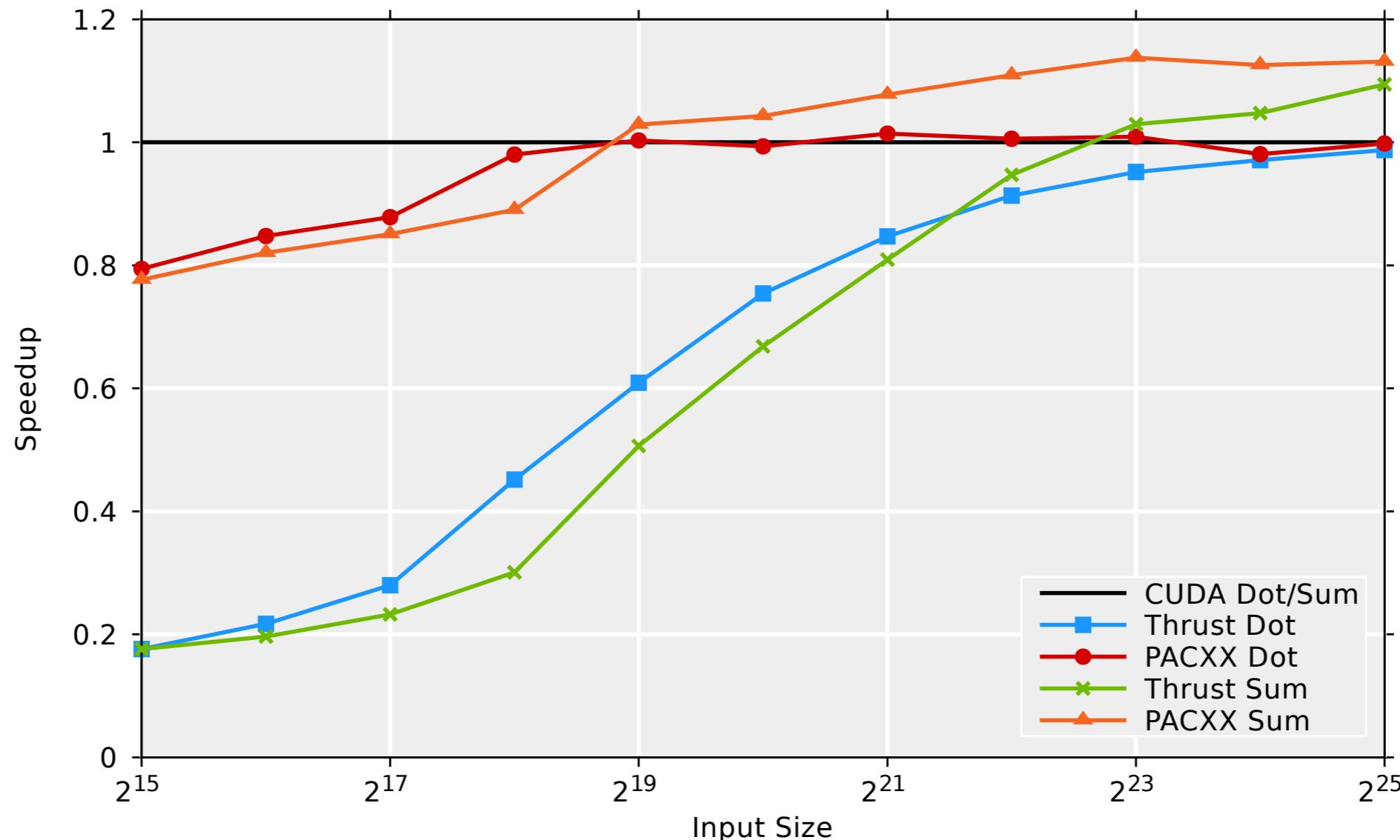
<https://ericniebler.github.io/range-v3/index.html#range-views>

# 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



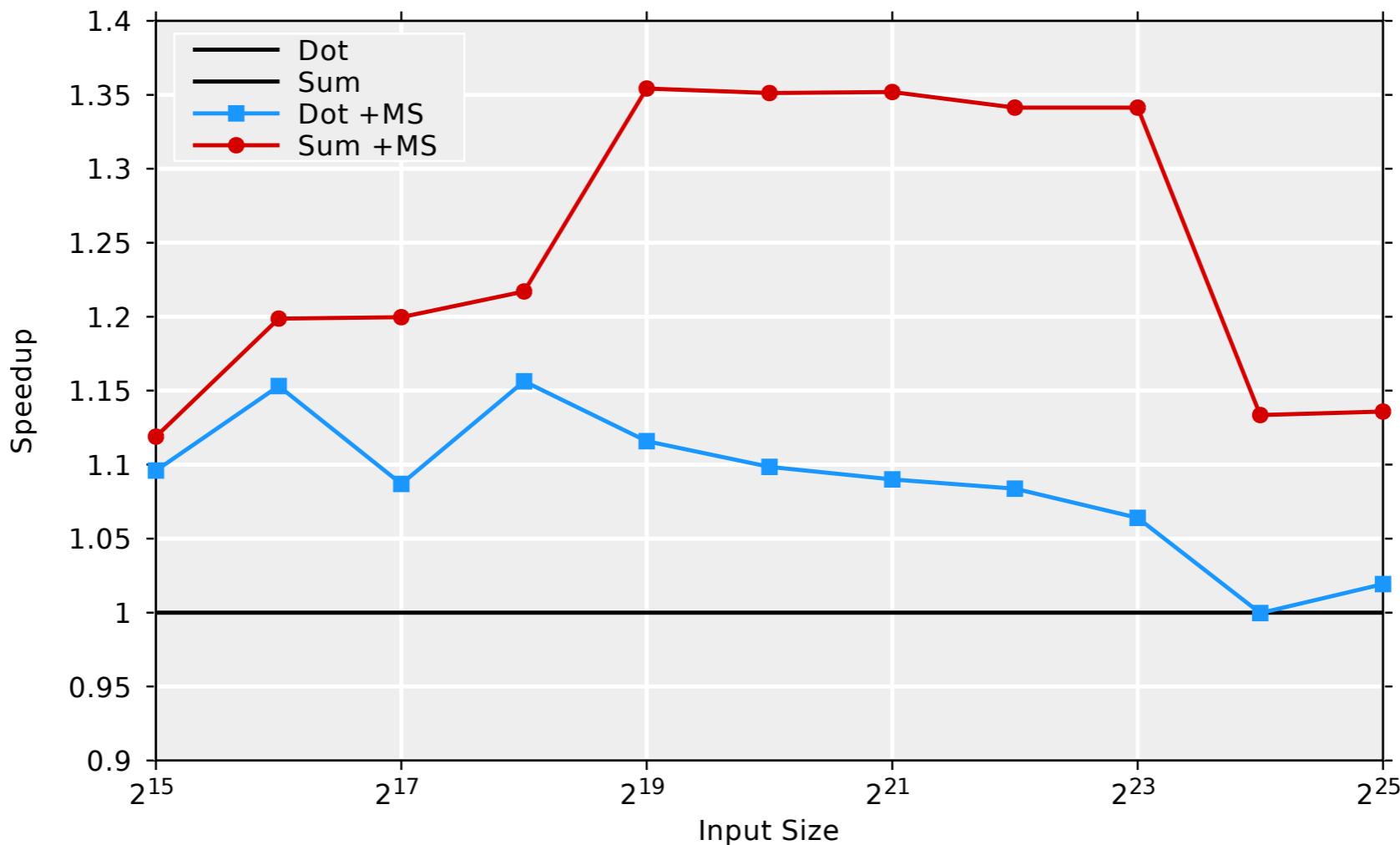
Performance comparable to Thrust and CUDA code

# Multi-Staging in PACXX

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

```
1  template <class InRng, class T, class Fun>
2  auto reduce(InRng&& in, T init, Fun&& fun) {
3      // 1. preparation of kernel call
4      ...
5      // 2. create GPU kernel
6      auto kernel = pacxx::kernel(
7          [fun](auto&& in, auto&& out,
8              int size, auto init) {
9              // 2a. stage elements per thread
10             int ept = stage(size / glbSize);
11             // 2b. start reduction computation
12             auto sum = init;
13             for (int x = 0; x < ept; ++x) {
14                 sum = fun(sum, *(in + gid));
15                 gid += glbSize; }
16             // 2c. perform reduction in shared memory
17             ...
18             // 2d. write result back
19             if (lid = 0) *(out + bid) = shared[0];
20         }, glbSize, lclSize);
21         // 3. execute kernel
22         kernel(in, out, distance(in), init);
23         // 4. finish reduction on the CPU
24         return std::accumulate(out, init, fun); }
```

# Performance Impact of Multi-Staging



Up to 1.35x performance improvement

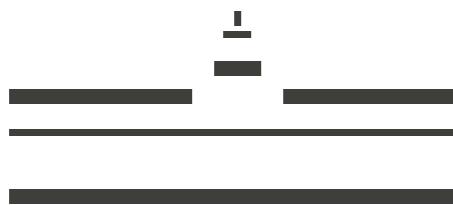
# Summary: Towards Composable GPU Programming

- GPU Programming with universal *composable* patterns
- Views vs. Actions/Algorithms determine kernel fusion
- Kernel fusion for views guaranteed  $\Rightarrow$  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