C++ Standard Template Library

C++ **STL** supplies a number of algorithms that are available in Thrust

STL provide efficient ways to store, access, manipulate and view data
also includes containers,
#include<numeric>
#include <algorithm>

Example: List of random numbers, sorted, and printed in 4 lines of code.

```
vector<int> myVector(NUM_INTS);                    //See Class4.pdf
generate(myVector.begin(), myVector.end(),  rand);
sort(myVector.begin(), myVector.end());
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, “\n"));
```

```
int A[] = {1, 4, 2, 8, 5, 7};
const int N = sizeof(A) / sizeof(int);
sort(A, A + N);
copy(A, A + N, ostream_iterator<int>(cout, " "));
// The output is " 1 2 4 5 7 8".
```

C++ Standard Template Library: Algorithms

Searching, sorting, reordering, permuting, creating, and destroying sets of data.

Over 50 of them: either in <numeric> or <algorithm>

http://www.cplusplus.com/reference/algorithm/

deque: double ended queue

Vectors allow insertion of data into the middle of the vector, but often just want to add at the ends.

Deque is a vector with insertion at both the back end: push_back
but also at the front end: push_front

pop_back to retrieve from back
pop_front

deque is a little slower than vector

Thrust Library

http://docs.nvidia.com/cuda/thrust/index.html
http://thrust.github.io

Data structures:
- thrust::device_vector
- thrust::host_vector
- etc

```
thrust::host_vector<int> A(10,1)   :  initialize A with 10 elements set to 1
thrust::host_vector<int> h_vec(12);

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

thrust::device_vector<int> d_vec = h_vec;

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

```
#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);
    H[0] = 14; H[1] = 20; H[2] = 38; H[3] = 46;
   std::cout << "H has size " << H.size() << std::endl;
   for(int i = 0; i < H.size(); i++)
        std::cout << "H[" << i << "] = " << H[i] << std::endl;
   H.resize(2);
   std::cout << "H now has size " << H.size() << std::endl;
   thrust::device_vector<int> D = H;
   D[0] = 99;
   D[1] = 88;
   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;
}
```

Thrust Library

Some algorithms—four generic types

transformation (for_each element, apply an operation)

```
thrust::sequence(X.begin(), X.end());
```

reduce (reduction of an array to a single number)

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

scan (prefix-sum)

```
thrust::inclusive_scan(data, data + 6, data); // in-place scan
```

sort

```
thrust::sort(A, A + 6);
```

Note: here all commands begin with thrust:: as that is the namespace for the thrust commands (so as not to confuse with STL commands)

Thrust functions

```
adjacent_difference.h   device_malloc_allocator.h   execution_policy.h   inner_product.h   random.h          sort.h              uninitialized_copy.h
advance.h               device_malloc.h             extrema.h            iterator          reduce.h          swap.h              uninitialized_fill.h
binary_search.h         device_new_allocator.h      fill.h               logical.h         remove.h          system              unique.h
copy.h                  device_new.h                find.h               memory.h          replace.h         system_error.h      version.h
count.h                 device_ptr.h                for_each.h           merge.h           reverse.h         tabulate.h
detail                  device_reference.h          functional.h         mismatch.h        scan.h            transform.h
device_allocator.h      device_vector.h             gather.h             pair.h            scatter.h         transform_reduce.h
device_delete.h         distance.h                  generate.h           partition.h       sequence.h        transform_scan.h
device_free.h           equal.h                     host_vector.h        random            set_operations.h  tuple.h
```

Thrust: Sum 1024 random numbers

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

__host__ static __inline__ float rand_01()
{
    return ((float)rand()/RAND_MAX);  //random number between 0 -1
}
int main(void)  {
  /* generate random data on the host */
  thrust::host_vector<float> h_vec(1024);
  thrust::generate(h_vec.begin(), h_vec.end(), rand_01);
  /* transfer to device and compute sum */
  thrust::device_vector<float> d_vec = h_vec;
  float x = thrust::reduce(d_vec.begin(), d_vec.end());
  return 0;
}
```

C++ Function Template

```
// function template to add numbers (type of T is variable)
template< typename T >
 T add(T a, T b)
{
        return a + b;
}
    // add integers
int x = 10; int y = 20; int z;
z = add<int>(x,y);                  // type of T explicitly specified
z = add(x,y);                       // type of T determined automatically

// add floats
float x = 10.0f; float y = 20.0f; float z;
z = add<float>(x,y); // type of T explicitly specified
z = add(x,y); // type of T determined automatically
```

Function Objects (Functors)

```
// templated functor to add numbers

template< typename T >
    class add
 {
        public:
        T operator()(T a, T b)
        {
            return a + b;
        }
 };

int x = 10; int y = 20; int z;
add<int> func; // create an add functor for T=int
z = func(x,y);        // invoke functor on x and y, which are ints

float x = 10; float y = 20; float z;
add<float> func; // create an add functor for T=float
z  = func(x,y);    // invoke functor on x and y, which are floats
```

Another Example

```
// this is a functor
struct add_x {
  add_x(int x) : x(x) {}
  int operator()(int y) { return x + y; }

private:
  int x;
};

// Now you can use it like this:
add_x add42(42); // create an instance of the functor class
int i = add42(8); // and "call" it
assert(i == 50); // and it added 42 to its argument

std::vector<int> in; // assume this contains a bunch of values)
std::vector<int> out;
// Pass a functor to std::transform, which calls the functor on every element
// in the input sequence, and stores the result to the output sequence
std::transform(in.begin(), in.end(), out.begin(), add_x(1));
assert(out[i] == in[i] + 1); // for all i
```

Calculate Pi (first part)

```
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <cmath>
….
// integrand<T> computes    f(x) using a functor (C++ function object)
template <typename T>
struct integrand
{
    __host__ __device__                    //means works on both host and device
        T operator()(const T& x) const {
            return 4./(1+ x * x);
        }
};
```

Calculate Pi (last part)

```
// initialize host array
   thrust::host_vector<double> x(N);  //initialize on host—dynamic container
   for (int i = 0; i<N; i++)
      x[i]= interval* ( (double) i + 0.5);

   // transfer to device
   thrust::device_vector<double> d=x;  //initialize of device and copy x to device vector

   // setup arguments
   integrand<double>      unary_op;
   thrust::plus<double>   binary_op;   //available in Thrust
   double init = 0;

   // compute norm
    double norm =  thrust::transform_reduce(d.begin(), d.end(), unary_op, init, binary_op);
    norm *= interval;
```

<thrust/functional.h>

struct thrust::plus< T >
struct thrust::minus< T >
struct thrust::multiplies< T >
struct thrust::divides< T >
struct thrust::modulus< T >
struct thrust::negate< T >
struct thrust::equal_to< T >
struct thrust::not_equal_to< T >
struct thrust::greater< T >
struct thrust::less< T >
struct thrust::greater_equal< T >
struct thrust::less_equal< T >
struct thrust::logical_and< T >
struct thrust::logical_or< T >
struct thrust::logical_not< T >
struct thrust::bit_and< T >
struct thrust::bit_or< T >
struct thrust::bit_xor< T >
struct thrust::identity< T >
struct thrust::maximum< T >
struct thrust::minimum< T >
.....

Thrust Summation

```
#include <thrust/reduce.h>

thrust::device_vector<int> D(10);

….define D….

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

initial value operator

or simpler

```
int sum = thrust::reduce(D.begin(), D.end());
```

Compute norm

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

using namespace std;

 thrust::device_vector<float> d_x ;

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

float init = 0;

// compute norm
float norm;

norm = sqrt( thrust::transform_reduce(d_x.begin(), d_x.end(), unary_op, init, binary_op) );
```

What is D?

```
int main(void)
{

    thrust::device_vector<int> D(10, 1);

    thrust::fill(D.begin(), D.begin() + 7, 9);

    thrust::host_vector<int> H(D.begin(), D.begin() + 5);

    thrust::sequence(H.begin(), H.end());


    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;
}
```

Scan

Prefix Sums:

thrust::inclusive_scan(data, data + 6, data); // in-place scan

For six elements add the preceding elements together one at a time: partial sums

#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}

Thrust Sort

Sort by keys:

```
#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'}
```

Using Thrust

Write a Thrust program to compute

y <— a x + y, where a is a constant and x and y are vectors (1024)

SAXPY (one way)

Given X, Y as defined, along with the constant A

```
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>());
```

```
const int N = 1000;
thrust::device_vector<float> V1(N);
thrust::device_vector<float> V2(N);
thrust::device_vector<float> V3(N);

thrust::sequence(V1.begin(), V1.end(), 1);
thrust::fill(V2.begin(), V2.end(), 75);

thrust::transform(V1.begin(), V1.end(), V2.begin(), V3.begin(),
                  thrust::multiplies<float>());
// V3 is now {75, 150, 225, ..., 75000}
```

Example of multiplies

Thrust

Example for SAXPY Problem (single precision a x + y)

```
 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;
         }
};

    int main() {
    // Y <- A * X + Y
     thrust::transform(X.begin(), X.end(), Y.begin(), Y.end(), saxpy_functor(A));

   or simpler
    thrust::transform(X.begin(), X.end(), Y.begin(), Y.end(), a*_1 + _2);
```

Comparison of scan and reduce (again)

Thrust on Host Only

```
#include <thrust/host_vector.h>
#include <thrust/sort.h>
#include <cstdlib>
#include <algorithm>
#include <iostream>
int main(void)
{
// serially generate 1M random numbers
    thrust::host_vector<int> vec(1 << 20);
    std::generate(vec.begin(), vec.end(), rand);
     // sort data on host
    thrust::sort(vec.begin(), vec.end());

 // report the largest number
    std::cout << "Largest number is " << vec.back() << std::endl;

    return 0;
}
```

/usr/local/bin/g++ -O3 thrust_on_host.cpp -I/Developer/NVIDIA/CUDA-6.5/include

Thrust and OpenMP on Host

```
 #include <thrust/host_vector.h>
#include <thrust/system/omp/vector.h>

#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <thrust/copy.h>

#include <cstdlib>
#include <algorithm>
 ……
// serially generate 1M random numbers
    thrust::host_vector<int> h_vec(1<<20);
    std::generate(h_vec.begin(), h_vec.end(), rand);

    //transfer data to OpenMP
    thrust::omp::vector<int> d_vec = h_vec;

    // sort data in parallel with OpenMP
    thrust::sort( d_vec.begin(), d_vec.end());

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