GPU TECHNOLOGY CONFERENCE

Thrust by Example

Advanced Features and Techniques

San Jose, California | Thursday, September 23, 2010 | Jared Hoberock (NVIDIA Research)

PRESENTED BY NVIDIA.

What is Thrust?

```
#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 (805 Mkeys/sec on GeForce GTX 480)
  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;
}
```

Background

- Thrust is a parallel implementation of the C++ STL
  - Containers and Algorithms
  - CUDA and OpenMP backends
- This talk assumes basic C++ and Thrust familiarity
  - Templates
  - Iterators
  - Functors

Roadmap

- CUDA Best Practices
- How to Realize Best Practices with Thrust
- Examples
- Extended example: 2D Bucket Sort
- Performance Analysis

Best Practices

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

Fusion

- Memory bandwidth is scarce
- Many computations have low computational intensity
- Keep intermediate results on chip

Fusion

- 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());
```

Fusion

- 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

Fusion

- 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());
```

GPU TECHNOLOGY CONFERENCE

Fusion

- 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
```

Example: 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>()));
```

GPU TECHNOLOGY CONFERENCE

Example: Fast Vector Norm

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

Example: Fast Vector Norm

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

Example: Fast Vector Norm

```
// 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)

Structure of Arrays

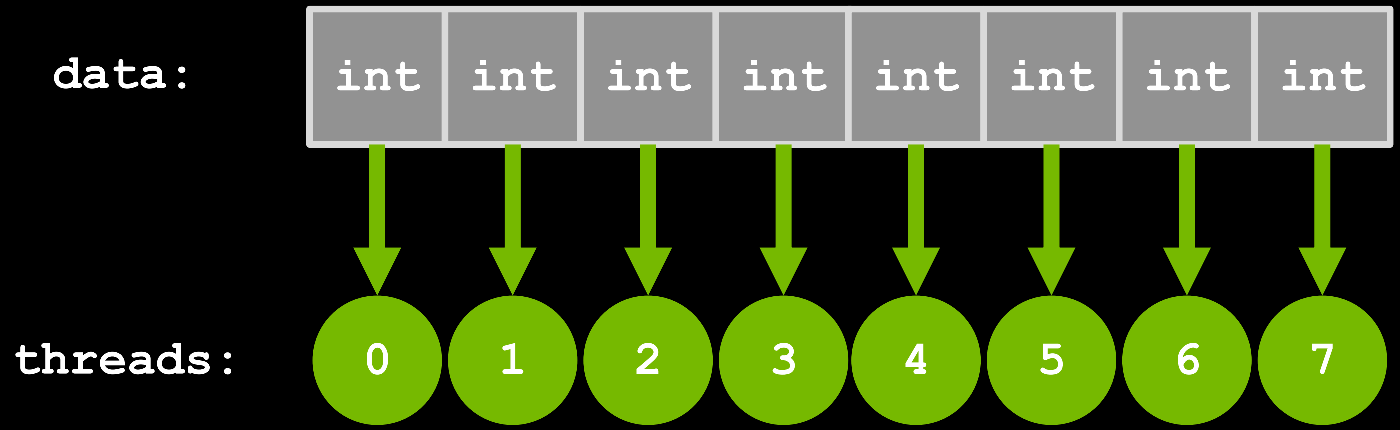
- Coalescing improves memory efficiency
- Accesses to arrays of arbitrary structures won't coalesce
- Reordering into structure of arrays ensures coalescing

```
struct float3
{
  float x;
  float y;
  float z;
};

float3 *aos;
...
aos[i].x = 1.0f;
```

```
struct float3_soa
{
  float *x;
  float *y;
  float *z;
};

float3_soa soa;
...
soa.x[i] = 1.0f;
```

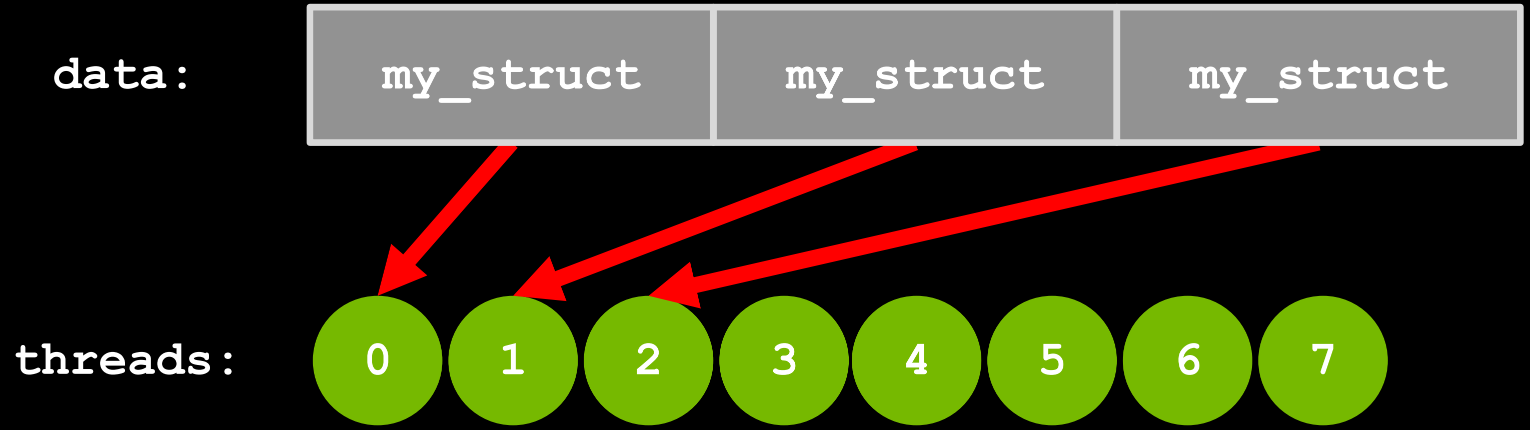
Structure of Arrays

- Coalescing (Simple version)
  - Contiguous 4, 8, or 16 byte accesses are good



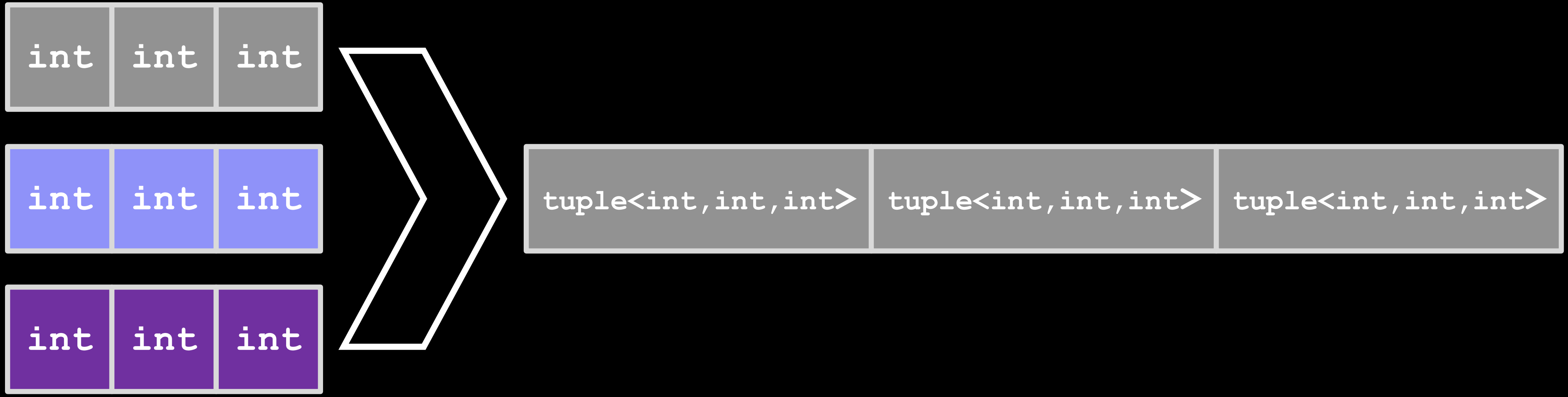
Structure of Arrays

- Coalescing (Simple version)
  - Arrays of arbitrary data structures won't necessarily coalesce
  - Large performance penalty on older hardware



Structure of Arrays

- `zip_iterator` "zips" up arrays into `tuple` on the fly
  - Performance benefit of coalescing
  - Conceptual goodness of structs



Example: 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> vec(n);
transform(vec.begin(), vec.end(), vec.begin(), rotate_float3());
```

Example: 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());
```

Example: 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());
```

Speedup: 1.3x (GTX 480)
2.5x (GTX 280)

Implicit Sequences

- Often we need ranges following a sequential pattern
  - Constant ranges
    - `[1, 1, 1, 1, ...]`
  - Incrementing ranges
    - `[0, 1, 2, 3, ...]`
- Implicit ranges require no storage
  - `constant_iterator`
  - `counting_iterator`

Example: Slow Min Index

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

device_vector<int> indices(vec.size());   // allocate storage for explicit indices
sequence(indices.begin(), indices.end()); // indices = [0, 1, 2, ...)

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);
```

Example: Fast Min Index

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

counting_iterator<int> indices_begin(0);  // create implicit range [0, 1, 2, ...)
counting_iterator<int> indices_end(vec.size());

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);
```

Example: Fast Min Index

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

counting_iterator<int> indices_begin(0);  // create implicit range [0, 1, 2, ...)
counting_iterator<int> indices_end(vec.size());

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);
```

Speedup: 2.7x (GTX 480)
3.2x (GTX 280)

Example: Fast Min Index

```
// min_element implements this operation directly
return min_element(vec.begin(), vec.end()) - vec.begin();
```

```
Speedup: 2.7x (GTX 480)
         3.2x (GTX 280)
```

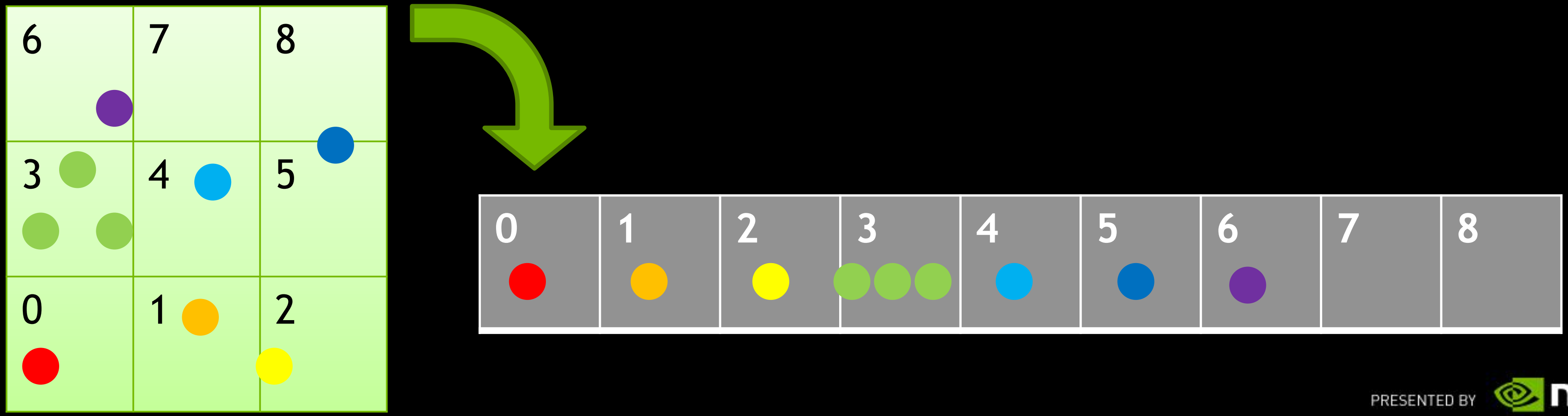
Recap: Best Practices

- Fusion
  - Eliminate superfluous memory traffic & storage
  - `transform_iterator` and `transform_` algorithms
- Structure of arrays
  - Ensures coalescing
  - `zip_iterator`
- Implicit sequences
  - Avoid explicitly storing and accessing regular patterns
  - `constant_iterator` & `counting_iterator`

Example: 2D Bucket Sort

Procedure:

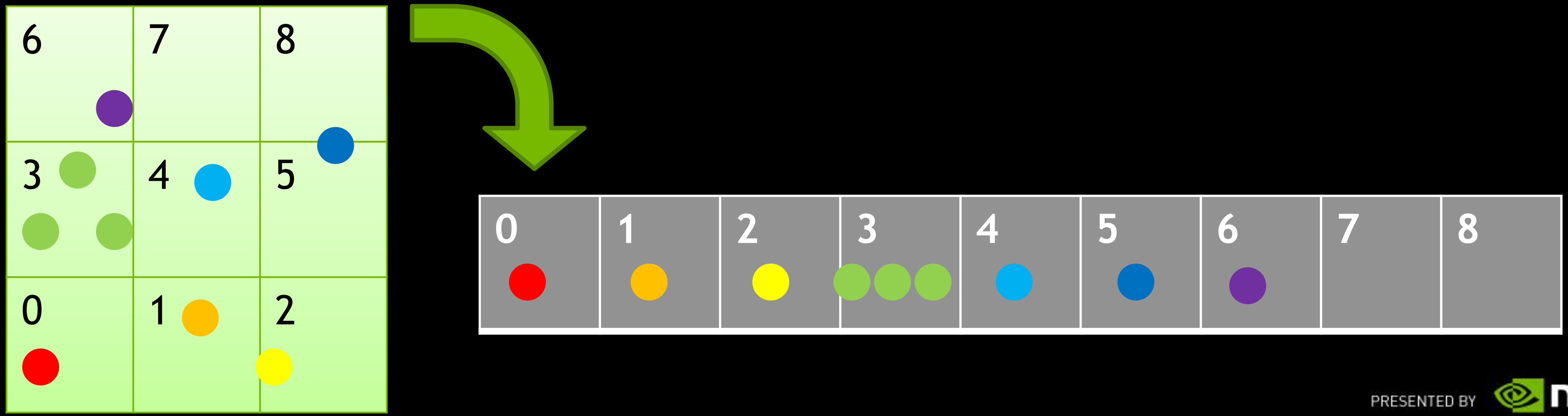
1. create random points
2. compute bucket index for each point
3. sort points by bucket index
4. count size of each bucket



Implementation Choices

Procedure:

1. create random points
2. compute bucket index for each point
3. sort points by bucket index
4. count size of each bucket



Implementation Choices

- Where to generate random input?
  - Host versus device?
- How to compute the size of each bucket
  - Binary search versus reduction?
- Performance versus concision

Step 1: Create random points

- On the host:

```
// return a random float2 in [0,1)^2
float2 make_random_float2(void)
{
  return make_float2( rand() / (RAND_MAX + 1.0f),
                      rand() / (RAND_MAX + 1.0f) );
}

// allocate some random points in the unit square on the host
host_vector<float2> h_points(N);
generate(h_points.begin(), h_points.end(), make_random_float2);

// transfer to device
device_vector<float2> points = h_points;
```

Step 1: Create random points

- On the device with an RNG:

```
struct make_random_float2
{
  __host__ __device__
  float2 operator()(int index)
  {
    default_random_engine rng;

    // skip past numbers used in previous threads
    rng.discard(2*index);
    return make_float2( (float)rng() / default_random_engine::max,
                        (float)rng() / default_random_engine::max);
  }
};

// generate random input directly on the device
device_vector<float2> points(N);
transform(make_counting_iterator(0),
          make_counting_iterator(N),
          points.begin(), make_random_float2());
```

Step 1: Create random points

- On the device with an integer hash:

```
struct make_random_float2
{
  __host__ __device__
  float2 operator()(int index)
  {
    return make_float2( (float)hash(2*index + 0) / UINT_MAX,
                        (float)hash(2*index + 1) / UINT_MAX);
  }
};

// generate random input directly on the device
device_vector<float2> points(N);
transform(make_counting_iterator(0),
          make_counting_iterator(N),
          points.begin(), make_random_float2());
```

GPU TECHNOLOGY CONFERENCE

PRESENTED BY NVIDIA.

Step 1: Create random points

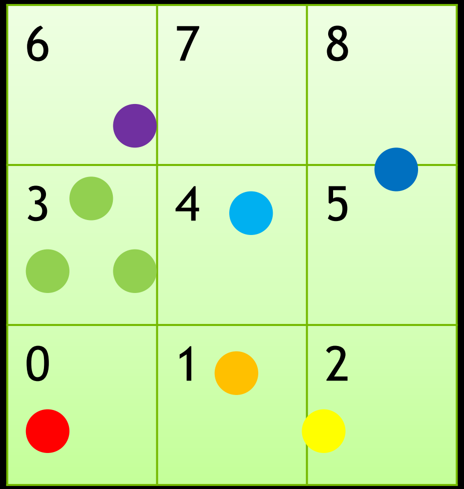
- Host implementation causes unnecessary serialization and copy
- Device implementation performed in situ but has different tradeoffs
  - `rng.discard(n)` is `lg(n)`
  - Integer hash is fast but low quality

Step 2: Compute bucket index for each point

```
// a functor hashing points to indices
struct point_to_bucket_index
{
  unsigned int w, h;

  __host__ __device__
  point_to_bucket_index(unsigned int width, unsigned int height)
    :w(width), h(height){}

  __host__ __device__
  unsigned int operator()(float2 p)
  {
    // coordinates of the grid cell containing point p
    unsigned int x = p.x * w;
    unsigned int y = p.y * h;

    // return the bucket's linear index
    return y * w + x;
  }
};
```

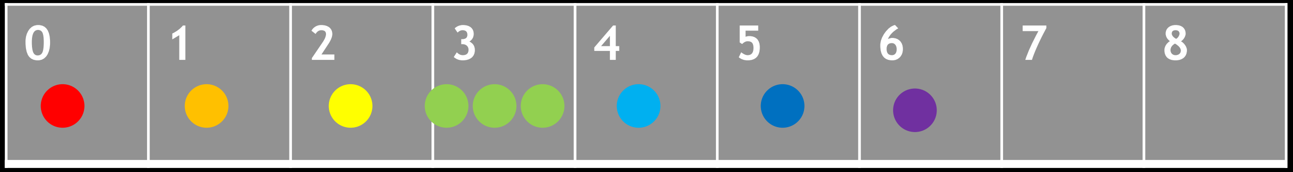
Step 2: Classify each point

```
// resolution of the 2D grid
unsigned int w = 200;
unsigned int h = 100;

// allocate storage for each point's bucket index
device_vector<unsigned int> bucket_indices(N);

// transform the points to their bucket indices
transform(points.begin(),
          points.end(),
          bucket_indices.begin(),
          point_to_bucket_index(w,h));
```



Step 3: Sort points by bucket index

```
sort_by_key(bucket_indices.begin(), bucket_indices.end(), points.begin());
```



Step 4: Compute the size of each bucket

- Using reduction:

```
// allocate space to hold per-bucket sizes
device_vector<int> bucket_sizes(width * height);

// allocate some random points in the unit square on the host
reduce_by_key(bucket_indices.begin(), bucket_indices.end(),
              make_constant_iterator(1),
              bucket_indices.begin(),
              bucket_sizes.begin());

// keys    = {0, 0, 2, 3, 3, 3, 4, 6, 7, ... } // bucket_indices
// values  = {1, 1, 1, 1, 1, 1, 1, 1, 1, ... } // constant_iterator
// ==>
// output_keys   = {0, 2, 3, 4, 6, 7, ... }    // bucket_indices
// output_values = {2, 1, 3, 1, 1, 1, ... }    // bucket_sizes
// note empty buckets do not appear in the output
```

Step 4: Compute the size of each bucket

- Using binary search:

```
// bucket_begin[i] indexes the first element of bucket i
// bucket_end[i] indexes one past the last element of bucket i
device_vector<unsigned int> bucket_begin(w*h);
device_vector<unsigned int> bucket_end(w*h);

// used to produce integers in the range [0, w*h)
counting_iterator<unsigned int> search_begin(0);

// find the beginning of each bucket's list of points
lower_bound(bucket_indices.begin(), bucket_indices.end(),
            search_begin, search_begin + w*h, bucket_begin.begin());

// find the end of each bucket's list of points
upper_bound(bucket_indices.begin(), bucket_indices.end(),
            search_begin, search_begin + w*h, bucket_end.begin());
```

GPU TECHNOLOGY CONFERENCE

PRESENTED BY NVIDIA.

Step 4: Compute the size of each bucket

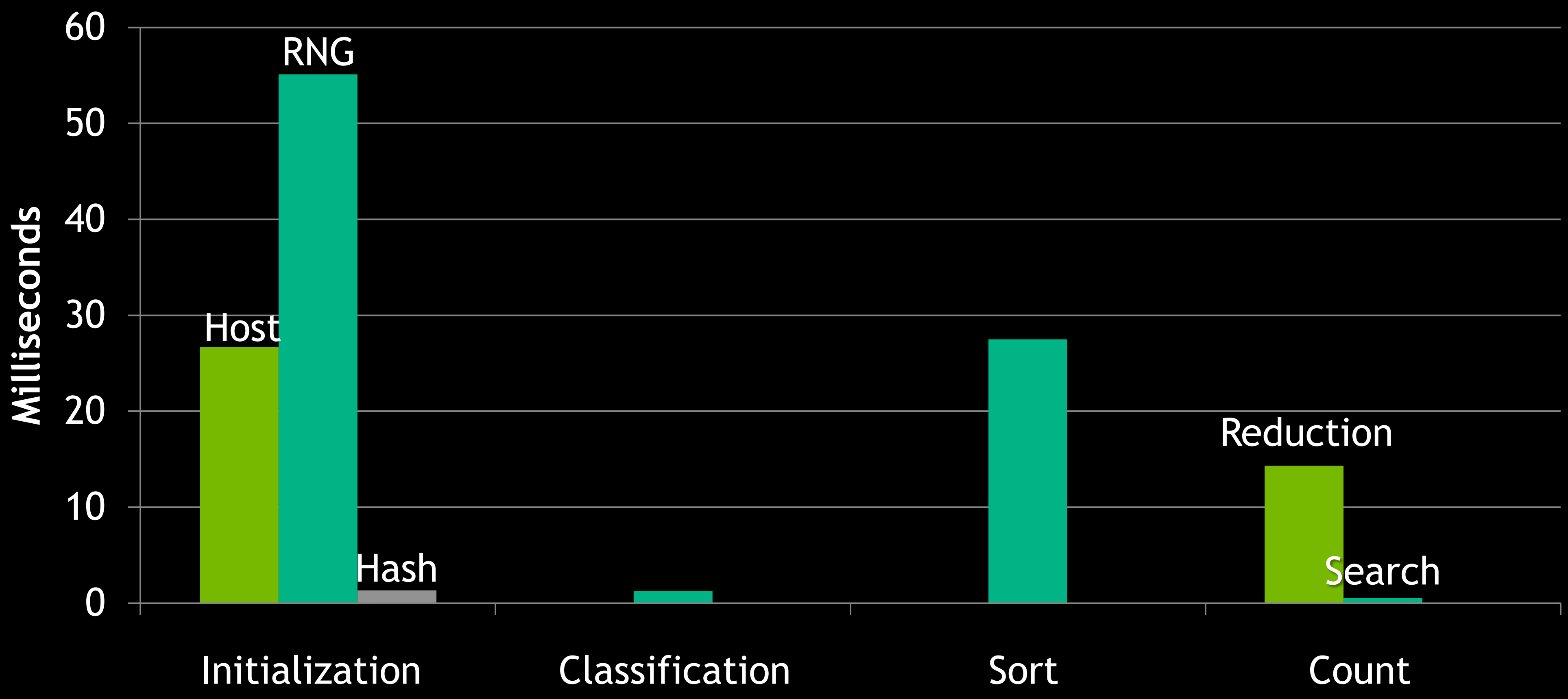
- Using binary search:

```
// allocate space to hold per-bucket sizes
device_vector<unsigned int> bucket_sizes(w*h);

// take the difference between bounds to find each bucket's size
transform(bucket_end.begin(), bucket_end.end(),
          bucket_begin.begin(),
          bucket_sizes.begin(),
          minus<unsigned int>());

// bucket_size[i] = bucket_end[i] - bucket_begin[i]

// note each bucket's size appears in the output even if it is empty
```

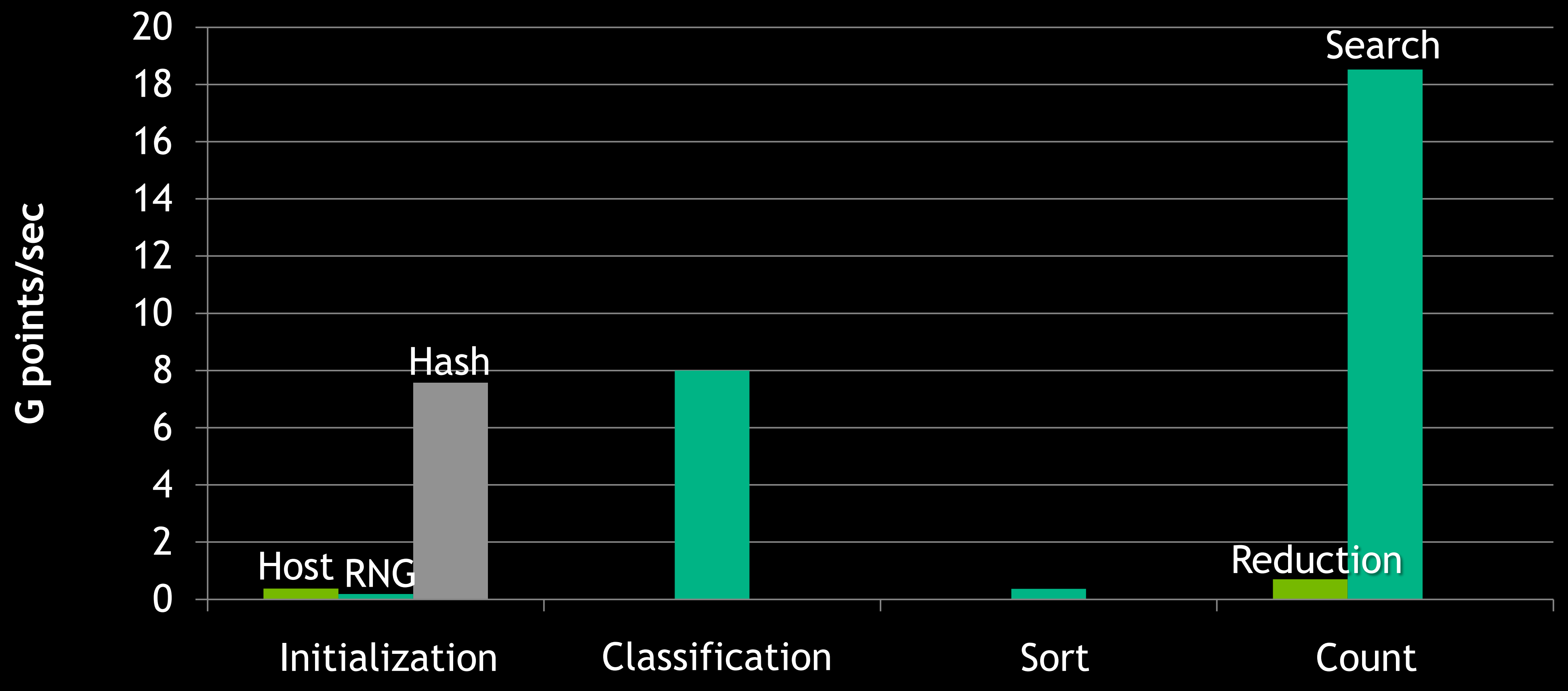
Performance Methodology

- 10M Points
- Software: Thrust 1.3 + nvcc 3.2
- Hardware: GeForce 480 GTX + Intel i7
- Memory allocation and copying is included in timings

Performance Profile (Lower is Better)



Throughput (Higher is Better)



Analysis

- Point generation
  - Host initialization is simple, but wastes memory and host->device bandwidth
  - Device initialization with RNG adds modest complexity but correctly avoiding correlation is expensive
  - Device initialization with hash is simple and fast but low quality
  - Generate input in situ when possible
  - Use a domain specific vendor RNG library if you need fast + high quality numbers

Analysis

- Sort
  - Dominates performance of the optimized implementation
  - Very fast when keys are primitives (`char`, `int`, `float`, etc.)
  - When in doubt, ask “Can `thrust::sort` solve this problem?”

Analysis

- Count
  - `reduce_by_key` is general & simple – use for prototyping
  - `transform` + `sort_by_key` + `reduce_by_key` = MapReduce
  - Common pattern: bring like items together and reduce
  - Examples: word count, histogram, rendering, run length encoding, etc.

Analysis

- Count
  - Binary search yielded a massive advantage over `reduce_by_key` given a little extra effort for this problem
  - Applicable to this case because keys happened to be sorted
  - Requires extra storage for interval bounds
  - `reduce_by_key` likely to get faster in the future

More Examples on Google Code Site

- Monte Carlo Integration
- Run-Length Encoding
- Summed Area Table
- Moving Average
- Word Count
- Voronoi Diagram
- Graphics Interop
- Mode
- Stream Compaction
- Lexicographical Sort
- Summary Statistics
- Histogram

And many more!

Summary

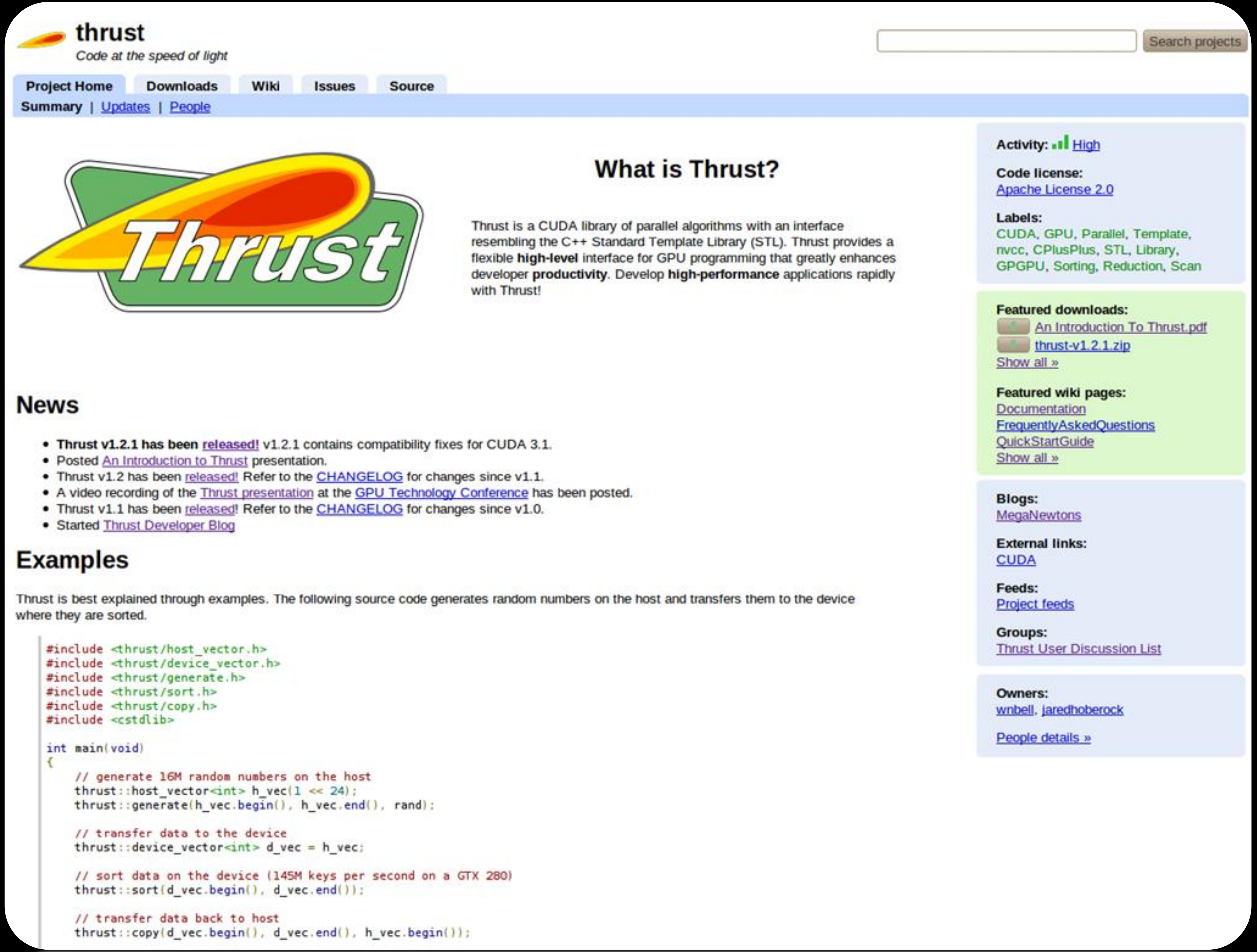
- CUDA best practices are easy with Thrust
  - Fusion: `transform_iterator`
  - SoA: `zip_iterator`
  - Sequences: `counting_iterator`
- Efficient use of memory
  - Maximize bandwidth
  - Eliminate redundancies
  - Perform operations in situ

Summary

- Thrust enables interesting CUDA applications with minimal effort
- `thrust::sort` everything
  - 805M keys/sec on GTX 480
  - Bring like items together, reduce
  - MapReduce

GPU TECHNOLOGY CONFERENCE

Thrust on Google Code



thrust
Code at the speed of light

Search projects

Project Home | Downloads | Wiki | Issues | Source

Summary | Updates | People

What is Thrust?

Thrust is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). Thrust provides a flexible **high-level** interface for GPU programming that greatly enhances developer **productivity**. Develop **high-performance** applications rapidly with Thrust!

News

- **Thrust v1.2.1 has been released!** v1.2.1 contains compatibility fixes for CUDA 3.1.
- Posted An Introduction to Thrust presentation.
- Thrust v1.2 has been released! Refer to the CHANGELOG for changes since v1.1.
- A video recording of the Thrust presentation at the GPU Technology Conference has been posted.
- Thrust v1.1 has been released! Refer to the CHANGELOG for changes since v1.0.
- Started Thrust Developer Blog

Examples

Thrust is best explained through examples. The following source code generates random numbers on the host and transfers them to the device where they are sorted.

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

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());
```

Activity: High

Code license:
Apache License 2.0

Labels:
CUDA, GPU, Parallel, Template, nvcc, CPlusPlus, STL, Library, GPGPU, Sorting, Reduction, Scan

Featured downloads:
An Introduction To Thrust.pdf
thrust-v1.2.1.zip
Show all »

Featured wiki pages:
Documentation
FrequentlyAskedQuestions
QuickStartGuide
Show all »

Blogs:
MegaNewtons

External links:
CUDA

Feeds:
Project feeds

Groups:
Thrust User Discussion List

Owners:
wnbell, jaredhoberock

People details »

PRESENTED BY NVIDIA

Resources

- Quick Start Guide
- API Documentation
- Source Code
- Mailing List: thrust-users on Google Groups

Questions?

jhoberock@nvidia.com

http://thrust.googlecode.com