**Report on**

**Sorting data on the GPU**

**using Bitonic Merge Sort**

Ram **Manohar** Oruganti

Submitted on 12/13/13

Instructor: Mr. Jason Lowden

TA: Ms. Samantha Kenyon

Lecture Section: 1

Professor: Dr. Sonia Lopez Alarcon

**Abstract:**

Sorting algorithms are widely employed in various fields and many a times the efficiency of the sorting algorithm in the application dictates the performance of the application. Hence, even before GPUs were used for general purpose computations, many sorting algorithms have been proposed for GPUs as it would not be efficient to send the data to the host for sorting data native to the applications of the device. One such algorithm that efficiently sorts data on the device and has few requirements in terms of computational capabilities is Bitonic Merge Sort, devised by Ken Batcher. The computational complexity of the algorithm is .

Now that GPUs can be used for general purpose computations as well, it is a good idea to see if the data local to the host can be sorted in the GPU more quickly. As the average computational complexity of various sorting algorithms implemented on CPUs is , there exists a scope for achieving a better performance when the data is sorted on the GPU once the data size is large enough to give the device the necessary degree of parallelism it needs. So this project implements Bitonic Merge Sort on the host and the device and compares the execution times of these implementations against bubble sort implementation on the host for the best case and worst case scenarios for various input sizes.

From the results it can be clearly seen that this approach to sorting the data comes with its set of advantages and disadvantages. The readily seen advantage is the speedup the implementation offers for large input sequences and the predictability of execution time for a given size immaterial of the data in the sequence which can be exploited by the application. The disadvantages are reduced performance benefits for input sequences whose length is not a power of 2 and lower utilization and performance of the system for input sequences which aren’t large enough to give the necessary degree of parallelism for GPUs.

**Design Methodology:**

Bitonic Merge Sort, being a data-independent algorithm, can be represented using a Sorting Network as shown in Fig1. A sorting network has steps and each step involves a few stages. Number of stages in a step is determined by the step number. Hence the computational complexity becomes . The fine lines in Fig1 indicate the end of a stage and the thick lines indicate the end of a step. An arrow between two lines indicates a comparison operation between data elements and the arrow-head always points to the larger value.

Step 3

Stage 3 Stage 2 Stage 1

Step 1

Stage1

Step 2

Stage 2 Stage 1

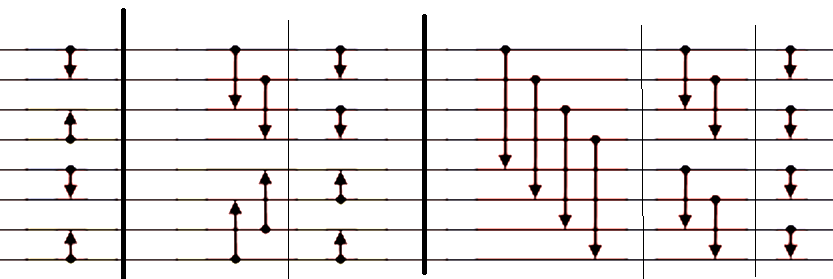


Fig1. A bitonic sorter network for an input sequence of length 8.

A bitonic sorter generates bitonic sequences at the end of each intermediate step and supplies it as an input to the next stage. A bitonic sequence is a combination of an ascending sequence and a descending sequence of equal length that is a multiple of 2. Each stage involves n/2 comparisons which can be done in parallel. The number of bitonic sequences reduces by a factor of 2 and the size of each bitonic sequence increases by 2 after each step. In the last step all the comparisons are ascending and thus it produces an output sequence sorted in ascending order.

The Bitonic Merge Sort algorithm is as follows:

1. Check if the number of elements in the input sequence is a power of 2. If not, pad the input sequence with the minimum number of zeros possible to satisfy the condition.
2. Split the input sequence into sequences of size that is a multiple of 2 starting from 2, such that the alternative sequences are ascending (and the remaining sequences are descending). To do so, start by comparing the and elements of the input when the sequence is being split into a size of and swap the elements of the to satisfy the requirement (ascending or descending). Proceed by diving by 2 until becomes 1. Thus each step will have stages.
3. Merge an ascending sequence with a consecutive descending sequence to obtain a bitonic sequence and use this sequence as an input for the next step.
4. Repeat step 2 for the next multiple of 2 and proceed to step 3. Continue the process until steps are completed , where n is the length of the input sequence after zero padding (if any).
5. After steps the input sequence will be sorted.

The implementation of Bitonic Merge Sort on CPU can be done by using either recursive functions or iterative loops. However the use of recursive function calls creates additional overheads and affects the performance of the program. Hence iterative loops should be used for a better performance. The host needs to append zeros if the length of the input sequence is not a power of 2. The primary loop iterates through all the steps in the sorting and the secondary loop iterates through the stages in each step. For each step the length of the bitonic sequence, and thus the length of ascending sequence and descending sequence that form it, is determined. This process is iterated for a few times to obtain the average execution time.

The implementation of Bitonic Merge Sort on GPU needs to be done keeping threads and parallelism in mind. The sorting of elements within a stage needs to be done concurrently to speed up the sorting procedure and as the number of threads in the parallel implementation as the number of elements in the input sequence (after zero padding, if any) and each comparison accesses two elements of the array, one of the two threads need to be dropped by using a branching statement. The implementation drops the thread with a higher index using a branching statement. This logic to drop the higher thread needs a bit-level analysis of the sorting network. This logic is illustrated using an 8-bit input bitonic sorter as shown in Fig 2. The numbers to the left indicate the index of the element and the numbers in the parentheses indicate the binary value of this array index.

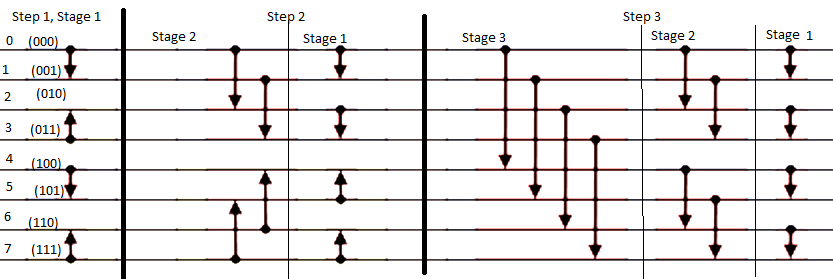


Fig 2. An 8-bit bitonic sorter

In the first step, first stage the numbers that are compared are (0,1) , (4,5) in ascending order and (2,3), (6,7) in descending order. As threads with higher index in the comparison are to be dropped the condition should be such that it allows threads 0,2,4,6 to pass through and rejects the rest. Similarly in the second step, stage 2, threads 0,1,4,5 should be retained and the rest should be rejected. In stage 1 of step 2 threads 0,2,4,6 should be retained and the rest should be rejected. In stage 3 of step 3 threads 0,1,2,3 should be retained, in stage 2 of step 3 threads 0,1,4,5 should be retained, in stage 1 of step 3 threads 0,2,4,6 should be retained. So, from the above analysis it can be concluded that the stage number decides the threads that should be retained. The step number on the other hand decides the direction of comparison (ascending or descending). When the bitwise XOR of and the thread index is computed and the resultant value is greater than the thread index itself, the thread falls into the set of threads that should be allowed to proceed. So using this logic the kernel decides as to which threads should be allowed to proceed for further computation. If the bitwise AND of and thread index is zero the comparison should arrange the numbers in ascending order and in descending order otherwise. The zero padding in the device could be done in parallel for all the elements using a kernel. The grid dimensions are dependent on the size of the input sequence and the block dimension for CUDA implementation is defined such that it utilizes the SMPs to the maximum possible extent.

In this project Bubble Sort was implemented on the CPU and its execution time was compared against the execution times of Bitonic Implementations on the CPU and GPU and the performance speedup was calculated with respect to Bubble Sort implementation for random case where rand function was used to generate the input sequences and for best case where an input sequence that had been ascending was fed as input to these implementations. The execution times and thus speedups were obtained for various input sizes.

**Results & Analysis:**

The execution times for the implementations of Bubble Sort on CPU, Bitonic Merge Sort on CPU, and Bitonic Merge Sort on GPU have been calculated for the best case and average case scenarios of Bubble Sort. A sample output for an input size of 65536 for an average case random input scenario has been shown in Fig 3.

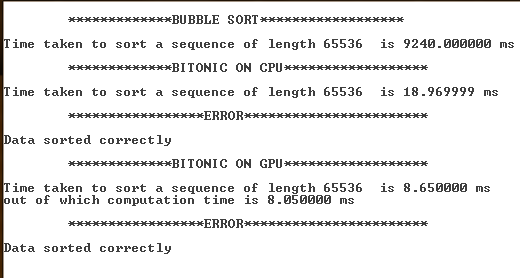


Fig 3. Sample output for an input size of 65536

The information about the execution times for random case and best case scenarios, speedups, GPU computation time and percentage of computation time on the GPU (GPU utilization), deviation of execution time for Bitonic and Bubble Sort implementations are tabulated in Table 1, Table 2, Table 3, Table 4 and Table 5 respectively.

|  |  |  |  |
| --- | --- | --- | --- |
| Input Size | Execution time for Bubble sort average case (ms) | Execution time for Bitonic Sequential (ms) | Execution time for Bitonic CUDA (ms) |
| 2 | 0 | 0 | 0.33 |
| 3 | 0 | 0 | 0.39 |
| 4 | 0 | 0 | 0.48 |
| 5 | 0 | 0 | 0.5 |
| 8 | 0 | 0 | 0.5 |
| 9 | 0 | 0 | 0.91 |
| 16 | 0 | 0 | 0.58 |
| 17 | 0 | 0 | 1.07 |
| 32 | 0 | 0 | 0.93 |
| 33 | 0 | 0 | 1.04 |
| 64 | 0 | 0 | 1.15 |
| 65 | 0.01 | 0.02 | 1.23 |
| 128 | 0.05 | 0.02 | 1.36 |
| 129 | 0.05 | 0.035 | 2.123 |
| 256 | 0.22 | 0.05 | 1.38 |
| 257 | 0.24 | 0.07 | 1.74 |
| 512 | 1.02 | 0.11 | 1.71 |
| 513 | 1.11 | 0.12 | 3.009 |
| 1024 | 4.03 | 0.25 | 2.26 |
| (continued) | | | |
| Input Size | Execution time for Bubble sort average case (ms) | Execution time for Bitonic Sequential (ms) | Execution time for Bitonic CUDA (ms) |
| 1025 | 4.34 | 0.42 | 2.61 |
| 2048 | 15.09 | 0.42 | 2.54 |
| 2049 | 15.87 | 0.59 | 3.16 |
| 4096 | 46.68 | 0.75 | 2.76 |
| 4097 | 48.13 | 1.29 | 3.12 |
| 8192 | 141.05 | 1.73 | 3.18 |
| 8193 | 147.38 | 2.96 | 3.92 |
| 16384 | 550.13 | 3.92 | 4.16 |
| 16385 | 567.54 | 6.3 | 6.54 |
| 32768 | 2261.29 | 8.48 | 6.59 |
| 32769 | 2276.15 | 13.86 | 6.73 |
| 65536 | 9240.82 | 18.97 | 6.85 |
| 65537 | 9254.32 | 30.46 | 9.34 |
| 131072 | 48379.32 | 42.49 | 9.96 |
| 131073 | 48504.68 | 68.75 | 16.85 |
| 262144 | 202031.5 | 91.9 | 17.04 |
| 262145 | 203038.42 | 153.13 | 27.67 |
| 524288 | 790369.56 | 203.07 | 30.9 |
| 524289 | 790403.28 | 337.97 | 52.96 |
| 1048576 | 1204719.25 | 442.91 | 58.29 |

Table 1. Execution times for average case scenario with random inputs for various input sizes

|  |  |  |  |
| --- | --- | --- | --- |
| Input Size | Execution time for Bubble sort best case (ms) | Execution time for Bitonic Sequential (ms) | Execution time for Bitonic CUDA (ms) |
| 2 | 0 | 0 | 0.331 |
| 3 | 0 | 0 | 0.397 |
| 4 | 0 | 0 | 0.48 |
| 5 | 0 | 0 | 0.49 |
| 8 | 0 | 0 | 0.48 |
| 9 | 0 | 0 | 0.93 |
| 16 | 0 | 0 | 0.61 |
| 17 | 0 | 0 | 1.13 |
| 32 | 0 | 0 | 0.97 |
| 33 | 0 | 0 | 0.99 |
| 64 | 0 | 0 | 1.18 |
| 65 | 0 | 0.03 | 1.29 |
| 128 | 0 | 0.03 | 1.31 |
| 129 | 0 | 0.05 | 2.05 |
| (continued) | | | |
| Input Size | Execution time for Bubble sort best case (ms) | Execution time for Bitonic Sequential (ms) | Execution time for Bitonic CUDA (ms) |
| 256 | 0 | 0.05 | 1.41 |
| 257 | 0 | 0.07 | 1.77 |
| 512 | 0 | 0.14 | 1.69 |
| 513 | 0 | 0.14 | 3.13 |
| 1024 | 0 | 0.29 | 2.21 |
| 1025 | 0 | 0.37 | 2.53 |
| 2048 | 0 | 0.39 | 2.46 |
| 2049 | 0 | 0.64 | 3.12 |
| 4096 | 0.01 | 0.76 | 2.77 |
| 4097 | 0.01 | 1.34 | 3.07 |
| 8192 | 0.03 | 1.85 | 3.26 |
| 8193 | 0.03 | 3.13 | 3.97 |
| 16384 | 0.06 | 4.01 | 4.07 |
| 16385 | 0.07 | 6.65 | 6.48 |
| 32768 | 0.11 | 8.54 | 6.42 |
| 32769 | 0.12 | 15.29 | 6.62 |
| 65536 | 0.19 | 18.97 | 7.01 |
| 65537 | 0.2 | 30.75 | 9.41 |
| 131072 | 0.37 | 41.82 | 10.15 |
| 1048576 | 2.03 | 442.88 | 57.59 |

Table 2. Execution times for best case scenario with pre-sorted inputs for various input sizes

|  |  |  |  |
| --- | --- | --- | --- |
| Input Size | Speedup of Bitonic CPU with respect to Bubble Sort | Speedup of Bitonic CUDA with respect to Bubble Sort | Speedup of Bitonic GPU with respect to Bitonic CPU |
| 2 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 |
| 17 | 0 | 0 | 0 |
| 32 | 0 | 0 | 0 |
| 33 | 0 | 0 | 0 |
| 64 | 0 | 0 | 0 |
| 65 | 0.5 | 0.008130081 | 0.016260163 |
| 128 | 2.5 | 0.036764706 | 0.014705882 |
| (continued) | | | |
| Input Size | Speedup of Bitonic CPU with respect to Bubble Sort | Speedup of Bitonic CUDA with respect to Bubble Sort | Speedup of Bitonic GPU with respect to Bitonic CPU |
| 129 | 1.428571429 | 0.023551578 | 0.016486105 |
| 256 | 4.4 | 0.15942029 | 0.036231884 |
| 257 | 3.428571429 | 0.137931034 | 0.040229885 |
| 512 | 9.272727273 | 0.596491228 | 0.064327485 |
| 513 | 9.25 | 0.36889332 | 0.039880359 |
| 1024 | 16.12 | 1.783185841 | 0.110619469 |
| 1025 | 10.33333333 | 1.662835249 | 0.16091954 |
| 2048 | 35.92857143 | 5.940944882 | 0.165354331 |
| 2049 | 26.89830508 | 5.022151899 | 0.186708861 |
| 4096 | 62.24 | 16.91304348 | 0.27173913 |
| 4097 | 37.31007752 | 15.42628205 | 0.413461538 |
| 8192 | 81.53179191 | 44.35534591 | 0.544025157 |
| 8193 | 49.79054054 | 37.59693878 | 0.755102041 |
| 16384 | 140.3392857 | 132.2427885 | 0.942307692 |
| 16385 | 90.08571429 | 86.77981651 | 0.963302752 |
| 32768 | 266.6615566 | 343.1396055 | 1.286798179 |
| 32769 | 164.2243867 | 338.2095097 | 2.059435364 |
| 65536 | 487.128097 | 1349.024818 | 2.769343066 |
| 65537 | 303.8187787 | 990.8265525 | 3.26124197 |
| 131072 | 1138.604848 | 4857.361446 | 4.266064257 |
| 131073 | 705.5226182 | 2878.616024 | 4.080118694 |
| 262144 | 2198.384113 | 11856.30869 | 5.393192488 |
| 262145 | 1325.921896 | 7337.853993 | 5.534152512 |
| 524288 | 3892.104004 | 25578.30291 | 6.57184466 |
| 524289 | 2338.678818 | 14924.53323 | 6.381608761 |
| 1048576 | 2720.009144 | 20667.68314 | 7.598387373 |

Table 3. Speedup offered for various input sizes

|  |  |  |  |
| --- | --- | --- | --- |
| Input Size | Execution time for Bitonic CUDA (ms) | CUDA compute time (ms) | Percentage of computation time in GPU |
| 2 | 0.33 | 0.07 | 21.21212121 |
| 3 | 0.39 | 0.12 | 30.76923077 |
| 4 | 0.48 | 0.12 | 25 |
| 5 | 0.5 | 0.25 | 50 |
| 8 | 0.5 | 0.19 | 38 |
| 9 | 0.91 | 0.47 | 51.64835165 |
| (continued) | | | |
| Input Size | Execution time for Bitonic CUDA (ms) | CUDA compute time (ms) | Percentage of computation time in GPU |
| 16 | 0.58 | 0.36 | 62.06896552 |
| 17 | 1.07 | 0.65 | 60.74766355 |
| 32 | 0.93 | 0.62 | 66.66666667 |
| 33 | 1.04 | 0.76 | 73.07692308 |
| 64 | 1.15 | 0.89 | 77.39130435 |
| 65 | 1.23 | 0.95 | 77.23577236 |
| 128 | 1.36 | 1.02 | 75 |
| 129 | 2.123 | 1.737 | 81.81818182 |
| 256 | 1.38 | 1.14 | 82.60869565 |
| 257 | 1.74 | 1.44 | 82.75862069 |
| 512 | 1.71 | 1.42 | 83.04093567 |
| 513 | 3.009 | 2.615 | 86.90594882 |
| 1024 | 2.26 | 2.07 | 91.59292035 |
| 1025 | 2.61 | 2.29 | 87.7394636 |
| 2048 | 2.54 | 2.22 | 87.4015748 |
| 2049 | 3.16 | 2.9 | 91.7721519 |
| 4096 | 2.76 | 2.49 | 90.2173913 |
| 4097 | 3.12 | 2.76 | 88.46153846 |
| 8192 | 3.18 | 2.88 | 90.56603774 |
| 8193 | 3.92 | 3.59 | 91.58163265 |
| 16384 | 4.16 | 3.86 | 92.78846154 |
| 16385 | 6.54 | 6.26 | 95.71865443 |
| 32768 | 6.59 | 6.27 | 95.14415781 |
| 32769 | 6.73 | 6.35 | 94.35364042 |
| 65536 | 6.85 | 6.36 | 92.84671533 |
| 65537 | 9.34 | 8.91 | 95.39614561 |
| 131072 | 9.96 | 9.27 | 93.07228916 |
| 131073 | 16.85 | 15.97 | 94.77744807 |
| 262144 | 17.04 | 16.04 | 94.1314554 |
| 262145 | 27.67 | 26.54 | 95.91615468 |
| 524288 | 30.9 | 28.91 | 93.55987055 |
| 524289 | 52.96 | 50.84 | 95.99697885 |
| 1048576 | 58.29 | 54.72 | 93.87545033 |

Table 4. GPU Utilization (for computation)

|  |  |  |  |
| --- | --- | --- | --- |
| Input Size | Percentage of deviation Bubble Sort execution time | Percentage of deviation Bitonic CUDA execution time | Percentage deviation in computation time CUDA |
| 2 | 0 | 0.303030303 | -14.28571429 |
| 3 | 0 | 1.794871795 | 8.333333333 |
| 4 | 0 | 0 | 0 |
| 5 | 0 | -2 | -4 |
| 8 | 0 | -4 | 5.263157895 |
| 9 | 0 | 2.197802198 | 4.255319149 |
| 16 | 0 | 5.172413793 | 2.777777778 |
| 17 | 0 | 5.607476636 | 6.153846154 |
| 32 | 0 | 4.301075269 | 4.838709677 |
| 33 | 0 | -4.807692308 | -5.263157895 |
| 64 | 0 | 2.608695652 | 5.617977528 |
| 65 | -100 | 4.87804878 | 7.368421053 |
| 128 | -100 | -3.676470588 | -4.901960784 |
| 129 | -100 | -3.438530382 | -2.705814623 |
| 256 | -100 | 2.173913043 | -0.877192982 |
| 257 | -100 | 1.724137931 | 2.777777778 |
| 512 | -100 | -1.169590643 | 1.408450704 |
| 513 | -100 | 4.021269525 | 2.868068834 |
| 1024 | -100 | -2.212389381 | -1.93236715 |
| 1025 | -100 | -3.0651341 | -5.240174672 |
| 2048 | -100 | -3.149606299 | -1.351351351 |
| 2049 | -100 | -1.265822785 | -2.413793103 |
| 4096 | -99.97857755 | 0.362318841 | 0.401606426 |
| 4097 | -99.97922294 | -1.602564103 | 0.724637681 |
| 8192 | -99.97873095 | 2.51572327 | 3.125 |
| 8193 | -99.97964446 | 1.275510204 | 3.064066852 |
| 16384 | -99.98909349 | -2.163461538 | -2.849740933 |
| 16385 | -99.98766607 | -0.917431193 | -4.632587859 |
| 32768 | -99.99513552 | -2.579666161 | -3.03030303 |
| 32769 | -99.99472794 | -1.634472511 | -1.732283465 |
| 65536 | -99.99794391 | 2.335766423 | 3.301886792 |
| 65537 | -99.99783885 | 0.749464668 | -0.112233446 |
| 131072 | -99.99923521 | 1.907630522 | 1.294498382 |
| 1048576 | -99.9998315 | -1.200892091 | -0.895467836 |

Table 5. Percentage of deviation in time between best case and average case scenarios

The graphs from Fig 4, Fig 5, Fig 6 and Fig 7 indicate the speedup plot, GPU Utilization in terms of computation, increase in execution time in comparison to previous input case and percentage deviation of execution time between best case and average case scenarios for Bitonic Merge Sort GPU implementation respectively.

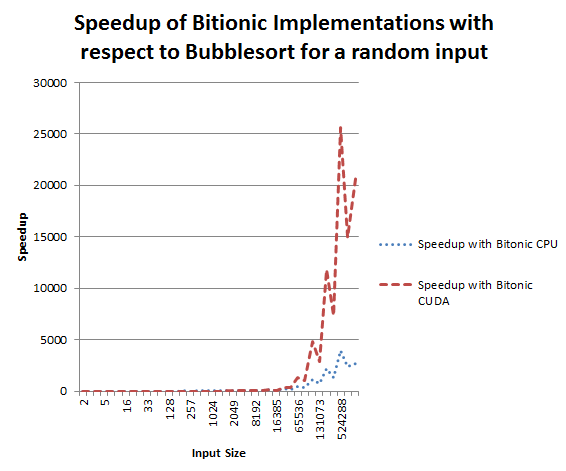


Fig 4. Speedup plot for various input sizes

From the above graph it can be readily seen, from the humungous speedups offered by the Bitonic Implementations on the host and the device, that Bitonic sort outperforms Bubble sort by a great margin. The other conspicuous part in the above graph is the presence of spikes in the speedup. The reason for these sudden falls of speedup can be accounted to the lack of inherent support for array sizes that are not powers of 2. Due to the zero padding the Bitonic Merge Sort algorithm sorts an array with higher number of elements than necessary. This leads to a penalty in terms of speedup and this penalty is very high for large array sizes, especially when the array size is very close to a power of 2 and exceeds it by just a few numbers. For bubble sort, on the other hand the execution time grows proportionally with size. This leads to the spikes in the speedup plot.

The plot in Fig 5 shows the percentage of time for which GPU is utilized for computation. Up until the GPU utilization is less than 80% the GPU’s performance is the worst as more than 20% of the execution time is not being utilized for computation. At a utilization of more than 80% for computation the GPU starts to catch up and once the percentage of utilization goes beyond 90 the GPU implementation beats both Bubble Sort CPU implementation and Bitonic CPU implementation. This is when the GPU implementation starts to have enough degree of parallelism to offer a speedup over other implementations. This problem has a relatively higher threshold for GPU and can be attributed to the branch divergence that reduces the number of threads and the presence of bitwise operations in these branching logics which can’t be effectively computed by the device, at least not as effectively as in the host. So amongst the n/2 comparisons that can be done in parallel, where n is the size of the input to bitonic sorter, n/4 threads need to execute compare and order numbers in ascending order and n/4 need to do it in descending order. So at any given point of time the degree of parallelism available for the GPU is n/4.

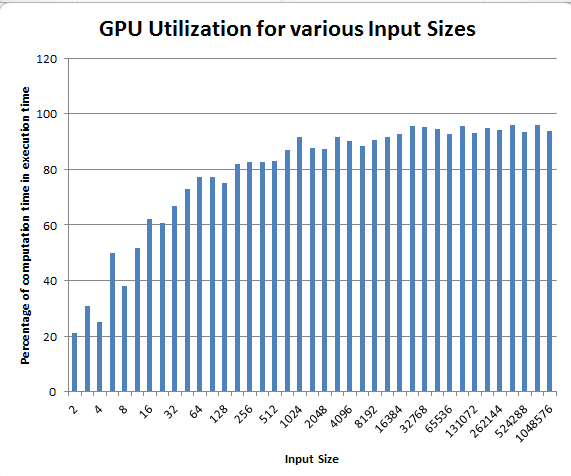


Fig 5. GPU Utilization in terms of computation for various input sizes

Fig 6. shows the plot of percentage increase in execution time with respect to the execution time for previous input for the GPU implementation. Again the interesting area in this plot is the points where the execution time rises sharply for a small change in input length. Even this can be attributed to the lack of inherent support for all sizes of input sequences. Hence the need for zero padding invokes another kernel which suffers all the more from the lack of necessary degree of parallelism for small input cases and thus can’t hide the latencies in accessing global memory. This also explains the presence of negative values in the graphs. The input sequences whose lengths are powers of 2 don’t need to invoke the kernel that does the zero padding and for small input sizes as the latency which can’t be hidden and shows up in the execution time otherwise, is not present for these input cases. Thus it leads to a lesser execution time even when the input size is larger.

Fig 7 shows the percentage of deviation of execution time for the GPU implementation of Bitonic Merge Sort for the best and average case scenarios for various input sizes. This graph serves as a conclusive proof of the fact that the bitonic merge sort algorithm is data independent and has the same best case and worst case computational complexity. The maximum deviation is around 5% and when the sizes grow larger this deviation goes on reducing.



Fig 6. Increase in execution time with respect to previous input execution time for the GPU implementation for various input sizes

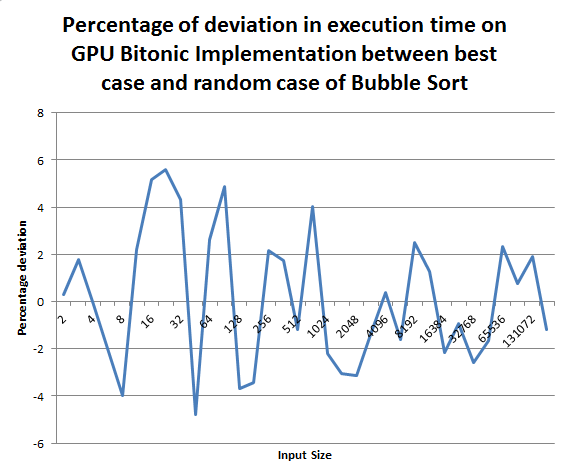


Fig 7. % deviation in execution time on GPU between best and average cases for various sizes

**Conclusion:**

It can be concluded that Bitonic Merge Sort implementation offers better performance over Bubble Sort except in the best case scenarios. But the possibility of occurrence of a best case scenario, i.e. completely sorted input is very rare and hence this approach can be employed to obtain a better performance, especially for large array sizes. It offers very high speedups for most of the cases. Although it doesn’t support sorting for all input sequences inherently, zero padding still allows offering great speedups but the speedups are not as good as the speedups obtained for sequences whose lengths are powers of 2. To ensure that the resources are not unnecessarily wasted, the data can be checked to verify that it is unsorted by using bubble sort. Also this algorithm can be used in applications that need predictable execution times owing to the data-independent nature of the algorithm. Even for smaller input sizes the execution time on the GPU can be further reduced by doing the zero padding in the host for smaller input sizes and using kernel for zero padding in large input sequences. As the copy time for small input sequences is less, the memory copy time would be less than the latencies incurred by the zero padding kernels.