CS 629/729 GPU Programming
Lab 3

Spring 2024

Learning Outcomes

- Understanding the limitations of atomic operations.
- Understanding how to use recursive kernel calls to implement a parallel reduce type problem.
- Understanding how to use shared memory to improve the performance of a parallel reduction problem.
- Understanding how to use warp shuffling to perform a parallel reduction problem.
- How to compare and contrast approaches through benchmarking.

Lab Background

The file '`Student.dat`' contains records of student assignment marks. The purpose of the exercise is to calculate the maximum mark and the student who achieved this mark.

There is no need to create new source code files for each task. We will implement all of our code within the same source files provided.

You may run your code on either Google Colab or GPU cluster server.

To compile the code: nvcc lab3.cu -o lab3

Task 0

The starting code (`lab3.cu`) contains a function `readRecords` which reads the student assessment data from the provided file and stores it in an array of (`student_record`) structures. Because an array of structures is a poor choice of memory layout for GPU memory access so an additional structure of arrays (`student_records`) has been provided.

0.1 Complete the placeholder comment by implementing a method to convert the array of structure (`recordsAOS`) data into structure of array format. The resulting structure of array can be saved into the host memory allocated to the `h_records` pointer.

Task 1

The atomic operation we learnt in class can be used to enable exclusive access to memory, so we can use atomicCAS function to create a critical section for updating global variables atomically. A placeholder function has been created to allow you to implement calculation of the maximum mark and associated student using atomic operation. Complete the `maximumMark_atomic` function and kernel (in `kernels.cuh`) by implementing the following:

1.1 Implement the `maximumMark_atomic_kernel` kernel that uses the atomicCAS function. The kernel should CUDA atomicCAS function to create a critical section that compare each individual mark with a global maximum mark and then update the global maximum mark.
1.2 Configure appropriate number of thread blocks and number of thread per block for launching your kernel in the `maximumMark_atomic` function
1.3 You should launch your kernel on the default stream. Remember to synchronize with device immediately after the kernel launch.
1.4 Copy maximum mark and associated student from device to host with cudaMemcpyFromSymbol
1.5 Implement a CPU version validation function to validate your results.
1.6 Compile and execute the program to measure the performance of the kernel. Record your result in *Table 1*.

Task 2

A placeholder function has been created to allow you to implement an improved recursive method of calculating the maximum mark and associated student. Complete the `maximumMark_recursive` function and kernel (in `kernels.cuh`) by implementing the following:

2.1 Each thread should load a student record into shared memory using a `student_record` structure to hold the data (there is no penalty for array of structure reading from shared memory). Synchronize the threads to ensure shared memory is fully populated.
2.2 For evenly numbered threads, read in two records from shared memory (the record saved by the thread and the record saved by the proceeding thread). Compare these values to find the maximum and write the maximum value to the `d_records_result` array so that it is compact (resulting in an array with half the number of values as the input data in `d_records`).
2.3 Implement the host side code to repetitively call `maximumMark_recursive_kernel`. If you have not statically defined the shared memory size you will need to pass this as a third argument in the kernel launch configuration. The first call should have `NUM_RECORDS` in the input array. The final call should result in `THREADS_PER_BLOCK` values in the resulting array. You will need to swap the outputs from a previous kernel call to be the new input and vice versa. *Hint: use a temporary pointer.*
2.4 Move the final `THREADS_PER_BLOCK` values from the GPU to CPU memory (in the memory allocated to `h_records_result`).
2.5 Reduce the final `THREADS_PER_BLOCK` values using a loop on the CPU. Save the maximum mark in the `max_mark` variable and the associated student id in the `max_mark_student_id` variable. Execute your code in release mode and record the results in *Table 1*. Ensure that you get the correct result (as produced by the CPU validation function).

Task 3

The recursive method improves performance but can be drastically improved by reducing all of the values that are saved to shared memory. A place holder function has been created to allow you to implement an improved shared memory method of calculating the maximum mark and associated student. Complete the `maximumMark_SM` function and kernel (in `kernels.cuh`) by implementing the following:

3.1 As with the previous kernel, each thread should load a single student record into shared memory.
3.2 Implement parallel reduction algorithm that reduce all values in shared memory into one value.
3.3 For the first thread in the block write the reduced value to the `d_reduced_records` array so that the result array contains a record for each thread block.
3.4 Within the `maximumMark_SM` host function implement a call to the kernel you have just implemented.
3.5 Copy back the reduced values from the GPU to CPU memory (in the memory allocated to `h_records_result`).
3.6 Reduce the final values using a loop on the CPU. Save the maximum mark in the `max_mark` variable and the associated student id in the `max_mark_student_id` variable. Execute your code in release mode and record the results in *Table 1*. Ensure that you get the correct result (as produced by the CPU validation function).

Task 4

For the final exercise we will consider the performance implications of using warp shuffles rather than caching values in shared memory. Complete the `maximumMark_shuffle` function and kernel (in `kernels.cuh`) by implementing the following:

4.1 Implement a warp shuffle reduction to find the maximum assignment mark and associated student id. Shared memory is not required for this exercise. *Hint: Make sure that your shuffle instructions are not dependent on conditions as two threads must execute the same shuffle instruction to exchange information.*

4.2 Execute the shuffle kernel. Copy the final result back. Reduce the warp results on the CPU. Execute your code in release mode and record the results in *Table 1*. Ensure that you get the correct result (as produced by the CPU validation function).

Table 1 - Performance Results

method	**Execution Time (ms)**
`maximumMark_atomic`	
`maximumMark_recursive`	
`maximumMark_SM`	
`maximumMark_shuffle`	

Task 5 (CS 729 only)

A much larger data file (`Student_large.bin`) has been provided with 2^{20} student records. Adjust your implementation to handle this large number of records and record the execution time of all four kernels. To begin, you will need to modify the readRecords function to read the larger data file and modify `NUM_RECORDS` value.