Spring 2011

Sorting on CUDA

Ayushi Sinha
Providence College

Department of Computer Science & Engineering

Research Experience for Undergraduates 2010

Sorting on CUDA

Author:
Ayushi Sinha

Supervisor:
Prof. Kunal Agrawal

August 20, 2010

1 Introduction

Sorting, the process of arranging items in a certain order, is one of the most researched topics in the field of computer science. With increase in the amount of data that can be stored on computers, increase in the efficiency of sorting algorithms has become increasingly important. Newer algorithms have been deviced in order to sort data faster. However, even the best sequential sorting algorithms will take a large amount of time to sort a large enough data set. The need for something faster than sequential sorting algorithms led to research in the field of parallel computing. Several parallel sorting algorithms are now being deviced and perfected in order to further optimize sorting.

While researching at Washington University in St. Louis, I implemented parallel sorting algorithms for quicksort, bitonic sort and radix sort in CUDA. The algorithms were tested on five data sets with different distribution patterns: normal distribution, gaussian distribution, bucket distribution, sorted distribution and zero distribution. All inputs are 32-bit integers. The time taken by the kernel to complete the execution of the parallel sorting algorithms is recorded and compared.

2 Algorithms

2.1 Quicksort

Quicksort is one of the two comparison sorts I implemented. Since CUDA 3.0 does not support recursion, I had to implement a non-recursive algorithm which made the sort less efficient. This algorithm calls the kernel function only once to sort one data set.

- Procedure

In the kernel function, I begin by declaring two arrays *start* and *end*. The first index of *start* is set to 0 while that of *end* is set to $N - 1$, where **N** is the size of the data set. Furthur code is executed under the condition that the thread index, idx, is greater than or equal to zero. If the condition is met, variables L and R are set to $start[idx]$ and $end[idx]$, respectively. L stores the leftmost required index of the data set.Therefore, to begin with, it is set to zero. Similarly, R stores the rightmost required index of the data set. Therfore, to begin with, it is set to $N - 1$.

If L is less than R, then the pivot value is set to the value stored at location L:

```
pivot = array[L]
```

array is the data set that contains the numbers to be sorted. The numbers in this data set are then compared to the pivot. We start from the right and decrement R till the first values less than the pivot is encountered. Once this value is found, it is copied to position L in the data set. Now, we compare from the left by incrementing L until the first value greater than the pivot is encountered. This value is copied to position R in the data set. This procedure is repeated as long as L is less than R. When L and R become equal, we copy the pivot to this position in the data set. Then, the next index in the arrays *start* and *end* are set, and the current index of *end* is updated.

```
values[L] = pivot;
start[idx + 1] = L + 1;
end[idx + 1] = end[idx];
end[idx++] = L;
```

Further, if the condition:

if $(end[idx] - start[idx] > end[idx-1] - start[idx-1])$

is true, then $start[idx]$ is swapped with $start[idx-1]$, and $end[idx]$ is swapped with $end[idx-1]$. The values in the arrays *start* and *end* are used to update L and R respectively when the code in the loop is repeated.

If L is not less than R, however, the thread index, idx, is decremented until L is set to a value less than R. When idx becomes less than zero, the data set is sorted and the kernel function terminates.

- Problems

As mentioned before, this algorithm is not recursive since CUDA 3.0 does not support recursion. Therefore, the algorithm is considerably slow.

- Results

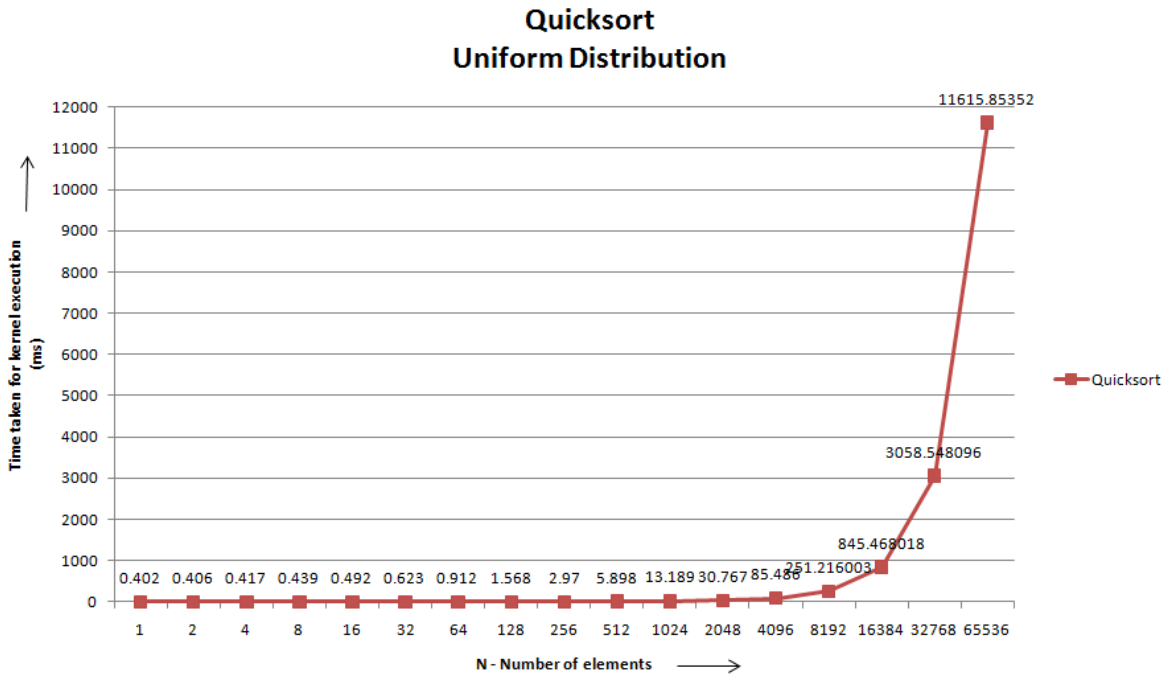
  With 512 numbers, these are the average results:

Data set distribution	Average kernel execution time
Uniform	05.2728000ms
Gaussian	16.1874002ms
Bucket	05.4554000ms
Sorted	68.8643996ms
Zero	67.8875992ms

The worst case for quicksort is evidently when the data set is sorted or has zero distribution. Although my algorithm can sort bigger data sets, it gets extremely slow as the data set gets bigger, especially in sorting sorted data sets or data sets with zero distribution. For 10,000 numbers, these are the average results:

Data set distribution	Average kernel execution time
Uniform	0356.177663ms
Gaussian	5043.411133ms
Bucket	0359.160003ms
Sorted	25606.53125ms
Zero	25216.32161ms

With 10,000 numbers, even though data sets with uniform and bucket distribution are sorted within a second, it takes over 25 seconds to sort sorted data sets and data sets with zero distribution.



2.2 Bitonic Sort

Bitonic sort is also a comparison sort. However, since this sorting technique is designed specially for parallel machines, it is very efficient. A data set with $N = 2^k$ numbers can be sorted by repeating the sorting procedure k times. In each repetition i, where $1 \leq i \leq k$,

$$\frac{N}{2^i}$$

sorted subsequences of length 2^i are pairwise merged. The sorted subsequences alternate between monotonically increasing and monotonically decreasing. Two such consecutive subsequences form a bitonic sequence. Therefore, each repetition gives rise to

$$\frac{N}{2^{i+1}}$$

bitonic subsequences of length 2^{i+1}. At the end of the last repetition, we are left with a monotonically increasing sorted sequence.

- Procedure

In my algorithm, the kernel function is invoked through a nested loop. The outer loop repeats the function call k times as required for the sorting to be successful. The loop runs from $k = 2$ to $\mathbf{N}$, incrementing by the power of 2, where $\mathbf{N}$ is the size of the data set. The inner loop runs from $j = k/2$ to 1, decrementing by the power of two. Therefore, k and j are always powers of two.

for (int $k = 2$; $k \leq N$; $k \ll= 1$) {
    for (int $j = k \gg 1$; $j > 0$; $j \gg= 1$) {
        ⋮
        Kernel_function <<< ... >>> (...);
        ⋮
    }
}

In the kernel function, each thread index, idx, is XOR-ed with j and stored in ixj. This ensures that ixj and idx are in the same subsequence in order to avoid redundant comparisons. Further execution of the code is restricted by the condition:

if ($ixj > idx$) {
    ⋮
}

This ensures that values at the two indices, ixj and idx, are compared only once - when ixj is greater than idx. If this condition is true, then the function checks whether the indices belong to the monotonically increasing subsequence of the bitonic subsequence, or the monotonically decreasing subsequence. If $idx \wedge k$[1] equals zero, the indices belong to the monotonically increasing subsequence of the bitonic subsequence. If it does not equal zero, then the indices belong to the monotonically decreasing subsequence of the bitonic subsequence. If $idx \wedge k$ equals zero and the values in the subsequence are not in monotonically increasing order, then the values are swapped. Similarly, if $idx \wedge k$ does not equal zero and the values in the subsequence are not monotonically decreasing, the values are swapped. Let $array$ contain the values to be sorted:

[1]Logical AND

```
if (ixj > idx) {
    if (idx & k == 0 && array[idx] > array[ixj])
        swap(idx, ixj);
    if (idx & k ≠ 0 && array[idx] < array[ixj])
        swap(idx, ixj);
}
```

At the end of the last repetition, we have only half a bitonic sequence. In other words, we are left with one monotonically increasing or sorted sequence.

- Problems

One limitation of this algorithm is that the size of the data set to be sorted must be a power of 2. That is, **N** must equal 2^k, for some non-negative k. Also, my algorithm can currently only sort data sets of sizes up to 2^{22}.

- Results

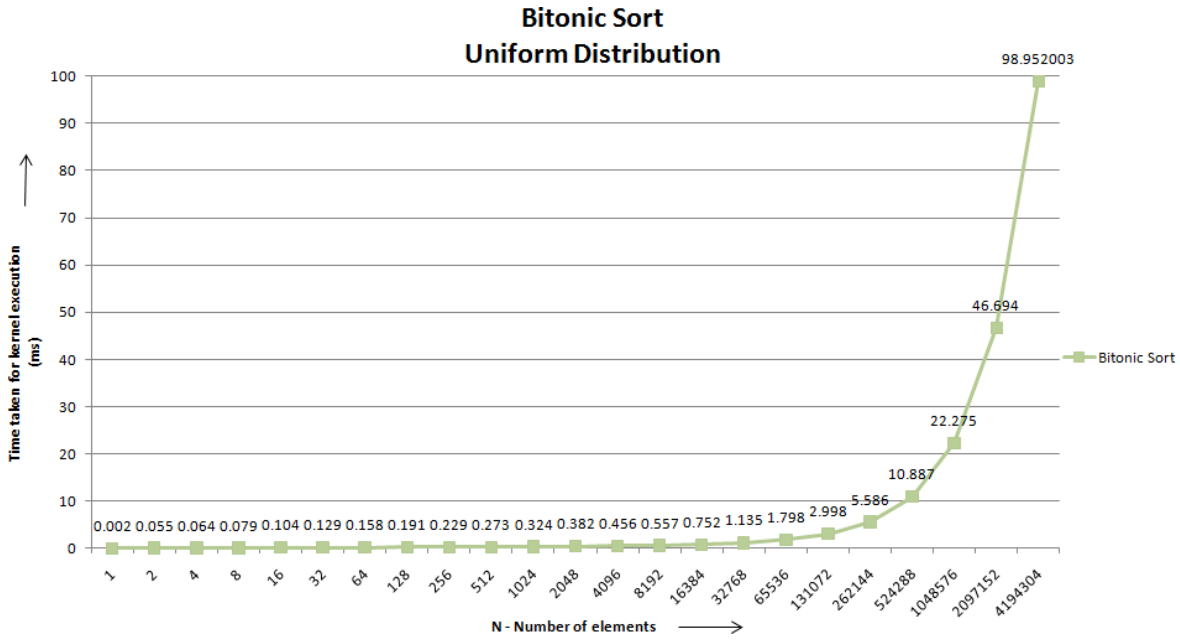
With 512 numbers, these are the average results:

Data set distribution	Average kernel execution time
Uniform	0.2730ms
Gaussian	0.2460ms
Bucket	0.2356ms
Sorted	0.2384ms
Zero	0.2326ms

The size of the largest data set that the code can currently work with is 4,194,304. The average results for this data set are:

Data set distribution	Average kernel execution time
Uniform	98.9497984ms
Gaussian	86.9412000ms
Bucket	98.1599992ms
Sorted	76.0807982ms
Zero	72.7174012ms

This bitonic sort algorithm can therefore sort over 4 million numbers in less than a second.



2.3 Radix Sort

Radix sort is different from quicksort and bitonic sort in that it is a non-comparison sort. It sorts integers by processing individual digits and clubbing together identical digits sharing the same significant position.

- Procedure

In my version of the algorithm, I converted each input from decimal to binary so that instead of having to track ten different digits, 0 to 9, only two digits, 0 and 1, would have to be tracked. After extracting the required digits from all numbers in the data set into an array, say $array1$, I set a 1 for all false sort keys (or 0s) and a 0 for all true sort keys (1s), and stored these in $array2$. Then I counted all the 1s in $array2$ by incrementing a counter everytime a 1 was encountered and stored the counter at

each step in *array*3. *array*3 now contains the destination address for each input that produced a false key. The total number of false keys for **N** inputs can be calculated as follows:

```
totalFalses = array2[N-1] + array3[N-1]
```

Then the destination address for each input that produced a true key is calculated and stored in *array*4. For each thread index, *idx*, the address can be calculated as:

```
array4[idx] = idx - array3[idx] + totalFalses
```

Finally, all destination addresses are stored in an array and the input is rearranged accordingly. The destination address pattern is a perfect permutation of the indices of the input array. Therefore, there are no write conflicts. These steps are repeated for each bit in a binary sequence. For instance, in my current code, I repeat the above steps 8 times since my data set contains numbers that can be represented by 8 bits.

- Problems

  The only kernel configuration that the code seems to work with is:

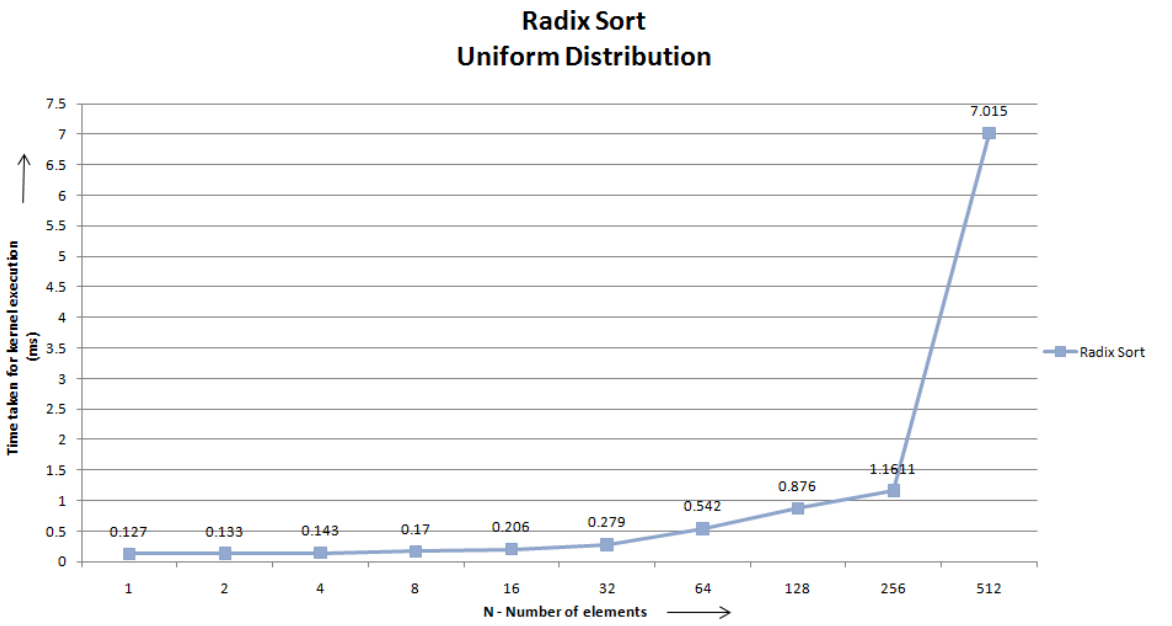
  ```
  <<< 1, N >>>
  ```

  The sort failed with other configurations I tried. With this configuration however, I cannot run the code for a data set larger than 512 numbers.

- Results

  With 512 numbers, these are the average results:

Data set distribution	Average kernel execution time
Uniform	7.0320ms
Gaussian	7.0000ms
Bucket	6.9716ms
Sorted	6.9570ms
Zero	6.9386ms



3 Conclusion

Parallel sorting algorithms are much faster than sequential sorting algorithms. As more research in the field of parallel computing produces more efficient parallel sorting algorithms, sorting large data sets will get easier. Among the algorithms I explored, bitonic sort is the most efficient.