2023

Architectures and Platforms for Artificial Intelligence

MERGE SORT USING CUDA-C PARALLEL PROGRAMING

SAMI MOHAMMED OSMAN

Summary

In this project, the objective was to experiment with parallel programming through the implementation of sorting algorithms for arrays. The project focused on studying and implementing sorting algorithms using CUDA C, with a focus on the ***Merge sort algorithm.*** The project was implemented using both global memory usage and L1 shared memory usage. The code was written with the possibility of running on multiple GPUs and a potential implementation for parallelization of GPU initialization using OpenMP. The project script takes inputs for the *array size*, *L1 memory usage flag*, and *GPU count* and generates three output files: Unsorted.txt with the original array of random integers, Sorted.txt with the sorted array, and Merge_out.txt with the execution time output analysis for weak and strong scaling.

Introduction:

A sorting algorithm most commonly orders data numerical or lexicographical order. Some common examples of sorting algorithms include bubble sort, insertion sort, selection sort, merge sort, and quick sort. Some sorting algorithms are convenient to parallelize than others. For my project I will use the ***Merge sort.***

- **Merge sort** is a good choice for parallel programming because it naturally divides the data into smaller, independent sub-problems that can be solved concurrently.
- **Quick sort** is also relatively easy to parallelize, as the partitioning step can be done in parallel, then the sub-arrays can be sorted in parallel as well.
- Bubble sort and insertion sort are not efficient to parallelize, due to the amount of swap operations needed and the dependency of each step on the previous one.

Strategy:

I decided to attempt to make bottom-up **merge sort** parallel. Recursive implementation would solve the problem quite well, but it is not possible to have recursion in GPU. I used for loop to sub divide the array that must be Merge sorted together. The basic idea is to use a loop to divide the input into pair element chunks. Use GPU threads to merge sort each chunk in parallel, increase the chunk size in the second round. At last Merge the two sorted chunks back together in a single thread, to produce the final sorted output.

Without L1 Shared Memory algorithm: [V1]

sortArray() function:

- Start by creating two lists: input array and a temporary array that is the same size. The width is defined as chunk and increases by a factor of 2 with each step.

mergeSortKernel() function:

- As the width grows, this function sorts each section of the list into the temporary array. Then, switch the pointers of the two lists so that you can avoid creating multiple small arrays or having to copy the temporary array back to the input array.
- This step is performed in parallel, where each thread is assigned a portion of the list to sort.
- **Note**: In this version of the code the array resides in the global memory. Read/Write operation is very costly.

With L1 Shared Memory algorithm. [V2]

mergeSortKernel_SHEMEM() function: This implementation follows the same process as previously described, dividing the array into two input arrays and utilizing a bottom-up approach. However, what sets this version apart is:

- The GPU threads Read/Write the array once from global memory to shared memory and vice versa.
- In this version the threads Read/Write operation during sort each section of the list into the temporary array uses the shared memory.
- Main draw back here is the size of the shared memory size restrict the size of the array we want to sort.

Implementation:

Merge_Multi_gpu.cu:

Contains the main() function. Responsible for controlling the utility functions and setting base variables like:

- **Array size** integer: Creating user defined size random value array to be sorted and save them in *Unsorted.txt* file. Save the output in sorted.txt file.
- **Shared Memory Flag** boolean: The project runs on two versions of memory usage:
    1. “true” Use the GPU shared memory to store the array during sorting.
    2. “false” Use the GPU global memory to store the array during sorting.
- ***GPU count:*** The amount of GPU that we have to use.

kernelFunctions.cu:

Contains important function responsible to perform the merge sort bottom-up approach. **sortArray()** the function takes parameters like array to be sorted, size of the array, GPU device index and shared memory flag. The code sets the GPU device, allocates memory for two arrays (main and auxiliary) on the GPU, transfers the input array from the host to the GPU, and launches the merge sort kernel in the GPU.

The merge sort is performed in multiple iterations, each with a different block size, until the entire array is sorted. The code also prints the execution time and in case of CUDA error or not sorted array it displays error messages for debugging purposes. In this function the input array is sliced in chunks and use loop to implement the bottom-up sorting steps as follows:

```
(chunk = cnk; chunk < 2 * count; chunk *= 2)
```

Where: **cnk** is the starting index to be sorted.

Count is the chunk array size.

Increment *2 take two chunks together on the next round.

The kernelFunctions.cu file also Contains two functions that are executed in the GPU. Almost similar structure but the first one sorts the array by accessing the global memory. While the next one transfers the array to shared memory, do the sorting then store back the sorted array in the global memory.

- **mergeSortKernel():** This function is defined as global function "mergeSortKernel" that performs the merge sort on blocks of an input array "arr" of integers that resides in the global memory and stores the sorted results in "aux". The input parameters "blockSize" and "last" are used to determine the number of elements in each block and the last index of the input array, respectively. The code calculates the index of the current thread and performs the merge sort on each block within the bounds of the input array. The function sorts the elements in each block by comparing and merging the two halves of the current block and copies the sorted elements back to the original array.

- **mergeSortKernel_SHEMEM():** Performs merge sort on an input array using shared memory. The kernel uses the shared memory to copy elements of the input array **arr1** to a shared memory array **arr**. The elements of the shared memory array **arr** are then sorted and the sorted elements are copied back to the original array **arr1**. The kernel uses a **__syncthreads()** call to ensure that all threads in the block have finished copying data to shared memory and/or copying data back to the original array before continuing.

Utilities.cu:

This file takes all the part of code that can be sectioned and put into callable function for clear code readability. Some of these functions are generating random integers for the array, save the array in a file, check if the sorting is correct etc... In this project I used a predefined quick sort to check if the output of the Merge sorting is correct or not.

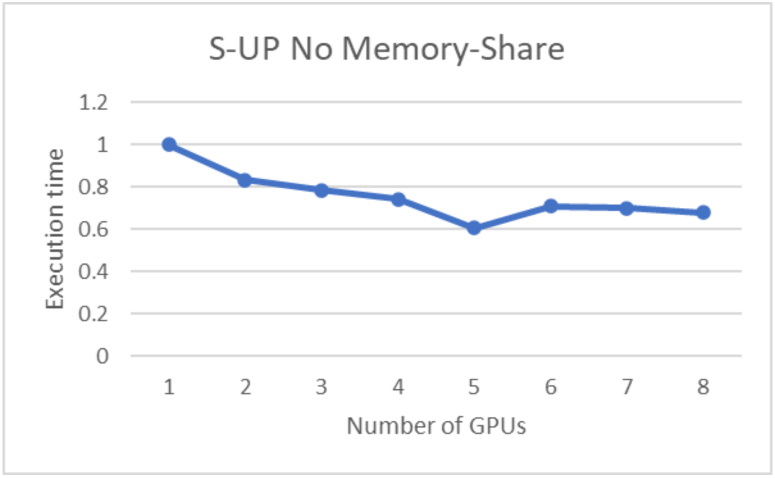
Scaling Analysis:

The performance analysis includes information about the weak and strong scaling of the algorithm, providing valuable insights into the effectiveness of the parallelization technique used.

Weak Scaling No L1 Memory Sharing

I wrote the code as if I am using 8 GPUs even though I am using one. The idea is executing them one after the other and calculate the execution time individually for each GPU call. This way the calculated execution time is as if the GPUs are running concurrently.

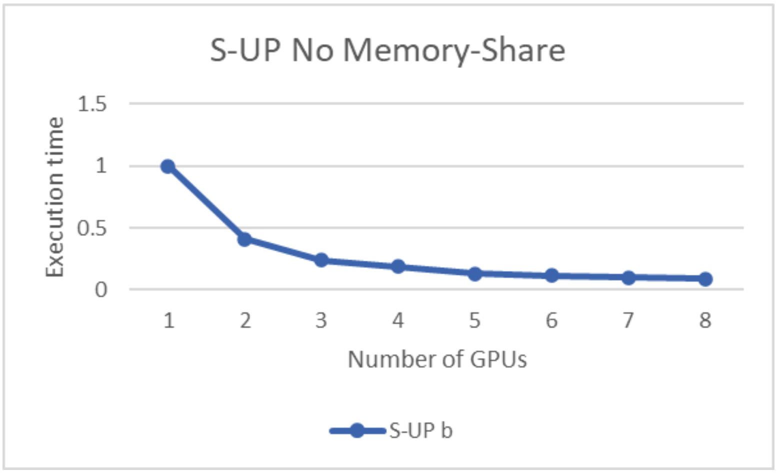
The execution of Merge_Multi_gpu for weak scaling is being run multiple times, with different GPU count, fixed array size and Shared memory flag "**false**" as the three parameters. The program is run five times for each GPU count.



There is a decline of performance at the beginning, but the graph shows that at certain point with the increase of number of GPUs there could be increase in performance, been the lowest point at GPU count of 5 in my case.

Strong Scaling No L1 Memory Sharing

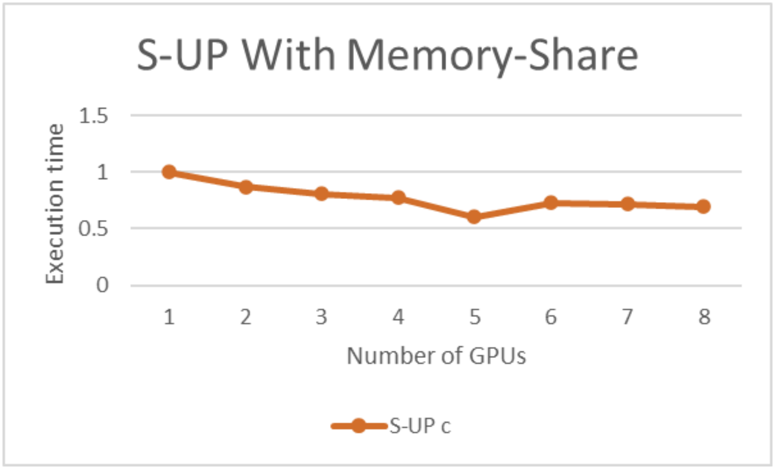
The program was run multiple times, with the GPU count ranging from 1 to 8, and the array size being updated in each iteration by multiplying the "array_size" variable by the current GPU count. The program was run five times for each GPU count and array size combination with Shared memory flag = "**false**".



Clearly there is a decline in the performance at the beginning, but the performance gets stabilized when we increase array size with the GPU counts. The increase in GPUs is helping in processing bigger array sizes with out significant decrease in performance. In my case starting from GPU count 6.

Weak Scaling with L1 Shared Memory

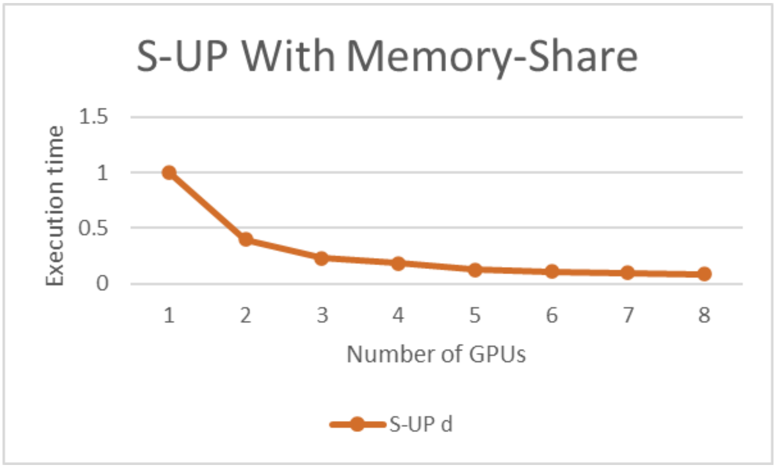
I am using the L1 shared memory to decrease the global memory access. Because This could potentially increase the performance as there is a huge read and write operations while sorting the arrays. The execution time is calculated as follows. Run the code multiple times, with different GPU count, fixed array size and Shared memory flag "**true**" as the three parameters.



The performance gets lower and lower at the beginning, but then it starts to rise as we add more GPUs. This could give us different result when we go with big sized arrays, but I wrote the code in way that whenever the L1 shared memory doesn't fit the array then it automatically uses the global memory because of the L1 shared memory constraint.

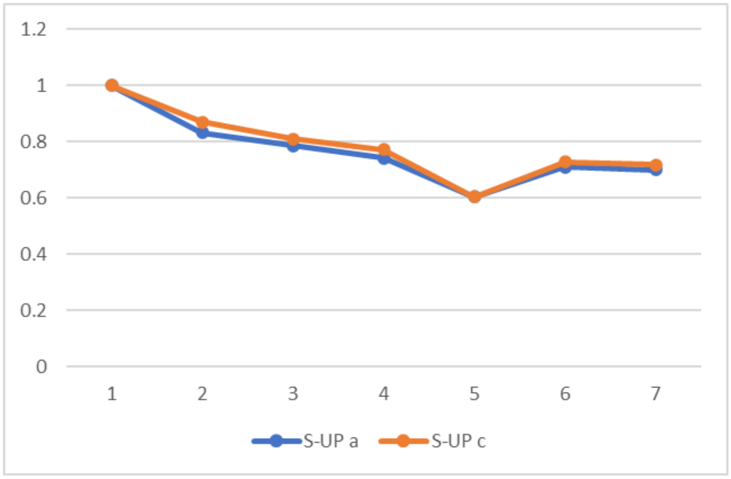
Strong Scaling with L1 Shared Memory

This time the execution is being run multiple times, with different GPU count, GPU relative array size and Shared memory flag "**true**" as the three parameters. The program is run five times for each GPU count.



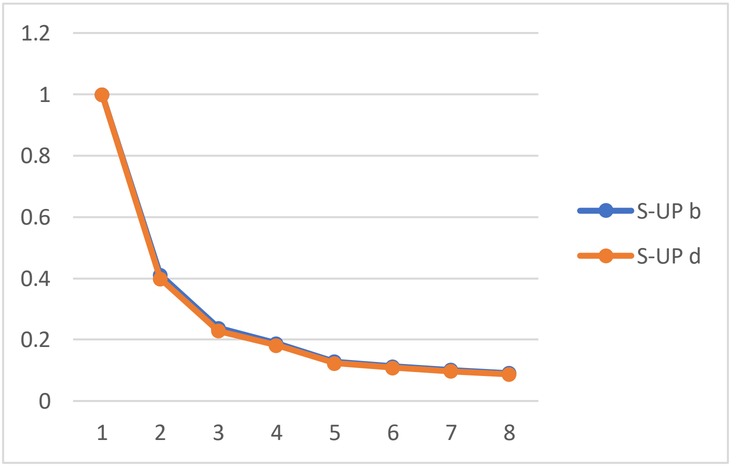
We have the same pattern as before where the performance goes down at the beginning and stays the same when we increase the input size with the number of relative GPUs.

Difference



Weak scaling difference between the use of L1 shared memory and without L1 shared memory. Where “S-Up a” is without shared memory and “S-Up c” is with shared memory.

There is some overhead by using L1 shared memory, but the algorithm needs more optimized way on using the shared memory to get even more overhead.



Strong scaling difference between the use of L1 shared memory and without L1 shared memory. Where “S-Up b” is without shared memory and “S-Up d” is with shared memory.

There is not much difference because the bigger the size gets the constraint of L1 shared memory becomes a problem. The first 3 GPU gets to use the shared memory, but the rest the size couldn’t fit in the shared memory.

Remarks:

1. I wrote a code where it is possible to run all 8 GPUs concurrently with the help of OpenMp. Where I start 8 OpenMp threads and each thread initiates GPUs with different device address and provides the section of the array to sort. I have commented the code for now and didn’t test it because I cannot send a job to the Slurm scheduler with GPU count more than 1.