Thrust Library

Dr. Alptekin Temizel
atemizel@metu.edu.tr

Thrust Library

- Distributed with CUDA Toolkit
- Header-only library
- Architecture agnostic
- Just compile and run!

Thrust Objectives

- Programmer productivity
  - Rapidly develop complex applications
  - Leverage parallel primitives
- Encourage generic programming
  - Don’t reinvent the wheel
- High performance
  - With minimal programmer effort
- Interoperability
  - Integrates with CUDA C/C++ code

Outline

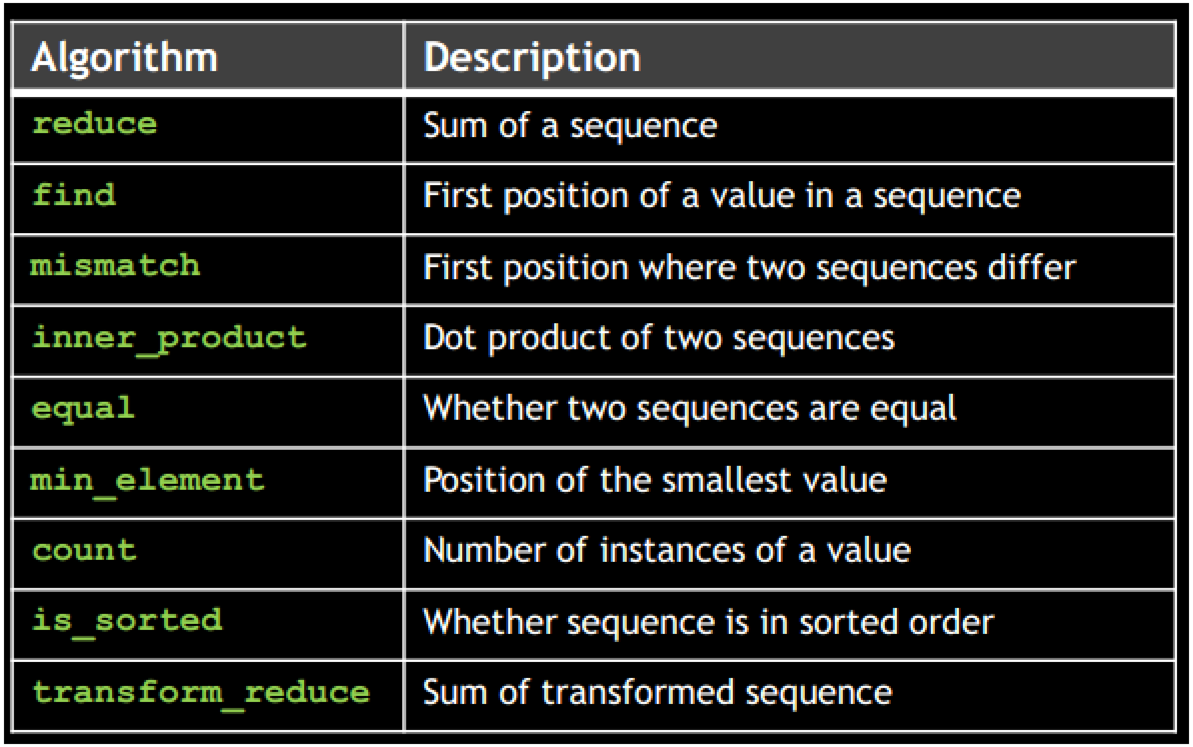
- Thrust Basics
    - Containers: host_vector, device_vector
    - Host – device copy operations
    - Initialization and modification: fill, sequence, generate, random
    - Device and raw pointers
    - STL integration
    - Templates and functors
    - Modifying vectors: replace, replace_if, reverse, reverse_copy
    - Namespaces

- Algorithms
    - Element wise: for_each, transform
    - Reduction: reduce, reduce_by_key, inner_product, find, equal …
    - Kernel fusion and transform_reduce
    - Iterators: constant_iterator, counting_iterator, zip_iterator
    - Prefix-sums: inclusive_scan, inclusive_scan_by_key
    - Sorting: sort, stable_sort, sort_by_key

Containers

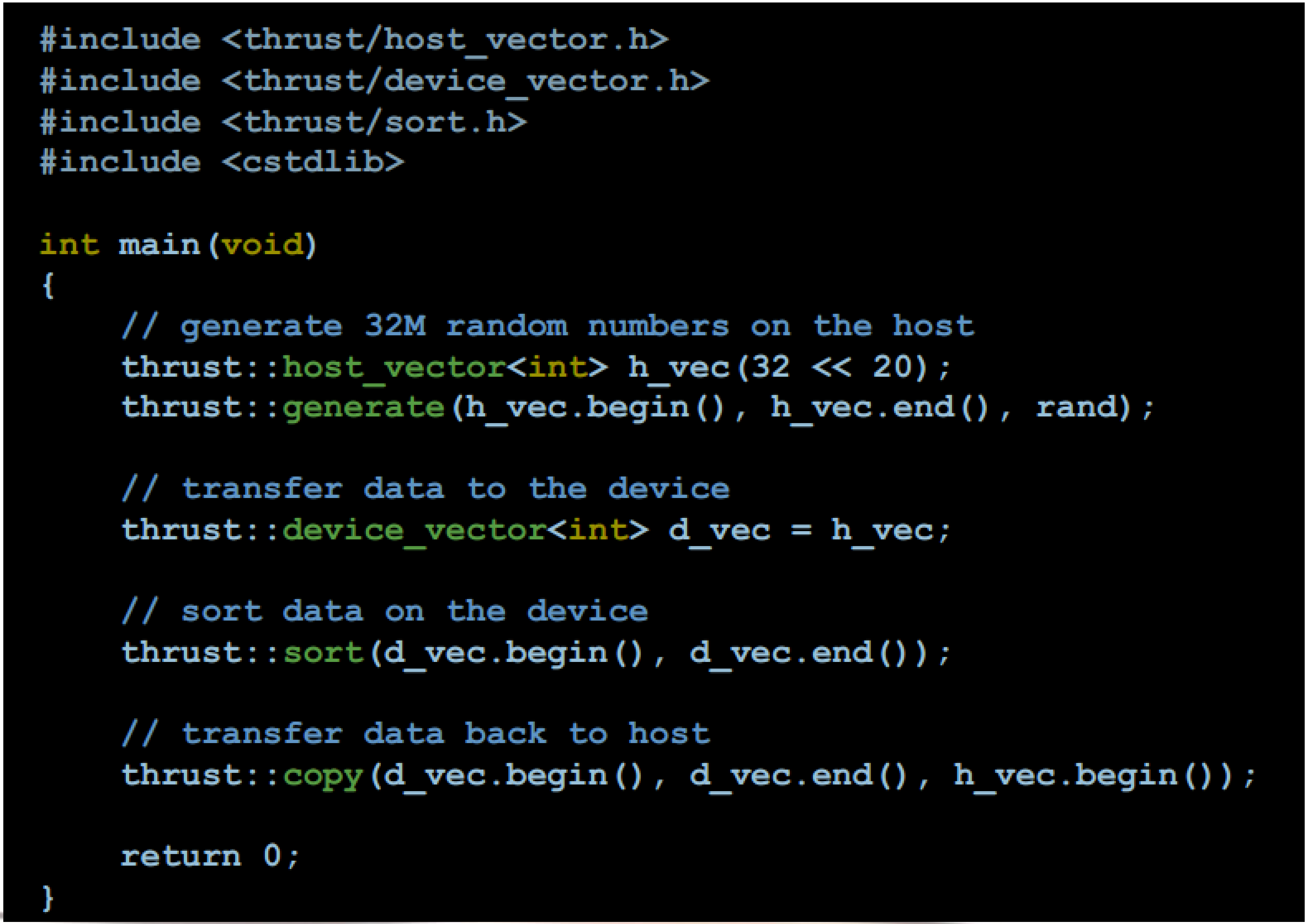
- C++ template library for CUDA
  - Mimics Standard Template Library (STL)
- Make common operations concise and readable
  - Hides cudaMalloc, cudaMemcpy and cudaFree
- Containers
  - thrust::host_vector<T>
    - CPU implementation via OpenMP
  - thrust::device_vector<T>
    - GPU implementation via CUDA
  - <T> stands for a generic data type

Thrust Algorithms

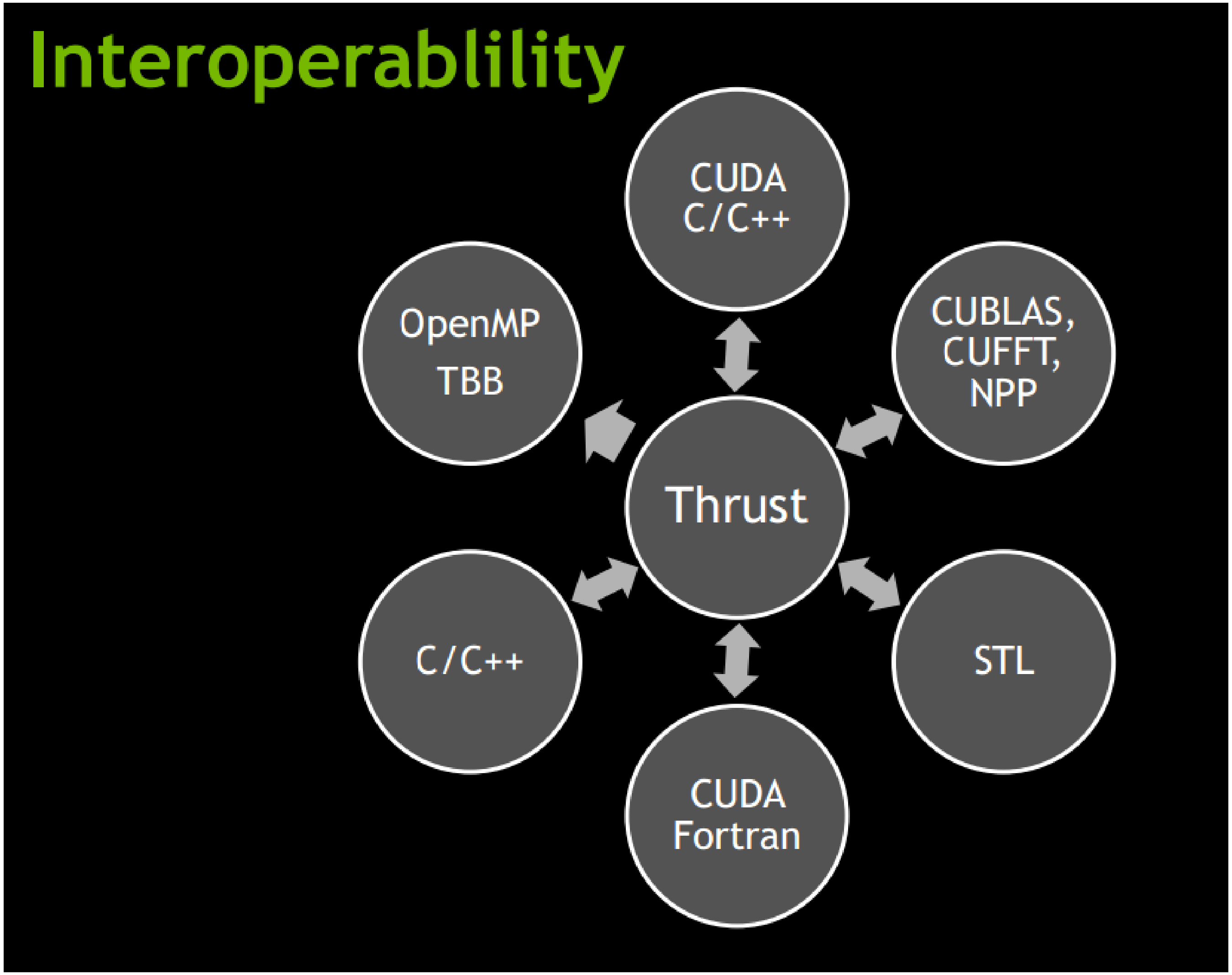


Algorithm	Description
`reduce`	Sum of a sequence
`find`	First position of a value in a sequence
`mismatch`	First position where two sequences differ
`inner_product`	Dot product of two sequences
`equal`	Whether two sequences are equal
`min_element`	Position of the smallest value
`count`	Number of instances of a value
`is_sorted`	Whether sequence is in sorted order
`transform_reduce`	Sum of transformed sequence

Example



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

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    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;
}
```



Interoperablility

CUDA C/C++

OpenMP TBB

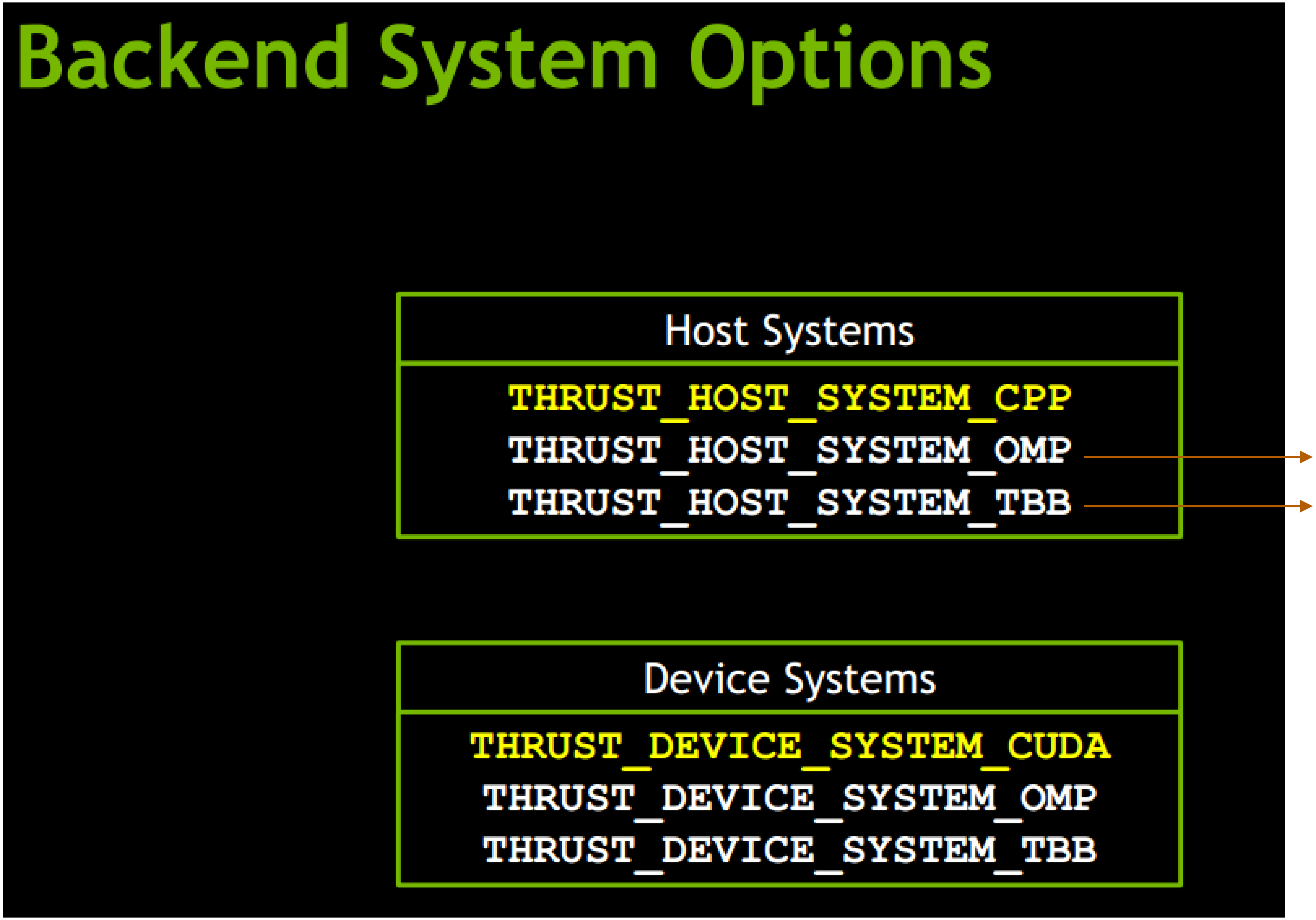
CUBLAS, CUFFT, NPP

Thrust

C/C++

STL

CUDA Fortran



Backend System Options

Host Systems

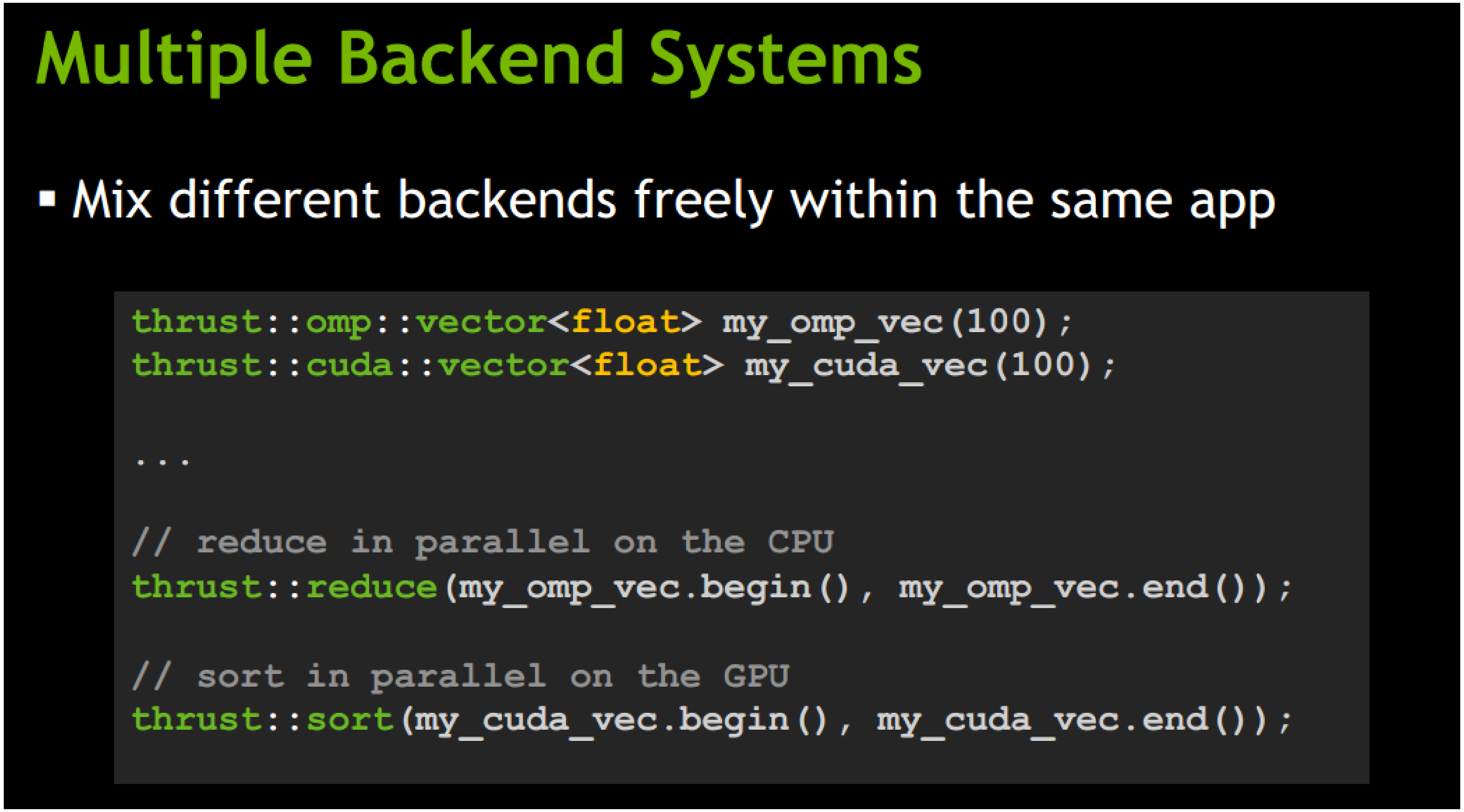
```
THRUST_HOST_SYSTEM_CPP
THRUST_HOST_SYSTEM_OMP
THRUST_HOST_SYSTEM_TBB
```

Device Systems

```
THRUST_DEVICE_SYSTEM_CUDA
THRUST_DEVICE_SYSTEM_OMP
THRUST_DEVICE_SYSTEM_TBB
```

THRUST_HOST_SYSTEM_OMP → OpenMP

THRUST_HOST_SYSTEM_TBB → Intel Threading Building Blocks



Multiple Backend Systems

- Mix different backends freely within the same app

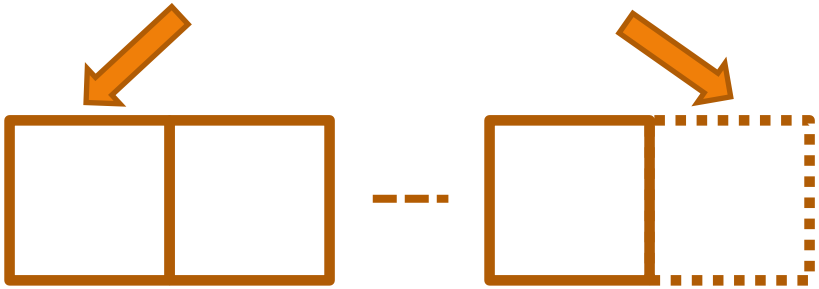
```
thrust::omp::vector<float> my_omp_vec(100);
thrust::cuda::vector<float> my_cuda_vec(100);

...

// reduce in parallel on the CPU
thrust::reduce(my_omp_vec.begin(), my_omp_vec.end());

// sort in parallel on the GPU
thrust::sort(my_cuda_vec.begin(), my_cuda_vec.end());
```

Host ↔ Device Copy Operations

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

// allocate host vector with 3 elements
thrust::host_vector<int> h_vec(3);
h_vec[0] = 13;         // array operator is overloaded
h_vec[1] = 27;
h_vec[2] = -36;
std::cout << “h_vec has size " << h_vec.size() << std::endl;

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

// manipulate device values from the host
d_vec[2] = 36;         // not efficient
```

Host ↔ Device Copy Operations

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

// copy all of d_vec back to the beginning of h_vec
thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
```



```
// [begin, end) pair defines a sequence of N elements
// N equals d_vec.size(), same as:
thrust::device_vector<int>::iterator begin = d_vec.begin();
thrust::device_vector<int>::iterator end   = d_vec.end();
// compute size of sequence [begin, end)
int N = end - begin;
```

Basics of Iterators

- Iterators act like pointers

```
// declare iterator variables
device_vector<int>::iterator begin = d_vec.begin();
device_vector<int>::iterator end   = d_vec.end();

// pointer arithmetic
begin++;

// dereference device iterators from the host
int a = *begin;
int b = begin[3];

// compute size of range [begin,end)
int size = end - begin;
```

Vector Initialization

```
#include <thrust/fill.h>
#include <thrust/sequence.h>
#include <thrust/generate.h>

// initialize all ten integers of a device_vector to 1
thrust::device_vector<int> d_vec(10, 1);

// set the first seven elements of a vector to 9
thrust::fill(d_vec.begin(), d_vec.begin() + 7, 9);

// set the elements of h_vec to 0, 1, 2, 3, ...
thrust::sequence(h_vec.begin(), h_vec.end());

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

Device and Raw Pointers

```
size_t N = 10;

int* dev_raw_ptr1;
cudaMalloc((void **)&dev_raw_ptr1, N * sizeof(int));
thrust::device_ptr<int> dev_ptr(dev_raw_ptr1 );

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

Device and Raw Pointers

```
size_t N = 10;

thrust::device_vector<int> d_vec(N);
int* dev_raw_ptr2 = thrust::raw_pointer_cast(&d_vec[0]);
```

Device and Raw Pointers

```
size_t N = 10;


thrust::device_ptr<int> dev_ptr = thrust::device_malloc<int>(N);
int* dev_raw_ptr3 = thrust::raw_pointer_cast( dev_ptr );
```

Interoperability

- Convert iterators to raw pointers

```
// allocate device vector
const int N = 65536;
thrust::device_vector<int> d_vec(N);

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

// use dev_raw_ptr in a CUDA C kernel

const int number = 29;
addNumber<<< N / 256, 256 >>>(N, dev_raw_ptr, number);
```

```
__global__ void addNumber(int N, int* data, int number)
{
   int i = blockIdx.x * 256 + threadIdx.x;
   if (i < N) {
      data[i] += number;
   }
}
```

STL Integration

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

```
// list container on host
std::list<int> h_list;
h_list.push_back(13);
h_list.push_back(27);
h_list.push_back(36);

// approach 1: create a device vector and copy an STL list into
   a device_vector
thrust::device_vector<int> d_vec(h_list.size());
thrust::copy(h_list.begin(), h_list.end(), d_vec.begin());

// approach 2: initialize a device_vector with the list
thrust::device_vector<int> d_vec(h_list.begin(), h_list.end());
```

Templates and Functors - I

- 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
```

Templates and Functors - II

- 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
```

Templates and Functors - III

- 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
```

Modifying Vectors - I

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

// create a device vector using a host array
const int N = 7;
int arr[N] = {1, 7, 1, 1, 3, 2, -1};
thrust::device_vector<int> d_vec(arr, arr + N) ;

// replace all the ones in d_vec with tens
thrust::replace(d_vec.begin(), d_vec.end(), 1, 10);

// replace via a predicate (E.g. greater than a number)
struct is_greater_than {
    int threshold;

    is_greater_than(int t) { threshold = t; }

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

is_greater_than pred(2);
thrust::replace_if(d_vec.begin(), d_vec.end(), pred, 10);
```

Modifying Vectors - II

- Reversing a vector

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

const int N = 6;
int data[N] = {0, 1, 2, 3, 4, 5};
thrust::device_vector<int> dev_vec(data, data + N);

// takes reverse of a vector inplace
thrust::reverse(dev_vec.begin(), dev_vec.end());
// dev_vec is now {5, 4, 3, 2, 1, 0}

// reverse_copy: reversed is written to a different output
thrust::device_vector<int> output(N);
thrust::reverse_copy(dev_vec.begin(), dev_vec.end(),
                     output.begin());
```

Namespaces

- Avoid name collisions

```
// allocate host memory
thrust::host_vector<int> h_vec(10);

// call STL sort
std::sort(h_vec.begin(), h_vec.end());

// call Thrust sort
thrust::sort(h_vec.begin(), h_vec.end());

// for brevity
using namespace thrust;

// without namespace
int sum = reduce(h_vec.begin(), h_vec.end());
```

Algorithms

- Element wise
  - for_each, transform
- Reduction
  - reduce, reduce_by_key, inner_product, find, equal …
- Kernel fusion
  - transform_reduce
- Iterators
  - constant_iterator, counting_iterator, zip_iterator
- Prefix-sums
  - inclusive_scan, inclusive_scan_by_key
- Sorting
  - sort, stable_sort, sort_by_key

Algorithms

- Thrust targets maximal occupancy and will compare the resource usage of the kernel (e.g., number of registers, amount of shared memory) with the resources of the target GPU to determine a launch configuration with highest occupancy.

- While the maximal occupancy heuristic is not necessarily optimal, it is straightforward to compute and effective in practice.

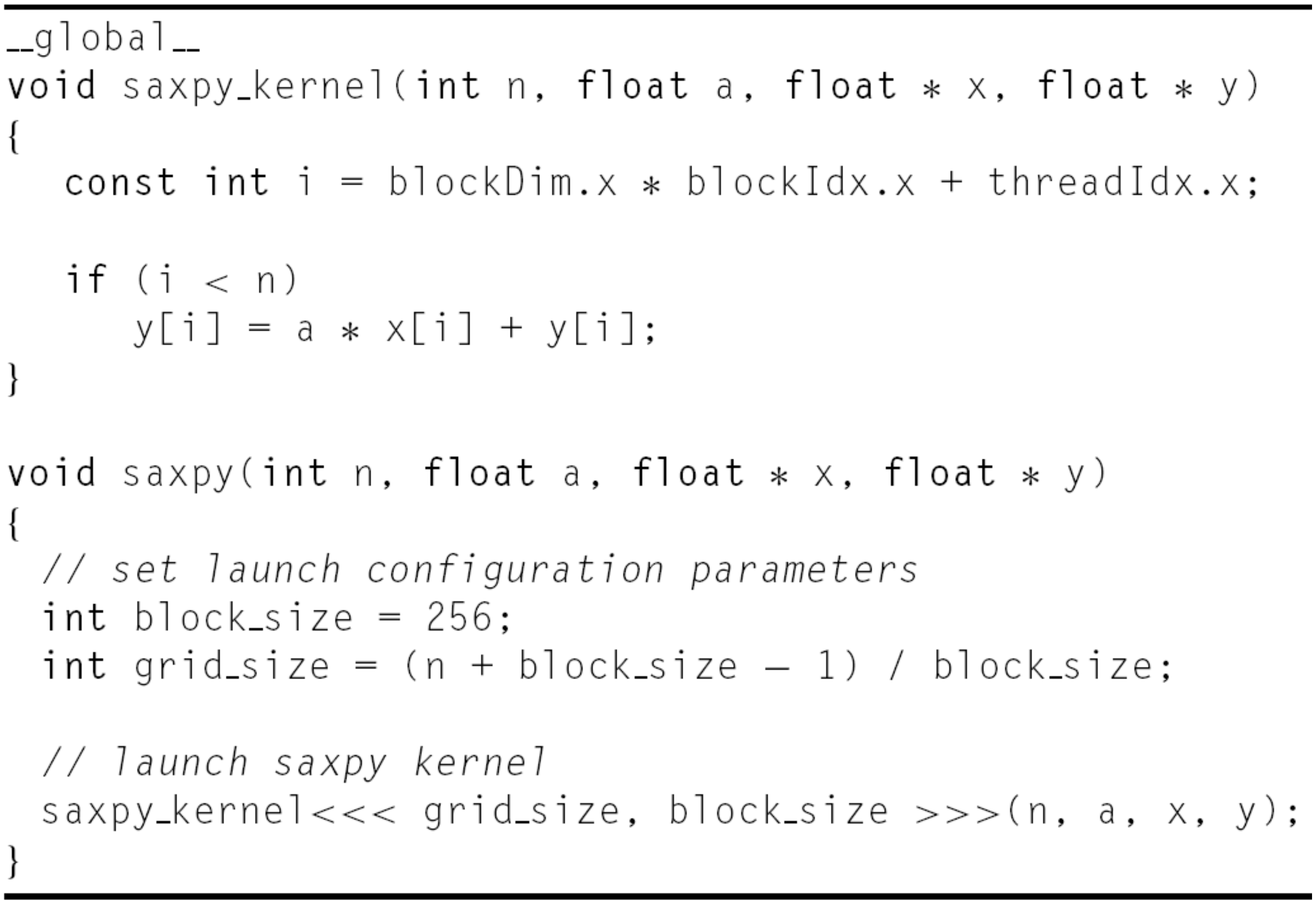
Element wise : for_each

```
struct normalize_functor
  : public unary_function<double4, double4> {
  __device__ __host__ double4 operator()(double4 v)
  {
        double len = sqrt(v.x*v.x + v.y*v.y + v.z*v.z);
        v.x /= len;
        v.y /= len;
        v.z /= len;
  }
}

device_vector<double4> d_vec = ...;
for_each(d_vec.begin(), d_vec.end(), normalize_functor() );
```

Element wise : transform

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

// allocate memory
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);

// transform A + B -> C
transform(A.begin(), A.end(),
          B.begin(), C.begin(),
          plus<int>());

// transform A - B -> C
transform(A.begin(), A.end(),
          B.begin(), C.begin(),
          minus<int> ());
```

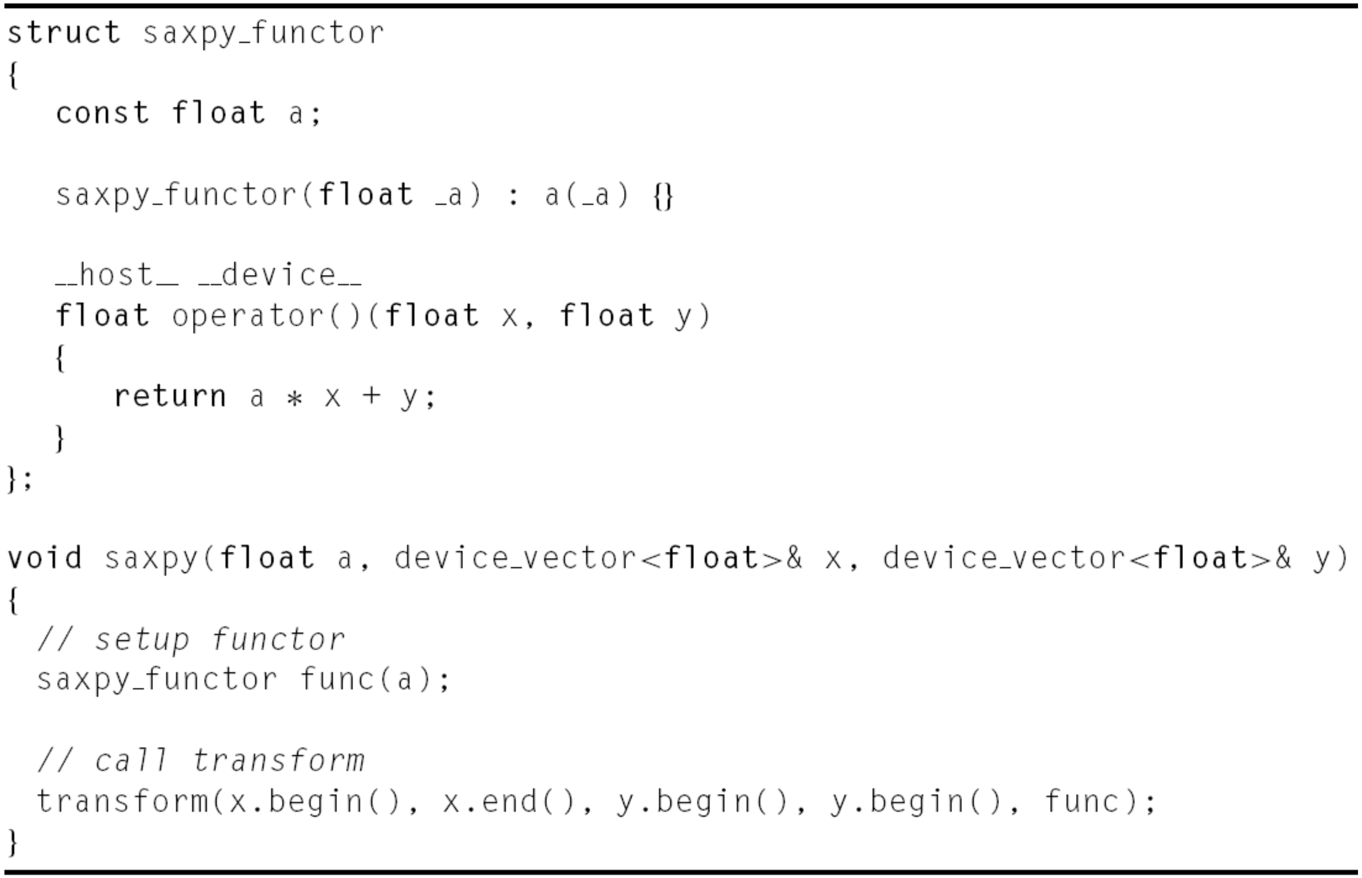
SAXPY via CUDA kernel



```
__global__
void saxpy_kernel(int n, float a, float * x, float * y)
{
   const int i = blockDim.x * blockIdx.x + threadIdx.x;

   if (i < n)
      y[i] = a * x[i] + y[i];
}

void saxpy(int n, float a, float * x, float * y)
{
  // set launch configuration parameters
  int block_size = 256;
  int grid_size = (n + block_size - 1) / block_size;

  // launch saxpy kernel
  saxpy_kernel<<< grid_size, block_size >>>(n, a, x, y);
}
```

(a) CUDA C

SAXPY via Thrust transform



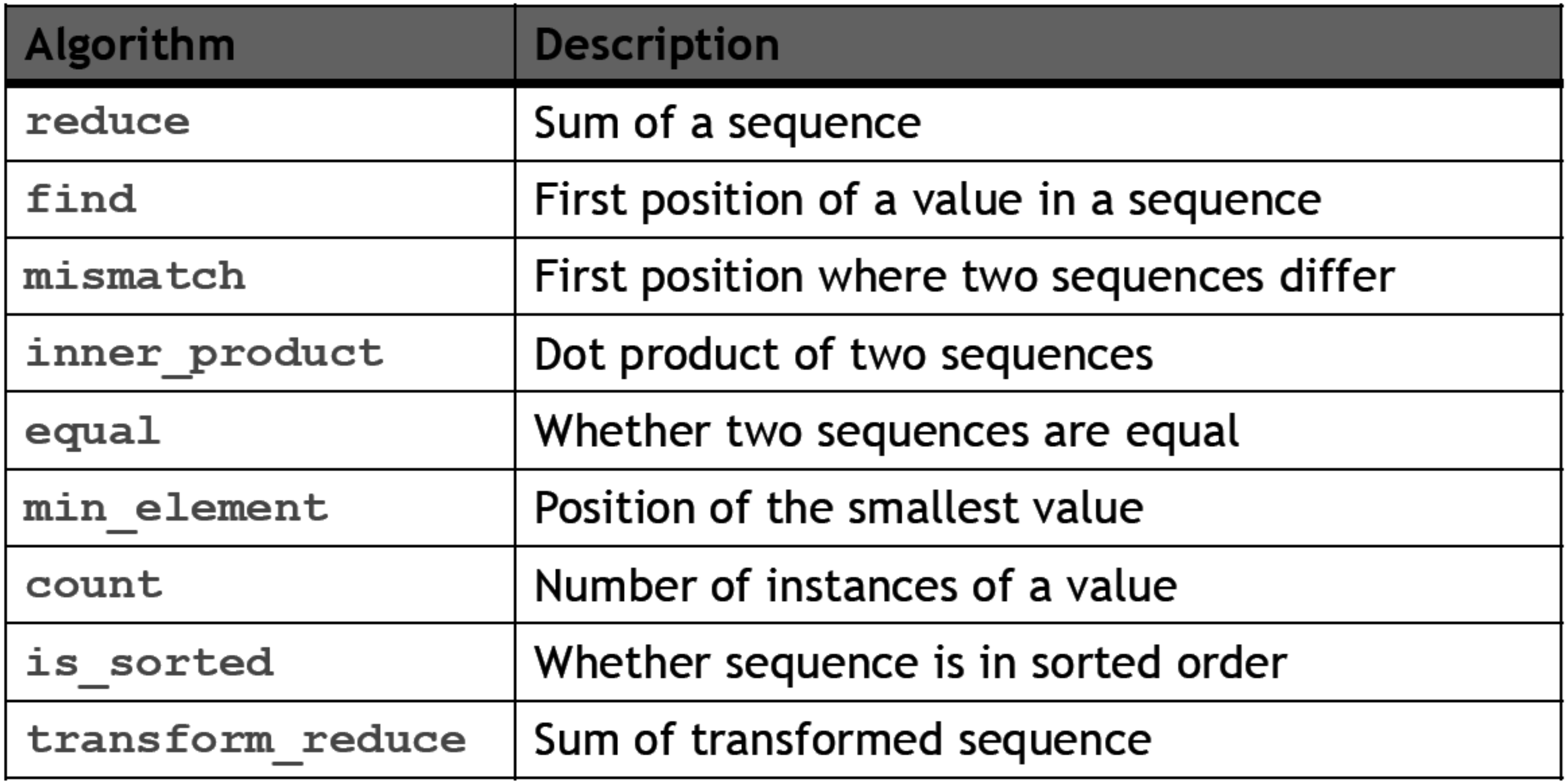
```
struct saxpy_functor
{
   const float a;

   saxpy_functor(float _a) : a(_a) {}

   __host__ __device__
   float operator()(float x, float y)
   {
      return a ∗ x + y;
   }
};

void saxpy(float a, device_vector<float>& x, device_vector<float>& y)
{
  // setup functor
  saxpy_functor func(a);

  // call transform
  transform(x.begin(), x.end(), y.begin(), y.begin(), func);
}
```

(b) Thrust

Element wise : custom transform

```
struct negate_float2 {
   __host__ __device__
   float2 operator()(float2a)
   {
     return make_float2(-a.x, -a.y);
   }
};

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

// create function object or ‘functor’
negate_float2 func;

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

Reduction



Algorithm	Description
`reduce`	Sum of a sequence
`find`	First position of a value in a sequence
`mismatch`	First position where two sequences differ
`inner_product`	Dot product of two sequences
`equal`	Whether two sequences are equal
`min_element`	Position of the smallest value
`count`	Number of instances of a value
`is_sorted`	Whether sequence is in sorted order
`transform_reduce`	Sum of transformed sequence

Reduction : reduce

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

// declare storage
host_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>());
```

Reduction : reduce – extrema calculation

template<typename ForwardIterator >	
ForwardIterator	thrust::min_element (ForwardIterator first, ForwardIterator last)
template<typename ForwardIterator , typename BinaryPredicate >	
ForwardIterator	thrust::min_element (ForwardIterator first, ForwardIterator last, BinaryPredicate comp)
template<typename ForwardIterator >	
ForwardIterator	thrust::max_element (ForwardIterator first, ForwardIterator last)
template<typename ForwardIterator , typename BinaryPredicate >	
ForwardIterator	thrust::max_element (ForwardIterator first, ForwardIterator last, BinaryPredicate comp)
template<typename ForwardIterator >	
thrust::pair< ForwardIterator, ForwardIterator >	thrust::minmax_element (ForwardIterator first, ForwardIterator last)
template<typename ForwardIterator , typename BinaryPredicate >	
thrust::pair< ForwardIterator, ForwardIterator >	thrust::minmax_element (ForwardIterator first, ForwardIterator last, BinaryPredicate comp)

Reduction : count_if

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

// count via a predicate (E.g. greater than a number)
struct is_greater_than {
    int threshold;

    is_greater_than(int t) { threshold = t; }

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

// create a device vector using a host array
const int N = 10;
int arr[N] = {1, 7, 1, 1, 3, 2, -1, 12, 6, 32};
thrust::device_vector<int> d_vec(arr, arr + N) ;

is_greater_than pred(2);
int count = thrust::count_if(d_vec.begin(), d_vec.end(), pred);
```

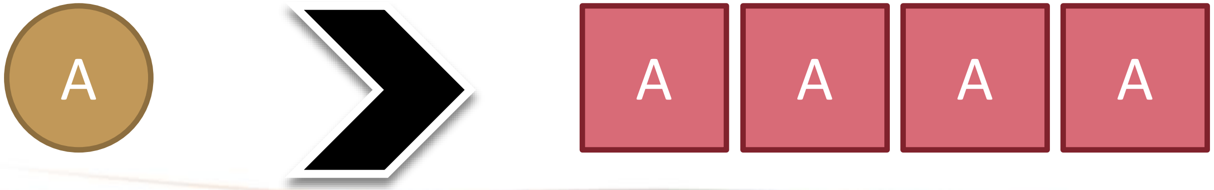
Iterators

- Behave like pointers
- Keep track of memory spaces
- Convertible to raw pointers
- Examples
  - constant_iterator,
  - counting_iterator,
  - transform_iterator
  - permutation_iterator
  - zip_iterator

Iterators - constant_iterator

- Mimics an infinite array filled with a constant value

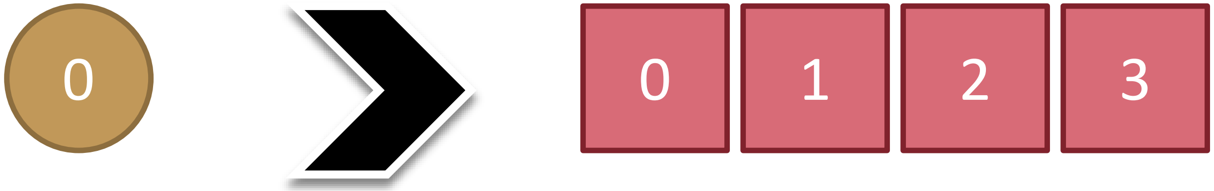
```
// create iterators
constant_iterator<int> begin(10);
constant_iterator<int> end = begin + 3;

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)
```



Iterators - counting_iterator

- Mimics an infinite array with sequential values

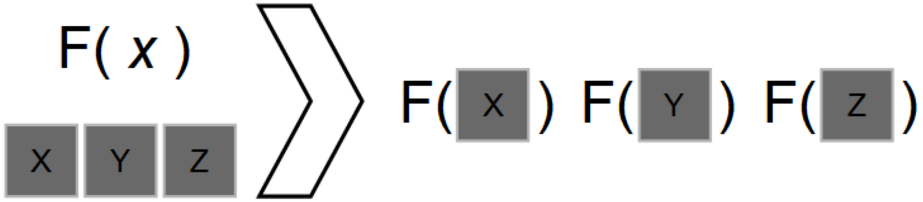
```
// create iterators
counting_iterator<int> begin(10);
counting_iterator<int> end = begin + 3;

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)
```



Iterators - transform_iterator

- Conserves memory capacity and bandwidth

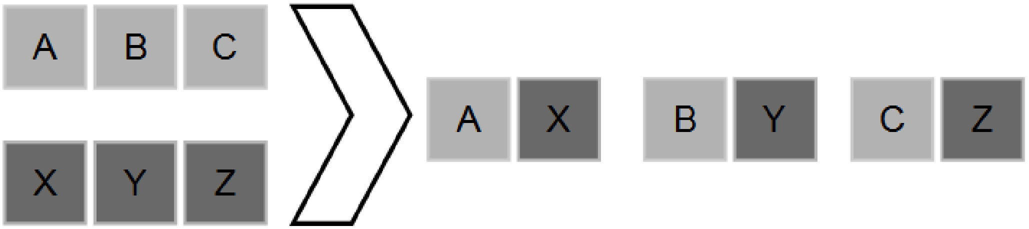
```
// initialize vector
device_vector<int> vec(3);
vec[0] = 10; vec[1] = 20; vec[2] = 30;

// 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)
```



Iterators – zip_iterator

- Support coalesced access via Structure of Arrays (SoA)

```
// 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')
```



Iterators - Slow Vector Rotation

```
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> v(n);
transform(v.begin(), v.end(), v.begin(), rotate_float3() );
```

Iterators - Fast Vector Rotation

```
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())),
   rotate_tuple() );
```

Kernel fusion - I

- `Consider z = g(f(x))`

```
device_vector<float> x(n);      // input
device_vector<float> f_x(n);    // temporary
device_vector<float> z(n);      // result

// compute f(x)
transform(x.begin(), x.end(), f_x.begin(), f() );

// compute g(f(x))
transform(f_x.begin(), f_x.end(), z.begin(), g() );
```

- Storage: 3 * n

- Bandwidth: 2 * n reads + 2 * n writes

- Temporaries: n

Kernel fusion - II

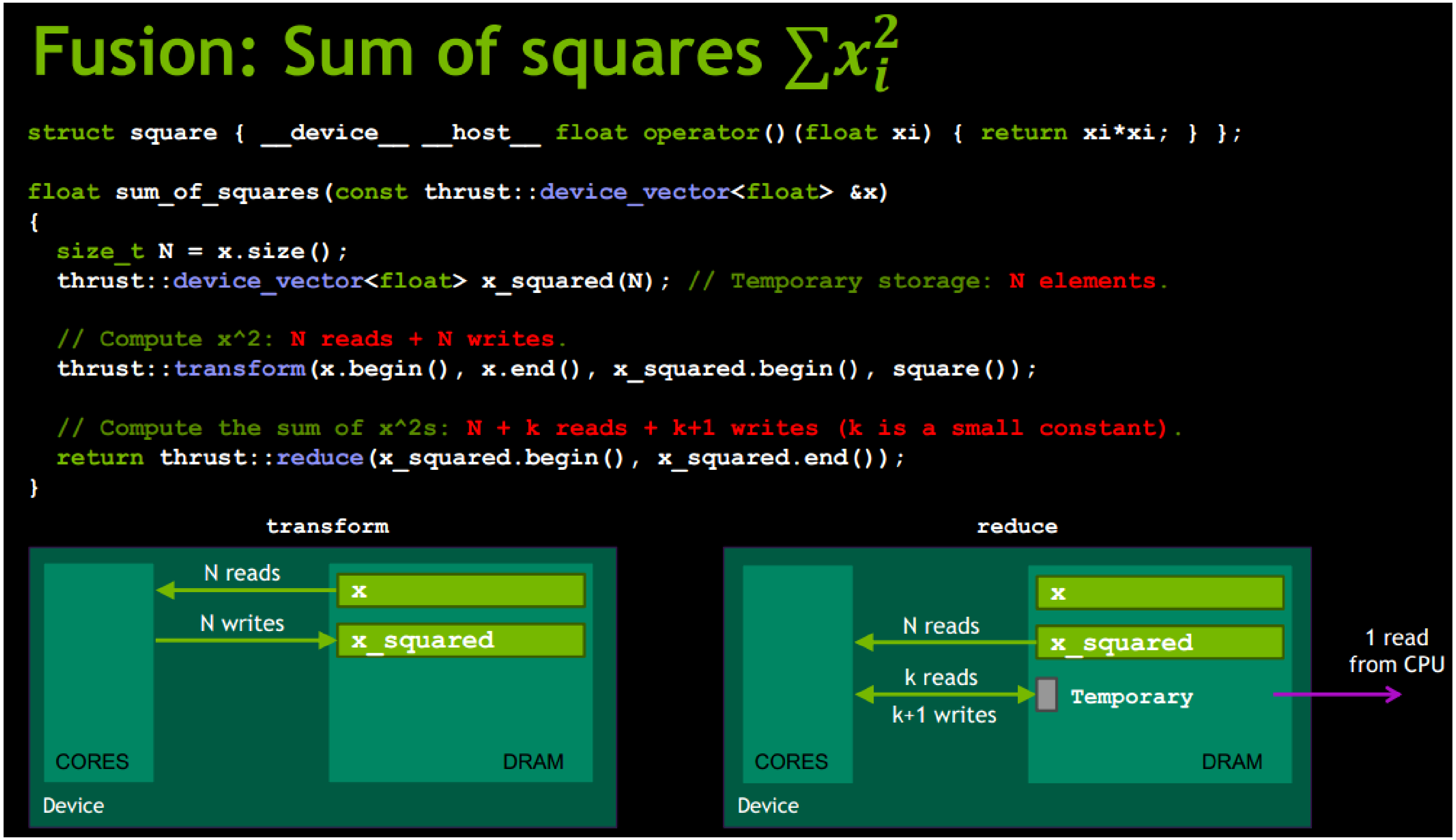
- A better way with **transform_iterator**

```
device_vector<float> x(n);      // input
device_vector<float> z(n);      // result

// compute g(f(x))
transform(make_transform_iterator(x.begin(), f() ),
          make_transform_iterator(x.end(), f() ),
          z.begin(),g() );
```

- Storage: 2 * n

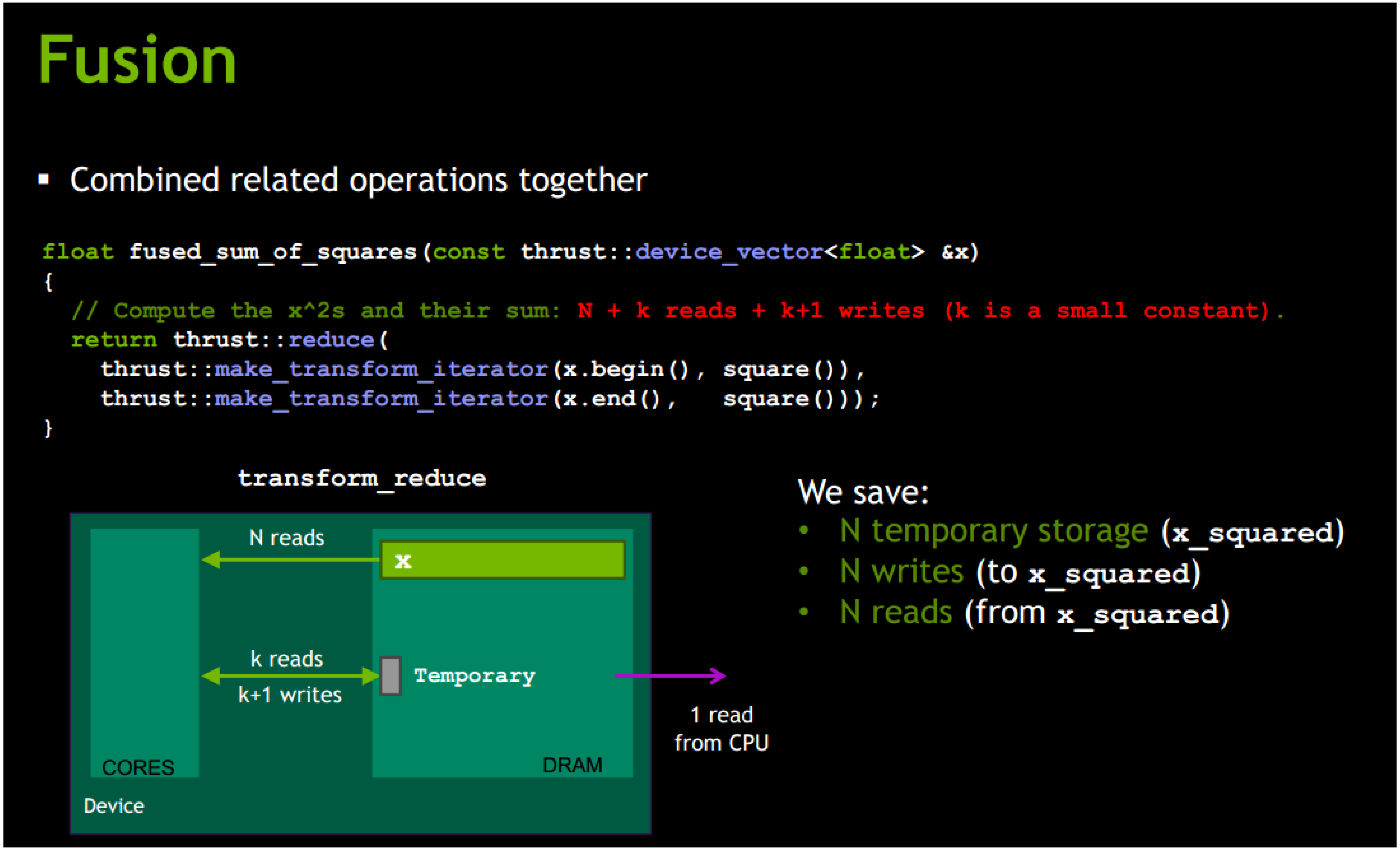
- Bandwidth: n reads + n writes

- Temporaries: 0



Fusion: Sum of squares $\sum x_i^2$

```
struct square { __device__ __host__ float operator()(float xi) { return xi*xi; } };

float sum_of_squares(const thrust::device_vector<float> &x)
{
  size_t N = x.size();
  thrust::device_vector<float> x_squared(N); // Temporary storage: N elements.

  // Compute x^2: N reads + N writes.
  thrust::transform(x.begin(), x.end(), x_squared.begin(), square());

  // Compute the sum of x^2s: N + k reads + k+1 writes (k is a small constant).
  return thrust::reduce(x_squared.begin(), x_squared.end());
}
```



Fusion

- Combined related operations together

```
float fused_sum_of_squares(const thrust::device_vector<float> &x)
{
  // Compute the x^2s and their sum: N + k reads + k+1 writes (k is a small constant).
  return thrust::reduce(
    thrust::make_transform_iterator(x.begin(), square()),
    thrust::make_transform_iterator(x.end(),   square()));
}
```

We save:

- N temporary storage (**x_squared**)
- N writes (to **x_squared**)
- N reads (from **x_squared**)

Kernel fusion - III

- Slow Vector Norm

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

device_vector<float> x_2(n); // temporary storage
transform(x.begin(), x.end(), x_2.begin(), square());
return sqrt(reduce(x_2.begin(), x_2.end()),
                    0.0f, plus<float>()));
```

Kernel fusion - IV

- Fast Vector Norm by Fusion

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

// fusion with transform_iterator
return sqrt(
  reduce(make_transform_iterator(x.begin(),square() ),
         make_transform_iterator(x.end(), square() ),
         0.0f, plus<float>() ) );
```

Kernel fusion - V

- Fast Vector Norm using transform_reduce

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

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

Speedup:
7.0x (GTX 480)
4.4x (GTX 280)

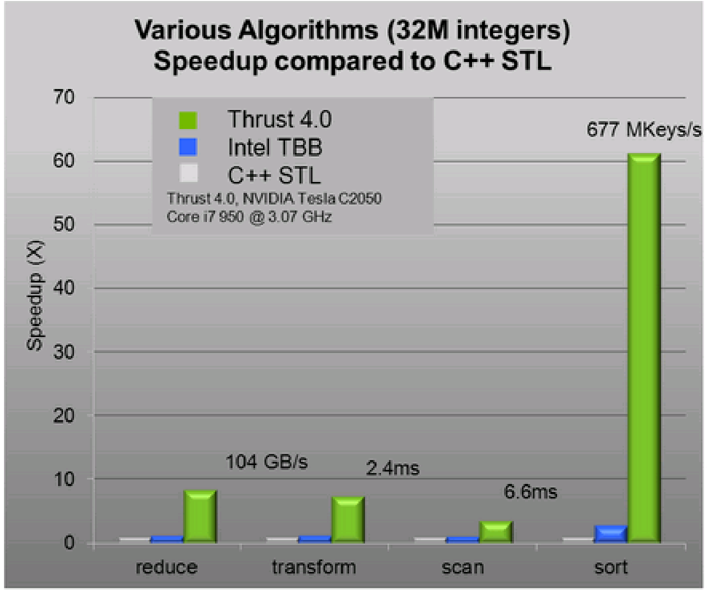
Prefix-Sums / Scan Operations

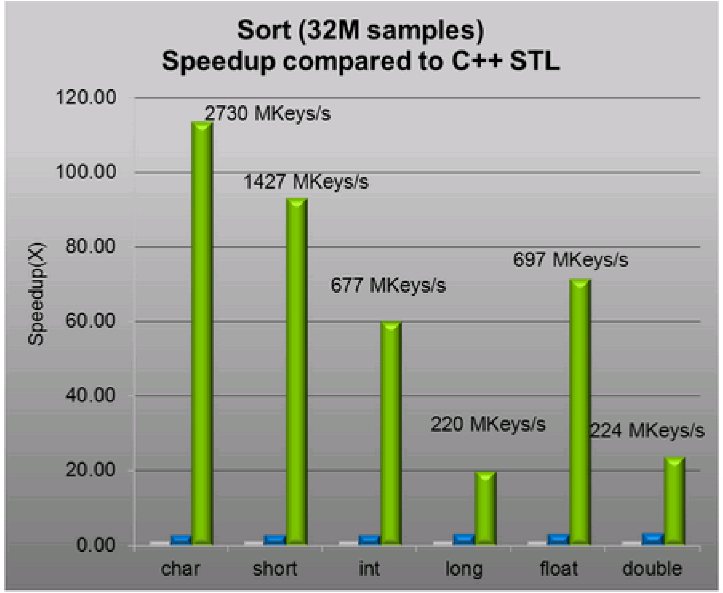
- Important building blocks in many parallel algorithms
  - such as stream compaction and radix sort
  - transform_inclusive_scan and transform_exclusive_scan applies a unary function before scan operation

```
// inclusive scan operation using the default plus operator
#include <thrust/scan.h>

int data [6] = {1, 0, 2, 2, 1, 3};
inclusive_scan(data , data + 6, data ); // in - place scan
// data is now {1, 1, 3, 5, 6, 9}

exclusive_scan(data , data + 6, data ); // in - place scan
// data is now {0, 1, 1, 3, 5, 6}
```

Sorting : sort

```
// 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());
```





Sorting : stable_sort, sort_by_key

- Stable sort preserves the relative ordering of equivalent elements

```
thrust::stable_sort(d_vec.begin(), d_vec.end());
```

- Sorts the data by given key

```
const int N = 6;
int keys[N] = { 1, 4, 2, 8, 5, 7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::sort_by_key(keys, keys + N, values );
// keys is now { 1, 2, 4, 5, 7, 8}
// values is now {'a ', 'c ', 'b ', 'e ', 'f ', 'd '}
```

Sources

- Nvidia Research
  - “CUDA Toolkit 4.0 - Thrust Quick Start Guide”
  - “An Introduction To Thrust”, Jared Hoberock & Nathan Bell
  - “High-Productivity CUDA Development with the Thrust Template Library”, Nathan Bell
  - “Thrust by Example - Advanced Features and Techniques”, Jared Hoberock
  - “Rapid Problem Solving using Thrust”, Nathan Bell
  - “Thrust: A Productivity-Oriented Library for CUDA”, Nathan Bell and Jared Hoberock
- Others
  - “CUDA Tricks”, Damodaran Ramani