Parallelism in Julia Spoke 18.S192
Homework 2

April 20, 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.

Submit any answers to questions typed, not handwritten!

We strongly encourage you to use `OhMyThreads.jl` for these problems, as it allows you to change the number of tasks you are running without restarting Julia and changing the number of threads.

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

Due Monday, April 28, 11:59PM

1. **Using the Perlmutter Supercomputer. You must follow the attached instructions to use Perlmutter correctly for this homework.** We saw in class how to submit computations on a supercomputer. Re-use your code from HW1 and re-run it for benchmarking. Please submit the ipython notebooks and benchmarking results, specifically:

   0. Run `Threads.nthreads()`. We hope you see 128. Let us know if you do not.
   1. **HW1 Question 1:** A table of the absolute execution time for the matmul and matadd operations with and without allocations in function of input matrix size. Use the code from question HW1 1a-c.
   2. **HW1 Question 2:** Two tables of of the absolute execution time of threaded Vectordouble, (HW1 question 2a) and Matrixsquare (HW1 question 2b), with a single task and multiple tasks, in function of the input matrix size. Play with

the number of tasks you are starting. For part (b), use matrices of size at least 1024×1024. Use the code from HW1 question 2.

3. **HW1 Question 3:** Two tables of the absolute execution time of parallel prefix sum (HW1 question 3a) and parallel prefix matrix multiply (HW1 question 3b), with a single thread and multiple threads, in function of the input matrix size. For part (b), use matrices of size at least 1024×1024. Play with the number of tasks you are using. Use the code from HW1 question 3.

2. **Playing with Tasks on Perlmutter** Matrix multiplication is the backbone of HPC, a significant fraction of the codes run on Perlmutter will be plain old matrix multiplies! We're going to explore the performance of tasks using matrix multiplication on Perlmutter. Use the following code for this section:

```
  function matmul!(C, A, B, nt = Threads.nthreads())
    m, k = size(A)
    k_check, n = size(B)

    # Validate dimensions
    k == k_check ||
      throw(DimensionMismatch("Inner dimensions must match"))
    size(C) == (m, n) ||
      throw(DimensionMismatch("Output matrix has incorrect dimensions"))

    # Initialize result matrix to zeros
    fill!(C, 0.0)

    # Perform the matrix multiplication with 3 nested loops
    OhMyThreads.@tasks for i in 1:m
    # set-blocks can be used to change parameters of the @tasks call!
      @set begin
        ntasks = nt
    end
      for j in 1:n
        for l in 1:k
            C[i, j] += A[i, l] * B[l, j]
        end
    end
    end
return C
end
```

1. Benchmark the code above and set `n={1,2,8,32,64,128}`. Report your results in a table for small matrices (100×100) and large matrices (4000×4000) (this isn't

really that large in the grand scheme of things, but since you're testing with 1 task it may take a while). Comment on what you observe.

2. There are 3 loop indices: `for i in 1:m, for j in 1:n`, and `for l in 1:k`. Experiment with different permutations of the order of these for loops (specifically what happens if you swap `for i in 1:m` with `for j in 1:n`. Try several of these permutations and report your results for 4000×4000 size matrices. Can you identify any reason for performance differences you may observe (hint: look up column-major order)? You may ask AI for help but use your own words.

3. **Amdahl's law.** Parallelizing serial programs is only possible up until serial bottlenecks. Compare the speed-up from a single task to multiple tasks on a supercomputer in function of the matrix size for vectordouble, matrixsquare, parallel prefix sum and parallel prefix matrix multiply.

   - In tables, report the ratio of the single-threaded and multithreaded computation time in function of the matrix size for vectordouble, matrixsquare, parallel prefix sum and parallel prefix matrix multiply. Make a plot to show this.
   - What ratio are the speed-ups for vectordouble, matrixsquare, parallel prefix sum and parallel prefix matrix multiply approaching as input size increases? If you do not see convergence, increase matrix size.
   - Use Amdahl's law to determine the fraction of the program that is parallelizable.
   - Did you expect this result? Why (not)? If not, make some educated guesses what might be causing this.

   Switch to a (shared) GPU node for the remaining quesetions! Use a low thread kernel, DO NOT USE A HIGH THREAD KERNEL ON SHARED NODES!

4. **Introduction to GPUs.** In the Perlmutter supercomputer, you have a few GPUs available. Run `nvidia-smi` to see their specifications. By allocating a GPU matrix and executing a mathematical function, julia is behind the scenes automatically calling CUDA or CUSOLVER or CUTLASS (Nvidia-optimized libraries) to execute your function so you don't need to write GPU kernels for every function yourself. To call the underlying functions on the hardware, we use the CUDA.jl package.

   (a) Allocate a matrix on the GPU and benchmark matrix additon and multiplication on the GPU, similar to question 1.1.

```
using CUDA
# define n =
eltype = Float32 #specify the data type (the default is Float64)
A = CUDA.randn(eltype,n,n)
B = CUDA.randn(eltype,n,n)
C = CUDA.zeros(eltype,n,n) # (or C = similar(A)  also creates a )
mul!(C,A,B) #1. benchmark matmul for different n
```

```
C = A * B   #2. compare with the line above, what is the difference
            #in speed and why?
C .= A .+ B #3. benchmark matadd  for different n
C = A + B   #4. compare with the line above, what is the difference
            #in speed and why?
```

To benchmark this correctly use `@belapsed CUDA.@sync C=A+B`. Submit your code and a table of the absolute execution time for the matmul and matadd operations with and without allocations in function of input matrix size.

(b) Generate the following plots and answer the questions. Use the results from question 1.1 for CPU timing and question 3a for GPU timing. Please use very small n and large n. Execution time for large n should be noticeable, but keep in mind the memory limits of each GPU on Perlmutter (typically 40GB).

- **Matmul and matadd without allocation:** Plot the CPU and GPU execution times of matmul and matadd without allocations. What is faster at which matrix size and why do you think this is the case? Is there a difference between when matmul and matadd become faster on the GPU? Why (not)? Tip: plot the matmul in one color and matadd in another color, and CPU in dashed vs GPU in solid lines. Make the plot logarithmic if helpful.
- **Matadd with and without allocations:** Plot the CPU and GPU execution times of matadd with and without allocations. Do you see a difference in how much impact allocations have on the CPU or GPU? Tip: plot the allocating function in one color and non-allocating function in another color, and CPU in dashed vs GPU in solid lines. Make the plot logarithmic if helpful.

(c) Test for different matrix size `@belapsed` without `CUDA.@sync`. What do you see? Look up the use of the synchronizing function and explain why you need it.

(d) Element-wise operations are in julia defined by a dot ., for example $C. = A. + B$ adds A to B element-wise and saves the results of this calculation element-wise in C. On the GPU, the element-wise addition is dispatch to vendor-optimized libraries or julia-native implementations. In HW1 question 2a you wrote a function for doubling each element of a vector. Allocate a gpu vector (one line function) and use element-wise operations to achieve the same (one line of code).

- Submit your code.
- Benchmark this for different sizes. Provide the table with the results.
- Generate a plot comparing these results to the results you obtained in question 1.2 of this homework for vectordouble.
- What do you see: at which sizes does which function take over in terms of speed? Why do you think this is the case?

(e) Let us take a look at a KernelAbstractions.jl kernel that executes the same operation. The following code is a skeleton, fill in the blanks. We will assign a single thread for a single element in the input matrix. Refer to https://juliagpu.github.io/KernelAbst for more detailed documentation if needed.

```
using KernelAbstractions,CUDA
backend=KernelAbstractions.get_backend(CUDA.zeros(1))
elty=Float32
const NUMTHREADSINBLOCK= 64 #threads per cuda block

@kernel function mykernel!(size_in::Int, input::AbstractGPUArray)
 idx = # calculate idx using (@index(Group, Linear)
       #and @index(Local,Linear)
    if (g<size_in)
        value = #load value into register from input
        #do operation on value
        input[idx] = value #save result back in global memory
    end
    return
end

reduction!(size_in::Int, A::AbstractGPUArray) =
          mykernel!(backend, (NUMTHREADSINBLOCK))(size_in, A,
          ndrange=size(A))
```

In the last line, ndrange determines the TOTAL number of threads needed over all blocks. Given NUMTHREADSINBLOCK, the GPU kernel calculates how many blocks it needs to generate the total number of threads. Every block gets launched with the same number of threads. Submit your code and test that it works.

(f) Benchmarking GPU kernels at small matrix sizes is challenging. Instead of belapsed, use the following code to benchmark KernelAbstractions kernels. (At larger vector sizes `@belapsed` should give the same results.)

```
    function benchmark_ms( myfunc, args...;kwargs...)
    elapsed=0.0
    best=100000
    i=0
    while(elapsed<200.0 || i<2)
        CUDA.synchronize()
        start = time_ns()
        for i=1:20
            myfunc(args...;kwargs...)
        end
        CUDA.synchronize()
        endtime = time_ns()
        thisduration=(endtime-start)/1e6
        elapsed+=thisduration
        best = min(thisduration/numruns,best)
        i+=1
    end
    return best
end

#example usage
benchmark_ms( mykernel!,length(a), a) #warmup first
timing= benchmark_ms( mykernel!,length(a), a) #use these results
```

Submit the following:

- Submit your code and table with results for Benchmark your element-wise code from question (e) against results you got from your custom kernel-code in question (f).
- A plot showing both.
- Which one is faster at which matrix sizes? Why?