# Massively-Parallel Algorithms
## Other Programming Paradigms

G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

# Thrust

- The "standard library" for CUDA
- Resembles the STL (Standard Template Library)
    - Pure header file, fully templatized
- Consists of: containers, algorithms
- In the examples in the following, I will omit all the includes usually necessary:

```
#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>
. . .
```

# Containers

- Mostly vectors (like STL, i.e., arrays with a bit of "intelligence")
  - Comes in two varieties: vectors on the host side, and on the device side

```
thrust::host_vector<int> h_vec(2);          // hides cudaMalloc

thrust::device_vector<int> d_vec = h_vec;   // hides cudaMemcpy
```

- Working on all elements of vectors: algorithms usually require bounds
  - Either use iterators (like in STL):

```
thrust::fill( d_vec.begin(), d_vec.end(), 0);
```

    - `begin()` is like pointer to first element, `end()` like pointer to one element *past last one*
  - Or, use "device pointer" objects:

```
thrust::device_ptr d_vec_ptr = &d_vec[0];
thrust::fill( d_vec_ptr, d_vec_ptr + d_vec.size(), 0);
```

- In the following, I will omit the namespace identifier `thrust::` , where clear by context (hopefully)
- In your source code, you might want to use

```
using namespace thrust;
```

# Transformations

- An operation (encoded as a function) to be applied to every element
- `fill()` is kind of a transformation
- Another example: negate every element

```
transform( X.begin(), X.end(), Y.begin(), thrust::negate<int>() );
```

  - Computes `Y = -X`
    - Assumes: `X`, `Y`, and `Z` are device vectors of `int`
- More examples:
  - Compute `Y[i] = X[i] mod Z[i]`:

```
transform( X.begin(), X.end(), Z.begin(), Y.begin(), modulus<int>() );
```

  - Replace all the 1's in Y with 10's:

```
replace( Y.begin(), Y.end(), 1, 10 );
```

# Defining Your Own Functors

- Example: the saxpy operation

  Y = a * X + Y

  (i.e., Y[i] = a * X[i] + Y[i] )

- Source code:

```
struct saxpy_fct
{
   const float m_a;
   saxpy_fct( float a ) : m_a( a ) {}
   __host__ __device__
   float operator()( const float & x, const float & y ) const
   {
           return m_a * x + y;
   }
};

void saxpy_fast( float a, device_vector<float> & X, device_vector<float> & Y )
{
   transform( X.begin(), X.end(), Y.begin(), Y.begin(), saxpy_fct(a) );
}
```

# Reduction

- You only have to provide the binary operator, many predefined exist
- Example:

```
int sum = thrust::reduce( d_vec.begin(), d_vec.end(), (int) 0, thrust::plus<int>() );
```

- Combine reduction and transformation in order to save bandwidth
- Example: computing the norm of a vector

```
float norm2 = transform_reduce( d_vec.begin(), d_vec.end(), square<float>, 0, plus<float> );
```

  - This is called "kernel fusion" in Thrust
- Many other predefined binary operators for reduction exist:
  `count_if(),min_element(),inner_product(),...`

# Other Building Blocks

- Prefix-sum (scan): `inclusive_scan()`, `exclusive_scan()`
- Stream compaction: `remove_if ()`, `copy_if()`
- Sorting:

# SIMD Parallelization on CPU's