CHAPTER

Thrust: A Productivity-Oriented Library for CUDA

26

Nathan Bell and Jared Hoberock

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. By capturing the programmer's intent at a high level, Thrust has the discretion to make informed decisions on behalf of the programmer and select the most efficient implementation.

The value of high-level libraries is broadly recognized in high-performance computing. For example, the widely-used BLAS standard provides an abstract interface to common linear algebra operations. First conceived more than three decades ago, BLAS remains relevant today in large part because it allows valuable, platform-specific optimizations to be introduced behind a uniform interface.

Whereas BLAS is focused on numerical linear algebra, Thrust provides an abstract interface to fundamental parallel algorithms such as scan, sort, and reduction. Thrust leverages the power of C++ templates to make these algorithms *generic*, enabling them to be used with arbitrary user-defined types and operators. Thrust establishes a durable interface for parallel computing with an eye towards generality, programmer productivity, and real-world performance.

26.2 DIVING IN

Before going into greater detail, let us consider the program in Listing 26.1, which illustrates the salient features of Thrust.

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

int main(void)
{
  // generate 16M random numbers on the host
  thrust::host_vector<int> h_vec(1 << 24);
  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;
}
```

Listing 26.1. A complete Thrust program which sorts data on the GPU.

Thrust provides two `vector` *containers*: `host_vector` and `device_vector`. As the names suggest, `host_vector` is stored in host memory while `device_vector` lives in device memory on the GPU. Like the vector container in the C++ STL, `host_vector` and `device_vector` are generic containers (i.e., they are able to store any data type) that can be resized dynamically. As the example shows, containers automate the allocation and deallocation of memory and simplify the process of exchanging data between the host and device.

The program acts on the vector containers using the `generate`, `sort`, and `copy` algorithms. Here, we adopt the STL convention of specifying *ranges* using pairs of *iterators*. In this example, the iterators `h_vec.begin()` and `h_vec.end()` can be thought of as a pair of `int` pointers, where the former points to the first element in the array and the latter to the element one past the end of the array. Together the pair defines a *range* of integers of size `h_vec.end() - h_vec.begin()`.

Note that even though the computation implied by the call to the `sort` algorithm suggests one or more CUDA kernel launches, the programmer has not specified a launch configuration. Thrust's interface *abstracts* these details. The choice of performance-sensitive variables such as grid and block size, the details of memory management, and even the choice of sorting algorithm are left to the discretion of the library implementor.

26.2.1 Iterators and Memory Spaces

Although vector iterators are similar to pointers, they carry additional information. Notice that we did not have to instruct the `sort` algorithm that it was operating on the elements of a `device_vector` or hint that the `copy` was from device memory to host memory. In Thrust the memory spaces of each range are automatically *inferred* from the iterator arguments and used to dispatch the appropriate implementation.

In addition to memory space, Thrust's iterators implicitly encode a wealth of information which can guide the dispatch process. For instance, our `sort` example above operates on `int`s, a primitive data type with a fundamental comparison operation. In this case, Thrust dispatches a highly-tuned Radix Sort algorithm [2] which is considerably faster than alternative comparison-based sorting algorithms such as Merge Sort [3]. It is important to realize that this dispatch process incurs no performance or storage overhead: metadata encoded by iterators exists only at compile time, and dispatch strategies based on it are selected *statically*. In general, Thrust's static dispatch strategies may capitalize on any information that is derivable from the type of an iterator.

26.2.2 Interoperability

Thrust is implemented entirely within CUDA C/C++ and maintains interoperability with the rest of the CUDA ecosystem. Interoperability is an important feature because no single language or library is the best tool for every problem. For example, although Thrust algorithms use CUDA features like `__shared__` memory internally, there is no mechanism for users to exploit `__shared__` memory directly through Thrust. Therefore, it is sometimes necessary for applications to access CUDA C directly to implement a certain class of specialized algorithms, as illustrated in the software stack of Figure 26.1.

Interfacing Thrust to CUDA C is straightforward and analogous to the use of the C++ STL with standard C code. Data that resides in a Thrust container can be accessed by external libraries by

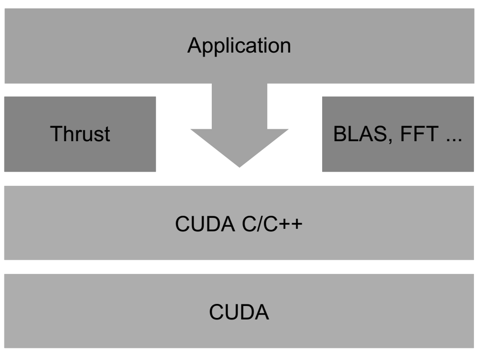

FIGURE 26.1

Thrust is an abstraction layer on top of CUDA C/C++.

```
size_t N = 1024;

// allocate Thrust container
device_vector<int> d_vec(N);

// extract raw pointer from container
int * raw_ptr = raw_pointer_cast(&d_vec[0]);

// use raw_ptr in non-Thrust functions
cudaMemset(raw_ptr, 0, N * sizeof(int));

// pass raw_ptr to a kernel
my_kernel<<<N / 128, 128>>>(N, raw_ptr);

// memory is automatically freed
```

(a) Interfacing Thrust to CUDA

```
size_t N = 1024;

// raw pointer to device memory
int * raw_ptr;
cudaMalloc(&raw_ptr, N * sizeof(int));

// wrap raw pointer with a device_ptr
device_ptr<int> dev_ptr = device_pointer_cast(raw_ptr);

// use device_ptr in Thrust algorithms
sort(dev_ptr, dev_ptr + N);

// access device memory through device_ptr
dev_ptr[0] = 1;

// free memory
cudaFree(raw_ptr);
```

(b) Interfacing CUDA to Thrust

Listing 26.2. Thrust interoperates smoothly with CUDA C/C++.

extracing a "raw" pointer from the vector. The code sample in Listing 26.2 illustrates the use of `raw_pointer_cast` to obtain an `int` pointer to the contents of a `device_vector`.

Applying Thrust algorithms to raw pointers is also straightforward. Once the raw pointer has been wrapped by a `device_ptr` it can be used like an ordinary Thrust iterator. The wrapped pointer provides the memory space information Thrust needs to invoke the appropriate algorithm implemention and also allows a convenient mechanism for accessing device memory from the host.

Thrust's native CUDA C interoperability is a powerful feature. Interoperability ensures that Thrust always *complements* CUDA C and that a Thrust plus CUDA C combination is never worse than either Thrust or CUDA C alone. Indeed, while it may be possible to write whole parallel applications entirely with Thrust functions, it is often valuable to implement domain-specific functionality directly in CUDA C. The level of abstraction targeted by native CUDA C affords programmers fine-grained control over the precise mapping of computational resources to a particular problem. Programming at this level provides developers the flexibility to implement exotic or otherwise specialized algorithms. Interoperability also facilitates an iterative development strategy: (1) quickly prototype a parallel application entirely in Thrust, (2) identify the application's hot spots, and (3) write more specialized algorithms in CUDA C and optimize as necessary.

26.3 GENERIC PROGRAMMING

Thrust presents a style of programming emphasizing genericity and composability. Indeed, the vast majority of Thrust's functionality is derived from four fundamental parallel algorithms: `for_each`, `reduce`, `scan`, and `sort`. For example, the `transform` algorithm is a derivative of `for_each` while `inner_product` is implemented with `reduce`.

Thrust algorithms are generic in both the type of the data to be processed and the operations to be applied to the data. For instance, the `reduce` algorithm may be employed to compute the sum of a range of integers (a `plus` reduction applied to `int` data) or the maximum of a range of floating point values (a `max` reduction applied to `float` data). This generality is implemented via C++ templates, which allows user-defined types and functions to be used in addition to built-in types such as `int` or `float` or Thrust operators such as `plus`.

Generic algorithms are extremely valuable because it is impractical to anticipate precisely which particular types and operators users will require. Indeed, while the computational structure of an algorithm is fixed, the number of *instantiations* of the algorithm is truly limitless. However, it is worth remarking that while Thrust's interface is general, the abstraction affords implementors the opportunity to specialize for specific types and operations known to be important use cases. As with inferences from memory space, these opportunities may be exploited statically.

In Thrust, user-defined operations take the form of C++ function objects, or *functors*. Functors allow the programmer to adapt a generic algorithm to implement a specific user-defined operation. For example, the code samples in Listing 26.3 implement SAXPY, the well-known BLAS operation, using CUDA C and Thrust respectively. Here, the generic `transform` algorithm is called with the user-defined `saxpy_functor`.

26.4 BENEFITS OF ABSTRACTION

In this section we'll describe the benefits of Thrust's abstraction layer with respect to programmer productivity, robustness, and real-world performance.

26.4.1 Programmer Productivity

Thrust's high-level algorithms enhance programmer productivity by automating the mapping of computational tasks onto the GPU. Recall the two implementations of SAXPY shown in Listing 26.3. In the CUDA C implementation of SAXPY the programmer has described a specific decomposition of the parallel vector operation into a grid of blocks with 256 threads per block. In contrast, the Thrust implementation does not prescribe a launch configuration. Instead, the only specifications are the input and output ranges and a functor to apply to them. Otherwise, the two codes are roughly the same in terms of length and code complexity.

Delegating the launch configuration to Thrust has a subtle yet profound implication: the launch parameters can be automatically chosen based on a model of machine performance. Currently, Thrust targets *maximal occupancy* and will compare the resource usage of the kernel (e.g., number of registers, amount of shared memory) with the resources of the target GPU to determine a launch configuration with highest occupancy. While the maximal occupancy heuristic is not necessarily optimal, it is straightforward to compute and effective in practice. Furthermore, there is nothing to preclude the use of more sophisticated performance models. For instance, a run-time tuning system that examined hardware performance counters could be introduced behind this abstraction without altering client code.

Thrust also boosts programmer productivity by providing a rich set of algorithms for common patterns. For instance, the map-reduce pattern is conveniently implemented with Thrust's `sort_by_key` and `reduce_by_key` algorithms, which implement key-value sorting and reduction respectively.

```
__global__
void saxpy_kernel(int n, float a, float * x, float * y)
{
   const int i = blockDim.x * blockIdx.x + threadIdx.x;

   if (i < n)
      y[i] = a * x[i] + y[i];
}

void saxpy(int n, float a, float * x, float * y)
{
  // set launch configuration parameters
  int block_size = 256;
  int grid_size = (n + block_size - 1) / block_size;

  // launch saxpy kernel
  saxpy_kernel<<< grid_size, block_size >>>(n, a, x, y);
}
```

(a) CUDA C

```
struct saxpy_functor
{
   const float a;

   saxpy_functor(float _a) : a(_a) {}

   __host__ __device__
   float operator()(float x, float y)
   {
      return a * x + y;
   }
};

void saxpy(float a, device_vector<float>& x, device_vector<float>& y)
{
  // setup functor
  saxpy_functor func(a);

  // call transform
  transform(x.begin(), x.end(), y.begin(), y.begin(), func);
}
```

(b) Thrust

Listing 26.3. SAXPY implementations in CUDA C and Thrust.

26.4.2 Robustness

Thrust's abstraction layer also enhances the robustness of CUDA applications. In the previous section we noted that by delegating the launch configuration details to Thrust we could automatically obtain maximum occupancy during execution. In addition to maximizing occupancy, the abstraction layer also ensures that algorithms "just work," even in uncommon or pathological use cases. For instance, Thrust automatically handles limits on grid dimensions (no more than 64K), works around limitations on the size of `__global__` function arguments, and accommodates large user-defined types in most algorithms. To the degree possible, Thrust circumvents such factors and ensures correct program execution across the full spectrum of CUDA-capable GPUs.

26.4.3 Real-World Performance

In addition to enhancing programmer productivity and improving robustness, the high-level abstractions provided by Thrust improve performance in real-world use cases. In this section we examine two instances where the discretion afforded by Thrust's high-level interface is exploited for meaningful performance gains.

To begin, consider the operation of filling an array with a particular value. In Thrust, this is implemented with the `fill` algorithm. Unfortunately, a straightforward implementation of this seemingly simple operation is subject to severe performance hazards. Recall that processors based on the G80 architecture (i.e., Compute Capability 1.0 and 1.1) impose strict conditions on which memory access patterns may benefit from memory coalescing [4]. In particular, memory accesses of sub-word granularity (i.e., less than four bytes) are not coalesced by these processors. This artifact is detrimental to performance when initializing arrays of `char` or `short` types.

Fortunately, the iterators passed to `fill` implicitly encode all the information necessary to intercept this case and substitute an optimized implementation. Specifically, when `fill` is dispatched for smaller types, Thrust selects a "wide" version of the algorithm that issues word-sized accesses per thread. While this optimization is straightforward to implement, users are unlikely to invest the effort of making this optimization themselves. Nevertheless, the benefit, shown in Table 26.1, is worthwhile, particularly on earlier architectures.

Like `fill`, Thrust's sorting functionality exploits the discretion afforded by the abstract `sort` and `stable_sort` functions. As long as the algorithm achieves the promised result, we are free to utilize

Table 26.1 Memory Bandwidth of Two `fill` Kernels

GPU	data type	naive `fill`	`thrust::fill`	Speedup
GeForce 8800 GTS	`char`	1.2 GB/s	41.2 GB/s	34.15x
	`short`	2.4 GB/s	41.2 GB/s	17.35x
	`int`	41.2 GB/s	41.2 GB/s	1.00x
	`long`	40.7 GB/s	40.7 GB/s	1.00x
GeForce GTX 280	`char`	33.9 GB/s	75.0 GB/s	2.21x
	`short`	51.6 GB/s	75.0 GB/s	1.45x
	`int`	75.0 GB/s	75.0 GB/s	1.00x
	`long`	69.2 GB/s	69.2 GB/s	1.00x
GeForce GTX 480	`char`	74.1 GB/s	156.9 GB/s	2.12x
	`short`	136.6 GB/s	156.9 GB/s	1.15x
	`int`	146.1 GB/s	156.9 GB/s	1.07x
	`long`	156.9 GB/s	156.9 GB/s	1.00x

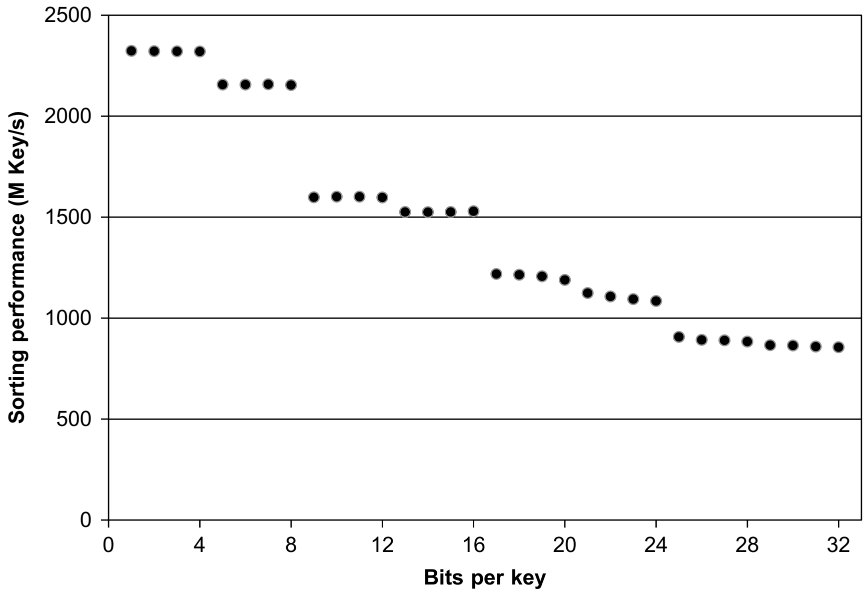

FIGURE 26.2

Sorting 32-bit integers on the GeForce GTX 480: Thrust's dynamic sorting optimizations improve performance by a considerable margin in common use cases.

sophisticated static (compile-time) and dynamic (run-time) optimizations to implement the sorting operation in the most efficient manner.

As mentioned in Section 26.2.1, Thrust statically selects a highly-optimized Radix Sort algorithm [2] for sorting primitive types (e.g., `char`, `int`, `float`, and `double`) with the standard `less` comparison operator. For all other types (e.g., user-defined data types) and comparison operators, Thrust uses a general Merge Sort algorithm. Because sorting primitives with Radix Sort is considerably faster than Merge Sort, this static optimization has significant value.

Thrust also applies dynamic optimizations to improve sorting performance. Before invoking the Radix Sort, Thrust quickly computes the minimum and maximum among the keys to be sorted. Since the cost of Radix Sort is proportional to the number of significant key bits, we can exploit knowledge of the extremal values to reduce the cost of sorting. For instance, when all integer keys are in the range [0, 16), only four bits must be sorted, and we observe a 2.71× speedup versus a full 32-bit sort. The relationship between key bits and radix sort performance is plotted in Figure 26.2.

26.5 BEST PRACTICES

In this section we highlight three high-level optimization techniques that programmers may employ to yield significant performance speedups when using Thrust.

26.5.1 Fusion

The balance of computational resources on modern GPUs implies that algorithms are often *bandwidth limited*. Specifically, computations with low *arithmetic intensity*, the ratio of calculations per memory access, are constrained by the available memory bandwidth and do not fully utilize the computational resources of the GPU. One technique for increasing the computational intensity of an algorithm is to *fuse* multiple pipeline stages together into a single operation. In this section we demonstrate how Thrust enables developers to exploit opportunities for kernel fusion and better utilize GPU memory bandwidth.

The simplest form of kernel fusion is scalar function composition. For example, suppose we have the functions $f(x) \to y$ and $g(y) \to z$ and would like to compute $g(f(x)) \to z$ for a range of scalar values. The most straightforward approach is to read x from memory, compute the value $y = f(x)$, and then write y to memory, and then do the same to compute $z = g(y)$. In Thrust this approach would be implemented with two separate calls to the `transform` algorithm, one for f and one for g. While this approach is straightforward to understand and implement, it needlessly wastes memory bandwidth, which is a scarce resource.

A better approach is to fuse the functions into a single operation $g(f(x))$ and halve the number of memory transactions. Unless f and g are computationally expensive operations, the fused implementation will run approximately twice as fast as the first approach. In general, scalar function composition is a profitable optimization and should be applied liberally.

Thrust enables developers to exploit other, less-obvious opportunities for fusion. For example, consider the following two Thrust implementations of the BLAS function SNRM2 shown in Listing 26.4, which computes the Euclidean norm of a `float` vector.

Note that SNRM2 has low arithmetic intensity: each element of the vector participates in only two floating point operations, one multiply (to square the value) and one addition (to sum values together). Therefore, SNRM2 is an ideal candidate for fusion and the `transform_reduce` implementation, which fuses the `square` transformation with a `plus` reduction should be considerably faster. Indeed this is true and `snrm2_fast` is fully 3.8 times faster than `snrm2_slow` for a 16M element vector on a Tesla C1060.

While the previous examples represent some of the more common opportunities for fusion, we have only scratched the surface. As we have seen, fusing a transformation with other algorithms is a worthwhile optimization. However, Thrust would become unwieldy if all algorithms came with a `transform_` variant. For this reason Thrust provides `transform_iterator` which allows transformations to be fused with any algorithm. Indeed, `transform_reduce` is simply a convenience wrapper for the appropriate combination of `transform_iterator` and `reduce`. Similarly, Thrust provides `permutation_iterator` which enables `gather` and `scatter` operations to be fused with other algorithms.

26.5.2 Structure of Arrays

In the previous section we examined how fusion minimizes the number of off-chip memory transactions and conserves bandwidth. Another way to improve memory efficiency is to ensure that all memory accesses benefit from *coalescing*, since coalesced memory access patterns are considerably faster than non-coalesced transactions.

```
struct square
{
  __host__ __device__
  float operator()(float x) const
  {
     return x * x;
  }
};


float snrm2_slow(const thrust::device_vector<float>& x)
{
  // without fusion
  device_vector<float> temp(x.size());
  transform(x.begin(), x.end(), temp.begin(), square());

  return sqrt( reduce(temp.begin(), temp.end()) );
}

float snrm2_fast(const thrust::device_vector<float>& x)
{
  // with fusion
  return sqrt( transform_reduce(x.begin(), x.end(), square(), 0.0f, plus<float>());
}
```

Listing 26.4. SNRM2 has low arithmetic intensity and therefore benefits greatly from fusion.

```
struct float3
{
  float x;
  float y;
  float z;
};

float3 * aos;
...

aos[0].x = 1.0f;
```

(a) Array of Structures

```
struct float3_soa
{
  float * x;
  float * y;
  float * z;
};

float3_soa soa;
...

soa.x[0] = 1.0f;
```

(b) Structure of Arrays

Listing 26.5. Data layouts for three-dimensional `float` vectors.

Perhaps the most common violation of the memory coalescing rules arises when using a so-called Array of Structures (AoS) data layout. Generally speaking, access to the elements of an array filled with C `struct` or C++ `class` variables will be uncoalesced. Only special structures such as `uint2` or `float4` satisfy the memory coalescing rules.

An alternative to the AoS layout is the Structure of Arrays (SoA) approach, where the components of each struct are stored in separate arrays. Listing 26.5 illustrates the AoS and SoA methods of representing a range of three-dimensional `float` vectors. The advantage of the SoA method is that regular

```
struct rotate_tuple
{
  __host__ __device__
  tuple<float,float,float> operator()(tuple<float,float,float>& t)
  {
    float x = get<0>(t);
    float y = get<1>(t);
    float z = get<2>(t);

    float rx = 0.36f * x + 0.48f * y + -0.80f * z;
    float ry =-0.80f * x + 0.60f * y + 0.00f * z;
    float rz = 0.48f * x + 0.64f * y + 0.60f * z;

    return make_tuple(rx, ry, rz);
  }
};

...

device_vector<float> x(N), y(N), z(N);

transform(make_zip_iterator(make_tuple(x.begin(), y.begin(), z.begin())),
          make_zip_iterator(make_tuple(x.end(), y.end(), z.end())),
          make_zip_iterator(make_tuple(x.begin(), y.begin(), z.begin())),
          rotate_tuple());
```

Listing 26.6. The `zip_iterator` facilitates processing of data in structure of arrays format.

access to the `x`, `y`, and `z` components of a given vector is coalesceable (because `float` satisfies the coalescing rules), while regular access to the `float3` structures in the AoS approach is not.

The problem with SoA is that there is nothing to logically encapsulate the members of each element into a single entity. Whereas we could immediately apply Thrust algorithms to AoS containers like `device_vector<float3>`, we have no direct means of doing the same with three separate `device_vector<float>` containers. Fortunately Thrust provides `zip_iterator`, which provides encapsulation of SoA ranges.

The `zip_iterator` [5] takes a number of iterators and *zips* them together into a virtual range of tuples. For instance, binding three `device_vector<float>` iterators together yields a range of type `tuple<float,float,float>`, which is analogous to the `float3` structure.

Consider the code sample in Listing 26.6 which uses `zip_iterator` to construct a range of three-dimensional `float` vectors stored in SoA format. Each vector is transformed by a rotation matrix in the `rotate_tuple` functor before being written out again. Note that `zip_iterator` is used for both input and output ranges, transparently packing the underlying scalar ranges into tuples and then unpacking the tuples into the scalar ranges. On a Tesla C1060, this SoA implementation is 2.85× faster than the analogous AoS implementation (not shown).

26.5.3 Implicit Ranges

In the previous sections we considered ways to efficiently transform ranges of values and ways to construct ad hoc tuples of values from separate ranges. In either case, there was some underlying data stored *explicitly* in memory. In this section we illustrate the use of *implicit* ranges, i.e., ranges whose values are defined programmatically and not stored anywhere in memory.

For instance, consider the problem of finding the index of the element with the smallest value in a given range. We could implement a special reduction kernel for this algorithm, which we'll call `min_index`, but that would be time-consuming and unnecessary. A better approach is to implement `min_index` in terms of existing functionality, such as a specialized reduction over (value, index) tuples, to achieve the desired result. Specifically, we can zip the range of values `v[0]`, `v[1]`, `v[2]`, ... together with a range of integer indices `0`, `1`, `2`, ... to form a range of tuples `(v[0], 0)`, `(v[1], 1)`, `(v[2],2)` ... and then implement `min_index` with the standard `reduce` algorithm. Unfortunately, this scheme will be much slower than a customized reduction kernel, since the index range must be created and stored explicitly in memory.

To resolve this issue Thrust provides `counting_iterator` [5], which acts just like the explicit range of values we need to implement `min_index`, but does not carry any overhead. Specifically, when `counting_iterator` is dereferenced it generates the appropriate value "on the fly" and yields that value to the caller. An efficient implementation of `min_index` using `counting_iterator` is shown in Listing 26.7.

```
struct smaller_tuple
{
  tuple<float,int> operator()(tuple<float,int> a, tuple<float,int> b)
  {
    // return the tuple with the smaller float value
    if (get<0>(a) < get<0>(b))
      return a;
    else
      return b;
  }
};

int min_index(device_vector<float>& values)
{
  // [begin,end) form the implicit sequence [0,1,2, ... value.size())
  counting_iterator<int> begin(0);
  counting_iterator<int> end(values.size());

  // initial value of the reduction
  tuple<float,int> init(values[0], 0);

  // compute the smallest tuple
  tuple<float,int> smallest = reduce(make_zip_iterator(make_tuple(values.begin(), begin)),
                                     make_zip_iterator(make_tuple(values.end(), end)),
                                     init,
                                     smaller_tuple());
  // return the index
  return get<1>(smallest);
}
```

Listing 26.7. Implicit ranges improve performance by conserving memory bandwidth.

Here `counting_iterator` has allowed us to efficiently implement a special-purpose reduction algorithm without the need to write a new, special-purpose kernel. In addition to `counting_iterator` Thrust provides `constant_iterator`, which defines an implicit range of constant value. Note that these implicitly-defined iterators can be combined with the other iterators to create more complex implicit ranges. For instance, `counting_iterator` can be used in combination with `transform_iterator` to produce a range of indices with nonunit stride.

In practice there is no need to implement `min_index` since Thrust's `min_element` algorithm provides the equivalent functionality. Nevertheless the `min_index` example is instructive of best practices. Indeed, Thrust algorithms such as `min_element`, `max_element`, and `find_if` apply the exact same strategy internally.

References

[1] J. Hoberock, N. Bell, Thrust: A parallel template library, 2011. Version 1.4.0.
[2] D. Merrill, A. Grimshaw, Revisiting sorting for gpgpu stream architectures, Technical Report CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, 2010.
[3] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore GPUs, in Proceedings 23rd IEEE Int'l Parallel & Distributed Processing Symposium, IEEE Computer Society, Washington, DC, 2009.
[4] NVIDIA Corporation, CUDA C Best Practices Guide v3.2, NVIDIA Corporation, Santa Clara, CA, 2010 (Section 3.2.1).
[5] Boost Iterator Library. www.boost.org/doc/libs/release/libs/iterator/.

Non-Print Items

Abstract

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.