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

```cpp
#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

```cpp
// 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!

```bash
nvcc -DTHRUST_DEVICE_SYSTEM=THRUST_HOST_SYSTEM_OMP
```

<table>
<thead>
<tr>
<th>NVIDIA GeForce GTX 580</th>
<th>Intel Core i7 2600K</th>
</tr>
</thead>
<tbody>
<tr>
<td>`$ time ./monte_carlo</td>
<td>`$ time ./monte_carlo</td>
</tr>
<tr>
<td>pi is approximately 3.14159</td>
<td>pi is approximately 3.14159</td>
</tr>
<tr>
<td>real 0m6.190s</td>
<td>real 1m26.217s</td>
</tr>
<tr>
<td>user 0m6.052s</td>
<td>user 11m28.383s</td>
</tr>
<tr>
<td>sys 0m0.116s</td>
<td>sys 0m0.020s</td>
</tr>
</tbody>
</table>
## Backend System Options

### Host Systems
- `THRUST_HOST_SYSTEM_CPP`
- `THRUST_HOST_SYSTEM_OMP`
- `THRUST_HOST_SYSTEM_TBB`

### Device Systems
- `THRUST_DEVICE_SYSTEM_CUDA`
- `THRUST_DEVICE_SYSTEM_OMP`
- `THRUST_DEVICE_SYSTEM_TBB`
Multiple Backend Systems

- Mix different backends freely within the same app

```cpp
thrust::omp::vector<float> my_omp_vec(100);
thrust::cuda::vector<float> my_cuda_vec(100);
...

// reduce in parallel on the CPU
thrust::reduce(my_omp_vec.begin(), my_omp_vec.end());

// sort in parallel on the GPU
thrust::sort(my_cuda_vec.begin(), my_cuda_vec.end());
```
Potential Workflow

- Thrust Implementation
- Profile Application
- Specialize Components

Application

Bottleneck

Optimized Code
Performance Portability

Thrust

CUDA
Transform | Scan | Sort | Reduce

OpenMP
Transform | Scan | Sort | Reduce

Radix Sort | Merge Sort
G80 | GT200 | Fermi | G80 | GT200 | Fermi
Performance Portability

Slashdot

Developers: Sorting Algorithm Breaks Giga-Sort Barrier, With GPUs

Posted by timothy on Sunday August 29, @10:22PM from the quick-like-double-time dept.

An anonymous reader writes

"Researchers at the University of Virginia have recently open sourced an algorithm capable of sorting at a rate of one billion (integer) keys per second using a GPU. Although GPUs are often assumed to be poorly suited for algorithms like sorting, their results are several times faster than the best known CPU-based sorting implementations."

Read More... 99 comments
gpu graphics hardware developers programming story

Your Rights Online: Network Neutrality Is Law In Chile

Posted by timothy on Sunday August 29, @07:25PM from the muy-bien-tal-vez dept.

An anonymous reader writes

"Chile is the first country of the world to guarantee by law the principle of network neutrality, according to the Telecommunications Market Commission. Chile is following the latest law from Europe. The official newspaper of the Chilean Republic published yesterday an article about the new law. The draft law was approved in the Senate, now it will go to the Chamber of Deputies to be discussed and approved."

Read More... 3 comments

GPU TECHNOLOGY CONFERENCE
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
- Mailing List
- Webinars
- Publications

thrust.github.com
Thrust by Example

Julien Demouth, Nvidia
BEST PRACTICES
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 $\sum x_i^2$

```c
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

- \textbf{struct} \texttt{Float3} { \texttt{float} \texttt{x, y, z}; };

- Array of 32 \texttt{Float3}: \texttt{Float3[32]} (32 \texttt{Float3} = 32x12B = 384B)

- Load the 32 \texttt{x}: 3 \times 128B. Same for \texttt{y} and \texttt{z} \Rightarrow 3\times3\times128B = 1.125KB (only 384B needed)\(^1\)

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

- Group xs, ys and zs

```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  \( \Rightarrow 3 \times 128B = 384B \) (all needed)
Structure of Arrays

- Example: Scale a sequence of Float3

```c++
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
  - `thrust::constant_iterator`
  - `thrust::counting_iterator`
EXAMPLES
Processing Rainfall Data

- day: [0 0 1 2 5 5 6 6 7 8 ... ]
- site: [2 3 0 1 1 2 0 1 2 1 ... ]
- measurement: [9 5 6 3 3 8 2 6 5 10 ... ]

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

- Array of structures

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

- Structure of arrays (Best Practice)

```cpp
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>()) + 1; // * functor
}
```

day

\[
\begin{array}{ccccccccccc}
0 & 0 & 1 & 2 & 5 & 5 & 6 & 6 & 7 & 8 & \ldots \\
\end{array}
\]

day shifted

\[
\begin{array}{ccccccccccc}
0 & 0 & 1 & 2 & 5 & 5 & 6 & 6 & 7 & 8 & \ldots \\
\end{array}
\]

inner_product\((x,y) = x[0]*y[0] + x[1]*y[1] + x[2]*y[2] + \ldots\)
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  = 0 1 = 1 = 1 = 1 2 = 2 = 2 3 ... ]
tmp_measurement [6 + 2 3 + 3 + 6 + 10 9 + 8 + 5 5 ... ]

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

```cpp
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; }
};
```

_1 > 5
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>measurement</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

Sort by day and site

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

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

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());
}

- 40M elements sorted on a Tesla M2090:
  - 1st version: 990.76ms
  - 2nd version: 131.05ms
Thrust in the CUDA Toolkit

Thrust in GPU Computing Gems

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++ (referred to as 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 fine-grained 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 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 on the GPU.
Thrust on Google Code

- Quick Start Guide
- Examples
- News
- Documentation
- Mailing List (thrust-users)
Sort Unsorted Input (Faster)

```cpp
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 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>())
;
}
```
Total Rainfall Between Given Days

```cpp
int compute_total_rainfall_between_days(int first_day, int last_day, const Data &data) {
    // Search first_day/last_day using binary searches.
    int first = thrust::lower_bound(data.day.begin(), data.day.end(), first_day) - data.day.begin();
    int last = thrust::upper_bound(data.day.begin(), data.day.end(), last_day) - data.day.begin();

    // Reduce the measurements between the two bounds.
    return thrust::reduce(data.measurement.begin() + first, data.measurement.begin() + last);
}
```

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