kalyan 140001011
Amarnath Reddy 140001019

Optimized Bitonic Sort on Graphic Processor

Introduction

Sorting is a popular Computer Science topic which receives much attention. Many sequential sorts take O(NlogN) time to sort N keys. Bitonic sort is based on repeatedly merging two bitonic sequences to form a larger bitonic sequence and takes only $O(N\log^2 N)$ time steps.

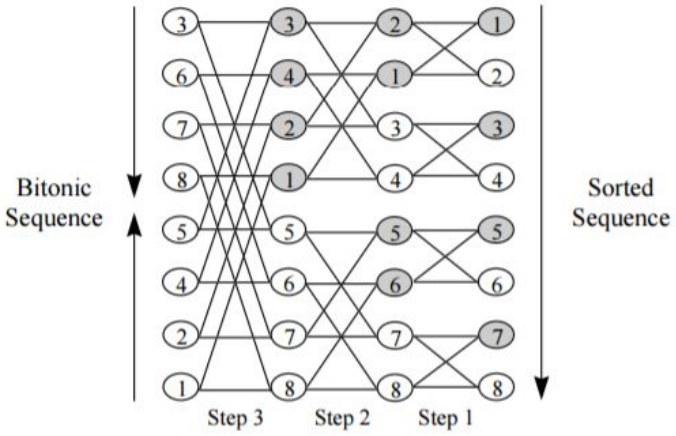
Objective

Our objective here is to implement the bitonic sorting network in a Graphical processing Unit and analysing its performance when compared to a sequential sorting method.

Bitonic Sequence

A bitonic sequence is a sequence with $x_0 \leq \ldots \leq x_k \geq \ldots x_{n-1}$ for some k, $0 \leq k < n$ or a circular shift of such a sequence.

On a bitonic sequence we can apply the operation called bitonic split which halves the sequence in two bitonic sequences so that all the elements of one sequence are smaller than all the elements of the other sequence. Thus, given a bitonic sequence we can recursively obtain shorter bitonic sequences using bitonic splits, until we obtain sequences of size one, at which point the input sequence is sorted

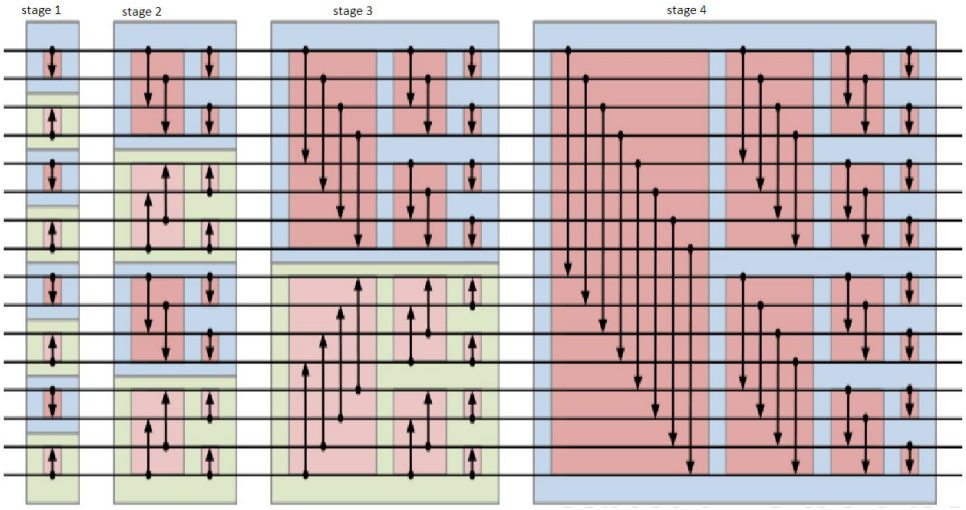


Bitonic Sorting Network

The bitonic sorting network for sorting N numbers consists of lg N bitonic sorting stages, where the i-th stage is composed of $N=2^i$ alternating increasing and decreasing bitonic merges of size 2^i.

Each node of the bitonic sorting network is identified by three integers which are the stage, the column inside the stage and the row of the node. We will see how we use this structure in our CUDA code.

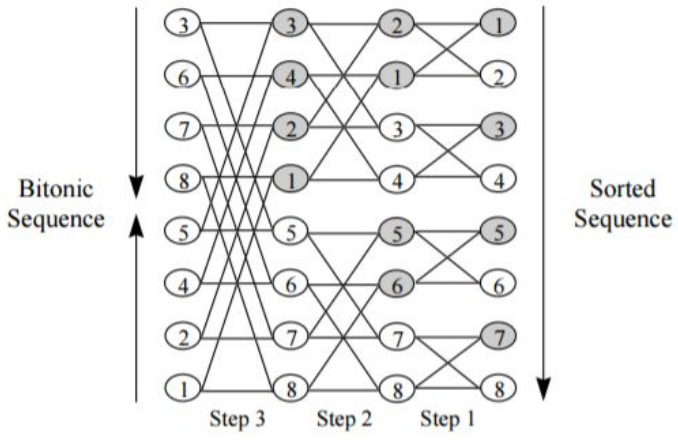
The bitonic Network looks like this:



Here we can see four stages each stage having variable (#columns = stage no.) no. of columns. Each column having N/2 comparisons. So we need N/2 processors to run this algorithm parallely.

Bitonic Merge Sort Algorithm

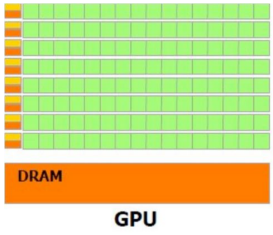
Given a sequence (Array) of numbers we will build bitonic sequences of size 4 and apply bitonic merge i.e., sorting the small bitonic sequences in such a way that the array splits into bitonic sequences of double size. So for every step the size of bitonic sequence doubles. If we apply this operation for $\log_2(N)$ times the size of bitonic sequence becomes 2N with its pivot at the end of the array, i.e., the values from beginning of the sequence to the end are in increasing order, so the array is sorted.



If the figure (to the right), we can see that merging a bitonic sequence of size N (with its pivot at the center) needs $\log_2(N)$ steps. At each step a bitonic sequence is divided into two small

bitonic sequence of half the length so that all the elements of one sequence are smaller than all the elements of the other sequence. Finally the sequence gets sorted in $\log_2(N)$ steps.

Graphics Processor



GPU is a parallel processor, their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel.

A GPU's structure contains N multiprocessors with M cores each. All the cores in the same processor share same instruction i.e., they are in SIMD

NVIDIA CUDA

CUDA (Compute Unified Device Architecture) is a language developed by NVIDIA for general purpose computing. Here we have to deal with both the host memory (primary memory) and the device memory (GPU's memory).

CUDA has its own thread layout which can be defined by the programmer. Here threads are organised into blocks and these blocks form a grid. Each block is executed by a multiprocessor. In CUDA only kernel functions can be executed in GPU.

Algorithm Implementation

Let us say we have N/2 processors each with index starting from 0 to N/2-1. Actually this problem can be easily implemented recursively, but here we will not use this method, instead we will use mapping (optimized implementation)where we map which processor has to deal with what elements at a stage g and step t and run the algorithm for all the stages and steps.

Let's say index is the processors index, then the two elements we deal with at stage g and step t are at positions m1 and m2 :

$m1 = map$

$m2 = map + (2^{(t-1)})$ where $map = (index / (2^{(t-1)}))*(2^t) + (index \% (2^{(t-1)}))$

$dir = (map / (2^g)) \% 2$ dir defines which elements gets bigger value after swapping

Since this operation has to be done by GPU we code it in kernel function, which looks like mentioned below:

```
__global__ void bitonicSort(int* a, int n, int g, int t) {

	int index = blockIdx.x * blockDim.x + threadIdx.x,
		map = (index / (1 << (t - 1)))*(1 << t) + (index % (1 << (t - 1))),
		pos = (map / (1 << g)) % 2,
		m1 = (pos == 0) ? map : (map + (1 << (t - 1))),
		m2=  (pos == 0) ? (map + (1 << (t - 1))):map;

	atomicMin(&a[m1], atomicMax(&a[m2], a[m1]));
	__syncthreads();
}
```

The driver function which decides which stage and which step has to occur and calls the kernel is written as below:

```
Void sortb(int* a, int size,int logn2) {

	int* array;
	cudaMalloc((void **)&array, sizeof(int)*size);
	cudaMemcpy(array, a, sizeof(int) * size, cudaMemcpyHostToDevice);
		int threadsPerBlock = 1024;
	int blocksPerGrid = ((size/2) + threadsPerBlock - 1) / threadsPerBlock;
	clock_t t, t1,td;
	t = clock();
	for (int g = 1; g <= logn2; g++) {
		for (int t = g; t > 0; t--) {
			bitonicSort <<<blocksPerGrid, threadsPerBlock >>>(array, size, g, t);

		}
	}
	cudaDeviceSynchronize();
	cudaMemcpy(a, array, size * sizeof(int), cudaMemcpyDeviceToHost);
	cudaFree(array);
}
```

In order to sort an array present in main memory using GPU we have to copy this array to device memory (GPU can perform operations only on its device memory) and perform sorting and copy the sorted array back to main memory.

kernel<<<>>> is an asynchronous call, so we need to synchronize using `cudaDeviceSynchronize()` if we want to measure the time taken for kernel operations.

Analysis and Report

This algorithm was tested on an NVIDIA GeForce 940M machine which has maxwell architecture and 384 CUDA cores.

Theoretical

Best known sequential sorting method has time complexity of O(n logn) for n array input

$T(1, N) = O(N \log N)$

Time complexity of bitonic sort algorithm:

We have log(N) stages, with each stage having increasing number of steps upto log(N), so total no of time steps required when running in N/2 processor machine are

$1 + 2 + \ldots\ldots + \log(N) = \log^2(N)$

$T(N/2 , N) = O(\log^2(N))$

Work complexity $= (N/2) * O(\log^2(N))$ all the processors are working at each time step

$= O(N \log^2(N))$

Speedup $S(p,N) = T(1,N) / T(p,N)$

If $p < N$ then let us assume the few parallel steps are executed sequentially

i.e., N parallel steps which takes O(1) when executed in N processor machine takes O(N/p) steps if executed in processor machine

$T(p,N) = O(N/p) * O(\log^2(N)) = O(N \log^2(N) /p)$

Speedup $= T(1,N) / T(p,N)$

$= (N \log N t_c) / ((N/2)\log^2(N) /p t_c)$

$= 2p/\log N$

For $N = 8192 = 2^{13}$

Theoritical speed up $= 2 * (384) / 13 = 59.07$

Practical

```
Analysis Report:   Max no of processors

   N       time over        time over        is sorted?    speedup:
   N      serial code      Bitonic CUDA code
      4       0.00000          0.00000          YES     0.000000
      8       0.00000          0.00000          YES     0.000000
     16       0.00000          0.00000          YES     0.000000
     32       0.00000          0.00000          YES     0.000000
     64       0.00000          0.00000          YES     0.000000
    128       0.00000          0.00000          YES     0.000000
    256       0.00000          0.00000          YES     0.000000
    512       0.00000          0.00100          YES     0.000000
   1024       0.00000          0.00100          YES     0.000000
   2048       0.00200          0.00100          YES     1.999998
   4096       0.00300          0.00100          YES     2.999997
   8192       0.00700          0.00300          YES     2.333333
  16384       0.01200          0.00400          YES     2.999999
  32768       0.04500          0.00800          YES     5.624999
  65536       0.04400          0.01800          YES     2.444444
 131072       0.08800          0.03900          YES     2.256410
 262144       0.17400          0.08200          YES     2.121951
 524288       0.38000          0.17900          YES     2.122905
1048576       0.68300          0.37600          YES     1.816489

speedUp = 0.0000000 or -nan(ind) means , more precision is needed to calculate speedup
```

Above is the output of the analyser, which is used to compute time taken for serial code, parallel code and speedup for various sizes of input N, ranging from 2^2 to 2^{20}

We have observed that the practical speedup has an average of 2.6720 which is not even near to the practical speedup.

This huge difference must be because of thread scheduling, cache updates, communication time and GPU time step is not same as CPU time step.

Analysis report when no of processors are varying

```
Analysis Report:  VARYING PROCESSORS # = 16

   N       time over        time over        is sorted?    speedup:
   N      serial code      Bitonic CUDA code
   512      0.00100          0.00600          YES     0.166667
   1024      0.00100          0.01500          YES     0.066667
   2048      0.00200          0.05400          YES     0.037037
   4096      0.00400          0.15700          YES     0.025478
   8192      0.01800          0.32700          YES     0.055046
```

```
Analysis Report:  VARYING PROCESSORS # = 32

   N       time over        time over        is sorted?    speedup:
   N      serial code      Bitonic CUDA code
     64       0.00000          0.00000          YES     -nan(ind)
    128       0.00000          0.00000          YES     -nan(ind)
    256       0.00000          0.00200          YES     0.000000
    512       0.00100          0.00700          YES     0.142857
   1024       0.00000          0.01600          YES     0.000000
   2048       0.00100          0.05000          YES     0.020000
   4096       0.00200          0.14000          YES     0.014286
   8192       0.02600          0.34300          YES     0.051802

Analysis Report:  VARYING PROCESSORS # = 64

   N       time over        time over        is sorted?    speedup:
   N      serial code      Bitonic CUDA code
    128       0.00000          0.00100          YES     0.000000
    256       0.00000          0.00200          YES     0.000000
    512       0.00000          0.00600          YES     0.000000
   1024       0.00100          0.01600          YES     0.062500
   2048       0.00100          0.05100          YES     0.019608
   4096       0.00670          0.14500          YES     0.046897
   8192       0.01500          0.32600          YES     0.046012

Analysis Report:  VARYING PROCESSORS # = 128

   N       time over        time over        is sorted?    speedup:
   N      serial code      Bitonic CUDA code
    256       0.00100          0.00500          YES     0.200000
    512       0.00100          0.01000          YES     0.100000
   1024       0.00200          0.02200          YES     0.090909
   2048       0.00300          0.05600          YES     0.053571
   4096       0.01000          0.13600          YES     0.073529
   8192       0.01000          0.32400          YES     0.030864

Analysis Report:  VARYING PROCESSORS # = 256

   N       time over        time over        is sorted?    speedup:
   N      serial code      Bitonic CUDA code
    512       0.00000          0.00900          YES     0.000000
   1024       0.00100          0.02200          YES     0.045455
   2048       0.00300          0.05300          YES     0.056604
   4096       0.00800          0.13400          YES     0.059701
   8192       0.00600          0.35100          YES     0.017094

speedUp = 0.0000000 or -nan(ind) means , more precision is needed to calculate speedu
```

If we focus on the parallel algorithm time consumption, we have this

```
#processors                                 speedup
        16                                  0.025478
        32                                  0.014286
        64                                  0.046897
        128                                 0.073529
        256                                 0.059701
```

As the # of processors increases, the speedup increases

This behaviour could be because of the increase in cache updates as the n of processors decreases

Conclusion

In this final report we have implemented the bitonic sorting using a mapping method on a GPU and analysed its performance. We have also attained an average speedup of 2.67 when compared to sequential sorting.