Parallel Programming Assignment-2: Sample Sort in Cuda

Swayambhu Nath Ray
CDS M Tech
SR No. - 13288

28 Feb 2017

The Program Outline:

The code attached with this report has been used for implementing sample sort in cuda. The results shown below is for an array size of 4194304 elements. The code can be divided into three steps:

1. **Step-1: Local Sorting** In this step each thread block takes it own part of the main array which are disjoint and sort them locally. In the code attached, each thread block is taking up its own part of the main array in its shared memory (for faster access) and performing a parallel bubble sort to sort its elements locally.

2. **Step-2: Splitter Selection** In this step splitters need to be seleceted which will be used in the final step for dividing the main array among the thread blocks. But in order to exploit more parallelism, the splitters have been selected in the cpu in parallel with the local sort in the gpu. Thus choosing random samples, then sorting them, then again choosing equi-spaced samples and finalising the splitters. These splitters are then sent to the gpu in the 2nd kernel call.

3. **Step-3: Local Sorting** In this step each thread block is provided with two splitters, defining their upper and lower bound of interested numbers. Then using many threads all the numbers within this range is brought to the shared memory of the thread block. Then begins the critical part. If a simple merge sort were to be performed then most of the part of the algorithm would have been serialized, wasting the resources of the gpu. Also experimenting with merge-sort it has been realized that the code performs poorly. Hence an ennumeration sort has been performed in the code attached with this. In ennumeration sort each and every thread is kept busy all throughout and hence utilizing the resources of the gpu well and also giving better performance than

the merge-sort in shared memory. At the end of this sort each thread block will have its own array of sorted elements with each thread block arrays sorted according to their rank.

Some results: The sorted array in thread block 0 after the program is executed for 41904304 elements is given in the following diagram:

Figure 0.1: Sorted elements in thread block 0 after the program is executed

As it is clear from the above diagram that the elements are well sorted and hence this algorithm gives reliable results.

Some splitters chosen is shown in the following diagram:

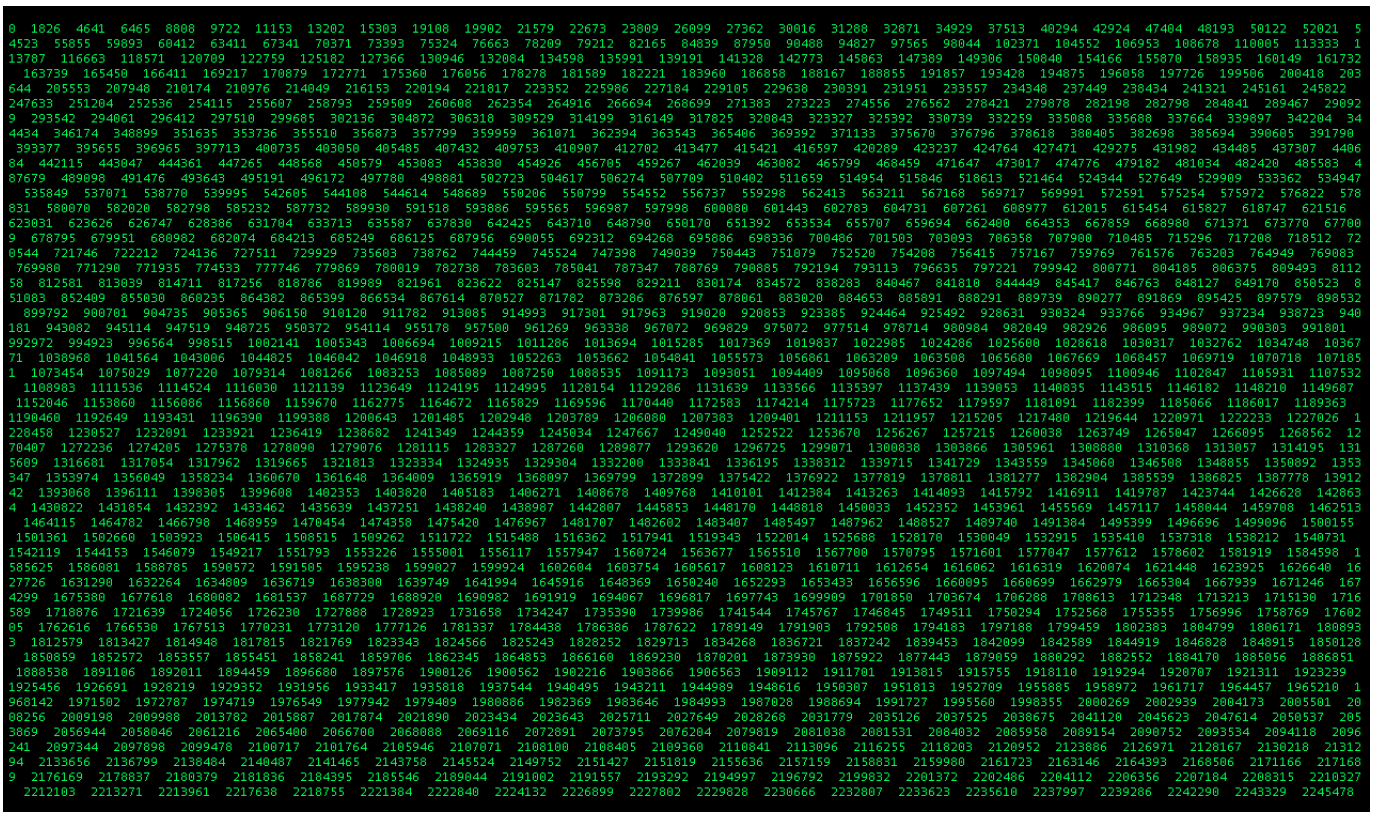

Figure 0.2: Splitters chosen.

Comparison: The above algorithm has been compared with a sequential merge sort in cpu and the results are shown below as follows:

No. of elements	Time for sample sort in gpu(usec)	Time for merge sort in cpu(usec)	speed-up
4194304	3061411	1456743	0.476
2097152	1645876	714604	0.434
1048576	848765	348842	0.411

Table 0.1: Comparisons.

Handling race conditions during final sort in shared memory.

In the code attached,all the threads are allowed to write to the same location for same elements in the array. Hence all the threads will eventually overwrite the same result for the same number in the array. Then for the unwritten places in the final array in shared memory each thread is filling up the non-filled places with valid elements before that place and hence the race condition is well taken care off, making the sorting stable.