MULTIPROCESSOR PROGRAMMING DV2606

LAB 2: GPU programming

Adnan Altukleh and Abdulkarim Dawalibi

Implementation:

Parallel odd-even sort implementation

In our parallel versions of the Odd-Even sort algorithm, we have utilized CUDA in C++ to harness the power of GPU processing, allowing for concurrent execution of operations to enhance the algorithm's performance. For the first version, we define a fixed number of threads (1024) and launch a kernel function, OES_kernel, which sequentially processes pairs of array elements with thread-level parallelism. The array elements are sorted through iterative comparisons and swaps, synchronized at each iteration using __syncthreads().

In the second version, we adapt the implementation to dynamically compute the number of blocks based on the array size and a modified thread count (500). The kernel, OES_kernel, now receives an additional phase parameter to handle odd and even phases separately. This version leverages block and thread parallelism, with each thread comparing and potentially swapping a distinct pair of elements in the array.

Both implementations follow a similar pattern in the main function. We initialize a large vector of random integers, allocate memory on the device, and copy the data to the device. The sorting is executed on the device using OES_kernel, which is called from the odd-even sort function in the host. After sorting is done, we synchronize the device, copy the sorted data back to the host, and free the GPU memory. The sorted status of the array is checked, and the total execution time is reported, demonstrating the efficiency of parallel sorting using CUDA.

Parallel Gauss-Jordan implementation

In our implementation of the Gaussian Jordan row reduction, we have utilized CUDA in C to leverage the capabilities of GPU processing for concurrent execution and enhanced algorithm performance. Our approach divides the matrix rows among a specified number of threads, defined as NUM_THREADS. Each thread is responsible for processing a designated portion of the rows, determined by the division N / NUM_THREADS.

The main function initiates the default values, reads options, and initializes the matrix. In the work function, we allocate memory on the GPU for matrix A, vectors b and y, and copy the data from host to device.

The row reduction process consists of several steps, each implemented as a separate kernel function:

- device_division: This kernel is responsible for the division step of the Gaussian Jordan method, normalizing the pivot elements of the matrix.
- device_elimination1 and device_elimination2: These kernels handle the elimination steps. They update the matrix rows by eliminating the coefficients, ensuring each thread operates on a distinct set of rows to avoid race conditions.
- add_device_elimination1 and add_device_elimination2: These additional kernels further assist in the elimination steps, updating vectors b and y accordingly.

After the completion of these steps, we synchronize the device to ensure all operations are completed, then copy the modified matrix and vectors back to the host. Finally, the allocated GPU memory is freed.

Measurements:

Odd-even sort task1;

-	Sequential	Parallel
Execution time(seconds)	7.4s	2.1s

Table 1: Presenting measurements for the Odd-Even Sort algorithm task 1 in both sequential and parallel.

Odd-even sort task2;

-	Sequential	Parallel
Execution time(seconds)	7.4s	0.9s

Table 2: Presenting measurements for the Odd Even Sort algorithm task 2 in both sequential and parallel.

Gauss-Jordan task3;

-	Sequential	Parallel
Execution time(seconds)	18.7s	1.3s

Table 2: Present measurements for the Gauss-Jordan in both sequential and parallel.

Discussion:

Odd-even sort

When we first started, it was a bit of a puzzle trying to wrap our heads around how the algorithm worked. We needed to figure out the role of each element and how everything played out in the sequential code. So, we tweaked the code a bit, adding print statements here and there. This helped us track what was happening step by step, making the whole thing a lot easier to understand.

Once we got the hang of the basics, we decided to take a stab at implementing the Odd-Even sort, but with a twist - using multiple blocks. Why? Well, it seemed more familiar because we had some examples from our lectures that used multiple blocks, and it just clicked with us.

So, we got down to it and put together an Odd-Even sort that worked with multiple blocks. But then came the real challenge - adapting this to work with just one block. At first, it was tricky, especially figuring out how to loop through everything in the kernel without messing up the data. We realized we could loop through by multiplying the thread number by two. This little trick helped us avoid overwriting data and made everything run smoothly.

Gauss-Jordan

Working on task 3, the 'Gauss-Jordan', was a bit easier since we had already worked on a similar algorithm in the previous assignment. We knew what to expect and just needed to make some changes to adapt it for CUDA.

But, as always, there was a twist in the tale. We hit a snag with synchronization, trying to figure out how to keep everything in line without tripping over race conditions, especially when the algorithm was setting elements to zero. We decided to create separate kernel functions to handle this part, stepping in right after the elimination process.