Com 4521 Parallel Computing with GPUs: Lab 07

Spring Semester 2021

Author: Dr Paul Richmond

Lab Assistants: Robert Chisholm, Luis Rene Montana Gonzalez, John Charlton

Department of Computer Science, University of Sheffield

Learning Outcomes

- Understanding the limitations of atomic operations on the Kepler architecture.
- 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

It's the future… and the popularity of COM4521 has grown almost out of control. The file '`COM4521.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. The starting code in the file '`exercise01.cu`' contains a naïve implementation of an atomic approach which creates a critical atomic section to compare each individual mark with a global maximum mark. The performance of this kernel is *terrible* as it serialises access to the global variable for each student record. We are going to try a number of alternative solutions to see which is the most effective.

Exercise 01

The starting code contains a function `readRecords` which reads the student assessment data from the provided file and stores it in an array of (`student_record`) structures. We know from the previous week's exercises that 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.

1.1 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.

1.2 You should now be able to compile and execute the program in release mode to note the performance of the naïve atomic approach. Record your result in *Table 1*.

Exercise 02

There is no need to create a new project for this lab we will implement all of our code within the same sources files in a single project.

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). Synchronise 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 given by the atomic version).

Exercise 03

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 (SM) 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 a strided shared memory conflict free reduction. *Hint: consult the lecture material.*
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 given by the atomic version).

Exercise 04

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; Important *Note: For all of exercise 4 you will need to build your code for compute mode 3.5 minimum to have shuffle instruction support.*

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 given by the atomic version).

Table 1 - Performance Results

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

If you have completed the exercise you can get help with your assignment or alternatively you could consider how to improve the performance further for very large `NUM_RECORDS` values. This would require recursively calling kernels where there is sufficient work (i.e. enough threads) for the GPU rather than reading the data back after the first reduction. A much larger data file (`com4521_large.bin`) has been provided for this purpose with 2^{20} records. A solution for this will not be provided.