An Introduction to the Thrust Parallel Algorithms Library
What is Thrust?

- High-Level Parallel Algorithms Library
- Parallel Analog of the C++ Standard Template Library (STL)
- Performance-Portable Abstraction Layer
- Productive way to program CUDA
Example

```c
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```
Easy to Use

- Distributed with CUDA Toolkit
- Header-only library
- Architecture agnostic
- Just compile and run!

$ nvcc -O2 -arch=sm_20 program.cu -o program
Why should I use Thrust?
Productivity

- **Containers**
  - `host_vector`
  - `device_vector`

- **Memory Management**
  - Allocation
  - Transfers

- **Algorithm Selection**
  - Location is implicit

```c++
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host data to device memory
thrust::device_vector<int> d_vec = h_vec;

// write device values from the host
d_vec[0] = 27;
d_vec[1] = 13;

// read device values from the host
int sum = d_vec[0] + d_vec[1];

// invoke algorithm on device
thrust::sort(d_vec.begin(), d_vec.end());

// memory automatically released
```
Productivity

- Large set of algorithms
  - ~75 functions
  - ~125 variations

- Flexible
  - User-defined types
  - User-defined operators

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>Sum of a sequence</td>
</tr>
<tr>
<td>find</td>
<td>First position of a value in a sequence</td>
</tr>
<tr>
<td>mismatch</td>
<td>First position where two sequences differ</td>
</tr>
<tr>
<td>inner_product</td>
<td>Dot product of two sequences</td>
</tr>
<tr>
<td>equal</td>
<td>Whether two sequences are equal</td>
</tr>
<tr>
<td>min_element</td>
<td>Position of the smallest value</td>
</tr>
<tr>
<td>count</td>
<td>Number of instances of a value</td>
</tr>
<tr>
<td>is_sorted</td>
<td>Whether sequence is in sorted order</td>
</tr>
<tr>
<td>transform_reduce</td>
<td>Sum of transformed sequence</td>
</tr>
</tbody>
</table>
Interoperability
Portability

- Support for CUDA, TBB and OpenMP
  - Just recompile!

```
nvcc -DTHRUST_DEVICE_SYSTEM=THRUST_HOST_SYSTEM_OMP
```

### NVIDIA GeForce GTX 580

```
$ time ./monte_carlo
pi is approximately 3.14159
real  0m6.190s
user  0m6.052s
sys   0m0.116s
```

### Intel Core i7 2600K

```
$ time ./monte_carlo
pi is approximately 3.14159
real  1m26.217s
user  11m28.383s
sys   0m0.020s
```
Extensibility

- Customize temporary allocation
- Create new backend systems
- Modify algorithm behavior
- New in Thrust v1.6
Robustness

- Reliable
  - Supports all CUDA-capable GPUs

- Well-tested
  - ~850 unit tests run daily

- Robust
  - Handles many pathological use cases
Openness

- Open Source Software
  - Apache License
  - Hosted on GitHub

Welcome to
- Suggestions
- Criticism
- Bug Reports
- Contributions
Resources

- **Documentation**

- **Examples**
  - https://github.com/thrust/thrust/tree/master/examples

- **Mailing List**
  - thrust-users googlegroup

- **Webinars**

- **Publications**

Thrust by Example
Thrust Basics

- Vectors and Iterators
- Algorithms and Functors
- Fancy Iterators
Thrust Basics - Vectors and Iterators

- Similar to STL Vector containers and iterators
- Two types of thrust vectors:
  - `thrust::host_vector<typename T>`
  - `thrust::device_vector<typename T>`
  - `thrust::host_vector<T>::iterator`
  - `thrust::device_vector<T>::iterator`

- `host/device` determines:
  - storage allocation (on the host or on the GPU device)
  - operation location (performed by CPU or performed by GPU)

Example: `thrust::device_vector<int> mydata(30);`
Thrust Basics - Vectors, Iterators and copying

thrust::device_vector<int> mydata(30);

- Implicit:
  thrust::host_vector<int> hostdata = mydata;
  int myitem = mydata[5];

- Explicit:
  thrust::copy(hostdata.begin(), hostdata.end(), mydata.begin());
Thrust Basics - Algorithms

thrust::device_vector<int> X(30), Y(30);
...

thrust::sort(X.begin(), X.end());
    // X is now sorted

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

int sum = thrust::reduce(X.begin(), X.end());
    // sum = \sum X_i
Thrust Basics - Algorithm Categories

Searching
Copying
Reductions
Merging
Reordering
  Partitioning
  Stream Compaction

Prefix-sums
Set Operations
Sorting
Transformations

http://thrust.github.io/doc/modules.html
Thrust Basics - Functors

▪ Built-in:

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

// Y_i = -X_i
```

▪ User-defined:

```cpp
struct square { __device__ __host__ int operator()(int xi) { return xi*xi; };

...  
```

```cpp
thrust::transform(X.begin(), X.end(), Y.begin(), square());

// Y_i = X_i^2
```
Thrust Basics - Fancy Iterators

- Constant Iterator
  - Auto-generates a constant sequence

- Counting Iterator
  - Auto-generates an incrementing or decrementing sequence

- Zip Iterator
  - Combines two or more sequences, producing “tuples”

- Transformation Iterator
  - Applies a transformation(functor) to a sequence “on the fly”

- Permutation Iterator
  - Vary the access order of a sequence “on the fly”
Simplified View of a GPU

DEVICE: Tesla M2090

1331 GFLOPS/s (Single Precision)

177 GB/s
Best Practices

- In general
  - Many applications are limited by memory bandwidth

- Best Practices
  - Fusion
    - Combined related operations together
  - Structure of Arrays
    - Ensure memory coalescing
  - Implicit sequences
    - Eliminate memory accesses and storage
Fusion: Sum of squares $\sum x_i^2$

```cpp
struct square { __device__ __host__ float operator()(float xi) { return xi*xi; });

float sum_of_squares(const thrust::device_vector<float> &x) {
    size_t N = x.size();
    thrust::device_vector<float> x_squared(N); // Temporary storage: N elements.

    // Compute $x^2$: N reads + N writes.
    thrust::transform(x.begin(), x.end(), x_squared.begin(), square());

    // Compute the sum of $x^2$s: N + k reads + k+1 writes (k is a small constant).
    return thrust::reduce(x_squared.begin(), x_squared.end());
}
```
Fusion

- Combined related operations together

```cpp
float fused_sum_of_squares(const thrust::device_vector<float> &x) {
    // Compute the x^2s and their sum: N + k reads + k+1 writes (k is a small constant).
    return thrust::reduce(
        thrust::make_transform_iterator(x.begin(), square()),
        thrust::make_transform_iterator(x.end(), square()));
}
```

We save:
- N temporary storage (x_squared)
- N writes (to x_squared)
- N reads (from x_squared)
Structure of Arrays

- `struct Float3 { float x, y, z; };`  
  
- Array of 32 Float3: `Float3[32]` (32 Float3 = 32x12B = 384B)

  1 memory load = 128B (involves 32 threads)

  ![Memory diagram]

  - Load the 32 x: 3 x 128B. Same for y and z \( \Rightarrow \) 3x3x128B = 1.125KB (only 384B needed)\(^1\)

\(^1\)GPUs based on Fermi and Kepler architectures have L1-cache to help here.
Structure of Arrays

- Group $x$s, $y$s and $z$s

```cpp
struct StructOfFloats {
    thrust::device_vector<float> x;
    thrust::device_vector<float> y;
    thrust::device_vector<float> z;
};
```

- Load $x$: 1 x 128B. Same for $y$ and $z$ $\Rightarrow$ 3x128B = 384B (all needed)
Structure of Arrays

Example: Scale a sequence of Float3

```cpp
struct scale
{
  typedef thrust::tuple<float, float, float> Float3;
  float s;
  scale(float s) : s(s) {}
  __host__ __device__ Float3 operator()(Float3 t)
  {
    float x = thrust::get<0>(t);
    float y = thrust::get<1>(t);
    float z = thrust::get<2>(t);
    return thrust::make_tuple(s*x, s*y, s*z);
  }
};

thrust::transform(
  thrust::make_zip_iterator(thrust::make_tuple(x.begin(), y.begin(), z.begin())),
  thrust::make_zip_iterator(thrust::make_tuple(x.end(), y.end(), z.end())),
  thrust::make_zip_iterator(thrust::make_tuple(x.begin(), y.begin(), z.begin())),
  scale(2.0f));
```
Implicit Sequences

- Often we need ranges following a sequential pattern
  - Constant ranges
    - \([1, 1, 1, 1, ...]\)
  - Incrementing ranges
    - \([0, 1, 2, 3, ...]\)

- Implicit ranges require no storage
  - \texttt{thrust::constant_iterator}
  - \texttt{thrust::counting_iterator}
EXAMPLES
Processing Rainfall Data

Notes
1) Time series sorted by day
2) Measurements of zero are excluded from the time series
Storage Options

- Array of structures

```c
struct Sample
{
    int day;
    int site;
    int measurement;
};
thrust::device_vector<Sample> data;
```

- Structure of arrays (Best Practice)

```c
struct Data
{
    thrust::device_vector<int> day;
    thrust::device_vector<int> site;
    thrust::device_vector<int> measurement;
};
Data data;
```
Number of Days with Any Rainfall

```cpp
int compute_number_of_days_with_rainfall(const Data &data) {
    return thrust::inner_product(data.day.begin(), data.day.end() - 1,
                                  data.day.begin() + 1,
                                  1,
                                  thrust::plus<int>(), // + functor
                                  thrust::not_equal_to<int>(); // * functor
}
```

day[0 0 1 2 5 5 6 6 7 8 ... ]

day shifted[0 0 1 2 5 5 6 6 7 8 ... ]

inner_product(x,y) = x[0]*y[0] + x[1]*y[1] + x[2]*y[2] + ...
Total Rainfall at Each Site

template<typename Vector>
void compute_total_rainfall_per_site(const Data &data, Vector &site, Vector &measurement)
{
    // Copy data to keep the original data as it is.
    Vector tmp_site(data.site), tmp_measurement(data.measurement);

    // Sort the “pairs” (site, measurement) by increasing value of site.
    thrust::sort_by_key(tmp_site.begin(), tmp_site.end(), tmp_measurement.begin());

    // Reduce measurements by site (Assumption: site/measurement are big enough).
    thrust::reduce_by_key(tmp_site.begin(), tmp_site.end(), tmp_measurement.begin(),
                          site.begin(),
                          measurement.begin());
}

tmp_site [0 1 1 1 1 2 2 2 3 ...
tmp_measurement [6 2 3 3 6 10 9 8 5 ...

site [0 1 2 3 ...
measurement [8 22 22 5 ...]
Number of Days where Rainfall Exceeded 5

```
using namespace thrust::placeholders;

int count_days_where_rainfall_exceeded_5(const Data &data) {
    size_t N = compute_number_of_days_with_rainfall(data);

    thrust::device_vector<int> day(N);
    thrust::device_vector<int> measurement(N);

    thrust::reduce_by_key(
        data.day.begin(), data.day.end(),
        data.measurement.begin(),
        day.begin(),
        measurement.begin());

    return thrust::count_if(measurement.begin(), measurement.end(), _1 > 5);
}

struct greater_than {
    int threshold;
    greater_than( int threshold ) : threshold( threshold ) {}  
    __device__ __host__ bool operator()( int i ) { return i > threshold; }
};
```
First Day where Total Rainfall Exceeded 32

```cpp
int find_first_day_where_total_rainfall_exceeded_32(const Data &data)
{
    // Allocate memory to store the prefix sums of measurement.
    thrust::device_vector<int> sums(data.measurement.size());

    // Compute prefix sums.
    thrust::inclusive_scan(data.measurement.begin(), data.measurement.end(), sums.begin());

    // Find the 1st day using a binary search (prefix sums are sorted - by definition).
    int day = thrust::lower_bound(sums.begin(), sums.end(), 33) - sums.begin();

    // Get the day.
    return data.day[day];
}
```

<table>
<thead>
<tr>
<th>day</th>
<th>[0 0 1 2 5 5 6 6 7 8 ... ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>meas</td>
<td>[9 5 6 3 3 8 2 6 5 10 ... ]</td>
</tr>
<tr>
<td>sums</td>
<td>[9 14 20 23 26 34 36 42 47 57 ... ]</td>
</tr>
</tbody>
</table>
### Sort Unsorted Input

<table>
<thead>
<tr>
<th>day</th>
<th>0 5 1 6 5 7 2 0 8 6 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>site</td>
<td>2 2 0 0 1 2 1 3 1 1 ...</td>
</tr>
<tr>
<td>measure</td>
<td>9 8 6 2 3 5 3 5 10 6 ...</td>
</tr>
</tbody>
</table>

Sort by day and site

<table>
<thead>
<tr>
<th>day</th>
<th>0 0 1 2 5 5 6 6 7 8 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>site</td>
<td>2 3 0 1 1 2 0 1 2 1 ...</td>
</tr>
<tr>
<td>measure</td>
<td>9 5 6 3 3 8 2 6 5 10 ...</td>
</tr>
</tbody>
</table>
Sort Unsorted Input

```cpp
struct day_site_cmp
{
  template <typename Tuple0, typename Tuple1>
  __device__ __host__ bool operator()(const Tuple0 &t0, const Tuple1 &t1)
  {
    int day0 = thrust::get<0>(t0);
    int day1 = thrust::get<0>(t1);
    int site0 = thrust::get<1>(t0);
    int site1 = thrust::get<1>(t1);

    return day0 < day1 || (day0 == day1 && site0 < site1);
  }
};

void sort_data(Data &data)
{
  thrust::sort_by_key(
    thrust::make_zip_iterator(thrust::make_tuple(data.day.begin(), data.site.begin())),
    thrust::make_zip_iterator(thrust::make_tuple(data.day.end(), data.site.end())),
    data.measurements.begin(),
    day_site_cmp());
}
```
Sort Unsorted Input (Faster)

- 40M elements sorted on a Tesla M2090:
  - 1st version: 990.76ms
  - 2nd version: 131.05ms

```cpp
void sort_data(Data &data)
{
    thrust::device_vector<int64> tmp(data.day.size());

    // Pack (day, site) pairs into 64-bit integers.
    thrust::transform(
        thrust::make_zip_iterator(thrust::make_tuple(data.day.begin(), data.site.begin())),
        thrust::make_zip_iterator(thrust::make_tuple(data.day.end(), data.site.end())),
        tmp.begin(),
        pack());

    // Sort using the 64-bit integers as keys.
    thrust::sort_by_key(tmp.begin(), tmp.end(), data.measurement.begin());

    // Unpack (day, site) pairs from 64-bit integers.
    thrust::transform(
        tmp.begin(),
        tmp.end(),
        thrust::make_zip_iterator(thrust::make_tuple(data.day.begin(), data.site.begin())),
        unpack());
}
```
This chapter demonstrates how to leverage the Thrust parallel template library to implement high-performance applications with minimal programming effort. Based on the C++ Standard Template Library (STL), Thrust brings a familiar high-level interface to the realm of GPU Computing while remaining fully interoperable with the rest of the CUDA software ecosystem. Applications written with Thrust are concise, readable, and efficient.

26.1 MOTIVATION

With the introduction of CUDA C/C++, developers can harness the massive parallelism of the GPU through a standard programming language. CUDA allows developers to make fine-grained decisions about how computations are decomposed into parallel threads and executed on the device. The level of control offered by CUDA C/C++ (henceforth CUDA C) is an important feature: it facilitates the development of high-performance algorithms for a variety of computationally demanding tasks which (1) merit significant optimization and (2) profit from low-level control of the mapping onto hardware. For this class of computational tasks CUDA C is an excellent solution.

Thrust (1) solves a complementary set of problems, namely those that are (1) implemented efficiently without a detailed mapping of work onto the target architecture or those that (2) do not merit or simply will not receive significant optimization effort by the user. With Thrust, developers describe their computation using a collection of high-level algorithms and completely delegate the decision of how to implement the computation to the library. This abstract interface allows programmers to describe what to compute without placing any additional restrictions on how to carry out the computation. The conservative low-level execution of a Thrust kernel (for the benefit of its execution in multiple threads) is transparent to the programmer, resulting in an abstraction that is both more high-level and more powerful than that of C++.

Thrust was designed to provide a Python interface with a high-level API that allowed users to quickly prototype and optimize CUDA-based applications. The Python interface offered the advantage of being able to rapidly create Python scripts that could be executed on any platform, with minimal effort required to port the code to the target GPU. However, the Python interface also had its limitations, particularly when it came to performance. As a result, the Thrust API for C++ was developed, providing a more efficient and powerful interface for developers who needed to write high-performance applications.

The C++ API for Thrust is significantly different from the Python API. The C++ API provides a more flexible and powerful interface, allowing developers to write more efficient and maintainable code. However, the Python API offers a more convenient and intuitive way to prototype and test applications. The choice between the two APIs depends on the specific needs of the developer. If performance is a top priority, the C++ API is the way to go. If ease of use and rapid prototyping are more important, the Python API is the better choice.
Thrust on (Google Code →) Github

- Quick Start Guide
- Examples
- News
- Documentation
- Mailing List (thrust-users googlegroup)
Thrust in the CUDA Toolkit

Sort Unsorted Input (Faster)

```
struct pack
{
    template <typename Tuple>
    __device__ __host__ int64 operator()(const Tuple &t)
    {
        return (static_cast<int64>(thrust::get<0>(t) << 32) | thrust::get<1>(t));
    }
};

struct unpack
{
    __device__ __host__ thrust::tuple<int, int> operator()(int64 p)
    {
        int d = static_cast<int>(p >> 32);
        int s = static_cast<int>(p & 0xffffffff);
        return thrust::make_tuple(d, s);
    }
};
```
Total Rainfall at a Given Site

```cpp
struct one_site_measurement
{
    int site;
    one_site_measurement(int site) : site(site) {}

    __host__ __device__ int operator()(thrust::tuple<int, int> t)
    {
        if (thrust::get<0>(t) == site)
            return thrust::get<1>(t);
        else
            return 0;
    }
};

int compute_total_rainfall_at_one_site(int i, const Data &data)
{
    // Fused thrust::transform_reduce (best practice).
    return thrust::transform_reduce(
        thrust::make_zip_iterator(thrust::make_tuple(data.site.begin(), data.measurement.begin())),
        thrust::make_zip_iterator(thrust::make_tuple(data.site.end(), data.measurement.end())),
        one_site_measurement(i),
        0,
        thrust::plus<int>())
;}
```