# GPU Computing with CUDA
# Lecture 6 - CUDA Libraries - Thrust

*Christopher Cooper*
*Boston University*

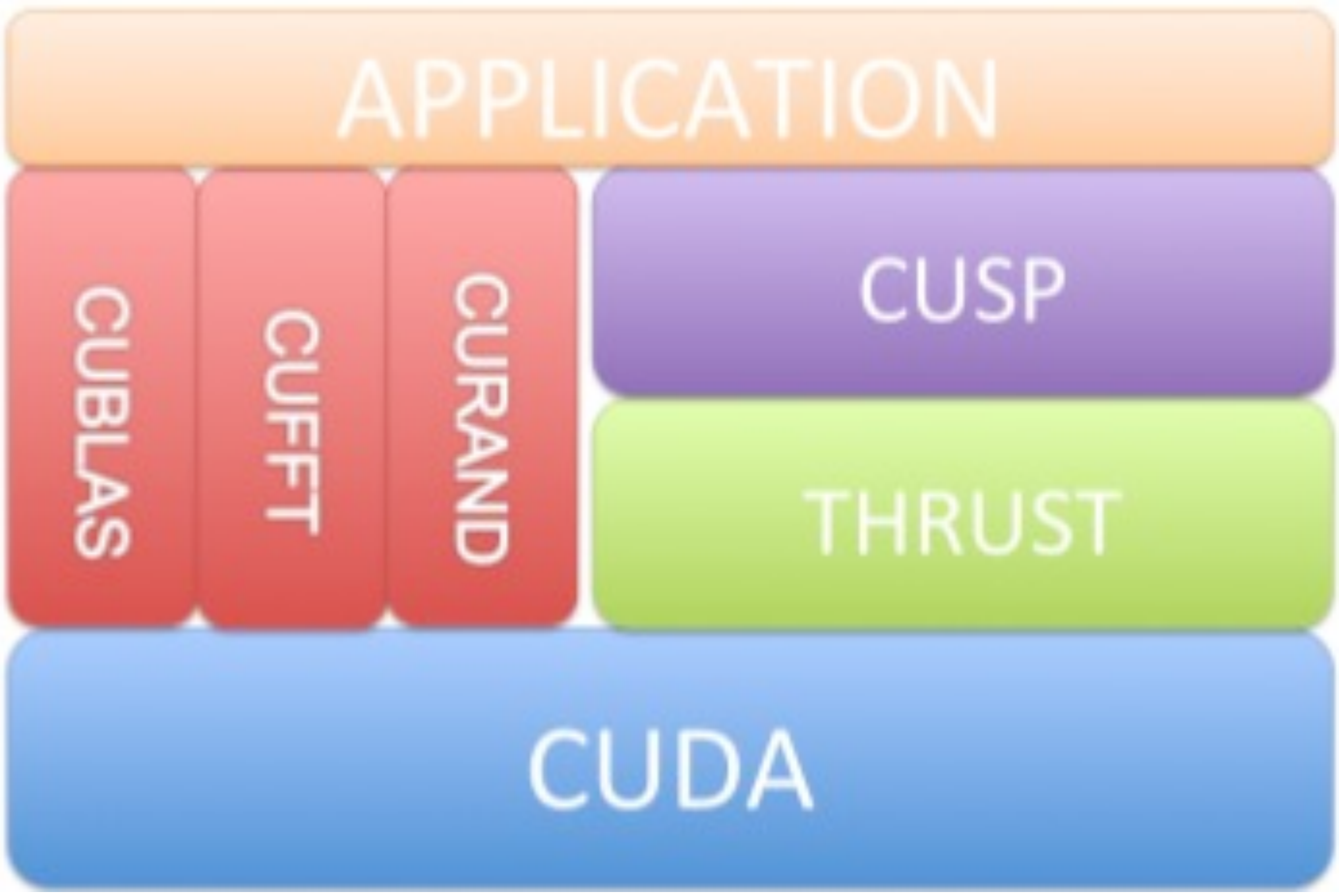
*August, 2011*
*UTFSM, Valparaíso, Chile*

# Outline of lecture

- CUDA Libraries
- What is Thrust?
- Features of thrust
- Best practices in thrust

# CUDA Libraries

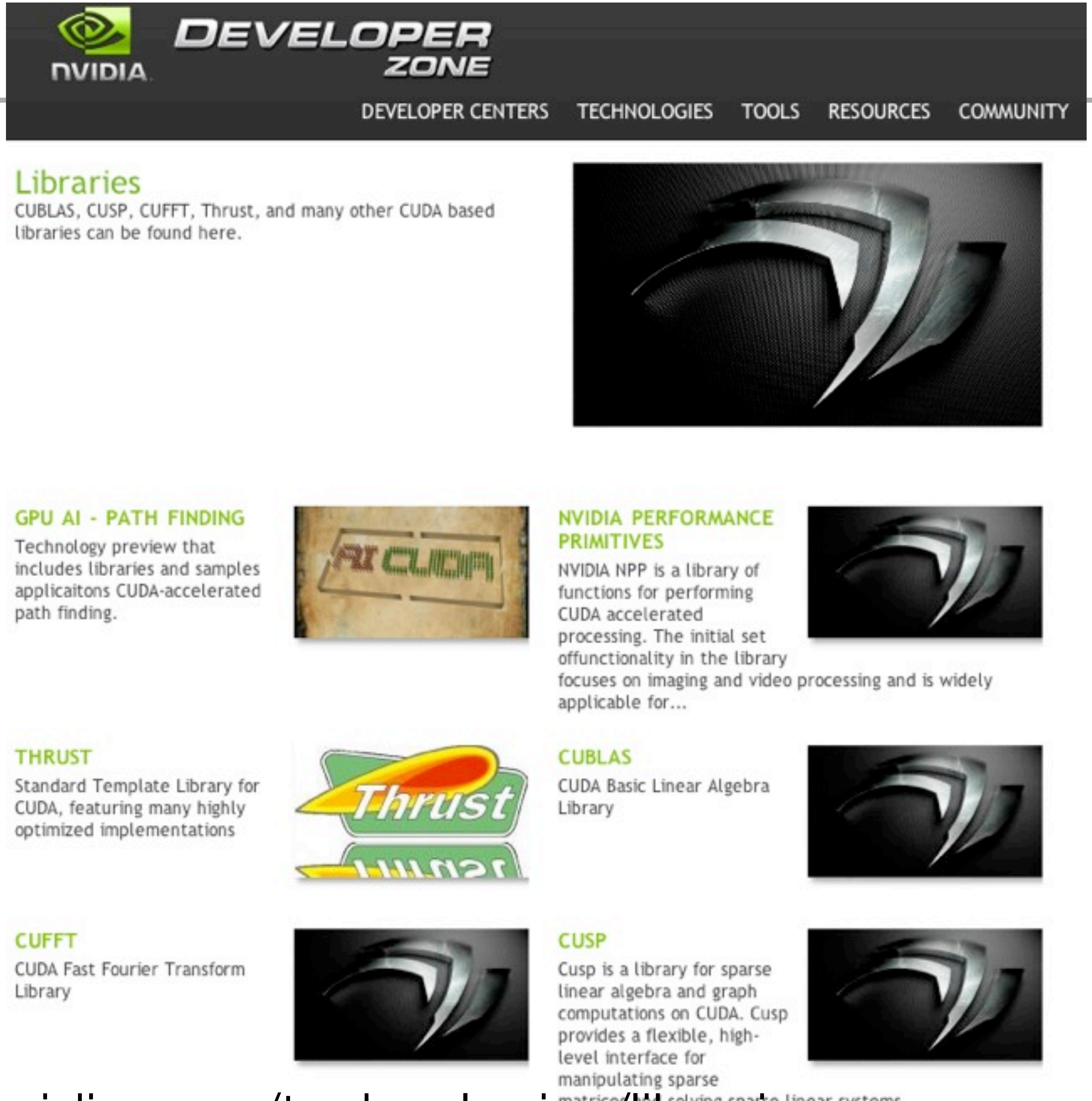
- NVIDIA has developed several libraries to abstract the user from CUDA




Bell, Dalton, Olson. Towards AMG on GPU

# CUDA Libraries

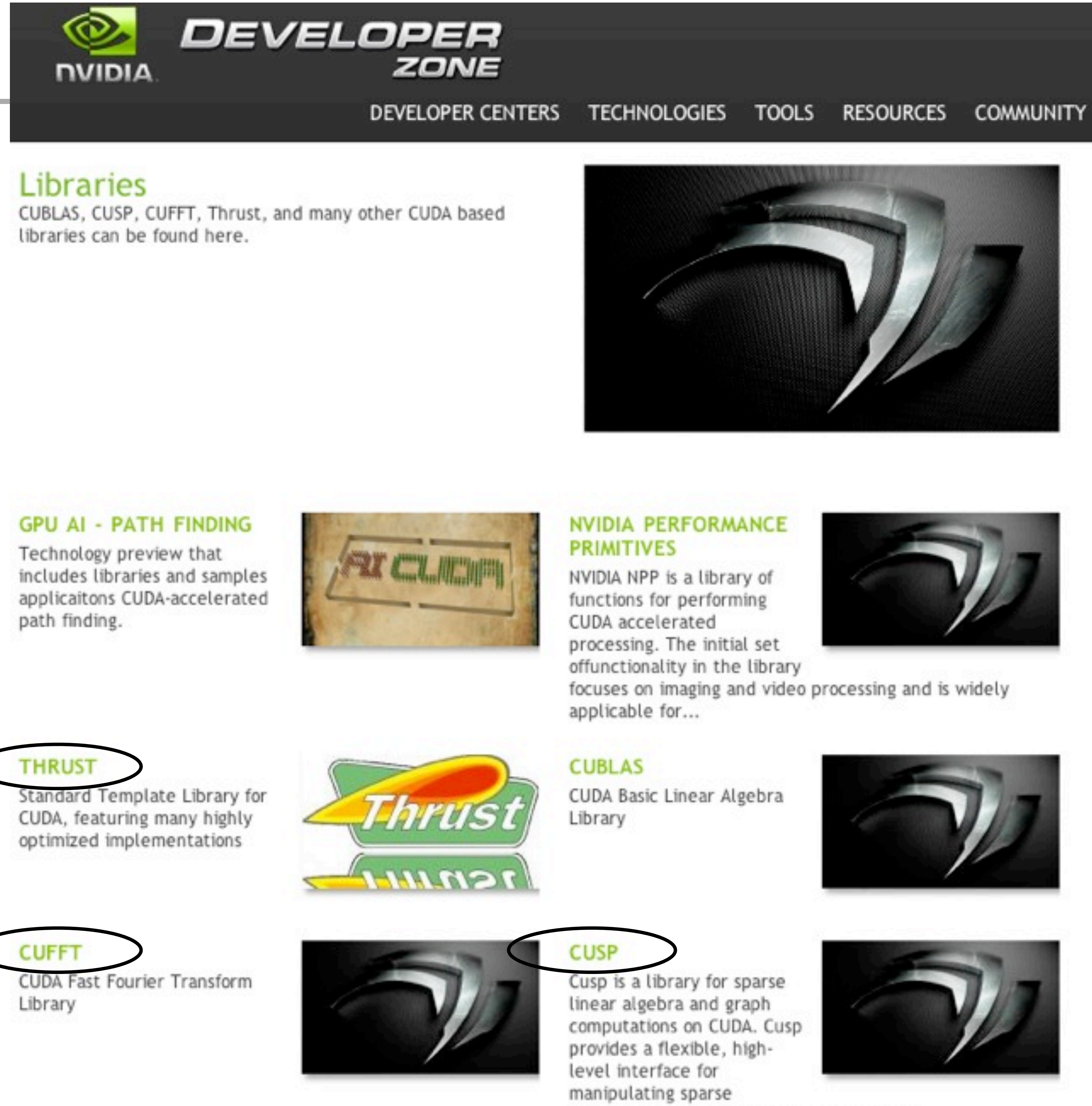
- CUDPP
- MAGMA
- IMSL
- VSIPL
- NVML
- CUPTI
- .....




http://developer.nvidia.com/technologies/libraries

# CUDA Libraries

- CUDPP
- MAGMA
- IMSL
- VSIPL
- NVML
- CUPTI
- .....




http://developer.nvidia.com/technologies/libraries

# Thrust - Introduction

- Template library for CUDA
  - Resembles C++ Standard Template Library (STL)
  - Collection of data parallel primitives
- Objectives
  - Programmer productivity
  - Encourage generic programming
  - High performance
  - Interoperability
- Comes with CUDA 4.0

# Thrust - Introduction

- Containers
  - `thrust::host_vector<T>`
  - `thrust::device_vector<T>`
- Algorithms
  - `thrust::sort()`
  - `thrust::reduce()`
  - `thrust::inclusive_scan()`
- http://code.google.com/p/thrust/
- Slides from Nathan Bell and Jared Hoberock - NVIDIA

# Thrust - Containers

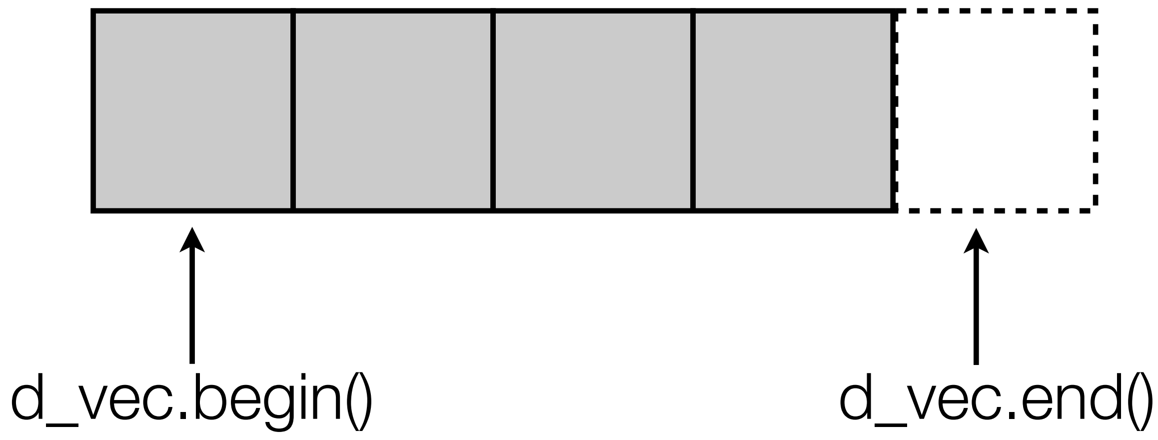
- Thrust provides two vector containers
  - `host_vector`: resides on CPU
  - `device_vector`: resides on GPU
- Hides cudaMalloc and cudaMemcpy

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

# Thrust - Iterators

- Iterators can be thought as pointers to array elements
  - They carry other information

```
// 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
```



# Thrust - 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;
```

# Thrust - Iterators

- Important: keep track of your memory space

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

# Thrust - 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!
```

# Thrust - Iterators

- Wrap raw pointers to use in thrust

```
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);
```

# Thrust - Algorithms

- Standard algorithms
  - Reductions
  - Transformations
  - Prefix sums
  - Sorting
- Many have straight analog to STL
- You can use your own user defined types

# Thrust - Algorithms

- Reductions

```
#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>());
```

# Thrust - Algorithms

- Thrust comes with lots of important built in transformations

```
#include<thrust/device_vector.h>
#include<thrust/transform.h>
#include<thrust/sequence.h>
#include<thrust/copy.h>
#include<thrust/fill.h>
#include<thrust/replace.h>
#include<thrust/functional.h>
#include<iostream>

//allocate three device_vectors with 10 elements
thrust::device_vector<int>X(10);
thrust::device_vector<int> Y(10);
thrust::device_vector<int>Z(10);
//initialize X to 0, 1, 2, 3,....
thrust::sequence(X.begin(),X.end());
//compute Y=-X
thrust::transform(X.begin(),X.end(),Y.begin(),thrust::negate<int>());
//fill Z with two s
thrust::fill(Z.begin(),Z.end(),2);
//compute Y = X mod 2
thrust::transform(X.begin(),X.end(),Z.begin(),Y.begin(),thrust::modulus<int>());
//replace all the ones in Y with tens
thrust::replace(Y.begin(),Y.end(),1,10);
//print Y
thrust::copy(Y.begin(),Y.end(),std::ostream_iterator<int>(std::cout,"\n"));
return 0;
```

# Thrust - Algorithms

- Thrust can do general types and operators using functors

```
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);
```

# Thrust - Algorithms

```
// 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);
```

# Thrust - Algorithms

- Prefix sums

```
#include<thrust/scan.h>
int data [6] = {1,0,2,2,1,3};
thrust::inclusive_scan(data, data + 6, data);//in-place scan
//data is now {1,1,3,5,6,9}
```

```
data[2] = data[0] + data[1] + data[2]
```

```
#include<thrust/scan.h>
int data [6] = {1,0,2,2,1,3};
thrust::exclusive_scan(data, data + 6, data);//in-place scan
//data is now {0,1,1,3,5,6}
```

```
data[2] = data[0] + data[1]
```

# Thrust - Algorithms

- Sorting

```
#include<thrust/sort.h>
. . .
const int N=6;
int A [N] = {1,4,2,8,5,7};
thrust::sort(A,A+N);
// A is now {1,2,4,5,7,8}
```

# Thrust - Fancy iterators

- Behave like “normal’” iterators
  - Also they can be seen as pointers
- Examples
  - `constant_iterator`
  - `counting_iterator`
  - `transform_iterator`
  - `permutation_iterator`
  - `zip_iterator`

# Thrust - Fancy iterators

- `constant_iterator`
  - Mimics an infinite array with constant values

```
// 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)
```

# Thrust - 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)
```

# Thrust - Fancy iterators

- `transform_iterator`
  - Allows us to fuse separate algorithms into one

```
// 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)
```

# Thrust - Fancy iterators

- `permutation_iterator`
  - Allows to fuse gather and scatter operations

```
#include<thrust/iterator/permutation_iterator.h>
...
//gather locations
thrust::device_vector<int> map(4);
map[0] = 3;
map[1] = 1;
map[2] = 0;
map[3] = 5;

//array to gather from
thrust::device_vector<int> source(6);
source[0] = 10;
source[1] = 20;
source[2] = 30;
source[3] = 40;
source[4] = 50;
source[5] = 60;

//fuse gather with reduction: sum = source[map[0]] + source[map[1]]+...
int sum = thrust::reduce(thrust::make_permutation_iterator(source.begin(),map.begin()),
thrust::make_permutation_iterator(source.begin(),map.end()));
```

# Thrust - Fancy iterators

- `zip_iterator`
  - Looks like an array of structs
  - Stored in structure of arrays

```
// 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’)
```

# Thrust - Best practices

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

# Thrust - Fusion

- Combine related operations together
  - Conserves memory bandwidth
- Example: norm of a vector
  - Square each element
  - Compute the sum of squares and take sqrt()

# Thrust - Fusion

- Unoptimized example

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

# Thrust - Fusion

- Optimized implementation (3.8x)

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

# Thrust - Structure of Arrays (SoA)

- Array of structures (AoS)
  - Often does not obey coalescing rules

```
device_vector<float3>
```

- Structure of arrays (SoA)
  - Obeys coealescing rules
  - Components stored in separate arrays

```
device_vector<float> x,y,z;
```

- Example: rotate 3D vectors

# Thrust - 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());
```

# Thrust - 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());
```

# Thrust - Implicit sequences

- Avoid storing sequences explicitly
  - Constant sequences
  - Incrementing sequences
- Implicit sequences require no storage

```
constant_iterator
counting_iterator
```

- Example
  - Index of the smallest element

# Thrust - 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);
}
```

# Thrust - 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);
}
```