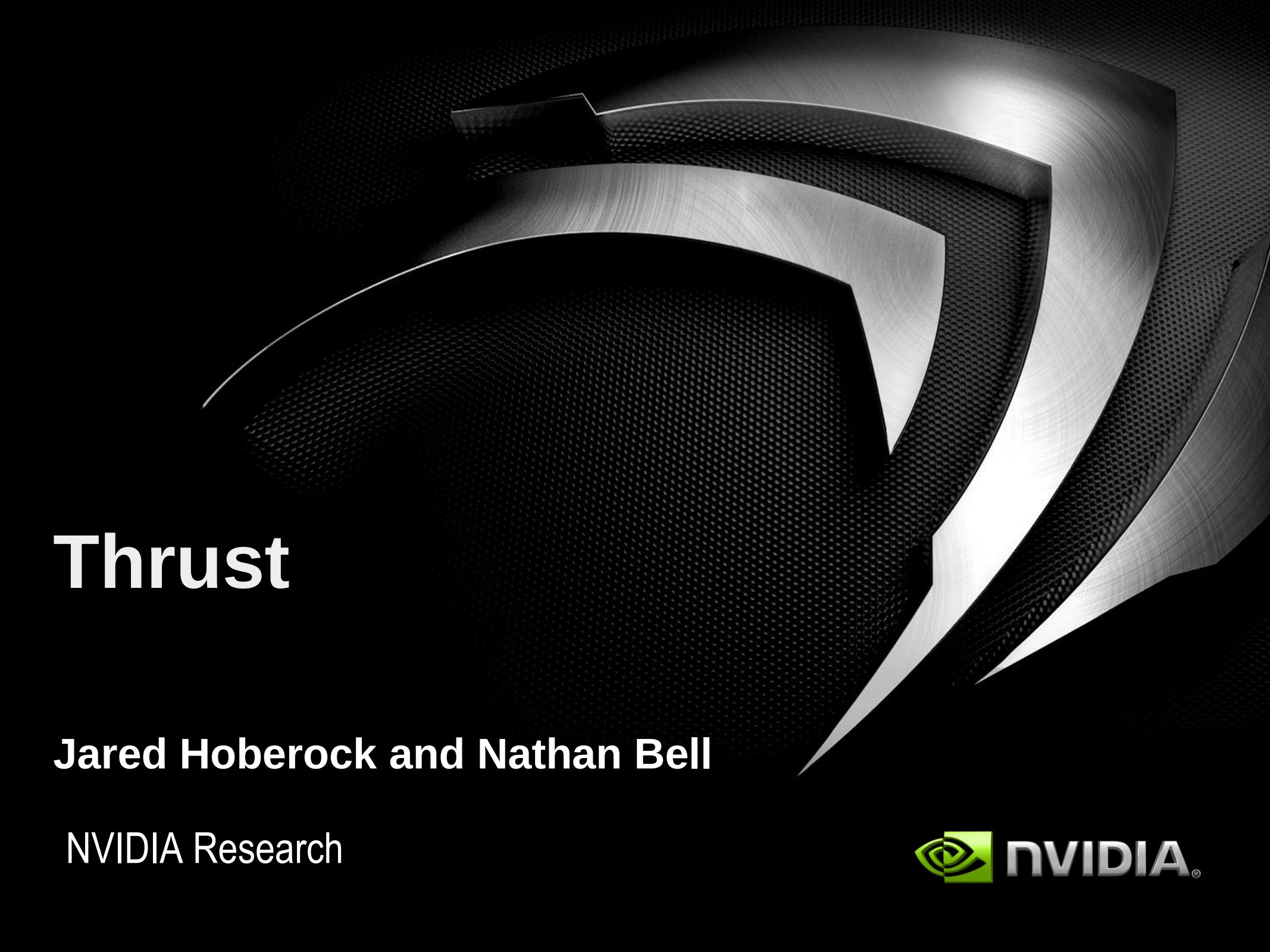


# Thrust

**Jared Hoberock and Nathan Bell**

NVIDIA Research

# 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

# Facts and Figures

- Thrust v1.0
  - Open source (Apache license)
  - 1,100+ downloads
- Development
  - 460+ unit tests
  - 25+ compiler bugs reported
  - 35k lines of code
    - Including whitespace & comments
- Uses CUDA Runtime API
  - Essential for template generation

# 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;
```

# 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

- 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

- Thrust provides ~50 algorithms
  - Reduction
  - Prefix Sums
  - Sorting
- Generic definitions
  - General Types
    - Builtin types (int, float, …)
    - User-defined structures
  - General Operators
    - reduce with plus(a,b)
    - scan with maximum(a,b)

# 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>());
```

# Halftime Summary

- Containers
  - Manage host & device memory
    - Automatic allocation and deallocation
    - Simplify data transfers
- Iterators
  - Behave like pointers
  - Associated with memory spaces
- Algorithms
  - Generic
    - Work for any type or operator
  - Statically dispatched based on iterator type
    - Memory space is known at compile-time

# Fancy Iterators

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

# Fancy Iterators

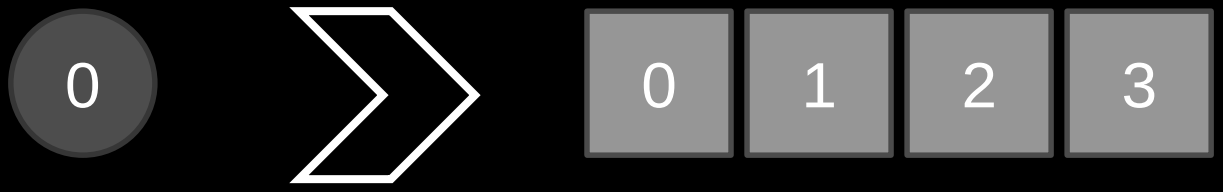
- constant_iterator
  - An infinite array filled with a constant value

```
// create iterators
constant_iterator<int> first(10);
constant_iterator<int> last = first + 3;

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

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

# Fancy Iterators

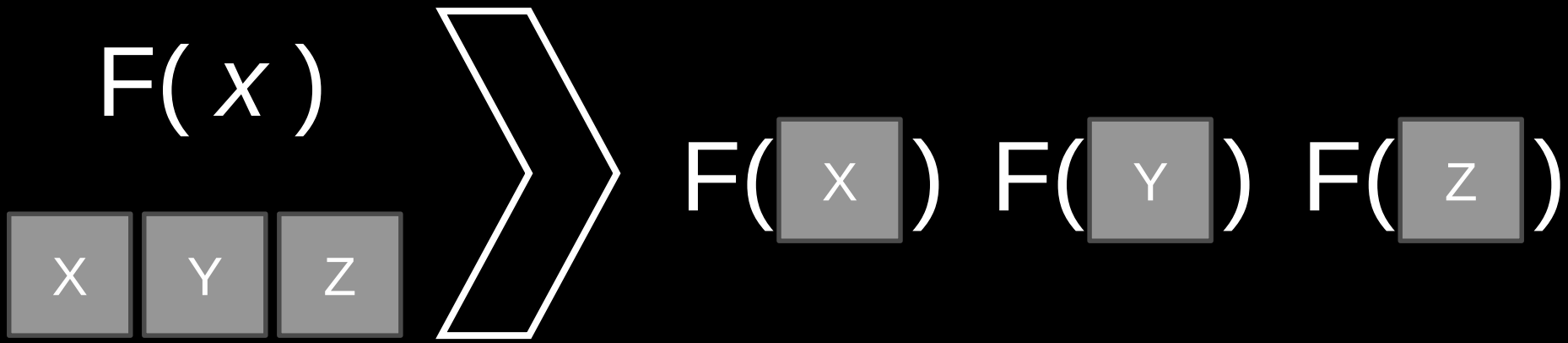
- counting_iterator
  - An infinite array with sequential values

```
// create iterators
counting_iterator<int> first(10);
counting_iterator<int> last = first + 3;

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

// sum of [first, last)
reduce(first, last);   // 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)
first = make_transform_iterator(vec.begin(), negate<int>());
last  = make_transform_iterator(vec.end(),   negate<int>());

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

// sum of [first, last)
reduce(first, last);   // 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

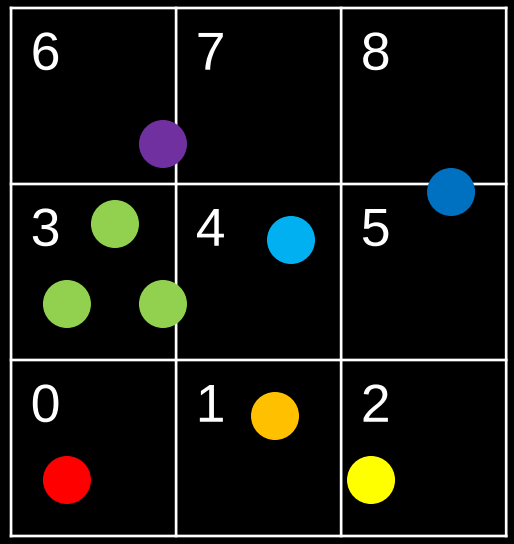
```
// initialize vectors
device_vector<int>  A(3);
device_vector<char> B(3);
A[0] = 10;  A[1] = 20;  A[2] = 30;
B[0] = 'x'; B[1] = 'y'; B[2] = 'z';

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

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

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

# Features & Optimizations

- gather & scatter
  - Works between host and device
- fill & reduce
  - Avoids G8x coalesing rules for char, short, etc.
- sort
  - Dispatches radix_sort  for all primitive types
    - Uses optimal number of radix_sort iterations
  - Dispatches merge_sort  for all other types

# Examples

- SNRM2
  - Computes norm of a vector
  - Level 1 BLAS function
- 2D Bucket Sort
  - Sorting points into cells of a 2D grid
  - Compute bounds for each bucket



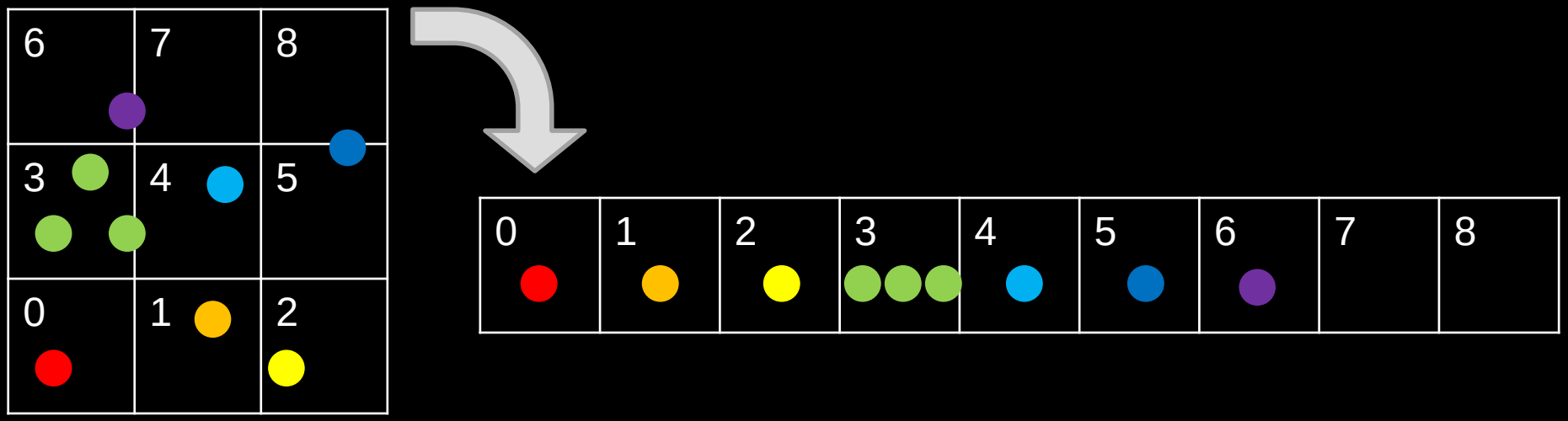
# Example: SNRM2

```
// 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: 2D Bucket Sort

**Procedure:**

- **[Step 1] create random points**
- **[Step 2] compute bucket index for each point**
- **[Step 3] sort points by bucket index**
- **[Step 4] compute bounds for each bucket**



# Example: 2D Bucket Sort

## [Step 1] create random points

```
// number of points
const size_t N = 100000;

// 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;
```

# Example: 2D Bucket Sort

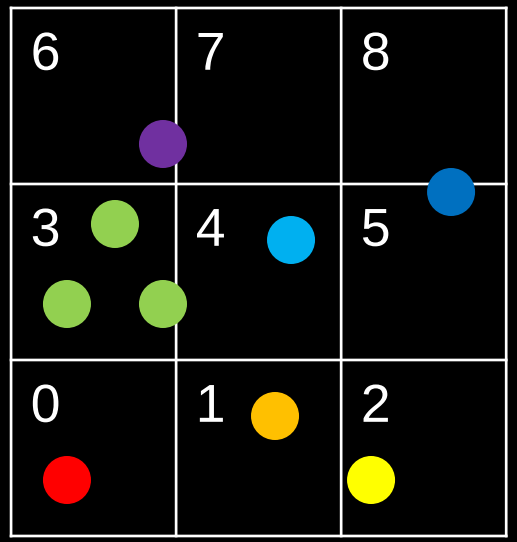
## [Step 2] compute bucket index for each point

```
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) const
  {
    // 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;
  }
};
```

# Example: 2D Bucket Sort

## [Step 2] compute bucket index for each point

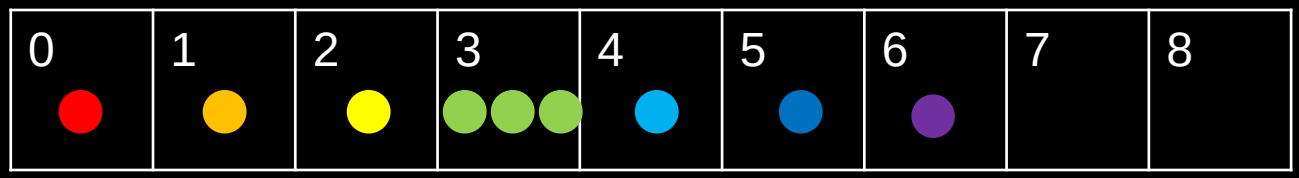
```
// resolution of the 2D grid
unsigned int w = 200;
unsigned int h = 100;

// 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));
```



# Example: 2D Bucket Sort

## [Step 3] sort points by bucket index

```
// sort the points by their bucket index
sort_by_key(bucket_indices.begin(),
            bucket_indices.end(),
            points.begin());
```



# Example: 2D Bucket Sort

## [Step 4] compute bounds for each bucket

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

# Roadmap

- Multicore backend
  - "Thrust Everywhere“
- Algorithms
  - merge, n_th, …
- Iterators
  - reverse_iterator, permutation_iterator, …
- More Features
  - PRNGs (parallel & serial)
  - Error reporting
  - Atomics
  - Range-based interface (a la Alexandrescu)
  - Promises & futures

# Wishlist

- Compiler
  - C++0x features
    - `auto` keyword makes code less verbose
    - Lambda expressions let us inline functor arguments
  - Allow overloading on __host__ & __device__
    - Code genericity
- CUDA Runtime API
  - Enhanced cudaSetDevice()
    - Enables distributed containers (multi-GPU)

# Acknowledgments


- Mark Harris, Michael Garland, et al.
  - Work on sort, scan, etc.
- Bastiaan Aarts et al.
  - Enhancements to nvcc

# Further Reading

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