Thrust library

- **Purpose: Hide low-level details of kernel invocations, threads per block, number of blocks.**
- **Many deliberate similarities with the standard library.**
- **Start with memory management:**

```
thrust::host_vector<int> h_vec(2);
h_vec[0] = 1; h_vec[1] = 54;

thrust::device_vector<int> d_vec = h_vec;
// Allocated and copied in one line.

d_vec[1] = 3;
// Equivalent to cudaMemcpy with offset

cout << d_vec[1] << endl;
// Automatic cudaMemcpy back to temporary variable.
```

- **Notice that each** `operator[]` **call to a** `device_vector` **has a** `cudaMemcpy` **under the hood.**

Tricks with vectors

- **Thrust QuickStart guide kindly offers multiple examples:**

```
// Initialize all ten integers of a device_vector to 1
thrust::device_vector<int> d_vec(10, 1);

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

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

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

// copy all of h_vec back to the beginning of d_vec
thrust::copy(h_vec.begin(), h_vec.end(), d_vec.begin());
```

- **Notice use of `d_vec.begin()`, `d_vec.end()`. These are iterators. As a reasonable approximation, we can think of them as being integer indices into arrays. Hence for example `d_vec.begin() + 7` means "the seventh entry of `d_vec`."**

- **Notice that `fill`, `copy`, `sequence` and the others are template functions. They work with iterators from outside Thrust as well:**

```
std::list<int> stl_list;
thrust::device_vector<int> d_vec(stl_list.begin(), stl_list.end());
```

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

- **I have the Power of Templates: Any class with a `next` method will do. (Don't try this at home; your mileage may vary.)**

Built-in Thrust algorithms

- **Borrowed from the QuickStart guide:**

```
using namespace thrust;
device_vector<int> X(10);
device_vector<int> Y(10);
device_vector<int> Z(10);

// initialize X to 0,1,2,3, ...
sequence(X.begin(), X.end());
// compute Y = -X
transform(X.begin(), X.end(), Y.begin(), negate<int>());
// compute Y = X mod 2
fill(Z.begin(), Z.end(), 2);
transform(X.begin(), X.end(), Z.begin(), Y.begin(), modulus<int>());
```

- **Notice structure of `transform`:**
  - **Start of data to transform**
  - **End of data**
  - **In the case of binary transforms, start of second piece of data**
  - **Start of working space: Storage for transformed data**
  - **(No need for end of working space or of second data: It's the same size as the first data, or larger. Otherwise we get memory violations.)**
  - **Finally, the unary operator functor that implements the transformation.**

More built-ins

- **Where there are transformations, there are also reductions:**

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

- **Structure:**
  - **Start and end of data**
  - **Initial value of sum**
  - **Operator - in this case binary, not unary.**
- **Some other operators:**
  - `count`, `count_if`: **Return number of elements in data, or number of elements that satisfy a condition (this also requires a functor to say whether a given element satisfies the condition).**
  - `min_element`, `max_element`: **Return smallest or largest element.**
  - `is_sorted`: **True if the list is sorted in ascending order (relies on there being a less-than operator).**

Still more built-ins

- **A very common pattern is to do a transformation followed by a reduction; hence Thrust has a function for that:**

```
transform_reduce(d_vec.begin(),
                 d_vec.end(),
                 absolute<int>(),
                 // Cheating - not builtin!
                 0,
                 plus<int>);
```

- **This sums the absolute values of each element in `d_vec`.**
- **Structure combines information from `transform` and `reduce` functions, but drops the working space:**
  - **Beginning and end of data to work on**
  - **Transformation functor (unary)**
  - **Initial value of sum**
  - **Reduction functor (binary)**

Custom functors

- **Not everything can be done with combinations of builtins.**
- **Again, the power of templates: The unary and binary operators just need to have an `operator()` function which takes a particular set of arguments and returns the third. For example, a custom plus functor:**

```
struct CustomPlus {
  __device__ float operator() (float one, float two) {
    return one + two;
  }
}
```

- **Functors may store state. For example, suppose we want to implement SAXPY: $\vec{z} = a\vec{x} + \vec{y}$. Then the functor may store the multiplier a:**

```
struct CustomSaxpy {
  float a;
  __device__ float operator() (float y_value, float x_value) {
    return y_value + a * x_value;
  }
}
```

- **This is the defining quality of a functor: It implements a function and has state. If it had no state it would be, in effect, a function pointer: For given arguments the result would always be the same. In the case of the SAXPY operator above, for given arguments the result still depends on a.**

- **We may think of this as a "frozen argument": We give *a* a value at the construction of the functor. Alternatively we could have passed it as an additional argument of the `operator` method. (Indeed this is just what happens under the hood.)**

- **Notice that in the case of SAXPY we could have done it with two builtins:**

```
device_vector d_xvec;
device_vector d_yvec;
float a = 3.0;
device_vector d_avec(d_yvec.size(), a);
transform(d_xvec.begin(), d_xvec.end(),
          d_avec.begin(), d_avec.begin(), multiplies<float>());
// Notice destructive update!
// d_avec now stores a*x.
transform(d_avec.begin(), d_avec.end(),
          d_yvec.begin(), d_yvec.begin(), plus<float>());
// Another destructive update:
// d_yvec now stores y + a*x.
```

- **Notice that in this case the working space (where the result is stored) is the same as the second data.**

- **This method requires more memory back-and-forth than the `CustomSaxpy` functor which combines the multiply and add operations; it is likely slower. Combining kernel calls into one functor is called kernel fusion.**

Fancy iterators

- **Even though Thrust will ensure that each thread works on a specific element, sometimes you still need to keep track of which thread you're in. Thrust offers the counting iterator:**

  ```
  counting_iterator<int> first(10);
  counting_iterator<int> last = first + 3;
  reduce(first, last, 0, plus<int>());
  ```

  This sums integers 10, 11, 12. Notice that the counting iterators act like arrays but don't actually take up memory space!

- **For passing constants, we can do so as state of the functor operator, or as a constant iterator:**

  ```
  constant_iterator<double*> someConstants;
  ```

- **Useful for (eg) mixing plain CUDA with Thrust.**

- **What if we need to pass several things to a functor? For example, we might want to work with three vectors at once and store the result in a fourth (code from `arbitrary_transformation.cu` in Thrust examples):**

```
struct arbitrary_functor {
template <typename Tuple>
  __device__ void operator () (Tuple t) {
    // D[i] = A[i] + B[i] * C[i];
    get<3>(t)  = get<0>(t);
    get<3>(t) += get<1>(t) * get<2>(t);
  }
};

int main (void) {
  // allocate storage
  device_vector<float> A(5);
  device_vector<float> B(5);
  device_vector<float> C(5);
  device_vector<float> D(5);

  // initialize input vectors
  A[0] = 3; B[0] = 6; C[0] = 2;
  A[1] = 4; B[1] = 7; C[1] = 5;
  A[2] = 0; B[2] = 2; C[2] = 7;
  A[3] = 8; B[3] = 1; C[3] = 4;
  A[4] = 2; B[4] = 8; C[4] = 3;

  // apply the transformation
  for_each(make_zip_iterator(make_tuple(A.begin(), B.begin(),
                                        C.begin(), D.begin())),
           make_zip_iterator(make_tuple(A.end(), B.end(),
```

```
                                            C.end(), D.end())),
            arbitrary_functor());

    return 0;
  }
```

- `for_each` **applies its function to each element in the range** `[first, last)`.
- **Another example: a pointer, the thread index, and the actual data.**

```
constant_iterator<int> eventSize(numVars);
constant_iterator<double*> arrayAddress(cudaDataArray);
counting_iterator<unsigned int> eventIndex(0);
lognorm += transform_reduce(make_zip_iterator(
                             make_tuple(eventIndex,
                                        arrayAddress,
                                        eventSize)),
                            make_zip_iterator(
                             make_tuple(eventIndex + numEntries,
                                        arrayAddress,
                                        eventSize)),
                            *logger, 0, plus<double>());

__device__ double MetricTaker::operator () (tuple<unsigned int,
                                                  double*,
                                                  int> t) const {
  int eventIndex = get<0>(t);
  int eventSize  = get<2>(t);
```

```
  double* eventAddress = get<1>(t) + (eventIndex * abs(eventSize));
  double ret = (*(reinterpret_cast<device_eventfunction_ptr>(
                 device_function_table[functionIdx])))
                 (eventAddress,
                  cudaArray,
                  paramIndices+parameters,
                  eventIndex);
  return (*(reinterpret_cast<device_metric_ptr>
           (device_function_table[metricIndex])))
           (ret, eventAddress + (abs(eventSize)-2), parameters);
}
```

Summary and resources

- **Thrust hides memory management and kernel launching behind abstractions.**
- **Transformations and reductions can be done in a single step, at the price that the code has to conform to Thrust convention.**
- **Thrust documentation is slightly limited. There is the QuickStart guide:** `http://code.google.com/p/thrust/wiki/QuickStartGuide`
- **Some examples:**

  `http://code.google.com/p/thrust/source/browse/#hg%2Fexamples`
- **There is also the raw API listing:** `http://code.google.com/p/thrust/wiki/Documentation`
- **Google discussion group, where the devs post:** `http://groups.google.com/group/thrust-users`

Extended example: Chi-square fit

- **Code at** `~ucn1122/goofitcourse/exercises/example2.cu`**.**
- **Uses** `TMinuit.hh` **and** `TRandom.hh` **ripped from full ROOT library.**
- **On oakley (within an interactive session):**

```
module load cuda
cd $TMPDIR
cp -r ~ucn1122/goofitcourse/exercises/* .
cd rootstuff
gmake
cd ..
./compexp2 # Contains nvcc command
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:./rootstuff/
./exp2
```

- **Let's look at code (in order of execution):**

```
// Pointers to avoid crash on exit.
device_vector<double>* dev_yvals;
device_vector<double>* dev_xvals;

int main (int argc, char** argv) {
  // Generate random data
  TRandom donram(42);
  host_vector<double> host_yvals;
  host_vector<double> host_xvals;
```

```
for (int i = 0; i < 100; ++i) {
  double currX = 0.1*i;
  double currVal =  2.0*currX*currX;
  currVal       -=  0.8*currX;
  currVal       +=  3.2;
  host_yvals.push_back(currVal + donram.Gaus(0.0, 0.1));
  host_xvals.push_back(currX);
}

// Move to device
dev_yvals = new device_vector<double>(host_yvals);
dev_xvals = new device_vector<double>(host_xvals);
```

- **Why not this?**

```
device_vector<double> dev_yvals;
// (...)
  dev_yvals =  host_yvals;
```

Because then there is a crash at the end of program execution, with error message `unload of CUDA runtime failed`**. Possibly a problem with Thrust cleaning its memory at the wrong time? The CUDA driver is a black box. At any rate, avoid the problem with pointers and explicit control of the cleanup.**

- **MINUIT setup:**

```
// Fit to degree-two polynomial
TMinuit minuit(3);
minuit.DefineParameter(0, "quad", 1.8, 0.01, -4.0, 4.0);
minuit.DefineParameter(1, "line", 1.0, 0.01, -4.0, 4.0);
minuit.DefineParameter(2, "cons", 3.0, 0.01, -4.0, 4.0);
minuit.SetFCN(chisq);
minuit.Migrad();
```

- **Now comes the Thrust code:**

```
void chisq (int& npar, double* deriv, double& fVal, double param[], int flag)
  // Extract parameters from Minuit, put into ChisqFunctor constructor.
  double quad = param[0];
  double line = param[1];
  double cons = param[2];
  ChisqFunctor functor(quad, line, cons, 0.1); // Note hardcoded error!
  double initVal = 0;
  fVal = transform_reduce(make_zip_iterator(make_tuple(dev_xvals->begin(),
                                                       dev_yvals->begin())),
                          make_zip_iterator(make_tuple(dev_xvals->end(),
                                                       dev_yvals->end())),
                          functor,
                          initVal,
                          thrust::plus<double>());
                          // 'plus' also exists in STL.
}
```

- **All kernel launches, moving of parameters to device, and reduction management is hidden away by Thrust.**
- **We still have to do the actual per-point chi-square calculation:**

```
struct ChisqFunctor :
public thrust::unary_function<tuple<double, double>, double> {
  ChisqFunctor (double q, double l, double c, double e)
    : quad(q), line(l), cons(c), error(e) {}

  __device__ double operator () (tuple<double, double> xypair) {
    // Extract x and y from tuple
    double xval = get<0>(xypair);
    double yval = get<1>(xypair);

    // Calculate expected y
    double expected = quad * xval * xval;
    expected       += line * xval;
    expected       += cons;

    // Chisquare
    double chisq = (expected - yval);
    chisq *= chisq;
    chisq /= error;
    return chisq;
  }

  double quad;
```

```
  double line;
  double cons;
  double error;
};
```

- **Finally, cleanup:**

```
  // Free device memory.
  delete dev_yvals;
  delete dev_xvals;
  return 0;
} // End of main method.
```