Bitonic Sort for C++ AMP

Steve Dower

http://stevedower.id.au

Swinburne University of Technology

Melbourne, Australia

March 6, 2012

1 Overview

Notably missing from most GPU processing interfaces are parallel sort algorithms. While a standard feature of CPU-based libraries and languages, the typical high throughput applications that GPUs are used for preclude a general implementation. Given the extent to which sort algorithms can be tuned for particular data structures, shifting the implementation burden onto developers is understandable. However, without even a basic implementation, many developers will find it easier to simply copy their data to the CPU, use the C++ standard library's sort algorithm and copy it back. For small data sets, this can be more efficient than sorting on the GPU, but for large arrays the transfer overhead becomes more significant.

This appendix describes the implementation of the bitonic sort algorithm for C++ AMP that is used for an implementation of ESDL (see [3]). Section 2 describes the general bitonic sort algorithm. Section 3 details the use of this implementation. Section 4 covers details of how the algorithm is implemented. The entire source code for this implementation is included in Section 4.

2 Bitonic Sort

Bitonic sorting networks are based on a fixed sequence of comparisons between every element in a dataset. A brief outline is given here; [1] and [2] are recommended for full descriptions and analysis of the algorithm.

The diagram in Figure 1 shows the sequence of comparison-exchange operations for a dataset with 16 values. Each arrow represents where the larger value appears after an exchange. Note that the number of elements in the dataset must be a power of two. Figure 2 simplifies the network by rearranging the exchanges to point in the same direction—in this case, towards the bottom of the figure.

Each lightly shaded block represents a sort *pass*: for a dataset of N elements, $\log_2 N$ passes are required. Within pass i, there are i *phases*. The first phase consists of $2^{-i}N$ equally sized blocks that compare each value with one mirrored around the middle of the block. Each subsequent phase consists of twice as many blocks as the previous, but compares each value with one

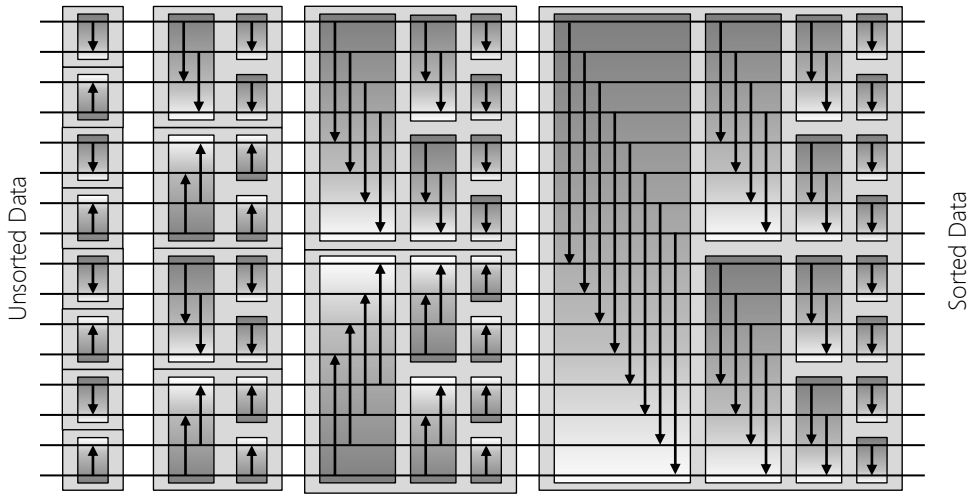

Figure 1: A bitonic sorting network for 16 values

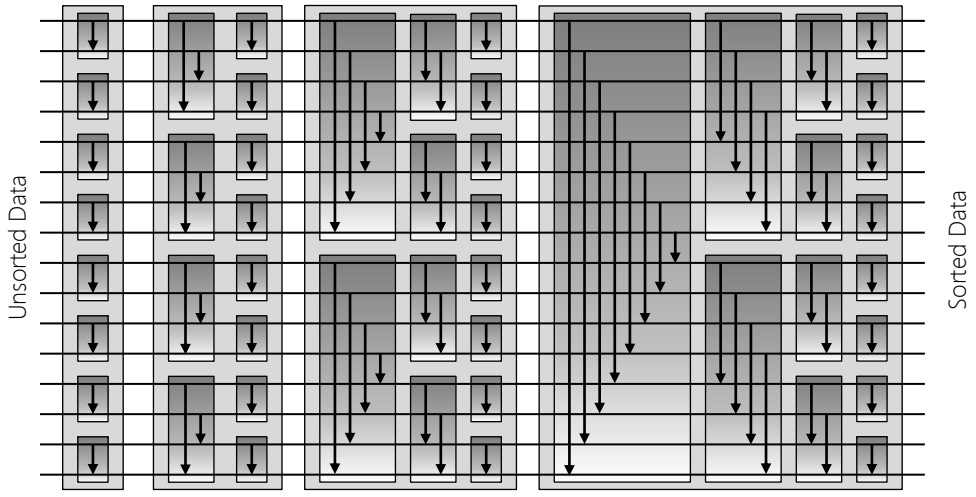

Figure 2: A simplified bitonic sorting network for 16 values

offset into the lower half rather than mirrored. This arrangement results in $\frac{1}{2}\log_2 N\,(\log_2 N + 1)$ comparisons for all cases. [1]

3 Usage

The implementation given here is contained in a single header file, `bitonic_sort.h`. It makes use of templates, in particular partial specialisation, to generate code optimised for the data type being sorted. Two functions are provided, `parallel_sort` and `parallel_sort_keys`, and a `key_index_type` structure. Signatures for these are shown in Listing 1.[1] For the simplest cases of sorting arrays of `float`, `int` or `unsigned int`, `parallel_sort` can be used as shown in Listing 2.

```
template<typename SourceType>
std::shared_ptr<concurrency::array<SourceType, 1>>
parallel_sort(const concurrency::array<SourceType, 1> &source,
              bool reverse=false);

template<typename SourceType>
std::shared_ptr<concurrency::array<typename key_index_type<SourceType>::type, 1>>
parallel_sort_keys(const concurrency::array<SourceType, 1> &source);

template<typename T>
struct key_index_type { typedef T type; };
```

Listing 1: Implied signatures of the methods provided by `bitonic_sort.h`

```
#include <amp.h>
#include <memory>
#include <bitonic_sort.h>

concurrency::array<float, 1> unsorted;
std::shared_ptr<concurrency::array<float, 1>> sorted, reverse_sorted;

// initialise unsorted with random values

sorted = bitonic_sort::parallel_sort(unsorted);
reverse_sorted = bitonic_sort::parallel_sort(unsorted, true);
```

Listing 2: Example of calling `parallel_sort` with a `float` array

[1] The signatures shown are intended to imply usage. The actual implementations have more complex overloads for optimisation purposes.

To sort types other than these, such as arrays or user-defined types, comparable structures are used that include an index into the original array. Specialising `key_index_type` allows the structure and comparison operator to be overridden for a particular source type. Listing 3 shows an example of sorting `a` user-defined type by one of its members. The `UDTKeyIndex` structure contains a sort key, the `y` member of `UDT`, an integer index and a comparison function. Any structure may be provided to properly sort any data type, with the caveat that all members must be able to run in `restrict(amp)` blocks and the warning that instances of this structure are copied often—size matters. Once a `key_index_type` has been defined for the user-defined type, arrays of that type may be sorted as for simple types.

The `parallel_sort_keys` method is used internally, but is also exposed for users who want sorted pointers into the original array but do not need the array itself reordered. Listing 4 shows an example of retrieving sorted indexes into the original array, followed by producing an array of differences between adjacent elements.

Source arrays do not need to be powers of two long and the result arrays are always the same length as the source. Arrays that are powers of two long are likely to have slightly better performance per element, but not significantly enough to justify manually padding.

```
#include <amp.h>
#include <memory>
#include <bitonic_sort.h>

struct UDT {
    float x;
    unsigned int y;
    int z;
};

struct UDTKeyIndex {
    unsigned int k;
    int i;
    UDTKeyIndex() restrict(cpu, amp) { }
    UDTKeyIndex(UDT key, int index) restrict(cpu, amp)
        : k(key.y), i(index) { }
    bool operator<(const UDTKeyIndex& other) restrict(cpu, amp) {
        return k < other.k;
    }
};

template<> struct bitonic_sort::key_index_type<UDT> {
    typedef UDTKeyIndex type;
};

concurrency::array<UDT, 1> unsorted;
std::shared_ptr<concurrency::array<UDT, 1>> sorted;

// initialise unsorted with random values

sorted = bitonic_sort::parallel_sort(unsorted);
```

Listing 3: Example of sorting an array of a user-defined type

```
std::shared_ptr<concurrency::array<UDTKeyIndex, 1>> sortKeys;
sortKeys = bitonic_sort::parallel_sort_keys(unsorted);

auto& sortKeyRef = *sortKeys;
concurrency::array<float, 1> result(sortKeyRef.extent.size());
concurrency::parallel_for_each(result.extent,
    [&](concurrency::index<1> i) restrict(amp) {
        int index = sortKeyRef[i].i;
        if (index + 1 >= unsorted.extent.size()) {
            result[i] = 0.0f;
        } else {
            result[i] = unsorted[index + 1].x - unsorted[index].x;
        }
    }
);
```

Listing 4: Example of using `parallel_sort_keys` with the user-defined type from Listing 3

4 Implementation

Most of the implementation is in the `bitonic_sort::details::parallel_sort_phase` function. Of the public facing functions, `parallel_sort` obtains the sorted keys from `parallel_sort_keys` and copies the source array into order, while `parallel_sort_keys` simply creates an array of sort keys and invokes `parallel_sort_inplace`. This method is internal and simply calls `parallel_sort_phase` the correct number of times. Listing 5 shows the section of `parallel_sort_inplace` responsible for calling `parallel_sort_phase`.

In Listing 5, `keys` is the list of key-index pairs (an instance of `concurrency::array<key_index_type<SourceType>::type, 1>`), `virtual_size` is the length of `keys` rounded up to the next power of two and `phase_count` is the base-2 log of `virtual_size`. For example, if the source set has 15 values, keys also has 15 values, `virtual_size` is 16 and `phase_count` is 4. The `parallel_sort_phase` function is templated over a Boolean (as well as the key type) that indicates the first call of each pass. Listing 6 shows the complete implementation of `parallel_sort_phase`.

```
int starting_block_size = 2;
while (--phase_count >= 0) {
  int current_block_size = starting_block_size;
  starting_block_size *= 2;

  details::parallel_sort_phase<true>(virt_size, current_block_size, dest);

  while ((current_block_size /= 2) >= 2) {
    details::parallel_sort_phase<false>(virt_size, current_block_size, dest);
  }
}
```

Listing 5: The section of `parallel_sort_inplace` that invokes `parallel_sort_phase`

```
1  template<bool First, typename KeyType>
2  void parallel_sort_phase(int virtual_size, int block_size,
3                           concurrency::array<KeyType, 1>& dest) {
4    const int actual_size = dest.extent.size();
5    const int half_block_size = block_size / 2;
6    const int mask = block_size - 1;
7    const int half_mask = half_block_size - 1;
8    concurrency::parallel_for_each(dest.accelerator_view,
9      concurrency::extent<1>(virtual_size / 2),
10     [=, &dest](concurrency::index<1> i) restrict(amp) {
11     const int i1 = ((i[0] & ~half_mask) << 1) | (i[0] & half_mask);
12     const int i2 = (First) ? ((i1 | mask) - (i1 & mask)) : (i1 + half_block_size);
13
14     if (i2 < actual_size && !(dest[i1] < dest[i2])) {
15       auto temp = dest[i1];
16       dest[i1] = dest[i2];
17       dest[i2] = temp;
18     }
19   });
20 }
```

Listing 6: Complete implementation of `parallel_sort_phase`

Line 9 in Listing 6 uses `virtual_size` to determine the number of kernels to invoke, while line 14 uses the actual size to avoid accessing elements that don't exist—the equivalent of padding the source data with elements that always sort last. For data sets that are slightly larger than a power of two, this avoids having to allocate almost double the memory required, while allowing the sort to operate efficiently. However, since kernels are still scheduled for non-existent elements, overall performance is determined by `virtual_size` rather than the actual

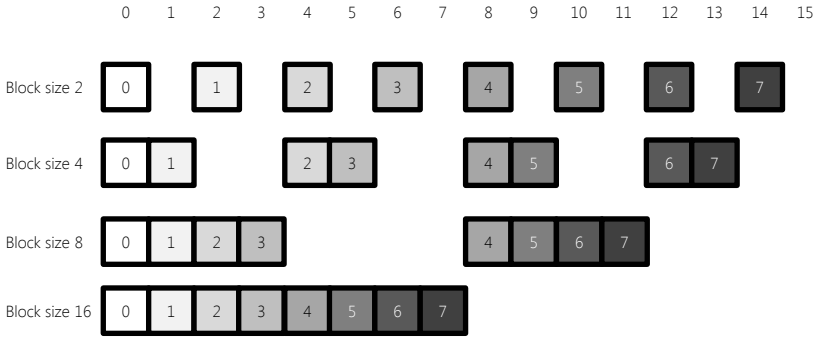

Figure 3: Values of `i1` based on block size

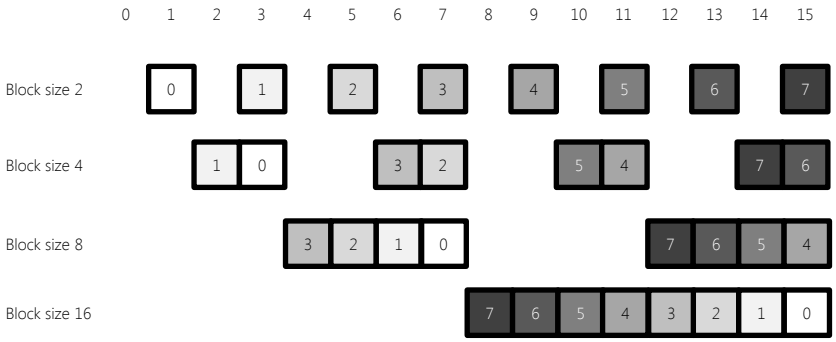

Figure 4: Values of `i2` based on block size during the first phase

size; arrays of 257 elements and 511 elements, which both have a `virtual_size` of 512, take similar amounts of time.

Lines 11 and 12 select the element indexes to compare based on the index `i`. Figures 3–5 show mappings from `i` to `i1` and `i2` in all phases. The values inside the boxes are the iteration index, from zero to half of `virtual_size`, while the actual index (horizontal position, and labelled across the top of each figure) is the value obtained. This scheme provides the set of comparisons that are shown in Figure 2 and is achieved with efficient arithmetic and bitwise operators.

Once `i1` and `i2` are known, the keys at those indexes are compared and exchanged if necessary. Because of the SIMD nature of the GPU, every kernel executes the same instructions; from a timing point-of-view, the exchange is always performed. As a result, it is important that the exchange be efficient. This is achieved primarily by using the `key_index_type` structure and sorting indexes into the source. The sort key is included to avoid indirection while sorting; DirectCompute supports random access reads but the hardware typically prefers to avoid them.

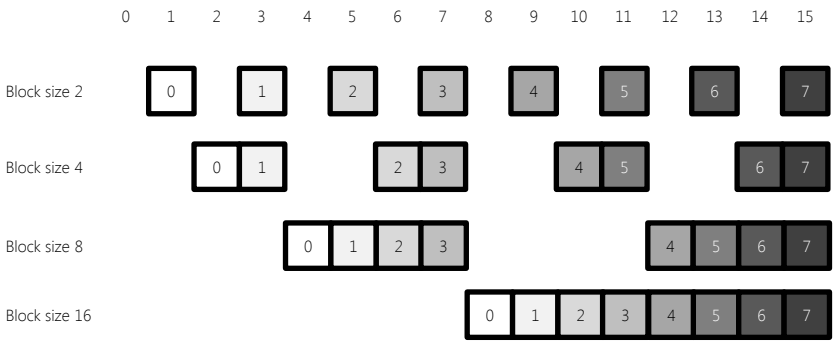

Figure 5: Values of `i2` based on block size during later phases

Unless a specialisation of `key_index_type` is provided (as shown in Listing 3), the values are sorted directly. Generic programming is used to identify cases where `key_index_type` returns the (internal) `SimpleKeyIndex` type and avoids the overhead of creating a keys array. However, for user-defined types that either cannot be compared directly or are too large to exchange often, providing a `key_index_type` specialisation can significantly improve performance. As long as the type returned from `key_index_type` has an index `i`, the two constructors in Listing 3 and a less-than comparison (`operator<()`), it can be used by `parallel_sort`.

5 Listing

Note: Listing 7 is written against the version of C++ AMP included with Visual Studio 11 Beta. Modifications may be required to compile and execute against the final release of Visual Studio 11.

```
#pragma once

#include <amp.h>
#include <memory>
#include <type_traits>

namespace bitonic_sort
{
  namespace details
  {
    template<typename SimpleType>
    struct SimpleKeyIndex {
      SimpleType k;
      int i;
      SimpleKeyIndex() restrict(cpu, amp) { }
      SimpleKeyIndex(SimpleType key, int index) restrict(cpu, amp)
        : k(key), i(index) { }
      bool operator<(const SimpleKeyIndex& other) restrict(cpu, amp) {
        return k < other.k;
      }
    };
  }

  template<typename T>
  struct key_index_type {
    typedef details::SimpleKeyIndex<T> type;
  };

  namespace details
  {
    template<typename T>
    struct uses_sort_key {
      static const bool value = !std::is_same<
        typename key_index_type<T>::type, SimpleKeyIndex<T>
      >::value;
    };

    template<bool First, typename KeyType>
    void parallel_sort_phase(int virtual_size, int block_size,
                concurrency::array<KeyType, 1>& dest) {
      const int actual_size = dest.extent.size();
      const int half_block_size = block_size / 2;
      const int mask = block_size - 1;
      const int half_mask = half_block_size - 1;
      concurrency::parallel_for_each(dest.accelerator_view,
        concurrency::extent<1>(virtual_size / 2),
        [=, &dest](concurrency::index<1> i) restrict(amp) {
        const int i1 = ((i[0] & ~half_mask) << 1) | (i[0] & half_mask);
        const int i2 = (First) ? ((i1 | mask) - (i1 & mask)) : (i1 + half_block_size);
```

```
      if (i2 < actual_size && !(dest[i1] < dest[i2])) {
        auto temp = dest[i1];
        dest[i1] = dest[i2];
        dest[i2] = temp;
      }
    });
  }

  template<typename KeyType>
  void parallel_sort_inplace(concurrency::array<KeyType, 1>& dest) {
    const int actual_size = dest.extent.size();
    int virt_size = 1, phase_count = 0;
    while (virt_size < actual_size) {
      virt_size *= 2;
      phase_count += 1;
    }

    int starting_block_size = 2;
    while (--phase_count >= 0) {
      int current_block_size = starting_block_size;
      starting_block_size *= 2;

      details::parallel_sort_phase<true>(virt_size, current_block_size, dest);

      while ((current_block_size /= 2) >= 2) {
        details::parallel_sort_phase<false>(virt_size, current_block_size, dest);
      }
    }
  }
}

template<typename SourceType>
std::shared_ptr<concurrency::array<typename key_index_type<SourceType>::type, 1>>
parallel_sort_keys(const concurrency::array<SourceType, 1>& source) {
  const int actual_size = source.extent.size();

  typedef typename key_index_type<SourceType>::type Key;
  auto pKeys = std::make_shared<concurrency::array<Key, 1>>(
    actual_size,
    source.accelerator_view
  );
  auto& keys = *pKeys;
  concurrency::parallel_for_each(keys.accelerator_view, keys.extent,
    [=, &keys, &source](concurrency::index<1> i) restrict(amp) {
    keys[i] = Key(source[i], i[0]);
  });

  details::parallel_sort_inplace(keys);

  return pKeys;
}

template<typename SourceType>
typename std::enable_if<
  details::uses_sort_key<SourceType>::value,
  std::shared_ptr<concurrency::array<SourceType, 1>>
>::type
```

```
  parallel_sort(const concurrency::array<SourceType, 1>& source, bool reverse=false) {
    auto pKeys = parallel_sort_keys(source);
    auto& keys = *pKeys;

    auto pResult = std::make_shared<concurrency::array<SourceType, 1>>(
      source.extent.size(),
      source.accelerator_view);
    auto& result = *pResult;

    if (!reverse) {
      concurrency::parallel_for_each(result.accelerator_view, result.extent,
        [&](concurrency::index<1> i) restrict(amp) {
        result[i] = source[keys[i].i];
      });
    } else {
      concurrency::parallel_for_each(result.accelerator_view, result.extent,
        [&](concurrency::index<1> i) restrict(amp) {
        result[i] = source[keys[keys.extent.size() - 1 - i].i];
      });
    }
    return pResult;
  }

  template<typename SourceType>
  typename std::enable_if<
    !details::uses_sort_key<SourceType>::value,
    std::shared_ptr<concurrency::array<SourceType, 1>>
  >::type
  parallel_sort(const concurrency::array<SourceType, 1>& source, bool reverse=false) {
    auto pResult = std::make_shared<concurrency::array<SourceType, 1>>(source);
    auto& result = *pResult;

    details::parallel_sort_inplace(result);

    if (reverse) {
      auto pNewResult = std::make_shared<concurrency::array<SourceType, 1>>(
          result.extent.size(),
          result.accelerator_view);
      auto& newResult = *pNewResult;
      concurrency::parallel_for_each(result.accelerator_view, result.extent,
        [&](concurrency::index<1> i) restrict(amp) {
        newResult[i] = result[result.extent.size() - 1 - i];
      });

      return pNewResult;
    }

    return pResult;
  }
}
```

Listing 7: C++ and C++ AMP code for `bitonic_sort.h`

References

[1] K. E. Batcher, "Sorting networks and their applications," in *AFIPS '68 (Spring) Proceedings of the April 30–May 2, 1968, spring joint computer conference.* ACM, New York, United States, 1968, pp. 307–314.

[2] G. Bilardi and A. Nicolau, "Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines," Cornell University, Tech. Rep., 1986.

[3] S. Dower, "ESDL and an abstraction for evolutionary algorithms," Ph.D. dissertation, Swinburne University of Technology, unpublished.

[4] Microsoft Corp., "C++ AMP : Language and programming model," January 2012, http://blogs.msdn.com/b/nativeconcurrency/archive/2012/02/03/c-amp-open-spec-published.aspx.