# Merge sort for CUDA

Mihail Petrov Stamenov

October 7, 2023

**Abstract**

In this short paper I have explained two different implementations of the algorithm Merge sort implemented in CUDA C.

## 1 Algorithms

### 1.1 Shared memory merge sort

The main deference between the two solutions is that the first one uses shared memory of the threads. The idea is that since each is going to be read at least $log(N)$ times. Because of that I tried partitioning the data in chunks as big as the max shared memory of block. Then each chunk is sorted. In the end of this operation all chunks are sorted. Then the whole array is sorted not using the shared memory because the memory would not be able to store that big parts of the array. Since the whole shared memory of a block must be used the each thread is given a segment of the array to copy. If the array is bigger then the shared memory of all the blocks a block should be able to work on multiple areas of the array. Thus I slice the array in chunks with size of the memory all threads in all blocks will use. I iterate through all this slices and copy each slice in the shared memory of the blocks, each block sorts it and then I write the result in the array and start working on the next slice. Each thread copies segments with size **GRAN**, which was created with the intention to be used as granularity. This makes the partitioning of the data regular. The number of slices is calculated as:

$$blocksSlices = (N/(allThreadsMemoryUsage) + 1$$

Then for each thread a device index is calculated:

$$deviceIndex = passedElemetsInDevice + (index * GRAN) + i;$$

where **passedElemetsInDevice** is defined as:

$$passedElemetsInDevice = (blkSlice * allThreadsMemoryUsage);$$

and **i** is the number of the element of a thread is copying. For the copying part an Embarrassingly Parallel pattern was used and each thread is copying **GRAN** elements per slice. The index where the thread should place copied element in the shared memory is $(t * GRAN) + i$. If we are in the end of the array and there are not enough elements in the array to fill GRAN array for the thread I put **INT_MAX**.

After a slice is copied I synchronize the threads so that I can start sorting the blocks. Then a for cycle on the merging step width is started . The step is named **width** and it is two at the beginning and is multiplied by 2 until it reaches value bigger then the whole memory of the block. In fact until **width** is not bigger than the **GRAN** the threads work in Embarrassingly Parallel pattern. After that the threads with bigger **id** are pointing to the end af the array and a reduction pattern appears since only one thread can merge two arrays. From the outside all threads are assigned multiple chunks from the shared memory with size width and they split the chunk in two and merge the two arrays. For this reason width starts from 2 and is multiplied by 2. In the beginning all couple of neighbouring numbers are sorted, then all groups of 4 are split in 2 arrays with 2 elements which are merged and so on. At one point the whole shared memory for the block is sorted array and the threads need to be synchronised because then they are copied back to the memory of the device.

After all slices are done the array is sorted the same way as the small chunks with the difference that now starting step width is 2 times bigger then the last of the shared memory sorting part.

## 1.2 Second implementation

The second implementation resembles the second part of the first algorithm. After copying the array in the device I start iterating on the step width starting from 2 to number bigger then **N**. This way for each iteration I call the global function **gpu_mergesort** with the width of the segments which first and second part is sorted and we only call the device function **merge** for them. Remaining of the array is not as big as the width we set the start as:

$$start = width * idx * slices$$

where **idx** is:

$$idx = threadIdx.x + blockIdx.x * blockDim.x;$$

and slices is the number of slices a thread should work on. It is calculated like this:

$$slices = N/((threadsCount) * width) + 1;$$

**width** is just the size of the slices which are in fact 2 slices of the previous width and need to be merged. While performing that operation every time we write the sorted slice in the the result array. Since we call **gpu_mergesort** multiple times (for each width once) I needed to have different arrays for each result to store, but that would have been too much memory wast. Thus i had only 2 arrays. Input array with the data and result array and two int pointers for each of them. When an iteration is finished in the result array we have the elements which are sorted in chunks of **width** and this array should be the input array for the next iteration where the input array for this iteration is useless. So I just switch the pointers and the pointer for input array now will point to the result array of the previous iteration. This way I can still have pointer to the input and result array for each iteration and I do not need to copyanything.

## 2 Test and results

| solution1 (local memory use) | solution 2 |
|---|---|
| 0.91792 | 0.176512 |
| 0.792576 | 0.177056 |
| 0.827392 | 0.198688 |
| 0.89504 | 0.24448 |
| 1.0104 | 0.394144 |
| 1.111744 | 0.423808 |
| 4.05104 | 3.410496 |
| 29.795328 | 28.700672 |
| 254.4328 | 194.074585 |
| 2631.354736 | 2575.912842 |
| 23599 | 23243.28711 |
| 213746.0781 | 210370.6875 |

Table 1: Times in milliseconds of solution 1 and solution 2 .

The two implementations were tested with randomly generated arrays with size N and each element of the arrays is between 1 and N. The program accepts input parameter N. After that a sample array is generated and given to the two algorithms to sort.

As we can see from table 1 the results for solution 2 are better. This might be due to the fact that in solution 1 while a block is getting ordered part of the threads are not doing anything, where as in the second solution if a thread finishes its slice starts the next one and since all the slices are more or less equal for the threads they can start the next iteration without having threads waiting for the others to finish.

In table 2 was calculated the throughput of the two solutions (number of processed data items/seconds as a function of the input size). Again we can see that solution two outperforms solution one but both of them have steady growth more prominent for solution two until 1000000 and reaches something like plateau. In the graph of solution 1 we see steady slow growth.

| N | solution1 (local memory use) | solution 2 |
|---|---|---|
| 20 | 21788.39115 | 113306.744 |
| 50 | 63085.43282 | 282396.5299 |
| 100 | 120861.6955 | 503301.6589 |
| 250 | 279317.1255 | 1022578.534 |
| 500 | 494853.5234 | 1268571.893 |
| 1000 | 899487.6518 | 2359559.046 |
| 10000 | 2468501.916 | 2932124.829 |
| 100000 | 3356230.883 | 3484238.975 |
| 1000000 | 3930310.872 | 5152658.191 |
| 10000000 | 3800323.789 | 3882118.928 |
| 100000000 | 4237467.689 | 4302317.462 |
| 1000000000 | 4678448.413 | 4753513.961 |

Table 2: Throughput of solution 1 and solution 2.

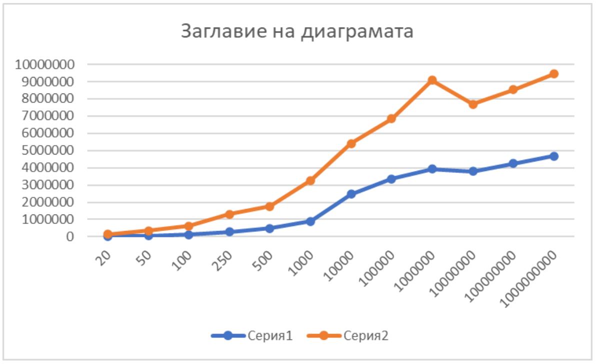

Figure 1: Enter Caption

To sum up, either because of the copies in the shared memory of the blocks or because of the not optimal distribution of work the first solution is slower. On the other havd by optimasing the distribution of work on threads I believe the first one can be faster since at the first part it will be reading only from the shared memory of a block.