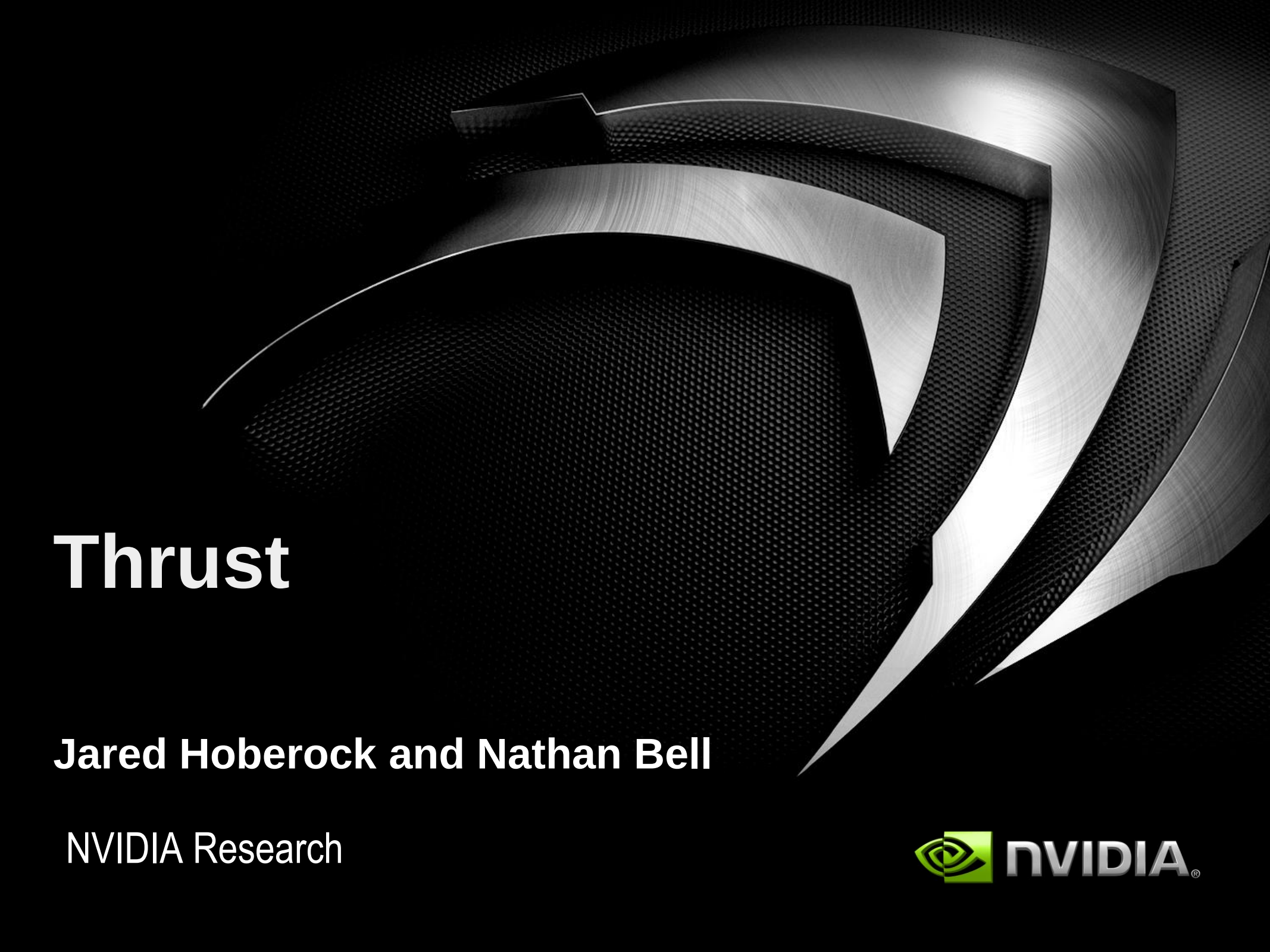


Thrust

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 (145M keys per second on a GTX 280)
    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

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()`
  - `thrust::segmented_inclusive_scan()`
  - Etc.

Containers

- Make common operations concise and readable
  - Hides cudaMalloc & cudaMemcpy

```
// 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;
```

Iterators & Memory Spaces

- 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());
```

Algorithms

- General types and operators

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

Trivial CUDA Interoperability

- 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!
```

Trivial CUDA Interoperability

- Wrap raw pointers with `device_ptr`

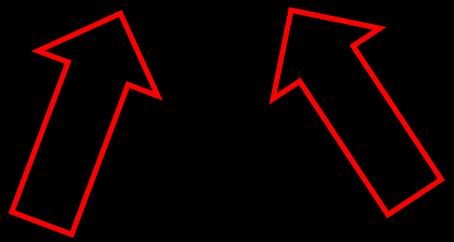
```
int N = 10;

// 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);
```

Why Bother?


```
__global__
void SAXPY(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];
}


SAXPY <<< n/256, 256 >>>(n, a, x, y);
```

Why Bother?

```
__global__
void SAXPY(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];
}


SAXPY <<< n/256, 256 >>>(n, a, x, y);
```



arbitrary constants

fails for N ≥ 16M

Why Bother?

- Arbitrary constants
  - Overly specific
  - Good for G80 is not necessarily good for Fermi
  - What runs on GT200 does not necessarily run on G80
- Sharp edges
  - 64K grid dimension limits
  - 256 byte limit for `__global__` function arguments
  - Device-dependent resource limits
  - Coalescing requirements
- Maintainability
  - One kernel per (unique) `for` loop is untenable

A Better Way


```
// C++ functor replaces __global__ function
struct saxpy
{
    const float a;

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

    __host__ __device__
    float operator()(const float& x, const float& y) const
    {
        return a * x + y;
    }
};


thrust::transform(x.begin(), x.end(), y.begin(), y.begin(), saxpy(a));
```

A Better Way

- Arbitrary constants eliminated
  - Computation is *implicit*
  - Launch configuration *deferred* to Thrust
    - Thrust maximizes occupancy
- Sharp edges eliminated
  - Arbitrary input sizes
  - Arbitrarily large types (e.g. `sizeof(T) > 256` is OK)
  - Gracefully handles excessive register/smem usage
  - Workaround coalescing requirements
    - e.g. char arrays on G80
- Robust abstraction layer

Observations

- Abstractions matter more now
  - Pervasive parallelism will kill the ad hoc for loop
  - OpenMP `#pragmas` are an interim solution
- Sequential programming
  - Not always a compelling case
    - `std::for_each` vs. novel for loop
    - `std::accumulate` vs. novel for loop
  - `std::sort()` and `std::vector<T>` are popular
- Parallel programming
  - C++ weirdness* is tenable now
    - `thrust::for_each` vs. novel CUDA kernel
    - `thrust::reduce` vs. novel CUDA kernel

*can be made more accessible with C++0x

Example: SNRM2

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

// setup arguments
square      unary_op;
plus<float> binary_op;
float init = 0;

// initialize vector
device_vector<float>  A(3);
A[0] = 20;  A[1] = 30;  A[2] = 40;

// compute norm
float norm = sqrt( transform_reduce(A.begin(), A.end(),
                                    unary_op, init, binary_op) );
```

Example: Run-Length Encoding

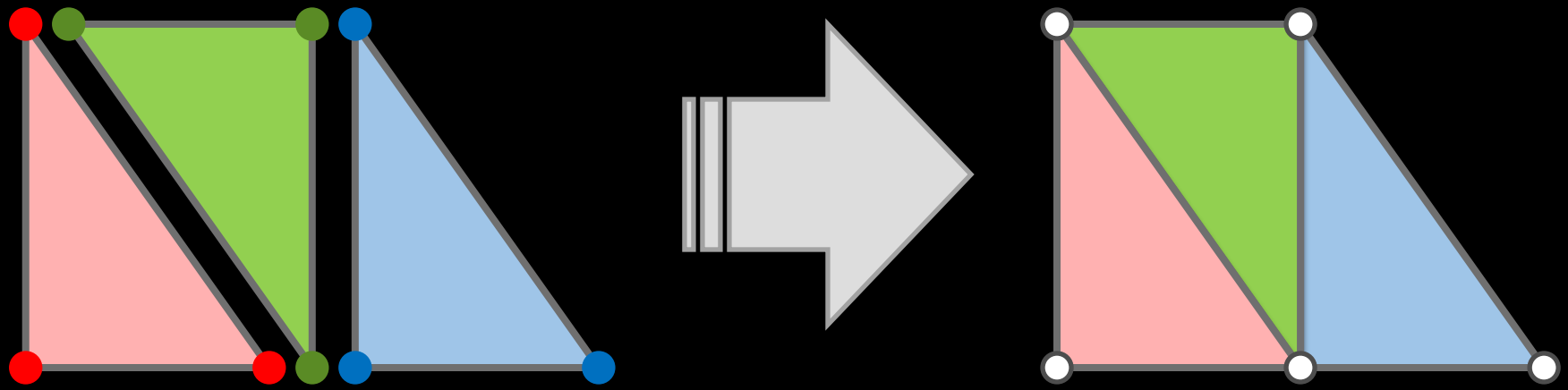

```
// input data on the host
const char data[] = "aaabbbbbcddeeeeeeeeeff";

const size_t N = (sizeof(data) / sizeof(char)) - 1;

// copy input data to the device
thrust::device_vector<char> input(data, data + N);

// allocate storage for output data and run lengths
thrust::device_vector<char> output(N);
thrust::device_vector<int> lengths(N);
```

Example: Run-Length Encoding


```
// compute run lengths -- one call!
size_t num_runs =
 thrust::reduce_by_key(
   input.begin(), input.end(),          // input key sequence
   thrust::constant_iterator<int>(1), // input value sequence
   output.begin(),                      // output key sequence
   lengths.begin()                      // output value sequence
   ).first - output.begin();            // compute the output size

// print the output
std::cout << "run-length encoded output:" << std::endl;
for(size_t i = 0; i < num_runs; i++)
  std::cout << "(" << output[i] << "," << lengths[i] << ")";
std::cout << std::endl;

// (a,3)(b,5)(c,1)(d,2)(e,9)(f,2) is printed
```

Example: Weld Vertices

- Problem: Marching Cubes produces “triangle soup”
- Want to “weld” vertices together into connected mesh



Example: Weld Vertices


```
// allocate memory for input mesh representation
thrust::device_vector<float2> input(9);

// create triangles...
input[0] = make_float2(0,0); // First Triangle
input[1] = make_float2(1,0);
input[2] = make_float2(0,1);
...

// allocate space for output mesh representation
thrust::device_vector<float2> vertices = input;
thrust::device_vector<unsigned int> indices(input.size());

// sort vertices to bring duplicates together
thrust::sort(vertices.begin(), vertices.end(), float2_less());
```

Example: Weld Vertices


```
// find unique vertices and erase redundancies
vertices.erase(thrust::unique(vertices.begin(), vertices.end(),
   float2_equal_to()), vertices.end());

// find index of each input vertex in the list of unique vertices
thrust::lower_bound(vertices.begin(), vertices.end(),
                    input.begin(), input.end(),
                    indices.begin(),
                    float2_less());

// ... done!
```

More Examples

- First four statistical moments in one pass
- Estimating pi with Monte Carlo sampling
- Lexicographical sorting
- Sorting into cells of 2d grid
- Reduction over padded 2d grid
- Bounding box of 2d point set
- Counting words

- Many more!

Power of Abstraction

- CUDA C is low-level and explicit
  - Desirable in a performance layer
  - Requires substantial knowledge to use effectively
- Thrust is high-level and abstract
  - Desirable in a productivity layer
  - Delivers high real-world performance
  - Affords a lot of discretion in implementation
    - Thrust : primitives :: BLAS : linear algebra

Power of Abstraction

- `thrust::sort()`
  - Static optimization
    - Radix Sort (`char, short, int, long, float, double`)
    - Merge Sort (structures, classes, non-standard comparisons)
  - Dynamic optimization
    - Compute min/max key to reduce Radix Sort cost
    - E.g. if all keys [0, 16) then 7x speedup
    - Full perf details on developer blog
- Allows for further improvement
  - Again, like BLAS

Power of Abstraction

- Performance Portability
  - Thrust-Multicore was (relatively) easy
- Toggle CUDA ↔ OpenMP with a compiler switch

GeForce GTX 280

```
$ time ./monte_carlo
pi is around 3.14164

real    0m0.222s
user    0m0.120s
sys     0m0.100s
```

Core2 Quad Q6600

```
$ time ./monte_carlo
pi is around 3.14163

real    0m2.090s
user    0m8.333s
sys     0m0.000s
```

Future Directions

- Segmented operations
- Nested parallelism
- Transparent scaling to multi-GPU
- Task parallelism
- Accessible interface

Summary

- Thrust is a parallel C++ STL for CUDA
- Productive and performant
- Performance portable
- Solves real problems today

Further Reading

- Thrust
  - Homepage
    - http://code.google.com/p/thrust/
  - Quick Start Guide
    - http://code.google.com/p/thrust/wiki/QuickStartGuide
  - Documentation
    - http://code.google.com/p/thrust/wiki/Documentation
- NVIDIA Research
  - http://www.nvidia.com/research
- CUDA
  - http://www.nvidia.com/cuda

Questions?

jhoberock@nvidia.com

http://www.nvidia.com/cuda