hw5

ECE 3803

Jeff Epstein

Contents

1 Introduction

In this assignment we explore parallel reduction and sorting techniques.

2 Assignment Part 1: Reduction (30 points)

Your task is to implement a parallel reduction function in CUDA, to find the maximum number in a `double` array.

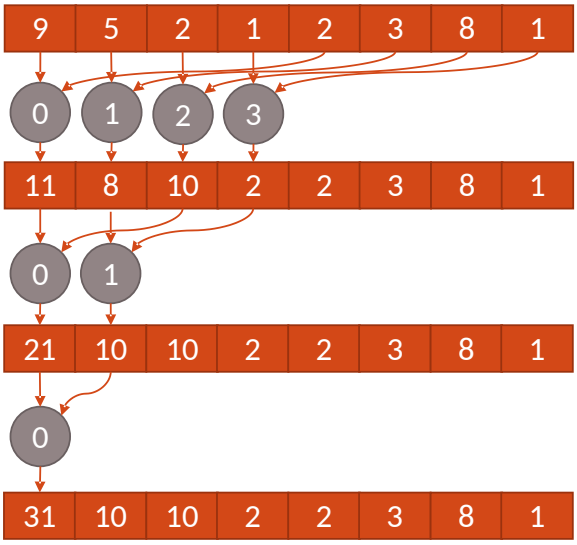

Figure 1: a parallel reduction, in this case of addition

Use the code from the slides as a starting point: refer to the example code for reduction. However, please note that the code from the slides is not necessarily complete or optimal, so you should make changes as you see fit. You can use techniques that we've discussed, included shared memory, static memory allocation, and atomics. Create any additional functions, kernel, and precompiler macros as necessary. Use `CHECK_ERROR` and `check_launch` where appropriate. Do not use cooperative groups for this assignment.

The strategy for your code should be similar to the strategy we discussed. You will need to efficiently do a parallel local reduction at the block level, then combine block-level results into global results. You can do this with just one kernel, but it may be easier to do it in two: one for block-level reduction, one for global reduction. Consider the following incomplete example, which omits considerations of error checking, memory allocation and copying, and other issues:

```
double reduce_max(double *nums) {
    double *block_level_result = ....;
    double *final_result = ....;
    const int num_blocks = ARRAY_SIZE/BLOCK_SIZE;
    ...
    // first pass: each block finds its local maximum, stores to block_level_result
    reduce_blocks<<<num_blocks, BLOCK_SIZE>>>(nums, block_level_result);

    // second pass: find maximum among block_level_result, store to final_result
    reduce_global<<<num_blocks/BLOCK_SIZE, BLOCK_SIZE>>>(block_level_result, final_result);
    ...
    return *final_result;
}
```

When you run the program, it will output its duration of execution, along with the duration of

C++'s CPU-based `std::max_element` function, for comparison. The program will verify that both algorithms produce the same results.

There are many ways to solve this problem. Faster performance will result in a better grade. Try to get execution time under 10ms.

Hints:

- You will probably need a global-memory array for storing the maximum value in each block.
- You can use CUDA's built-in `fmax` function, which will return the largest of its two `double` parameters.
- You can use the macro `DBL_MAX` (found in the C header `float.h`), which represents the largest possible `double` value. Its negation, `-DBL_MAX`, represents the smallest possible `double` value.
- We would like to use the `atomicMax` function, however CUDA does not have a version of this function for `double` values. Fortunately, we can implement our own `atomicMax` based on `atomicCAS`; the code for this has been provided.

3 Assignment Part 2: Radix sort (30 points)

Your task is to implement a parallel radix sort in CUDA. As part of this, you will have to implement parallel prefix scan.

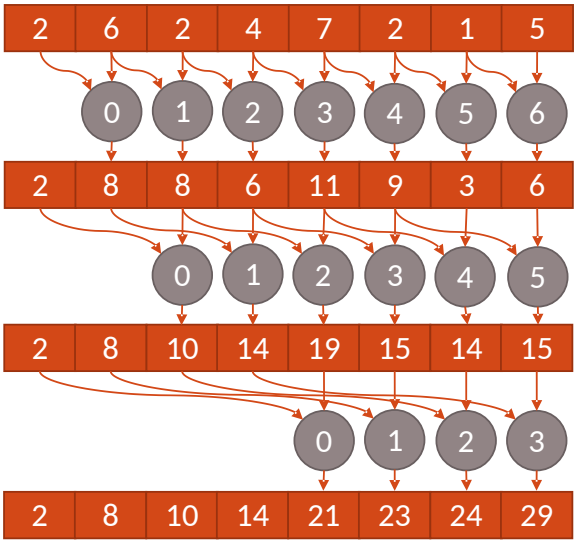

Figure 2: a parallel prefix scan

Use the code from the slides as a starting point: refer to the example code for both radix sort and prefix scan. However, please note that the code from the slides is not necessarily complete or optimal, so you should make changes as you see fit. You can use techniques that we've discussed, included shared memory, static memory allocation, and atomics. Create any additional functions, kernel, and precompiler macros as necessary. Use `CHECK_ERROR` and `check_launch` where appropriate.

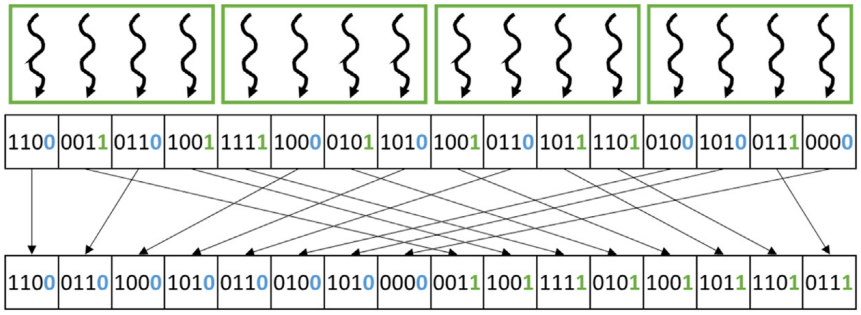

Figure 3: a parallel radix sort

The strategy for your code should be similar to the strategy we discussed. For prefix scan, you will need to efficiently do a parallel local scan at the block level, then combine block-level results into global results. For radix sort, you will use a 32-pass implementation, one for each of the 32 bits in the `unsigned int` data type. For each pass, you will invoke a prefix scan, and use its output to determine the destination location of each element for the next iteration. After the final iteration, all elements should be correctly sorted.

Note that your implementation will likely need inter-block synchronization. Atomic functions can help with this, but you will probably need more. For example, using the code from the slides, in order to implement prefix scan, you need to ensure that all previous blocks have stored their sum in global memory before later blocks read from that variable. `__syncthreads` won't help with this, as it synchronizes only threads in the same block. The best approach for this probably is to simply break your kernel into two smaller kernels, both called from host code: this way, you can be sure that the first kernel has completed before starting the next. Do not use cooperative groups for this assignment.

When you run the program, it will output its duration of execution, along with the duration of C++'s CPU-based `std::sort` function, for comparison. The program will verify that both algorithms produce the same results.

There are many ways to solve this problem. Faster performance will result in a better grade. Try to get execution time under 100ms.

4 Assignment Part 3: Questions (5 points)

Answer the following questions in a plain text file named `hw5.txt`.

1. In your implementation of radix sort, you built a parallel prefix scan. One challenge of this assignment is to ensure synchronization *between*, not merely within, each block. Describe your strategy for overcoming this challenge. Be detailed. If you used multiple kernels, describe their inputs and outputs, and why each separate kernel was necessary (i.e. why couldn't you have used fewer kernels). If you used another strategy for synchronization, describe in detail.
2. The synchronization strategies we've discussed so far are limited to:
    - `__syncthreads` for thread synchronization within a block.

- multiple kernels, for synchronization across the whole grid.
- and finally, atomic operations, which do not actually synchronize, but do facilitate inter-thread communication.

We have not (yet) discussed other mechanisms for synchronization between blocks.

Imagine that you are an Nvidia developer, tasked with creating synchronization primitives for their next GPU. Brainstorm an idea for a better way to handle synchronization. Answer the following questions:

(a) What kind of additional synchronization, beyond those we've already discussed, do you think programmers would want to use?
(b) How would programmers use your proposal? Consider the software API. Illustrate your proposal with code.
(c) How would your proposal be implemented at the hardware level? Consider the performance impact of your proposal.
(d) Specifically, how would your proposal make *this assignment* easier? Outline a solution to the radix sort problem that leverages your proposal.

5 Rules

5.1 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.

5.2 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.

5.3 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.

6 Submission

Submit only your completed `hw5.txt`, `max.cu`, and `radix.cu` files on Gradescope. Do not submit binary executable files. Do not submit data files.