CUDA Toolkit 4.2
Thrust Quick Start Guide

PG-05688-040_v01 | March 2012

Document Change History

Version	Date	Authors	Description of Change
01	2011/1/13	NOJW	Initial import from wiki.

Introduction

Thrust is a C++ template library for CUDA based on the Standard Template Library (STL). Thrust allows you to implement high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with CUDA C.

Thrust provides a rich collection of data parallel primitives such as scan, sort, and reduce, which can be composed together to implement complex algorithms with concise, readable source code. By describing your computation in terms of these high-level abstractions you provide Thrust with the freedom to select the most efficient implementation automatically. As a result, Thrust can be utilized in rapid prototyping of CUDA applications, where programmer productivity matters most, as well as in production, where robustness and absolute performance are crucial.

This document describes how to develop CUDA applications with Thrust. The tutorial is intended to be accessible, even if you have limited C++ or CUDA experience.

Installation and Versioning

Installing the CUDA Toolkit will copy Thrust header files to the standard CUDA include directory for your system. Since Thrust is a template library of header files, no further installation is necessary to start using Thrust.

In addition, new versions of Thrust continue to be available online through the Google Code Thrust project page. The version of Thrust included in this version of the CUDA Toolkit corresponds to version 1.4.0 from the Thrust project page.

Vectors

Thrust provides two vector containers, `host_vector` and `device_vector`. As the names suggest, `host_vector` is stored in host memory while `device_vector` lives in GPU device memory. Thrust's vector containers are just like `std::vector` in the C++ STL. Like `std::vector`, `host_vector` and `device_vector` are generic containers (able to store any data type) that can be resized dynamically. The following source code illustrates the use of Thrust's vector containers.

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

#include <iostream>

int main(void)
{
    // H has storage for 4 integers
    thrust::host_vector<int> H(4);

    // initialize individual elements
    H[0] = 14;
    H[1] = 20;
    H[2] = 38;
    H[3] = 46;

    // H.size() returns the size of vector H
    std::cout << "H has size " << H.size() << std::endl;

    // print contents of H
    for(int i = 0; i < H.size(); i++)
        std::cout << "H[" << i << "] = " << H[i] << std::endl;

    // resize H
    H.resize(2);

    std::cout << "H now has size " << H.size() << std::endl;

    // Copy host_vector H to device_vector D
    thrust::device_vector<int> D = H;

    // elements of D can be modified
    D[0] = 99;
    D[1] = 88;

    // print contents of D
    for(int i = 0; i < D.size(); i++)
```

```
        std::cout << "D[" << i << "] = " << D[i] << std::endl;

    // H and D are automatically deleted when the function returns
    return 0;
}
```

As this example shows, the = operator can be used to copy a `host_vector` to a `device_vector` (or vice-versa). The = operator can also be used to copy `host_vector` to `host_vector` or `device_vector` to `device_vector`. Also note that individual elements of a `device_vector` can be accessed using the standard bracket notation. However, because each of these accesses requires a call to `cudaMemcpy`, they should be used sparingly. We'll look at some more efficient techniques later.

It's often useful to initialize all the elements of a vector to a specific value, or to copy only a certain set of values from one vector to another. Thrust provides a few ways to do these kinds of operations.

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/sequence.h>

#include <iostream>

int main(void)
{
    // initialize all ten integers of a device_vector to 1
    thrust::device_vector<int> D(10, 1);

    // set the first seven elements of a vector to 9
    thrust::fill(D.begin(), D.begin() + 7, 9);

    // initialize a host_vector with the first five elements of D
    thrust::host_vector<int> H(D.begin(), D.begin() + 5);

    // set the elements of H to 0, 1, 2, 3, ...
    thrust::sequence(H.begin(), H.end());

    // copy all of H back to the beginning of D
    thrust::copy(H.begin(), H.end(), D.begin());

    // print D
    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "] = " << D[i] << std::endl;

    return 0;
}
```

Here we've illustrated use of the `fill`, `copy`, and `sequence` functions. The `copy` function can be used to copy a range of host or device elements to another host or device vector. Like the corresponding STL function, `thrust::fill` simply sets a range of elements to a specific value. Thrust's `sequence` function can be used to a create a sequence of equally spaced values.

Thrust Namespace

You'll notice that we use things like `thrust::host_vector` or `thrust::copy` in our examples. The `thrust::` part tells the C++ compiler that we want to look inside the `thrust` *namespace* for a specific function or class. Namespaces are a nice way to avoid name collisions. For instance, `thrust::copy` is different from `std::copy` provided in the STL. C++ namespaces allow us to distinguish between these two `copy` functions.

Iterators and Static Dispatching

In this section we used expressions like `H.begin()` and `H.end()` or offsets like `D.begin() + 7`. The result of `begin()` and `end()` is called an *iterator* in C++. In the case of vector containers, which are really just arrays, iterators can be thought of as pointers to array elements. Therefore, `H.begin()` is an iterator that points to the first element of the array stored inside the `H` vector. Similarly, `H.end()` points to the element *one past* the last element of the `H` vector.

Although vector iterators are similar to pointers they carry more information with them. Notice that we did not have to tell `thrust::fill` that it was operating on a `device_vector` iterator. This information is captured in the type of the iterator returned by `D.begin()` which is different than the type returned by `H.begin()`. When a Thrust function is called, it inspects the type of the iterator to determine whether to use a host or a device implementation. This process is known as *static dispatching* since the host/device dispatch is resolved at compile time. Note that this implies that there is *no runtime overhead* to the dispatch process.

You may wonder what happens when a "raw" pointer is used as an argument to a Thrust function. Like the STL, Thrust permits this usage and it will dispatch the host path of the algorithm. If the pointer in question is in fact a pointer to device memory then you'll need to wrap it with `thrust::device_ptr` before calling the function. For example:

```
size_t 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);
```

To extract a raw pointer from a `device_ptr` the `raw_pointer_cast` should be applied as follows:

```
size_t N = 10;

// create a device_ptr
thrust::device_ptr<int> dev_ptr = thrust::device_malloc<int>(N);

// extract raw pointer from device_ptr
int * raw_ptr = thrust::raw_pointer_cast(dev_ptr);
```

Another reason to distinguish between iterators and pointers is that iterators can be used to traverse many kinds of data structures. For example, the STL provides a linked list container (`std::list`) that provides bidirectional (but not random access) iterators. Although Thrust does not provide device implementations of such containers, it is compatible with them.

```
#include <thrust/device_vector.h>
#include <thrust/copy.h>
#include <list>
#include <vector>

int main(void)
{
    // create an STL list with 4 values
    std::list<int> stl_list;

    stl_list.push_back(10);
    stl_list.push_back(20);
    stl_list.push_back(30);
    stl_list.push_back(40);

    // initialize a device_vector with the list
    thrust::device_vector<int> D(stl_list.begin(), stl_list.end());

    // copy a device_vector into an STL vector
    std::vector<int> stl_vector(D.size());
    thrust::copy(D.begin(), D.end(), stl_vector.begin());

    return 0;
}
```

For Future Reference: The iterators we've covered so far are useful, but fairly basic. In addition to these normal iterators, Thrust also provides a collection of *fancy iterators* with names like `counting_iterator` and `zip_iterator`. While they look and feel like normal iterators, fancy iterators are capable of more exciting things. We'll revisit this topic later in the tutorial.

Algorithms

Thrust provides a large number of common parallel algorithms. Many of these algorithms have direct analogs in the STL, and when an equivalent STL function exists, we choose the name (e.g. `thrust::sort` and `std::sort`).

All algorithms in Thrust have implementations for both host and device. Specifically, when a Thrust algorithm is invoked with a host iterator, then the host path is dispatched. Similarly, a device implementation is called when a device iterator is used to define a range.

With the exception of `thrust::copy`, which can copy data between host and device, all iterator arguments to a Thrust algorithm should live in the same place: either all on the host or all on the device. When this requirement is violated the compiler will produce an error message.

Transformations

Transformations are algorithms that apply an operation to each element in a set of (zero or more) input ranges and then stores the result in a destination range. One example we have already seen is `thrust::fill`, which sets all elements of a range to a specified value. Other transformations include `thrust::sequence`, `thrust::replace`, and of course `thrust::transform`. Refer to the documentation for a complete listing.

The following source code demonstrates several of the transformation algorithms. Note that `thrust::negate` and `thrust::modulus` are known as *functors* in C++ terminology. Thrust provides these and other common functors like `plus` and `multiplies` in the file `thrust/functional.h`.

```
#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>
```

```
int main(void)
{
    // 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 twos
    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;
}
```

While the functors in `thrust/functional.h` cover most of the built-in arithmetic and comparison operations, we often want to do something different. For example, consider the vector operation `y <- a * x + y` where `x` and `y` are vectors and `a` is a scalar constant. This is the well-known SAXPY operation provided by any BLAS library.

If we want to implement SAXPY with Thrust we have a few options. The first is to use two transformations (one addition and one multiplication) and a temporary vector filled with the value `a`. A better choice is to use a single transformation with a user-defined functor that does exactly what we want. We illustrate both approaches in the source code below.

```
struct saxpy_functor
{
    const float a;

    saxpy_functor(float _a) : a(_a) {}

    __host__ __device__
        float operator()(const float& x, const float& y) const {
            return a * x + y;
        }
};

void saxpy_fast(float A, thrust::device_vector<float>& X, thrust::device_vector<↩
    float>& Y)
{
    // Y <- A * X + Y
    thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(), saxpy_functor(A));
}
```

```
void saxpy_slow(float A, thrust::device_vector<float>& X, thrust::device_vector<↩
    float>& Y)
{
    thrust::device_vector<float> temp(X.size());

    // temp <- A
    thrust::fill(temp.begin(), temp.end(), A);

    // temp <- A * X
    thrust::transform(X.begin(), X.end(), temp.begin(), temp.begin(), thrust::↩
        multiplies<float>());

    // Y <- A * X + Y
    thrust::transform(temp.begin(), temp.end(), Y.begin(), Y.begin(), thrust::plus<↩
        float>());
}
```

Both `saxpy_fast` and `saxpy_slow` are valid SAXPY implementations, however `saxpy_fast` will be significantly faster than `saxpy_slow`. Ignoring the cost of allocating the `temp` vector and the arithmetic operations we have the following costs:

- `fast_saxpy`: performs 2N reads and N writes
- `slow_saxpy`: performs 4N reads and 3N writes

Since SAXPY is *memory bound* (its performance is limited by memory bandwidth, not floating point performance) the larger number of reads and writes makes `saxpy_slow` much more expensive. In contrast, `saxpy_fast` will perform about as fast as SAXPY in an optimized BLAS implementation. In memory bound algorithms like SAXPY it is generally worthwhile to apply *kernel fusion* (combining multiple operations into a single kernel) to minimize the number of memory transactions.

`thrust::transform` only supports transformations with one or two input arguments (e.g. $f(x) \to y$ and $f(x, y) \to z$). When a transformation uses more than two input arguments it is necessary to use a different approach. The arbitrary_transformation example demonstrates a solution that uses `thrust::zip_iterator` and `thrust::for_each`.

Reductions

A reduction algorithm uses a binary operation to *reduce* an input sequence to a single value. For example, the sum of an array of numbers is obtained by reducing the array with a plus operation. Similarly, the maximum of an array is obtained by reducing with a operator that takes two inputs and returns the maximum. The sum of an array is implemented with `thrust::reduce` as follows:

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

The first two arguments to `reduce` define the range of values while the third and fourth parameters provide the initial value and reduction operator respectively. Actually, this

kind of reduction is so common that it is the default choice when no initial value or operator is provided. The following three lines are therefore equivalent:

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

Although `thrust::reduce` is sufficient to implement a wide variety of reductions, Thrust provides a few additional functions for convenience (like the STL). For example, `thrust::count` returns the number of instances of a specific value in a given sequence:

```
#include <thrust/count.h>
#include <thrust/device_vector.h>
...
// put three 1s in a device_vector
thrust::device_vector<int> vec(5,0);
vec[1] = 1;
vec[3] = 1;
vec[4] = 1;

// count the 1s
int result = thrust::count(vec.begin(), vec.end(), 1);
// result is three
```

Other reduction operations include `thrust::count_if`, `thrust::min_element`, `thrust::max_element`, `thrust::is_sorted`, `thrust::inner_product`, and several others. Refer to the documentation for a complete listing.

The SAXPY example in the Transformations section showed how *kernel fusion* can be used to reduce the number of memory transfers used by a transformation kernel. With `thrust::transform_reduce` we can also apply kernel fusion to reduction kernels. Consider the following example which computes the norm of a vector.

```
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <cmath>

// square<T> computes the square of a number f(x) -> x*x
template <typename T>
struct square
{
    __host__ __device__
        T operator()(const T& x) const {
            return x * x;
        }
};

int main(void)
{
    // initialize host array
    float x[4] = {1.0, 2.0, 3.0, 4.0};
```

```
    // transfer to device
    thrust::device_vector<float> d_x(x, x + 4);

    // setup arguments
    square<float>        unary_op;
    thrust::plus<float> binary_op;
    float init = 0;

    // compute norm
    float norm = std::sqrt( thrust::transform_reduce(d_x.begin(), d_x.end(), ↩
        unary_op, init, binary_op) );

    std::cout << norm << std::endl;

    return 0;
}
```

Here we have a *unary operator* called `square` that squares each element of the input sequence. The sum of squares is then computed using a standard `plus` reduction. Like the slower version of the SAXPY transformation, we could implement `norm` with multiple passes: first a `transform` using `square` or perhaps just `multiplies` and then a `plus` reduction over a temporary array. However this would be unnecessarily wasteful and considerably slower. By fusing the square operation with the reduction kernel we again have a highly optimized implementation which offers the same performance as hand-written kernels.

Prefix-Sums

Parallel prefix-sums, or *scan* operations, are important building blocks in many parallel algorithms such as stream compaction and radix sort. Consider the following source code which illustrates an *inclusive scan* operation using the default `plus` operator:

```
#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}
```

In an inclusive scan each element of the output is the corresponding partial sum of the input range. For example, `data[2] = data[0] + data[1] + data[2]`. An *exclusive scan* is similar, but shifted by one place to the right:

```
#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}
```

So now `data[2] = data[0] + data[1]`. As these examples show, `inclusive_scan` and `exclusive_scan` are permitted to be performed in-place. Thrust also provides the functions `transform_inclusive_scan` and `transform_exclusive_scan` which apply a unary function to the input sequence before performing the scan. Refer to the documentation for a complete list of scan variants.

Reordering

Thrust provides support for *partitioning* and *stream compaction* through the following algorithms:

- `copy_if` : copy elements that pass a predicate test
- `partition` : reorder elements according to a predicate (true values precede false values)
- `remove` and `remove_if` : remove elements that fail a predicate test
- `unique`: remove consecutive duplicates within a sequence Refer to the documentation for a complete list of reordering functions and examples of their usage.

Sorting

Thrust offers several functions to sort data or rearrange data according to a given criterion. The `thrust::sort` and `thrust::stable_sort` functions are direct analogs of sort and stable_sort in the STL.

```
#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}
```

In addition, Thrust provides `thrust::sort_by_key` and `thrust::stable_sort_by_key`, which sort key-value pairs stored in separate places.

```
#include <thrust/sort.h>
...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::sort_by_key(keys, keys + N, values);
// keys is now   {  1,   2,   4,   5,   7,   8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```

Like their STL brethren, the sorting functions also accept user-defined comparison operators:

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

Fancy Iterators

Fancy iterators perform a variety of valuable purposes. In this section we'll show how fancy iterators allow us to attack a broader class problems with the standard Thrust algorithms. For those familiar with the Boost C++ Library, note that our fancy iterators were inspired by (and generally derived from) those in Boost Iterator Library.

constant_iterator

Arguably the simplest of the bunch, `constant_iterator` is simply an iterator that returns the same value whenever we access it. In the following example we initialize a constant iterator with the value 10.

```
#include <thrust/iterator/constant_iterator.h>
...
// create iterators
thrust::constant_iterator<int> first(10);
thrust::constant_iterator<int> last = first + 3;

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

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

Whenever an input sequence of constant values is needed, `constant_iterator` is a convenient and efficient solution.

counting_iterator

If a sequence of increasing values is required, then `counting_iterator` is the appropriate choice. Here we initialize a `counting_iterator` with the value 10 and access it like an array.

```
#include <thrust/iterator/counting_iterator.h>
...
// create iterators
thrust::counting_iterator<int> first(10);
thrust::counting_iterator<int> last = first + 3;

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

// sum of [first, last)
thrust::reduce(first, last);   // returns 33 (i.e. 10 + 11 + 12)
```

While `constant_iterator` and `counting_iterator` act as arrays, they don't actually require any memory storage. Whenever we dereference one of these iterators it generates the appropriate value on-the-fly and returns it to the calling function.

transform_iterator

In the Algorithms section we spoke about kernel fusion, i.e. combining separate algorithms like `transform` and `reduce` into a single `transform_reduce` operation. The `transform_iterator` allows us to apply the same technique, even when we don't have a special `transform_xxx` version of the algorithm. This example shows another way to fuse a transformation with a reduction, this time with just plain `reduce` applied to a `transform_iterator`.

```
#include <thrust/iterator/transform_iterator.h>
// initialize vector
thrust::device_vector<int> vec(3);
vec[0] = 10; vec[1] = 20; vec[2] = 30;

// create iterator (type omitted)
... first = thrust::make_transform_iterator(vec.begin(), negate<int>());
... last  = thrust::make_transform_iterator(vec.end(),   negate<int>());

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

// sum of [first, last)
thrust::reduce(first, last);   // returns -60 (i.e. -10 + -20 + -30)
```

Note, we have omitted the types for iterators `first` and `last` for simplicity. One downside of `transform_iterator` is that it can be cumbersome to specify the full type of the iterator, which can be quite lengthy. For this reason, it is common practice to simply put the call to `make_transform_iterator` in the arguments of the algorithm being invoked. For example,

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

allows us to avoid creating a variable to store `first` and `last`.

permutation_iterator

In the previous section we showed how `transform_iterator` is used to fuse a transformation with another algorithm to avoid unnecessary memory operations. The `permutation_iterator` is similar: it allows us to fuse gather and scatter operations with Thrust algorithms, or even other fancy iterators. The following example shows how to fuse a gather operation with a reduction.

```
#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↩
                             ()));
```

Here we have used the `make_permutation_iterator` function to simplify the construction of the `permutation_iterators`. The first argument to `make_permutation_iterator` is the source array of the gather operation and the second is the list of map indices. Note that we pass in `source.begin()` for the first argument in both cases, but vary the second argument to define the beginning and end of the sequence.

When a `permutation_iterator` is used as an output sequence of a function it is equivalent to fusing a scatter operation to the algorithm. In general `permutation_iterator` allows you to operate on specific set of values in a sequence instead of the entire sequence.

zip_iterator

Keep reading, we've saved the best iterator for last! The `zip_iterator` is an *extremely* useful gadget: it takes multiple input sequences and yields a sequence of tuples. In this example we "zip" together a sequence of `int` and a sequence of `char` into a sequence of `tuple<int,char>` and compute the `tuple` with the maximum value.

```
#include <thrust/iterator/zip_iterator.h>
...
// 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')
```

What makes `zip_iterator` so useful is that most algorithms accept either one, or occasionally two, input sequences. The `zip_iterator` allows us to combine many independent sequences into a single sequence of tuples, which can be processed by a broad set of algorithms.

Refer to the arbitrary_transformation example to see how to implement a ternary transformation with `zip_iterator` and `for_each`. A simple extension of this example would allow you to compute transformations with multiple *output* sequences as well.

In addition to convenience, `zip_iterator` allows us to implement programs more efficiently. For example, storing 3d points as an array of `float3` in CUDA is generally a bad idea, since array accesses are not properly coalesced. With `zip_iterator` we can store the three coordinates in three separate arrays, which does permit coalesced memory access. In this case, we use `zip_iterator` to create a *virtual* array of 3d vectors which we can feed in to Thrust algorithms. Refer to the dot_products_with_zip example for additional details.

Additional Resources

This guide only scratches the surface of what you can do with Thrust. The following resources can help you learn to do more with Thrust or provide assistance when problems arise.

- Comprehensive Documentation of Thrust's API
- A list of Frequently Asked Questions
- MegaNewtons: A blog for Thrust developers
- Collection of example programs

We strongly encourage users to subscribe to the thrust-users mailing list. The mailing list is a great place to seek out help from the Thrust developers and other Thrust users.

Notice

Trademarks

Copyright