Thrust

Jared Hoberock and Nathan Bell

NVIDIA Research

Diving In

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void)
{
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

Objectives

- Programmer productivity
  - Rapidly develop complex applications
  - Leverage parallel primitives
- Encourage generic programming
  - Don’t reinvent the wheel
  - E.g. one reduction to rule them all
- High performance
  - With minimal programmer effort
- Interoperability
  - Integrates with CUDA C/C++ code

What is Thrust?

- C++ template library for CUDA
  - Mimics Standard Template Library (STL)
- Containers
  - `thrust::host_vector<T>`
  - `thrust::device_vector<T>`
- Algorithms
  - `thrust::sort()`
  - `thrust::reduce()`
  - `thrust::inclusive_scan()`
  - Etc.

Containers

- **Make common operations concise and readable**
  - **Hides `cudaMalloc`, `cudaMemcpy` and `cudaFree`**

```
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;

// manipulate device values from the host
d_vec[0] = 13;
d_vec[1] = 27;

std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;

// vector memory automatically released w/ free() or cudaFree()
```

Containers

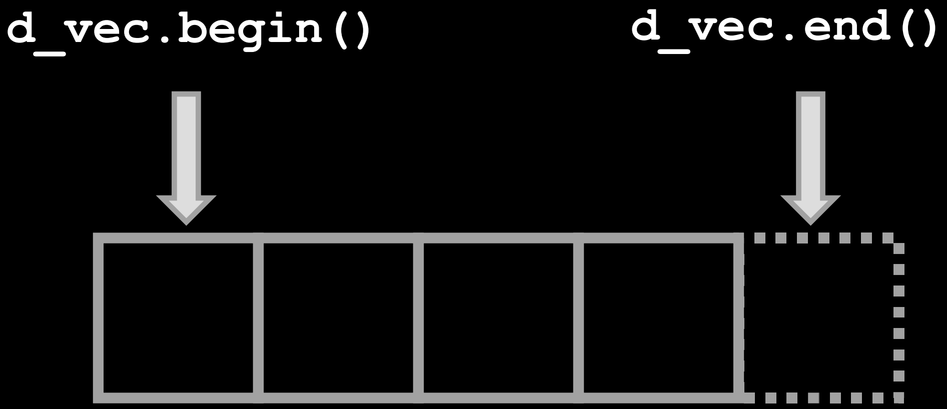
- Compatible with STL containers
  - Eases integration
  - `vector`, `list`, `map`, ...

```
// list container on host
std::list<int> h_list;
h_list.push_back(13);
h_list.push_back(27);

// copy list to device vector
thrust::device_vector<int> d_vec(h_list.size());
thrust::copy(h_list.begin(), h_list.end(), d_vec.begin());

// alternative method
thrust::device_vector<int> d_vec(h_list.begin(), h_list.end());
```

Iterators

- **Sequences defined by pair of iterators**

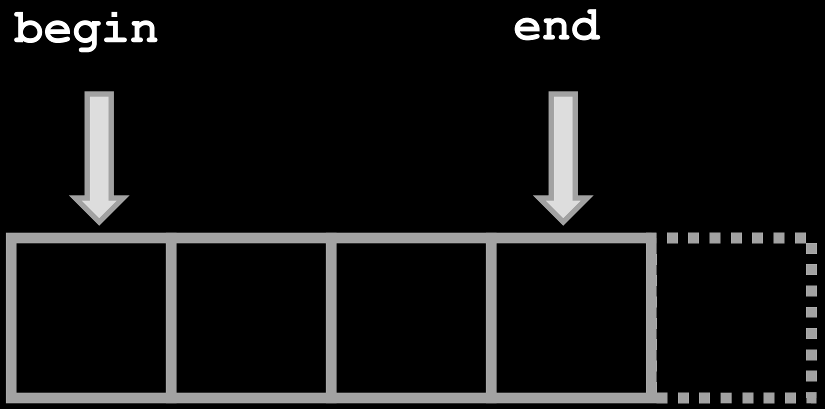
```
// allocate device vector
thrust::device_vector<int> d_vec(4);

d_vec.begin(); // returns iterator at first element of d_vec
d_vec.end()    // returns iterator one past the last element of d_vec

// [begin, end) pair defines a sequence of 4 elements
```



Iterators

- Iterators act like pointers

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

thrust::device_vector<int>::iterator begin = d_vec.begin();
thrust::device_vector<int>::iterator end   = d_vec.end();

int length = end - begin; // compute size of sequence [begin, end)

end = d_vec.begin() + 3;  // define a sequence of 3 elements
```



Iterators

- **Use iterators like pointers**

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

thrust::device_vector<int>::iterator begin = d_vec.begin();

*begin = 13;          // same as d_vec[0] = 13;
int temp = *begin;    // same as temp = d_vec[0];

begin++;              // advance iterator one position

*begin = 25;          // same as d_vec[1] = 25;
```

Iterators

- Track memory space (host/device)
  - Guides algorithm dispatch

```
// initialize random values on host
thrust::host_vector<int> h_vec(1000);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
thrust::device_vector<int> d_vec = h_vec;

// compute sum on host
int h_sum = thrust::reduce(h_vec.begin(), h_vec.end());

// compute sum on device
int d_sum = thrust::reduce(d_vec.begin(), d_vec.end());
```

Iterators

- **Convertible to raw pointers**

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector's memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<<N/256, 256>>>(N, ptr);


// Note: ptr cannot be dereferenced on the host!
```

Iterators

- Wrap raw pointers with `device_ptr`

```
int N = 10;

// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));

// wrap raw pointer with a device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);

// access device memory through device_ptr
dev_ptr[0] = 1;

// free memory
cudaFree(raw_ptr);
```

Namespaces

- C++ supports namespaces
  - Thrust uses `thrust` namespace
    - `thrust::device_vector`
    - `thrust::copy`
  - STL uses `std` namespace
    - `std::vector`
    - `std::list`
- Avoids collisions
  - `thrust::sort()`
  - `std::sort()`
- For brevity
  - `using namespace thrust;`

Recap

- Containers
  - Manage host & device memory
  - Automatic allocation and deallocation
  - Simplify data transfers
- Iterators
  - Behave like pointers
  - Keep track of memory spaces
  - Convertible to raw pointers
- Namespaces
  - Avoids collisions

C++ Background

- **Function templates**

```
// function template to add numbers (type of T is variable)
template< typename T >
T add(T a, T b)
{
   return a + b;
}

// add integers
int x = 10; int y = 20; int z;
z = add<int>(x,y);     // type of T explicitly specified
z = add(x,y);          // type of T determined automatically

// add floats
float x = 10.0f; float y = 20.0f; float z;
z = add<float>(x,y);   // type of T explicitly specified
z = add(x,y);          // type of T determined automatically
```

C++ Background

- Function objects (Functors)

```
// templated functor to add numbers
template< typename T >
class add
{
    public:
    T operator()(T a, T b)
    {
       return a + b;
    }
};

int x = 10; int y = 20; int z;
add<int> func;     // create an add functor for T=int
z = func(x,y);     // invoke functor on x and y

float x = 10; float y = 20; float z;
add<float> func;   // create an add functor for T=float
z = func(x,y);     // invoke functor on x and y
```

C++ Background

- Generic Algorithms

```
// apply function f to sequences x, y and store result in z
template <typename T, typename Function>
void transform(int N, T * x, T * y, T * z, Function f)
{
   for (int i = 0; i < N; i++)
      z[i] = f(x[i], y[i]);
}

int N = 100;
int x[N]; int y[N]; int z[N];

add<int> func;                       // add functor for T=int

transform(N, x, y, z, func);         // compute z[i] = x[i] + y[i]

transform(N, x, y, z, add<int>()); // equivalent
```

Algorithms

- Thrust provides many standard algorithms
  - Transformations
  - Reductions
  - Prefix Sums
  - Sorting
- Generic definitions
  - General Types
    - Built-in types (`int`, `float`, …)
    - User-defined structures
  - General Operators
    - reduce with `plus` operator
    - scan with `maximum` operator

Algorithms

- **General types and operators**

```
#include <thrust/reduce.h>

// declare storage
device_vector<int>   i_vec = ...
device_vector<float> f_vec = ...

// sum of integers (equivalent calls)
reduce(i_vec.begin(), i_vec.end());
reduce(i_vec.begin(), i_vec.end(),    0, plus<int>());

// sum of floats (equivalent calls)
reduce(f_vec.begin(), f_vec.end());
reduce(f_vec.begin(), f_vec.end(), 0.0f, plus<float>());

// maximum of integers
reduce(i_vec.begin(), i_vec.end(),    0, maximum<int>());
```

Algorithms

- General types and operators

```
struct negate_float2
{
    __host__ __device__
    float2 operator()(float2 a)
    {
        return make_float2(-a.x, -a.y);
    }
};

// declare storage
device_vector<float2> input  = ...
device_vector<float2> output = ...

// create functor
negate_float2 func;

// negate vectors
transform(input.begin(), input.end(), output.begin(), func);
```

Algorithms

- General types and operators

```
// compare x component of two float2 structures
struct compare_float2
{
   __host__ __device__
   bool operator()(float2 a, float2 b)
   {
       return a.x < b.x;
   }
};

// declare storage
device_vector<float2> vec = ...

// create comparison functor
compare_float2 comp;

// sort elements by x component
sort(vec.begin(), vec.end(), comp);
```

Algorithms

- **Operators with State**

```
// compare x component of two float2 structures
struct is_greater_than
{
    int threshold;

    is_greater_than(int t) { threshold = t; }

    __host__ __device__
    bool operator()(int x) { return x > threshold; }
};

device_vector<int> vec = ...

// create predicate functor (returns true for x > 10)
is_greater_than pred(10);

// count number of values > 10
int result = count_if(vec.begin(), vec.end(), pred);
```

Recap

- Algorithms
  - Generic
    - Support general types and operators
  - Statically dispatched based on iterator type
    - Memory space is known at compile time
  - Have default arguments
    - `reduce(begin, end)`
    - `reduce(begin, end, init, binary_op)`

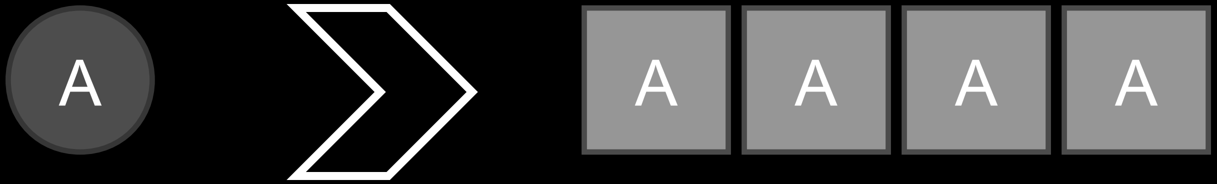
Fancy Iterators

- Behave like “normal” iterators
  - Algorithms don't know the difference
- Examples
  - `constant_iterator`
  - `counting_iterator`
  - `transform_iterator`
  - `permutation_iterator`
  - `zip_iterator`

Fancy Iterators

- constant_iterator
  - Mimics an infinite array filled with a constant value

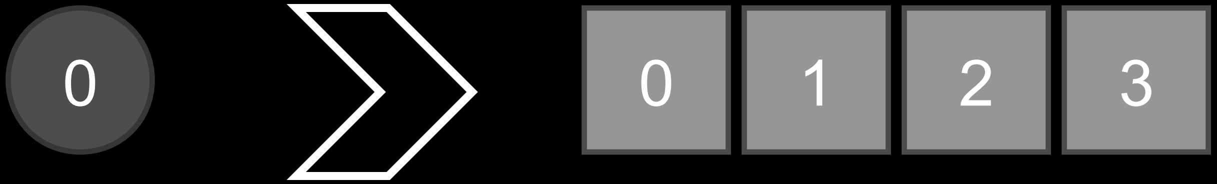
```
// create iterators
constant_iterator<int> begin(10);
constant_iterator<int> end = begin + 3;

begin[0]   // returns 10
begin[1]   // returns 10
begin[100] // returns 10

// sum of [begin, end)
reduce(begin, end);   // returns 30 (i.e. 3 * 10)
```



Fancy Iterators

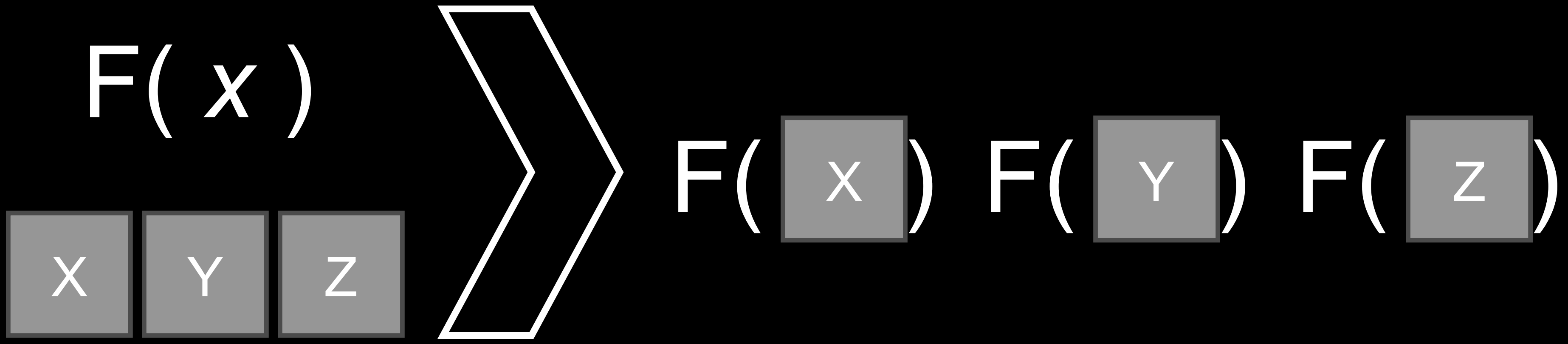
- `counting_iterator`
  - Mimics an infinite array with sequential values

```
// create iterators
counting_iterator<int> begin(10);
counting_iterator<int> end = begin + 3;

begin[0]   // returns 10
begin[1]   // returns 11
begin[100] // returns 110

// sum of [begin, end)
reduce(begin, end);   // returns 33 (i.e. 10 + 11 + 12)
```



Fancy Iterators

- `transform_iterator`
  - Yields a transformed sequence
  - Facilitates kernel fusion



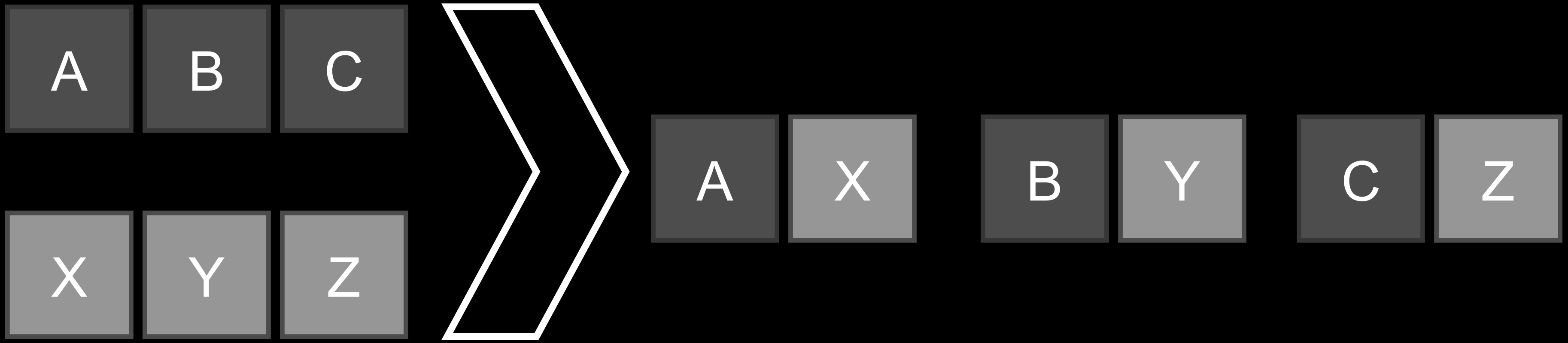
Fancy Iterators

- `transform_iterator`
  - Conserves memory capacity and bandwidth

```
// initialize vector
device_vector<int> vec(3);
vec[0] = 10; vec[1] = 20; vec[2] = 30;

// create iterator (type omitted)
begin = make_transform_iterator(vec.begin(), negate<int>());
end   = make_transform_iterator(vec.end(),   negate<int>());

begin[0]   // returns -10
begin[1]   // returns -20
begin[2]   // returns -30

// sum of [begin, end)
reduce(begin, end);   // returns -60 (i.e. -10 + -20 + -30)
```

Fancy Iterators

- `zip_iterator`
  - Looks like an array of structs (AoS)
  - Stored in structure of arrays (SoA)



Fancy Iterators

- zip_iterator

```
// initialize vectors
device_vector<int>  A(3);
device_vector<char> B(3);
A[0] = 10;  A[1] = 20;  A[2] = 30;
B[0] = 'x'; B[1] = 'y'; B[2] = 'z';

// create iterator (type omitted)
begin = make_zip_iterator(make_tuple(A.begin(), B.begin()));
end   = make_zip_iterator(make_tuple(A.end(),   B.end()));

begin[0]   // returns tuple(10, 'x')
begin[1]   // returns tuple(20, 'y')
begin[2]   // returns tuple(30, 'z')

// maximum of [begin, end)
maximum< tuple<int,char> > binary_op;
reduce(begin, end, begin[0], binary_op); // returns tuple(30, 'z')
```

Best Practices

- Fusion
  - Combine related operations together
- Structure of Arrays
  - Ensure memory coalescing
- Implicit Sequences
  - Eliminate memory accesses

Fusion

- Combine related operations together
  - Conserves memory bandwidth
- Example: SNRM2
  - Square each element
  - Compute sum of squares and take `sqrt()`

Fusion

- **Unoptimized implementation**

```
// define transformation f(x) -> x^2
struct square
{
    __host__ __device__
    float operator()(float x)
    {
        return x * x;
    }
};

float snrm2_slow(device_vector<float>& x)
{
  // without fusion
  device_vector<float> temp(x.size());
  transform(x.begin(), x.end(), temp.begin(), square());

  return sqrt( reduce(temp.begin(), temp.end()) );
}
```

Fusion

- **Optimized implementation (3.8x faster)**

```
// define transformation f(x) -> x^2
struct square
{
    __host__ __device__
    float operator()(float x)
    {
        return x * x;
    }
};

float snrm2_fast(device_vector<float>& x)
{
  // with fusion
  return sqrt( transform_reduce(x.begin(), x.end(),
                                square(), 0.0f, plus<float>());
}
```

Structure of Arrays (SoA)

- Array of Structures (AoS)
  - Often does not obey coalescing rules
    - `device_vector<float3>`
- Structure of Arrays (SoA)
  - Obeys coalescing rules
  - Components stored in separate arrays
    - `device_vector<float> x, y, z;`
- Example: Rotate 3d vectors
  - SoA is 2.8x faster

Structure of Arrays (SoA)

```
struct rotate_float3
{
  __host__ __device__
  float3 operator()(float3 v)
  {
    float x = v.x;
    float y = v.y;
    float z = v.z;

    float rx = 0.36f*x +  0.48f*y + -0.80f*z;
    float ry =-0.80f*x +  0.60f*y +  0.00f*z;
    float rz = 0.48f*x +  0.64f*y +  0.60f*z;

    return make_float3(rx, ry, rz);
  }
};

...

device_vector<float3> vec(N);

transform(vec.begin(), vec.end,
          vec.begin(),
          rotate_float3());
```

Structure of Arrays (SoA)

```
struct rotate_tuple
{
  __host__ __device__
  tuple<float,float,float> operator()(tuple<float,float,float> v)
  {
    float x = get<0>(v);
    float y = get<1>(v);
    float z = get<2>(v);

    float rx = 0.36f*x +  0.48f*y + -0.80f*z;
    float ry =-0.80f*x +  0.60f*y +  0.00f*z;
    float rz = 0.48f*x +  0.64f*y +  0.60f*z;

    return make_tuple(rx, ry, rz);
  }
};

...

device_vector<float> x(N), y(N), z(N);

transform(make_zip_iterator(make_tuple(x.begin(), y.begin(), z.begin())),
          make_zip_iterator(make_tuple(x.end(),   y.end(),   z.end())),
          make_zip_iterator(make_tuple(x.begin(), y.begin(), z.begin())),
          rotate_tuple());
```

Implicit Sequences

- Avoid storing sequences explicitly
  - Constant sequences
    - `[1, 1, 1, 1, … ]`
  - Incrementing sequences
    - `[0, 1, 2, 3, … ]`
- Implicit sequences require no storage
  - `constant_iterator`
  - `counting_iterator`
- Example
  - Index of the smallest element

Implicit Sequences

```
// return the smaller of two tuples
struct smaller_tuple
{
  tuple<float,int> operator()(tuple<float,int> a, tuple<float,int> b)
  {
    if (a < b)
      return a;
    else
      return b;
  }
};

int min_index(device_vector<float>& vec)
{
  // create explicit index sequence [0, 1, 2, ... )
  device_vector<int> indices(vec.size());
  sequence(indices.begin(), indices.end());

  tuple<float,int> init(vec[0],0);
  tuple<float,int> smallest;

  smallest = reduce(make_zip_iterator(make_tuple(vec.begin(), indices.begin())),
                    make_zip_iterator(make_tuple(vec.end(),   indices.end())),
                    init,
                    smaller_tuple());

  return get<1>(smallest);
}
```

Implicit Sequences

```
// return the smaller of two tuples
struct smaller_tuple
{
  tuple<float,int> operator()(tuple<float,int> a, tuple<float,int> b)
  {
    if (a < b)
      return a;
    else
      return b;
  }
};

int min_index(device_vector<float>& vec)
{
  // create implicit index sequence [0, 1, 2, ... )
  counting_iterator<int> begin(0);
  counting_iterator<int> end(vec.size());

  tuple<float,int> init(vec[0],0);
  tuple<float,int> smallest;

  smallest = reduce(make_zip_iterator(make_tuple(vec.begin(), begin)),
                    make_zip_iterator(make_tuple(vec.end(),   end)),
                    init,
                    smaller_tuple());

  return get<1>(small);
}
```

Recap

- Best Practices
  - Fusion
    - 3.8x faster
  - Structure of Arrays
    - 2.8x faster
  - Implicit Sequences
    - 3.4x faster

Additional Resources

- Thrust
  - Homepage
  - Quick Start Guide
  - Documentation
  - Examples
  - MegaNewtons (blog)
  - thrust-users (mailing list)
- Other
  - NVIDIA Research
  - CUDA