The Cut and Thrust of CUDA

Luke Hodkinson

Center for Astrophysics and Supercomputing
Swinburne University of Technology
Melbourne, Hawthorn 32000, Australia

May 16, 2013

Table of contents

Purpose

- A review of required C++ fundamentals.
  - Just the parts we need.
- An introduction to thrust basics.
- Two case studies.

What is Thrust?

- A parallel algorithms library.
- Resembles the C++ Standard Template Library.
- Enhances **productivity**.

Architectures

Thrust is not limited to GPUs...

- Multicore CPUs.
- GPUs.
- CUDA.
- OpenMP.
- Intel Threading Building Blocks.

Current Statuts

- Open source.
- Apache License 2.0.
- Version 1.6.0.
- Actively developed.
- 240+ active users.
- Now distributed with CUDA toolkit.

Results

The target problem

- Conversion of an MPI parallel selection algorithm.
- Used to find median of large distributed arrays.
- Running with 10,000,000 elements per process.
- Written in C++ using generic algorithms (STL).

Results

Programming effort

Number of characters modified

24*

* *Not including build scripts or including headers.*

Results

Programming time

Time spent modifying algorithm

$$\approx 1 \text{ minute}^*$$

* *Not including build scripts.*

Results

Speedup

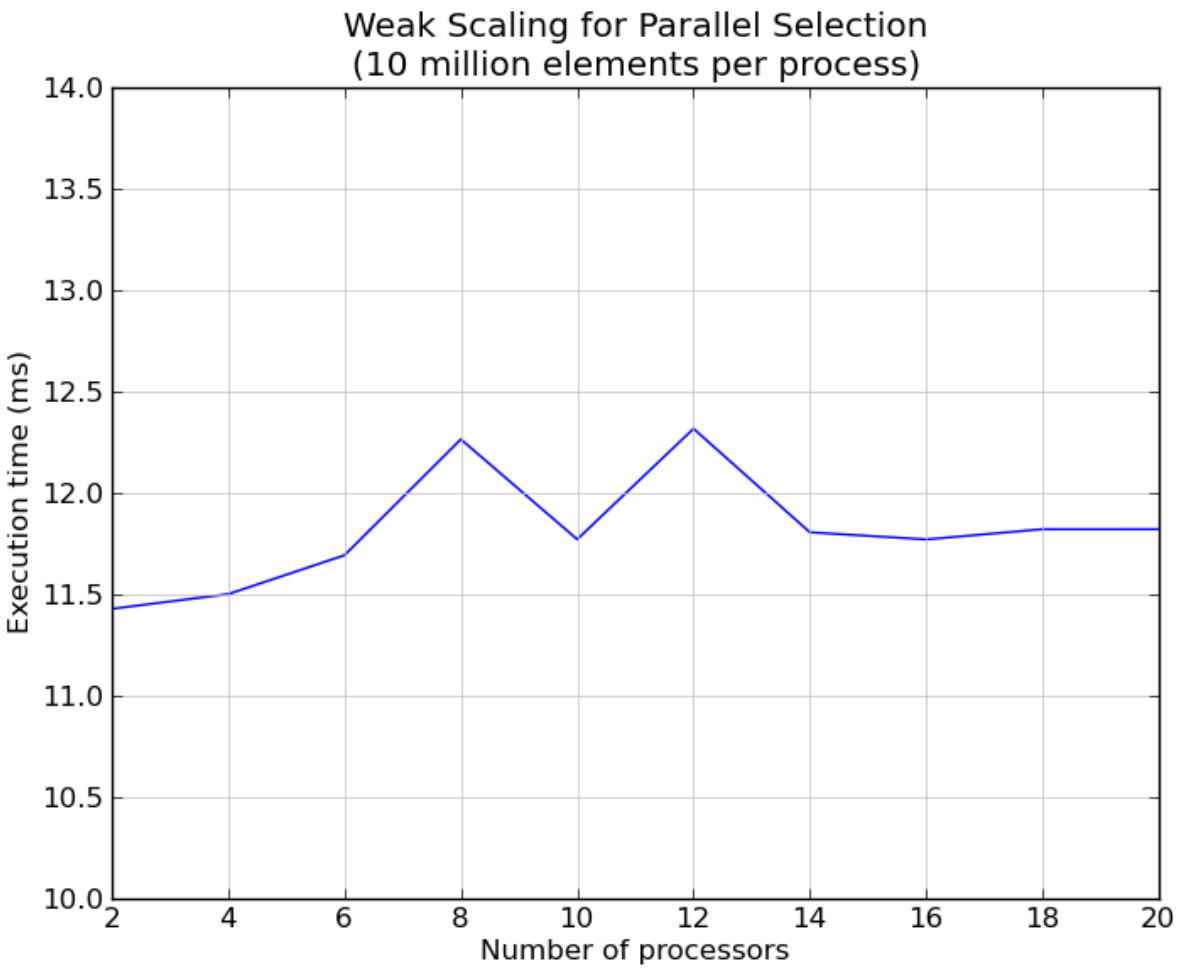
Measured speedup

$122x^*$

* *Measured on gSTAR.*

Results

Scaling



Results

Metrics

Metrics

$x5$ per character changed.
$x2$ per second spent.

Results

Of course...

- results may vary,
- already written in C++,

which brings us to ...

Object Oriented Programming

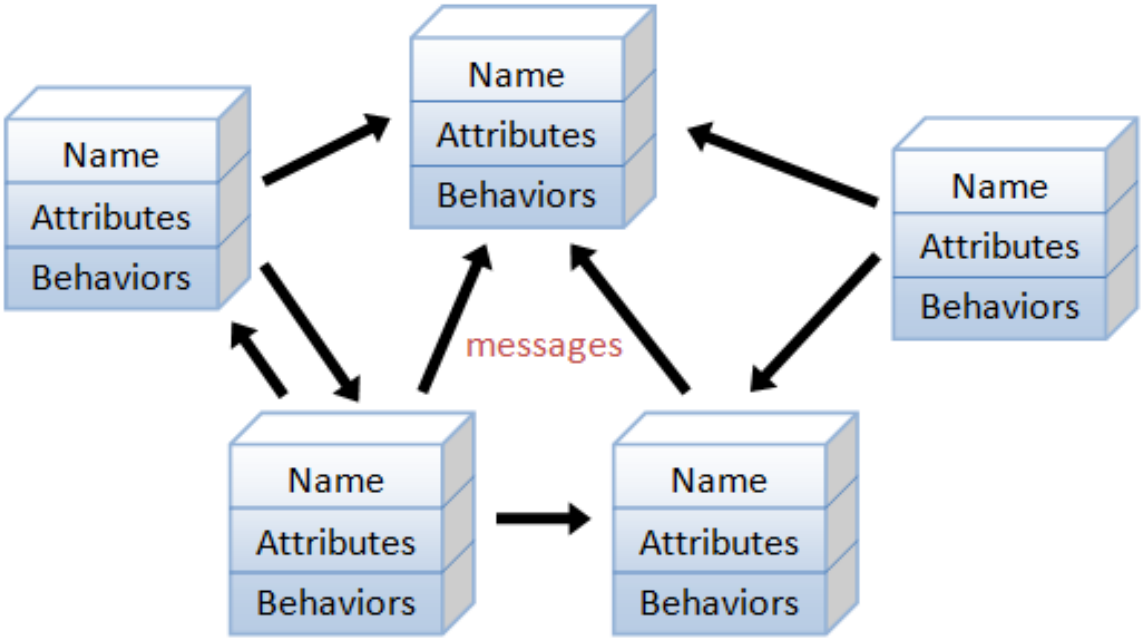
Why do we need it?

- Allows us to bind data with functions.
- Operator overloading allows us to create callable data structures.
- Both of the above allow for easy kernels.

Object Oriented Programming

What is it?

A programming paradigm characterised by representation of concepts as "objects" that have both data and procedures which interact with one another.



Object Oriented Programming

Some definitions:

class A structure to hold data and functions.

member Used to describe a peice of data in a class.

method Used to describe a function/behavior of a class.

object An instantiaion of a class. A class defines an object, but no allocation is given until instantiation.

Object Oriented Programming

Defining a class:

Example

```
struct point_type {
  double x, y;
};
```

```
class point_type {
public:
  double x, y;

  double norm() {
    return sqrt(x*x + y*y);
  }
};
```

Object Oriented Programming

Defining a class:

Example

Declared with class.

```
struct point_type {
  double x, y;
};
```

```
class point_type {
public:
  double x, y;

  double norm() {
    return sqrt(x*x + y*y);
  }
};
```

Object Oriented Programming

Defining a class:

Example

Needs scope modifier.

```
struct point_type {
  double x, y;
};
```

```
class point_type {
public:
  double x, y;

  double norm() {
    return sqrt(x*x + y*y);
  }
};
```

Object Oriented Programming

Defining a class:

Example

Class's data.

```
struct point_type {
  double x, y;
};
```

```
class point_type {
public:
  double x, y;

  double norm() {
    return sqrt(x*x + y*y);
  }
};
```

Object Oriented Programming

Defining a class:

Example

Class's behavior.

```
struct point_type {
  double x, y;
};
```

```
class point_type {
public:
  double x, y;

  double norm() {
    return sqrt(x*x + y*y);
  }
};
```

Object Oriented Programming

Instantiating a class into an object:

Example

```
int main() {
  // Make an object of type "point_type".
  point_type point;

  // Call the "norm" method.
  double value = point.norm();
}
```

Object Oriented Programming

- Can use `struct` instead of `class`.
- Data members are automatically accessible to methods.
- Methods callable from objects using ".".

Object Oriented Programming

Constructors

A special method that facilitates initialisation of data.

Example

```
struct point_type {
  // Default constructor.
  point_type()
    : x( 0.0 ),
      y( 0.0 ) { }

  // Constructor.
  point_type( double x_in, double y_in )
    : x( x_in ),
      y( y_in ) { }
};
```

Object Oriented Programming

Constructors

Is automatically called upon instantiation.

Example

```
int main() {
  // Construct with defaults;
  point_type point_a;

  // Construct with values.
  point_type point_b( 2.0, 4.0 );
};
```

Object Oriented Programming

Functors

C++ allows certain operators to be "overloaded".

`x + y`	`point_type& operator+( const point_type& y )`
`x - y`	`point_type& operator-( const point_type& y)`
`x*y`	`point_type& operator*( const point_type& y )`
`x/y`	`point_type& operator/( const point_type& y)`
`x()`	`void operator()()`

Object Oriented Programming

Functors

So, we can make function-like objects (functors) with attached data...

Example

```
struct square_plus {
  double to_add;

  square_plus( double value )
    : to_add( value ) {}

  double operator()( double x ) {
    return x*x + to_add;
  }
};
```

Object Oriented Programming

Functors

Example

```
int main() {

  square_plus op( 10 ); // create a "square_plus"
                        // object with "to_add"
                        // of 10

  op( 2 ); // call object, returns 40
  op( 3 ); // call object, returns 90

  square_plus( 10 )( 2 ); // create and call,
                          // can be very efficient

}
```

Object Oriented Programming

Functors

Example

```
int main() {

  square_plus op( 10 ); // create a "square_plus"
                        // object with "to_add"
                        // of 10

  op( 2 ); // call object, returns 40
  op( 3 ); // call object, returns 90

  square_plus( 10 )( 2 ); // create and call,
                          // can be very efficient

}
```

Call the constructor to initialise to “10”.

Object Oriented Programming

Functors

Example

```
int main() {

  square_plus op( 10 ); // create a "square_plus"
                        // object with "to_add"
                        // of 10

  op( 2 ); // call object, returns 40
  op( 3 ); // call object, returns 90

  square_plus( 10 )( 2 ); // create and call,
                          // can be very efficient

}
```

Call functor with argument "2".

Object Oriented Programming

Summary

Summary

- Bind data with functions using classes/objects.
- Initialize objects with constructors.
- Create function-like objects with operator overloading.

Object Oriented Programming

Object oriented concepts we won't be looking at:

- inheritance,
- polymorphism,
- encapsulation,
- dynamic dispatch.

There's a lot more to it!

What is Generic Programming?

Some quotes

Generic programming is a paradigm for developing efficient, reusable software libraries.

Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations.

Generic programming is a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters.

What is Generic Programming?

Some quotes

Generic programming is a paradigm for developing efficient, reusable software libraries.

Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations.

Generic programming is a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters.

What is Generic Programming?

Some quotes

Generic programming is a paradigm for developing efficient, reusable software libraries.

Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations.

Generic programming is a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters.

What is Generic Programming?

Example problem

Example

```
// Sum a set of numbers.
int sum( int array[], int size ) {
  int result = 0;
  for( unsigned ii = 0; ii < size; ++ii )
    result += array[ii];
  return result;
}
```

What is Generic Programming?

Example problem

Example

```
// Sum a set of numbers.
int sum( int array[], int size ) {
  int result = 0;
  for( unsigned ii = 0; ii < size; ++ii )
    result += array[ii];
  return result;
}
```

But what about different types?

What is Generic Programming?

Example problem

Potential solutions

- Write different versions for each type.
  - Fast, but unmanageable.
- Use pointers to generalise type.
  - mallocs for ints?! Too slow.
  - Lose type information.
- **Use generic programming.**

Templates

Introduction

Templates are how C++ handles generic programming.

Example

```
// Sum a set of numbers.
template< typename T >
T sum( T array[], int size ) {
  T result = 0;
  for( unsigned ii = 0; ii < size; ++ii )
    result += array[ii];
  return result;
}
```

Templates

Introduction

Example

```
int main() {
  int array_of_ints[10];
  double array_of_doubles[10];
  // TODO: Initialise arrays.

  sum<int>( array_of_ints, 10 );
  sum<double>( array_of_doubles, 10 );

  sum( array_of_ints, 10 );
  sum( array_of_doubles, 10 );
}
```

Pointers

Currently our summation precludes summation over a subset.
But, if we were to use pointers...

Example

```
// Sum a set of numbers.
template< typename T >
T sum( T* start, T* finish ) {
  T result = 0;
  while( start != finish )
    result += *start++;
  return result;
}
```

Pointers

Example

```
int main() {
  int array_of_ints[10];
  double array_of_doubles[10];
  // TODO: Initialise arrays.

  sum<int>( array_of_ints + 2,
            array_of_ints + 8 );
  sum<double>( array_of_doubles,
               array_of_doubles + 4 );
}
```

Iterators

Object Iterators

- What if we have a more complex container? Say, a linked list?
- Using the addition operator, ++, would not work.
- However, remember operator overloading...
- ...we can override these operators using C++ objects!
- What if we define a kind of object to represent positions in containers?

Iterators

Object Iterators

Example

```
// Sum a set of numbers.
template< typename Iterator >
typename Iterator::value_type
sum( Iterator start, Iterator finish ) {
  typename Iterator::value_type result = 0;
  while( start != finish )
    result += *start++;
  return result;
}
```

Iterators

Object Iterators

Example

```
// Sum a set of numbers.
template< typename Iterator >
typename Iterator::value_type
sum( Iterator start, Iterator finish ) {
  typename Iterator::value_type result = 0;
  while( start != finish )
    result += *start++;
  return result;
}
```

Looks complicated...

Iterators

Object Iterators

Example

```
// Sum a set of numbers.
template< typename Iterator >
typename Iterator::value_type
sum( Iterator start, Iterator finish ) {
  typename Iterator::value_type result = 0;
  while( start != finish )
    result += *start++;
  return result;
}
```

Iterators

Object Iterators

Example

```
int main() {
  std::list<int> list;
  std::set<double> set;
  // TODO: Fill list and set with elements.

  sum<int>( list.begin(), list.end() );
  sum<double>( set.begin(), set.end() );
}
```

Iterators

Object Iterators

Example

```
int main() {
  std::list<int> list;
  std::set<double> set;
  // TODO: Fill list and set with elements.

  sum<int>( list.begin(), list.end() );
  sum<double>( set.begin(), set.end() );
}
```

STL uses "std" namespace

Iterators

Object Iterators

Example

```
int main() {
  std::list<int> list;
  std::set<double> set;
  // TODO: Fill list and set with elements.

  sum<int>( list.begin(), list.end() );
  sum<double>( set.begin(), set.end() );
}
```

"begin" points to container start

Iterators

Object Iterators

Example

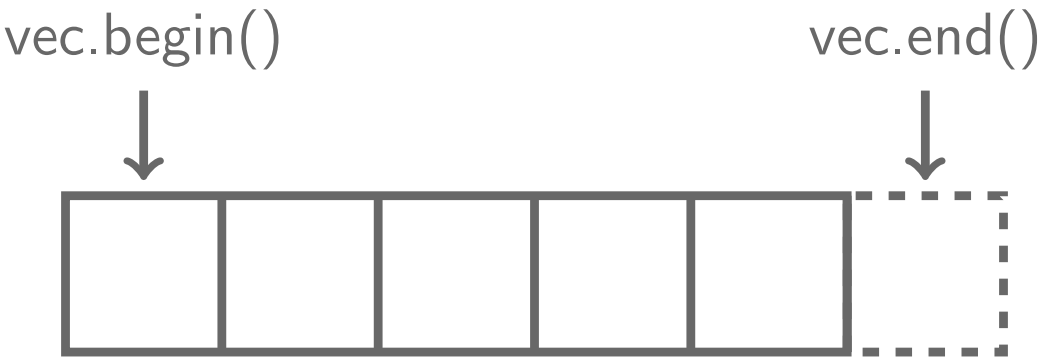
```
int main() {
  std::list<int> list;
  std::set<double> set;
  // TODO: Fill list and set with elements.

  sum<int>( list.begin(), list.end() );
  sum<double>( set.begin(), set.end() );
}
```

"end" points to container end

Iterators

Object Iterators



Algorithms

Introduction

C++ STL provides a suite of generic algorithms written in terms of iterators.

- sort
- copy
- generate
- reverse
- fill
- find
- count
- search

Algorithms

Transform

Example

```
template< typename InputIterator,
          typename OutputIterator,
          typename UnaryOperation >
void transform( InputIterator first,
                InputIterator last,
                OutputIterator result,
                UnaryOperation op )
{
  while( first != last ) {
    *result = op( *first );
    ++first;
    ++result;
  }
}
```

Algorithms

Transform

Example

```
template< typename T >
struct square {
  T operator()( T x ) {
    return x*x;
  }
};

int main() {
  std::vector<int> vec( 100 );
  std::fill<int>( vec.begin(), vec.end(), 10 );
  std::transform<int>( vec.begin(), vec.end(),
                       square<int>() );
}
```

Summary

- Give functions/objects variable types.
- Very efficient, different code for each type.
- Manageable code.
- *Allows separation of data and algorithms.*

Thrust

Introduction

- So how can Thrust help us write CUDA capable code?
- The Thrust library provides replacements for STL containers and algorithms.
  - `vector`, `transform`, `copy`, `count`, `count_if`
- And also some new ones.
  - `reduce`, `transform_reduce`
- **They are all hardware optimised.**

Thrust

Introduction

- So how can Thrust help us write CUDA capable code?
- The Thrust library provides replacements for STL containers and algorithms.
  - `vector`, `transform`, `copy`, `count`, `count_if`
- And also some new ones.
  - `reduce`, `transform_reduce`
- **They are all hardware optimised.**

Thrust

Introduction

- So how can Thrust help us write CUDA capable code?
- The Thrust library provides replacements for STL containers and algorithms.
  - `vector`, `transform`, `copy`, `count`, `count_if`
- And also some new ones.
  - `reduce`, `transform_reduce`
- They are all hardware optimised.

Thrust

Introduction

- So how can Thrust help us write CUDA capable code?
- The Thrust library provides replacements for STL containers and algorithms.
  - `vector`, `transform`, `copy`, `count`, `count_if`
- And also some new ones.
  - `reduce`, `transform_reduce`
- **They are all hardware optimised.**

Thrust

Introduction

- Just like the STL is declared inside the `std::` namespace...
- ... all thrust symbols are declared inside a `thrust::` namespace.
- Symbols can be the same name without conflicts.

Example

```
int main() {
  std::transform( vec.begin(), vec.end(),
                  square<int>() );
  thrust::transform( vec.begin(), vec.end(),
                     square<int>() );
}
```

Thrust Vectors

Device Memory

Device memory is distinct from host memory. We need a special container class to refer to device memory:

Various vectors

```
template< typename T >
class std::vector<T>;

template< typename T >
class thrust::host_vector<T>;

template< typename T >
class thrust::device_vector<T>;
```

Thrust Vectors

Creating a Vector

Device vectors may be constructed with a size...

Example

```
thrust::device_vector<float> dev_vec( 100 );
```

... or resized afterwards, just like a normal vector.

Example

```
thrust::device_vector<float> dev_vec;
dev_vec.resize( 100 );
```

Copying To/From Device

Values can be easily transferred from host to device and vice-versa.

Example

```
std::vector<float> host_vec( 100 );
thrust::device_vector<float> dev_vec( 100 );
thrust::copy( host_vec.begin(), host_vec.end(),
              dev_vec.begin() );
// Do some GPU processing.
thrust::copy( dev_vec.begin(), dev_vec.end(),
              host_vec.begin() );
```

Thrust Algorithms

Introduction

- Thrust's algorithms are its real power.
- Each algorithm is optimised for every supported hardware type.
- Like the STL, algorithms are customised with functors.
- What kinds of algorithms can be optimised for multicore hardware?
  - **Transformation**,
  - **reduction**,
  - **sorting**,
  - and others.

Thrust Algorithms

Introduction

- Thrust's algorithms are its real power.
- Each algorithm is optimised for every supported hardware type.
- Like the STL, algorithms are customised with functors.
- What kinds of algorithms can be optimised for multicore hardware?
  - **Transformation**,
  - **reduction**,
  - **sorting**,
  - and others.

Thrust Algorithms

Introduction

- Thrust's algorithms are its real power.
- Each algorithm is optimised for every supported hardware type.
- Like the STL, algorithms are customised with functors.
- What kinds of algorithms can be optimised for multicore hardware?
  - **Transformation**,
  - **reduction**,
  - **sorting**,
  - and others.

Thrust Algorithms

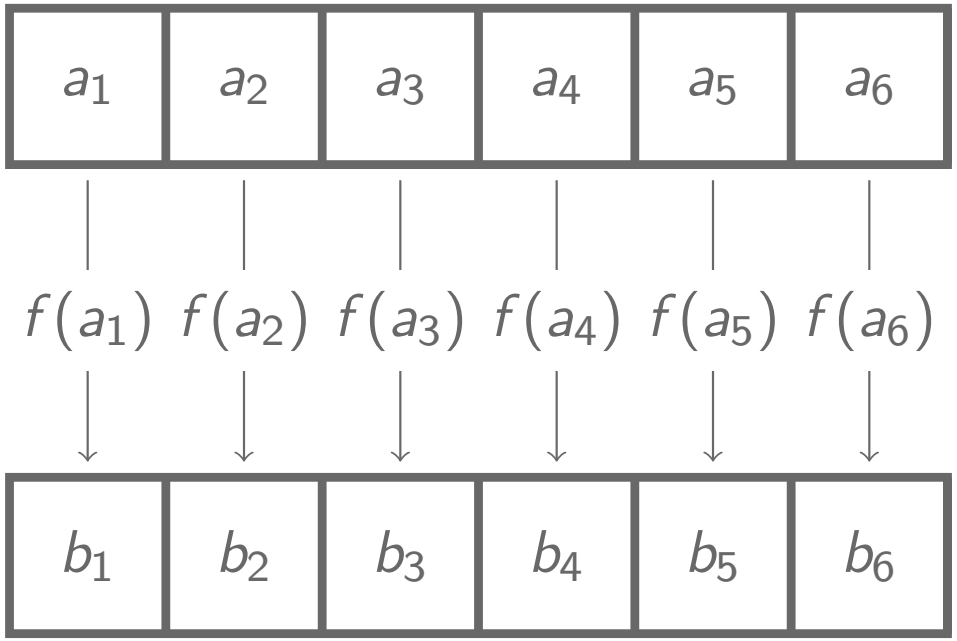
Introduction

- Thrust's algorithms are its real power.
- Each algorithm is optimised for every supported hardware type.
- Like the STL, algorithms are customised with functors.
- What kinds of algorithms can be optimised for multicore hardware?
  - **Transformation**,
  - **reduction**,
  - **sorting**,
  - and others.

Transformation

Definition

Apply a unary operation to each element of a vector and store the result in another (or the same) vector.



Transformation

Example

Example

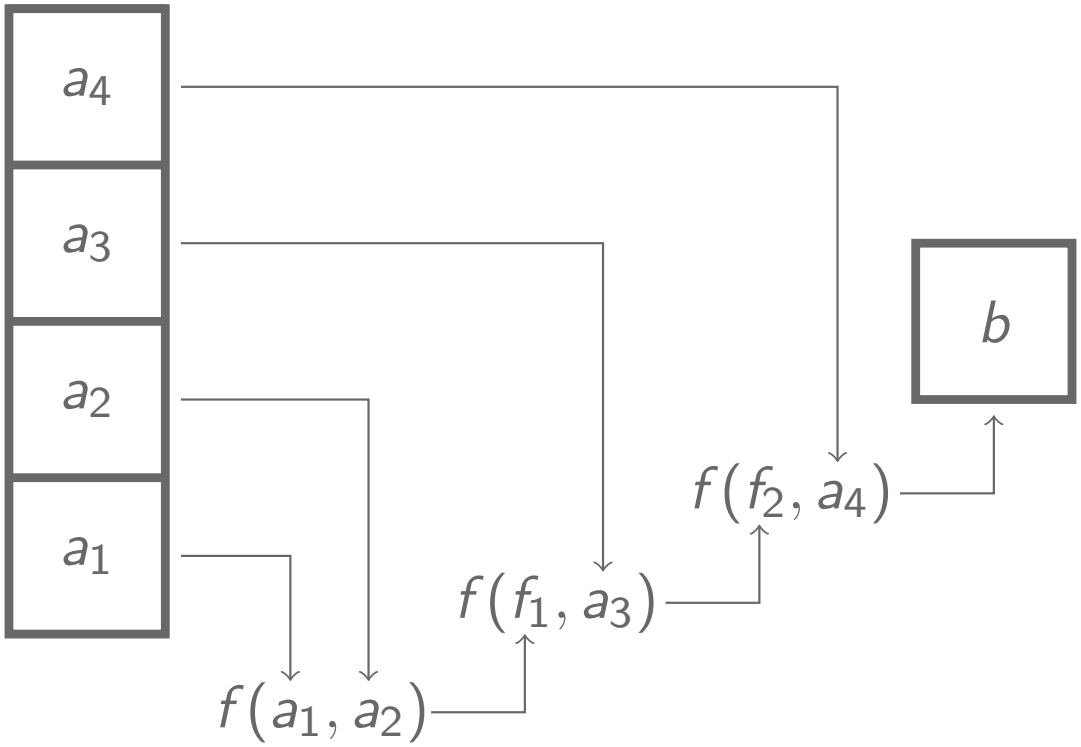
```
struct norm {
  __device__
  float operator()( float x ) {
    return sqrt( x*x );
  }
};

int main() {
  thrust::device_vector<float> src( 100 ),
                               dst( 100 );
  // TODO: Initialise "src".
  thrust::transform( src.begin(), src.end(),
                     dst.begin(),
                     norm() );
}
```

Reduction

Definition

Apply a binary operation to successive elements of a vector and return the scalar result.



Reduction

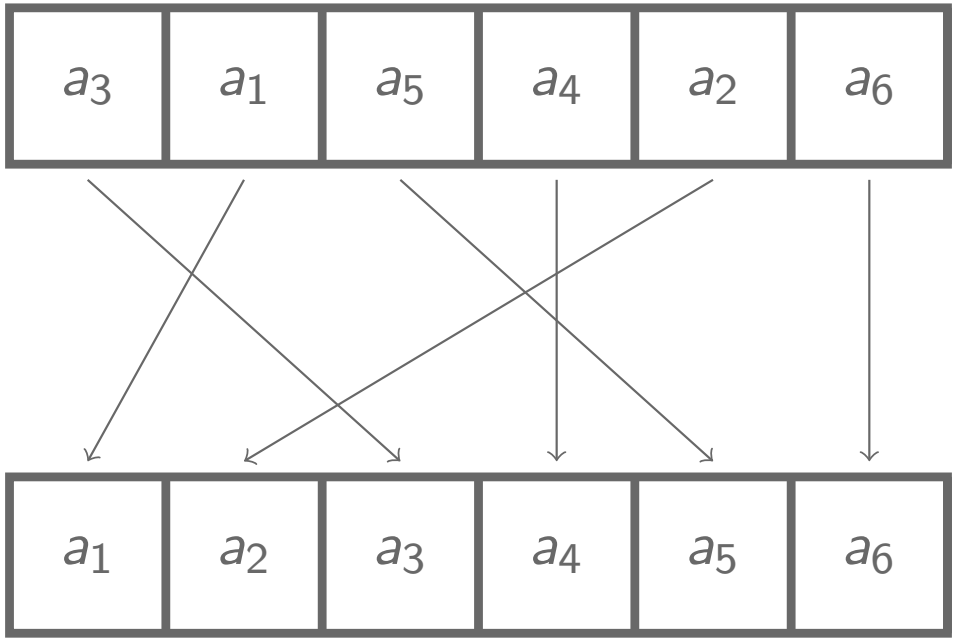
Example

Example

```
int main() {
  thrust::device_vector<float> src( 100 );
  // TODO: Initialise "src".
  thrust::reduce( src.begin(), src.end(),
                  thrust::plus<float>() );
}
```

Sorting

Definition



Sorting

Example

Example

```
int main() {
  thrust::device_vector<float> src( 100 );
  // TODO: Initialise "src".
  thrust::sort( src.begin(), src.end(),
                thrust::greater<float>() );
}
```

Application

Just a few fundamental algorithms may not seem like much...

Application
Examples

... but it can do a lot!

Example

- Discrete Voronoi
- Bucket sort
- Bounding box
- Histograms
- Lexicographical sort
- Monte carlo disjoint sequences
- Padded grid reduction
- Run-length encoding
- SAXPY
- Set operations

Application

Strategy

1. Does your algorithm subscribe to a fundamental algorithm (transform, reduce, sort)? Done.
2. Can your algorithm be decomposed into simpler components? Goto 1 for each component.
3. Can't decompose any further and doesn't match a fundamental algorithm? Write your own kernel.

Application

Strategy

1. Does your algorithm subscribe to a fundamental algorithm (transform, reduce, sort)? Done.
2. Can your algorithm be decomposed into simpler components? Goto 1 for each component.
3. Can't decompose any further and doesn't match a fundamental algorithm? Write your own kernel.

Application

Strategy

1. Does your algorithm subscribe to a fundamental algorithm (transform, reduce, sort)? Done.
2. Can your algorithm be decomposed into simpler components? Goto 1 for each component.
3. Can't decompose any further and doesn't match a fundamental algorithm? Write your own kernel.

Advanced Iterators

Overview

There are other powerful features of Thrust, designed to improve flexibility and performance.

- `thrust::constant_iterator`,
- `thrust::counting_iterator`,
- `thrust::transform_iterator` and
- `thrust::zip_iterator`.

Advanced Iterators

zip_iterator

- Algorithm's tend to accept a single input sequence.
- You may want to operate on multiple sequences,
- or your functor may *require* multiple sequences.
- `thrust::zip_iterator` provides a powerful solution.

Advanced Iterators

zip_iterator

Example

```
// Initialize vectors.
thrust::device_vector<int>  A(3);
thrust::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 = thrust::make_zip_iterator(thrust::make_tuple(A.begin(), B.begin()));
last  = thrust::make_zip_iterator(thrust::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).
thrust::maximum< tuple<int,char> > binary_op;
thrust::tuple<int,char> init = first[0];
thrust::reduce(first, last, init, binary_op); // returns tuple(30, 'z')
```

Advanced Iterators

zip_iterator

Example

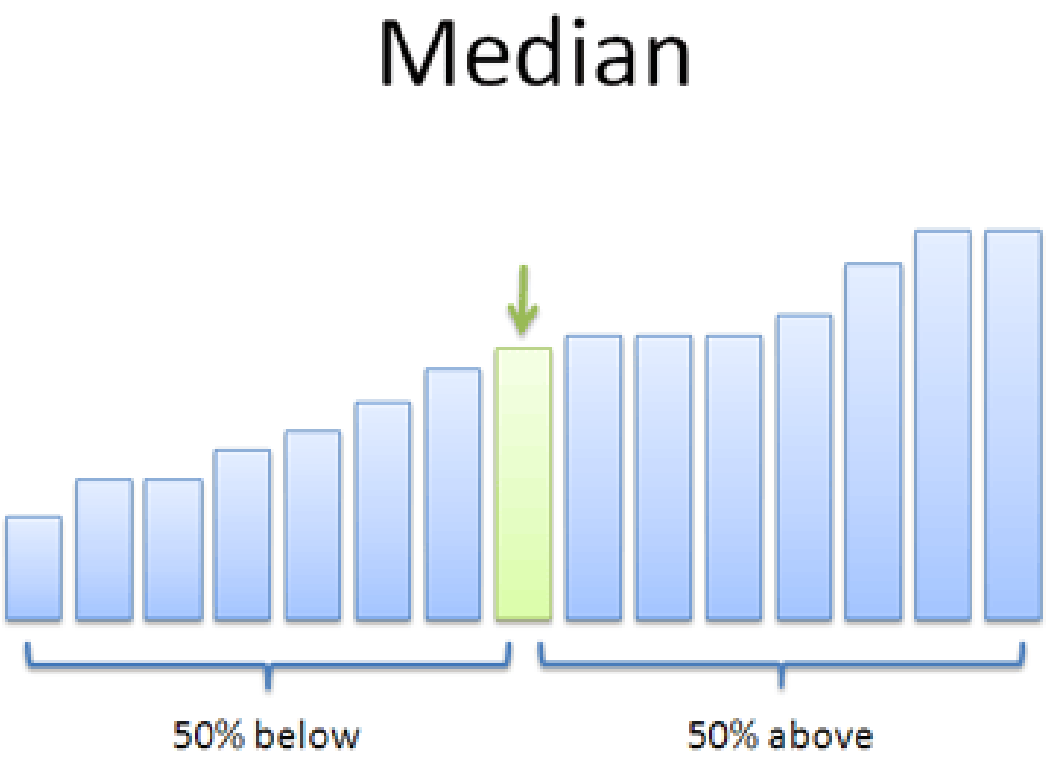
```
typedef thrust::tuple<float,float,float> float3;
using thrust::get;

struct dot_prod {
  __device__
  float operator()( float3 a, float3 b ) {
    return get<0>(a)*<0>get(b) + get<1>(a)*<1>get(b) + get<2>(a)*<2>get(b);
  }
};

int main() {
  thrust::device_vector<float> A0, A1, A2;
  thrust::device_vector<float> B0, B1, B2;
  thrust::device_vector<float> result;
  // Initialise vectors.
  thrust::transform(
    thrust::make_zip_iterator( A0.begin(), A1.begin(), A2.begin() ),
    thrust::make_zip_iterator( A0.end(),   A1.end(),   A2.end() ),
    thrust::make_zip_iterator( B0.begin(), B1.begin(), B2.begin() ),
    result.begin(),
    dot_prod()
  );
}
```

Distributed Median

Description



Distributed Median

Serial

In serial median can be calculated fairly easily by sorting the array and taking the $\frac{n}{2}$ element.

Example

```
thrust::device_vector<float> values( 100 );
thrust::sort( values.begin(), values.end() );
float median = values[values.size()/2];
```

Distributed Median

Parallel

Define a balance function,

$$b(x) = N_l(x) - \frac{n}{2}$$

which represents how close we are to the median by

$$b(x) = 0 \ .$$

Here $N_l(x)$ is a count of how many elements are below the point x and n is the number of elements in the array.

Distributed Median

Now we just need to find where $b(x) = 0$. We can do this with any root finder, but the Ridders algorithm will work particularaly well for us.

Distributed Median

And already had one written.

Example

```
template< class Function,
          class T >
T
ridders( Function func,
         T x1,
         T x2 );
```

Distributed Median

Example

```
template< typename Iterator >
struct median_function
{
  Iterator start, finish;
  long position;
  mpi::comm comm;

  long
  operator()( const value_type& x )
  {
    long sum_left = std::count_if(
      start, finish,
      std::bind2nd(
        std::less<value_type>(), x
      )
    );
    return comm.all_reduce( sum_left ) - position;
  }
};
```

Distributed Median

Example

```
template< typename Iterator >
struct median_function
{
  long position;
  mpi::comm comm;

  __device__
  long
  operator()( const value_type& x )
  {
    long sum_left = thrust::count_if(
      start, finish,
      thrust::bind2nd(
        thrust::less<value_type>(), x
      )
    );
    return comm.all_reduce( sum_left ) - position;
  }
};
```

Distributed Median

Example

```
template< typename Iterator >
struct median_function
{
  long position;
  mpi::comm comm;

  __device__
  long
  operator()( const value_type& x )
  {
    long sum_left = thrust::count_if(
      start, finish,
      thrust::bind2nd(
        thrust::less<value_type>(), x
      )
    );
    return comm.all_reduce( sum_left ) - position;
  }
};
```

Add device modifier.

Distributed Median

Example

```
template< typename Iterator >
struct median_function
{
  long position;
  mpi::comm comm;

  __device__
  long
  operator()( const value_type& x )
  {
    long sum_left = thrust::count_if(
      start, finish,
      thrust::bind2nd(
        thrust::less<value_type>(), x
      )
    );
    return comm.all_reduce( sum_left ) - position;
  }
};
```

Change to thrust namespace.

Distributed Median

Example

```
int main() {
  thrust::device_vector<float> vec( 1000 );
  thrust::generate(
    vec.begin(), vec.end(),
    thrust::counting_iterator<float>( 0 )
  );
  float median = ridders(
    median_function<float>( start, finish ),
    0, 1000
  );
}
```