Parallelism in Julia Spoke 18.S192
Homework 2

May 5, 2025

Please submit a PDF of your answers on GradeScope. This homework is graded. Use of ChatGPT or equivalent is allowed, perhaps encouraged (even for generating code, really!), but please write solutions to questions in your own words and comment on what LLM-generated code does and how you verified correctness, give credit, and please tell us, if you use an AI, which prompts for the AI worked and which ones did not work so well. (... and never never trust an AI without checking.)

Make sure to run all benchmarks long enough, go up to LARGE matrices and vectors: where possible 60s or 60,000ms computation time. You will not be able to see all effects on small matrices only. You may use the provided timing alternative code to `@belapsed CUDA.@sync` if it takes too long or write your own(if it is well tested and reasoned!), but make sure to @synchronize so you measure kernel runtime rather than launch time. If you write your own, demonstrate its output is reproducible over multiple runs.

Submit any answers to questions typed, not handwritten!

DO NOT LAUNCH NOTEBOOK KERNELS WITH MANY THREADS ON LOGIN NODES OR SHARED NODES. IF YOU ARE USING A SHARED GPU NODE, LAUNCH A LOW-THREAD NOTEBOOK KERNEL WITH AT MOST 32 THREADS.

Due Monday, May 5, 11:59PM

1. **Memory levels: compute-bound versus memory-bound.** In order to optimize code performance, it is important to understand whether your code is compute-bound or memory-bound.

   - Verify the GPUs you have by running `CUDA.versioninfo()`. Look up on the technical specifications online - what is the FP32 TFLOPS rate and the GPU memory bandwidth for this type of GPU?
   - For matrix multiply and matrix addition answer the following questions in a table, per matrix size in the rows and the questions in the column. Create separate tables for matrix addition and multiplication. Assume float32 datatype (4bytes).
     1. How many distinct bytes does the processor need to access?
     2. Assume perfect data-caching(i.e. every value only needs to be read once), and the algorithm's main bottleneck being memory. Use the GPU memory bandwidth to calculate theoretical expected speed.

3. How many bytes does the processor need to access in total? Count multiple times for multiple accesses of the same data.
4. Assume no data-caching (every value needs to be read from global memory the amount of times it is used in the algorithm), and the algorithm's main bottleneck being memory. Use the GPU memory bandwidth to calculate theoretical expected speed.
5. How many flops does the processor need to execute for this matrix size?
6. Assume the algorithm's main bottleneck is computations. Using the FLOPS rate, what do you expect the runtime to be?
7. What were your execution time for matrix multiply and matrix add built-in functions on HW2? (question 4b)

- Is matrix addition memory bound or compute bound? What about matrix multiplication? How well does your measured speed approach the theoretical speed?

2. **Memory access optimization.** We will continue working on the vector element doubling kernel you wrote in HW2 question 4(e). We will try different ways of distributing the workload across threads and blocks to investigate access patterns. For this question, use a vector of length 2^{28} of type Float32. In your previous homework, you likely assigned every thread one element to double and used NUMTHREADSINBLOCK= 64 (make this value non-constant for this question).

   - Create a parameter NUMELPERTHREAD and have every thread process a number of non-conseuctive elements: i.e. if the total number of threads over all blocks is n, have thread 1 access element 1, $n + 1$, $2n + 1$, ... Test a range of values for NUMELPERTHREAD and report at which value there is no longer a significant speed-up. Why do you think this is?
   - Keeping the NUMELPERTHREAD, vary the NUMTHREADSINBLOCK up and down - what is the optimal value? Why do you think this is?
   - Now we will make better use of the cache line and try at least 3 different strategies for memory access:
     - Have every thread access NUMELPERTHREAD consecutive elements. Vary NUMELPERTHREAD and report on performance.
     - Calculate the total number of blocks given NUMELPERTHREAD and NUMTHREADSINBLOCK. Seperate the vector into this many segments. i.e. if you have two blocks, block one would access elements 1 to n/2 and block two elements n/2+1 to n. Have each thread access non-consecutive elements within the block's section. I.e. have thread 1 access elements 1, NUMTHREADSINBLOCK+1, 2NUMTHREADSINBLOCK+1, ... Vary NUMELPERTHREAD.
     - Have every thread access 4 consecutive elements within a block section, and find the next 4 elements non-contiguously. I.e. have thread 1 access elements 1,2,3,4, 4NUMTHREADSINBLOCK+1, 4NUMTHREADSINBLOCK+2, ... Vary NUMELPERTHREAD and change 4 elements to 2 or 8.

Write a kernel for each memory access pattern and benchmark them for different parameter values. Which access pattern is the fastest? Why do you think?

- Change the algorithm: we will now double every elements 1 to 16, 33 to 48, ... and square every element 17 to 32, 48 to 64, ...
  - Why might this prompt be more challenging to optimize?
  - Test at least two different element access strategies and vary at least one parameter in each. What is the fastest and why? Bonus points might be awarded to insightful, creative or performant solutions!

3. **2D access patterns.** Data is saved linearly on hardware and threads are linear as well. 2D representation is only for the programmer's convenience. Julia is column-major: columns will be stored consecutively one after the other. Copy the naive matrix multiply algorithm we discussed in class and consider a square matrix of size 2^{13}. The standard algorithm uses a square block size of size (16,16).

   - Test at least six other block sizes: (1,16*16), (16*16,1), (8,8), (32,32), and define two more block sizes yourself. Report the performance on each and explain why you think this is the case.
   - Have every thread handle more than one element, vary which elements every thread is handling and how many. Test at least every thread handling (2,2) elements, (1,4) and (4,1). For this subquestion, position the elements every thread is handling consecutively.
   - Use your experience with the previous question to propose at least one other way to divide the elements across the threads. Report on performance.

4. **Start of Parallel prefix kernel.** In HW1 question 3 you implemented a parallel prefix sum algorithm. Due to data dependencies, the algorithm had to be split up into parts that can run in parallel in between which we synchronize. Remember that on a GPU you can synchronize between threads in a block, but the only way to synchronize between blocks is to launch a new kernel!

   - Write a naive kernel for parallel prefix sum, i.e. do not use shared memory, only global memory. Test the kernel at different sizes (only powers of two larger than a 2^8) and make sure it works.
   - Based on your experience with optimizing global memory access, propose and test at least one other way to structure this algorithm that you think might be performant. Test this against your first implementation at a large enough vector size to occupy the GPU (ideally 2^{28}, but use a smaller size if this takes too long).
   - What makes data access optimization more complex for parallel prefix than for element doubling?
   - Rewrite the kernel making use of workgroup-shared memory (declare `@local eltype(input)` and thread-exclusive register memory (`@private eltype(input) (vectorsize)`). Try to conceptually avoid bank conflicts.

5. **Seam Carving in Parallel**

- Familiarize yourself with seam carving if you are not familiar with it already. In particular, choose an image and comment on the seams you observe in that image using the attached notebook (embed the image in your submission).
- In the attached Pluto notebook fill in the blanks for the `NaiveTasks` method for `least_edgy` by parallelizing each row using OhMyThreads.jl
- Complete the implementation of the triangle based parallelism method (`TriangleTasks` method). Debug carefully with print statements as needed. If you encounter any issues or need any help debugging this come to office hours or email the TA.
- Parallelize the triangle tasks method and benchmark vs the other two methods. Comment on what you observe. Ensure you are using relatively large triangles. The code for benchmarking is commented out just below the implementations.
- Submit your Pluto notebooks.