Fusion requires a special type of iterator, whose type is horrendous to write. So, Thrust provides the **make_transform_iterator()** function, which we call to produce the special iterator needed, and then put the result directly into the second phase of our fusion, in this case into **scatter()**.

Essentially our use of **make_transform_iterator()** is telling Thrust, "Don't apply **transidx()** to **seq** yet. Instead, perform that operation as you go along, and feed each result of **transidx()** directly into **scatter()**." That word *directly* is the salient one here; it means we save n memory reads and n memory writes.[5] Moreover, we save the overhead of the kernel call, if our backend is CUDA.

Note that we also had to be a little bit more elaborate with data typing issues, writing the first line of our struct declaration as

```
struct transidx : public thrust::unary_function<int,int>
```

It won't work without this!

It would be nice to be able to use a counting iterator in the above code, but apparently the compiler encounters problems with determining where the end of the counting sequence is. There is similar code in the examples directory that comes with Thrust, and that one uses **gather()** instead of **scatter()**. Since the former specifies a beginning and an end for the map array, counting interators work fine.

6.12 A Timing Comparison

Let's look at matrix transpose one more time. First, we'll use the method, shown in earlier sections, of passing a device vector iterator to a functor. For variety, let's use Thrust's **for_each()** function. The following will be known as Code 1:

```
1  // matrix transpose, for_each version
2
3  #include <stdio.h>
4  #include <thrust/device_vector.h>
5
6  // functor; holds iterators for the input and output matrices, and each
7  // invocation of the function copies from one element from the former to
8  // the latter
9  struct copyelt2xp
10 {
11    int nrow;
12    int ncol;
13    const  thrust::device_vector<int>::iterator m;  // input matrix
```

[5] We are still writing to temporary storage, but that will probably be in registers (since we don't create the entire map at once), thus fast to access.

```
14      const  thrust::device_vector<int>::iterator mxp;  // output matrix
15      int *m1,*mxp1;
16      copyelt2xp(thrust::device_vector<int>::iterator _m,
17               thrust::device_vector<int>::iterator _mxp,
18               int _nr, int _nc):
19         m(_m), mxp(_mxp), nrow(_nr), ncol(_nc) {
20            m1 = raw_pointer_cast(&m[0]);
21            mxp1 = raw_pointer_cast(&mxp[0]);
22         }
23      __device__
24      void operator()(const int i)
25      // copies the i-th element of the input matrix to the output matrix
26      {  // elt i in input is row r, col c there
27         int r = i / ncol; int c = i % ncol;
28         // that elt will be row c and col r in output, which has nrow
29         // cols, so copy as follows
30         mxp1[c*nrow+r] = m1[r*ncol+c];
31      }
32  };
33
34  // transpose nr x nc inmat to outmat
35  void transp(int *inmat, int *outmat, int nr, int nc)
36  {
37      thrust::device_vector<int> dmat(inmat,inmat+nr*nc);
38      // make space for the transpose
39      thrust::device_vector<int> dxp(nr*nc);
40      thrust::counting_iterator<int> seqb(0);
41      thrust::counting_iterator<int> seqe = seqb + nr*nc;
42      // for each i in seq, copy the matrix elt to its spot in the
43      // transpose
44      thrust::for_each(seqb,seqe,
45         copyelt2xp(dmat.begin(),dxp.begin(),nr,nc));
46      thrust::copy(dxp.begin(),dxp.end(),outmat);
47  }
48
49  int rand16()  // generate random integers mod 16
50  {  return rand() % 16;  }
51
52  // test code:  cmd line args are matrix size, then row, col of elt to be
53  // checked
54
55  int main(int argc, char **argv)
56  {  int nr = atoi(argv[1]); int nc = nr;
57      int *mat = (int *) malloc(nr*nc*sizeof(int));
58      int *matxp = (int *) malloc(nr*nc*sizeof(int));
59      thrust::generate(mat,mat+nr*nc,rand16);
60      int checkrow = atoi(argv[2]);
61      int checkcol = atoi(argv[3]);
62      printf("%d\n",mat[checkrow*nc+checkcol]);
63      transp(mat,matxp,nr,nc);
```

```
64      printf("%d\n",matxp[checkcol*nc+checkrow]);
65  }
```

The **for_each()** function does what the name implies: It calls a function/functor for each element in a sequence, doing so in a parallel manner. Note that this also obviates our earlier need to use a discard iterator.

For comparison, we'll use the matrix transpose code that is included in Thrust's **examples/** file, to be referred to as Code 2:

```
1  // matrix transpose, from the Thrust package examples
2
3  #include <thrust/host_vector.h>
4  #include <thrust/device_vector.h>
5  #include <thrust/functional.h>
6  #include <thrust/gather.h>
7  #include <thrust/scan.h>
8  #include <thrust/iterator/counting_iterator.h>
9  #include <thrust/iterator/transform_iterator.h>
10 #include <iostream>
11 #include <iomanip>
12 #include <stdio.h>
13
14 // convert a linear index to a linear index in the transpose
15 struct transpose_index : public thrust::unary_function<size_t,size_t>
16 {
17   size_t m, n;
18
19   __host__ __device__
20   transpose_index(size_t _m, size_t _n) : m(_m), n(_n) {}
21
22   __host__ __device__
23   size_t operator()(size_t linear_index)
24   {
25       size_t i = linear_index / n;
26       size_t j = linear_index % n;
27
28       return m * j + i;
29   }
30 };
31
32 // convert a linear index to a row index
33 struct row_index : public thrust::unary_function<size_t,size_t>
34 {
35   size_t n;
36
37   __host__ __device__
38   row_index(size_t _n) : n(_n) {}
39
40   __host__ __device__
```

```
41    size_t operator()(size_t i)
42    {
43        return i / n;
44    }
45  };
46
47  // transpose an M-by-N array
48  template <typename T>
49  void transpose(size_t m, size_t n, thrust::device_vector<T>& src, thrust::device_vector<T>& dst
50  {
51    thrust::counting_iterator<size_t> indices(0);
52
53    thrust::gather
54      (thrust::make_transform_iterator(indices, transpose_index(n, m)),
55       thrust::make_transform_iterator(indices, transpose_index(n, m)) + dst.size(),
56       src.begin(),
57       dst.begin());
58  }
59
60  void transp(int *inmat, int *outmat, int nr, int nc)
61  {
62     thrust::device_vector<int> dmat(inmat,inmat+nr*nc);
63     // make space for the transpose
64     thrust::device_vector<int> dxp(nr*nc);
65     transpose(nr,nc,dmat,dxp);
66     thrust::copy(dxp.begin(),dxp.end(),outmat);
67  }
68
69  int rand16()  // generate random integers mod 16
70  {  return rand() % 16;  }
71
72  // test code:  cmd line args are matrix size, then row, col of elt to be
73  // checked
74  int main(int argc, char **argv)
75  {  int nr = atoi(argv[1]); int nc = nr;
76     int *mat = (int *) malloc(nr*nc*sizeof(int));
77     int *matxp = (int *) malloc(nr*nc*sizeof(int));
78     thrust::generate(mat,mat+nr*nc,rand16);
79     int checkrow = atoi(argv[2]);
80     int checkcol = atoi(argv[3]);
81     printf("%d\n",mat[checkrow*nc+checkcol]);
82     transp(mat,matxp,nr,nc);
83     printf("%d\n",matxp[checkcol*nc+checkrow]);
84  }
```

This approach is more efficient than ours in Section 6.8, making use of **gather()** instead of **scatter()**, to better take advantage of fancy iterators.

Code 1 is a lot easier to program than Code 2, but is it efficient? It turns out, though, that—good news!–the simpler code, i.e. Code 1, is just as good as Code 2 in the case of a CUDA backend, actually a little faster, and Code 1 is a lot faster in the OpenMP case.

Here we ran on CUDA backends, on a 10000x10000 matrix:

device	Code 1	Code 2
GeForce 9800 GTX	3.67	3.75
Tesla C2050	3.43	3.50

What about OpenMP? Here are some timing runs on a multicore machine (many more cores than the 16 we tried), using an input matrix of size 6000x6000:

# threads	Code 1	Code 2
2	9.57	23.01
4	5.17	10.62
8	3.01	7.42
16	1.99	3.35

6.13 Example: Transforming an Adjacency Matrix

Here is a Thrust approach to the example of Sections 4.13 and 5.11. To review, here is the problem:

Say we have a graph with adjacency matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{6.1}$$

with row and column numbering starting at 0, not 1. We'd like to transform this to a two-column matrix that displays the links, in this case

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 0 \\ 3 & 1 \\ 3 & 2 \end{pmatrix} \tag{6.2}$$