hw4

ECE 3803

Jeff Epstein

Contents

1 Introduction **1**

2 Assignment Part 1: Occupancy (5 points) **1**

3 Assignment Part 2: Binning with atomics (20 points) **2**

4 Assignment Part 3: Binning with atomics and shared memory (30 points) **3**

5 Assignment Part 4: Questions (5 points) **4**

6 Rules **4**

6.1 Development environment 4

6.2 Evaluation 4

6.3 Academic integrity 5

7 Submission **5**

1 Introduction

This assignment introduces dynamic block size and atomic functions.

2 Assignment Part 1: Occupancy (5 points)

In the previous assignment, we built a matrix multiply program and determined its optimal block size experimentally. CUDA provides a way to calculate the block so as to optimize theoretical occupancy. Although using this approach to improve theoretical occupancy does not necessarily lead to better performance, it's a good starting point for improving occupancy.

Your task is to modify your `matrix.cu` to use the `cudaOccupancyMaxPotentialBlockSizeVariableSMem` function for calculating the block size that achieves maximum potential occupancy for each of your kernels, and to use that calculated block size for launching your kernels.

Please read the official documentation on the `cudaOccupancyMaxPotentialBlockSizeVariableSMem` function.

Hints:

- You will need to write a function to pass as `cudaOccupancyMaxPotentialBlockSizeVariableSMem`'s fourth parameter that takes a potential block size and returns the amount of shared memory that your kernel needs. You will need such a function for each of your kernels.
- Because the block size is calculated at runtime, you will no longer be able to represent it as a precompiler macro. Find another way to communicate the block size to your kernels.
- As the needed amount of shared memory depends on the block size, you must dynamically allocate the appropriate amount of shared memory when launching your kernel.

3 Assignment Part 2: Binning with atomics (20 points)

Binning is the grouping of values according to some attribute of each value. Each value from a set is assigned to exactly one *bin*. Each bin has a unique number.

In this case, we are given an array of three-dimensional points, where for each point (x, y, z), each element $x, y, z \in [0, 1]$. The *binning function* will assign each point to subdivision of space. We want to write a program that will construct a new array where points belonging to the same bin (i.e. belonging to the same spacial division) are adjacent.

You are given starter code in `binning.cu`, containing a complete CPU-based implementation of such a program, in the function `host_binning`. This function uses the provided function `bin_index` to map a point to a bin. Your task is to write a GPU-based function named `device_binning_global` which produces identical results to `host_binning`. In this version, do not use shared memory in your implementation.

The particles are passed in an array `particles`. The bins each have a maximum capacity of `bin_size`, and are stored in an array `bins`, such that the contents of bin i are stored at `bins`$[i*$`bin_size`$] \ldots$ `bins`$[(i+1)*$`bin_size`$-1]$. The actual number of elements in each bin is stored in an array `bin_counters`, such that the number of used elements in bin i is stored at `bin_counters`$[i]$. Each element in the `bins` array is the index (i.e. position in the `particles` array) of a particle, or -1 if unoccupied. If a bin contains less than `bin_size` elements, the remaining values are ignored.

In the below diagram, we show two bins, bin 0 and bin 1. Each bin has a maximum capacity of `bin_size` of 8, although in practice `bin_size` will be larger. As shown in the `bin_counters` array, bin 0 contains 2 elements, and bin 1 contains 4 elements. In this case, points 0 and 3 are in bin 0, while points 1, 2, 4, and 5 are in bin 1.

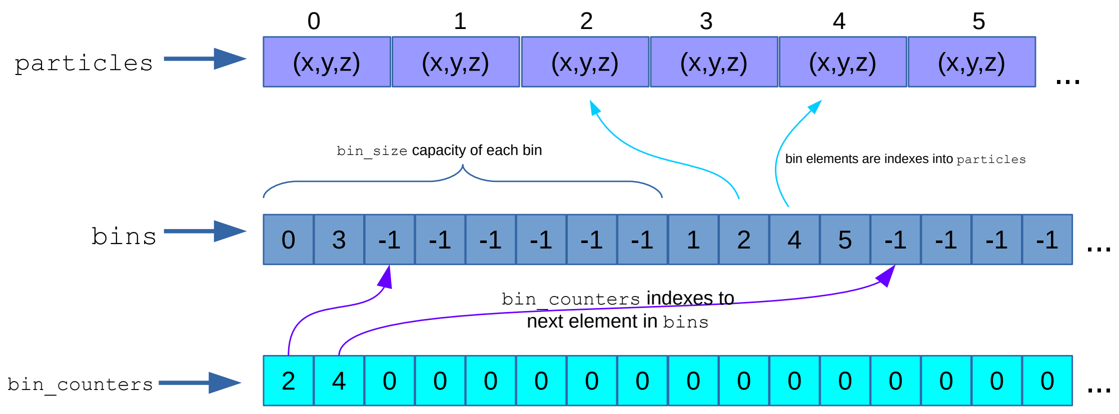

Figure 1: bin data structures

Use the `make` command to compile your program. Use the `make test-binning-global` command to compare the output of your code to the CPU version.

The `gridding` parameter describes the layout of the space into which we are allocating points. Each field stores the number of bits expressing the number of subdivisions. By default `gridding.x == gridding.y == gridding.z == 6`, which means that each of the three dimensions is divided into $2^6 = 64$ divisions, producing a total of $64 \times 64 \times 64 = 262144$ bins. See the `bin_index` function for more detail.

You must use at least one atomic function that we discussed. You may find their official documentation enlightening. You need to use atomics because your GPU-based implementation of binning requires multiple blocks to maintain a count of the number of elements in each bin. Do not use shared memory. Use `CHECK_ERROR` and `check_launch` where appropriate.

You can assume the bins will not overflow, i.e. there are fewer than `bin_size` elements in every bin. In other words, you don't need to check that `bin_counters[i] < bin_size`.

4 Assignment Part 3: Binning with atomics and shared memory (30 points)

Your task is to write a GPU-based function named `device_binning_shared` which performs the binning procedure, producing identical results to the other implementations. This time, use atomics and shared memory. You should use a *hierarchical* approach to binning:

- In the first pass, accumulate bin allocations into shared memory; in the second pass, merge the results.
- Due to limited availability of shared memory, the first pass might allocate particles to a coarse grid, before their final allocation.
- You can write this in just one kernel, but it might be easier to use two kernels called sequentially, where the second operates on the output of the first.

Use the `make` command to compile your program. Use the `make test-binning-shared` command to compare the output of your code to the CPU version.

As in the previous exercise, you will need to use atomic functions. You must use shared memory. Use `CHECK_ERROR` where appropriate.

5 Assignment Part 4: Questions (5 points)

Answer the following questions in a plain text file named `hw4.txt`.

1. What results did you get from CUDA's runtime calculation of block size in your revised matrix multiply program? Give a response for each of your kernels. Explain this result.
2. Describe in detail your solution for both of your binning programs. Some issues you should discuss:
   - What overall approach did you use?
   - How did you use atomic functions in your solution?
   - How did you use shared memory in your solution? Describe how you allocated and used shared memory.
   - How did you approach synchronization? Describe how you used `__syncthreads`.
3. Atomic operations safe communication between threads. In the past, we've discussed the use of `__syncthreads` to perform safe synchronization of threads within a block. Can we use atomic operations *instead* of `__syncthreads`? That is, can we replace an invocation of `__syncthreads` with code based on atomics? Provide a concrete example in code to support your opinion.

6 Rules

6.1 Development environment

Your code will be tested in our ICE environment, as described in the syllabus. Please test your code in that environment before submission. Your code must work with the operating system and CUDA development tool chain installed there.

You may use the CUDA libraries, as described in the ICE tutorial; as well as standard libraries that form part of the compiler. Do not use any other libraries. Do not use CUDA libraries or features that have not yet been introduced in this course.

6.2 Evaluation

Your submitted work will be evaluated on the following metrics, among others:

- your submission's fulfillment of the goals and requirements of the assignment. That is, does your program work?
- your submission's correct application of the techniques and principles studied in this course. That is, does your program demonstrate your understanding of the course material?
- your submission's performance characteristics. That is, does your program run fast enough?

Failure to satisfy the assignment's requirements, to apply relevant techniques, or to meet performance expectations may result in a grade penalty.

Only work submitted before the due date will be evaluated. Exceptions to this policy for unusual circumstances are described in the syllabus.

Ensure that your program compiles and runs without errors or warnings in the course's development environment. If your code cannot be compiled and run, it will not be graded.

In addition, you are obligated to adhere to the stylistic conventions of quality code:

- Indentation and spacing should be both consistent and appropriate, in order to enhance readability.
- Names of variables, parameters, types, fields, and functions should be descriptive. Local variables may have short names if their use is clear from context.
- Where appropriate, code should be commented to describe its purpose and method of operation. If your undocumented code is so complex or obscure that the grader can't understand it, a grade penalty may be applied.
- Your code should be structured to avoid needless redundancy and to enhance maintainability.

In short, your submitted code should reflect professional quality. Your code's quality is taken into account in assigning your grade.

6.3 Academic integrity

You should write this assignment entirely on your own. You are specifically prohibited from submitting code written or inspired by someone else, including code written by other students. Code may not be developed collaboratively. Do not use any artificial intelligence resource at any point in your completion of this assignment. Please read the course syllabus for detailed rules and examples about academic integrity.

7 Submission

Submit only your completed `hw4.txt`, `matrix.cu`, and `binning.cu` files on Gradescope. Do not submit binary executable files. Do not submit data files.

Acknowledgements

This assignment based on work by W. Randolph Franklin.