Homework 4

Dr. Michal A. Kopera

ME 471 / 571, Spring 2019 **due date : May 8th**

Homework done by: Arash Modaresi Rad

This is the final homework for the course, and it is an individual assignment. You can score a maximum of 30 points, unless you complete the extra credits.

The homework consists of two parts. In the first one I ask you to test your CUDA kernel for 1D derivative computation. For an extra credit, you can instead do the same analysis on the 1D shallow water kernel, if you managed to write it.

In the second part, I ask you to evaluate the performance of two codes for 2D shallow water equations: MPI and CUDA. You wrote the MPI code for HW3, so please use it, and I have provided for you the CUDA code for the same problem.

Unless marked otherwise, all questions are mandatory for both ME 471 and ME 571 students. Questions marked with (ME 571) are mandatory for 571 students, and extra credit for 471 students.

Part 1: 1D derivative computation using CUDA (10 pts) 8 pts

During lectures, you have received the instructions on how to create 1D derivative kernel (and subsequently 1D shallow water code). Please evaluate the performance of the kernel of your choice (either derivative or shallow water) by answering the following questions.

Answer: the codes for shallow water 1D for both global and local and similarly for derivative is provided in zip file under the name “1D”

1.1. Show that your kernel performs correctly for different number of points (nx) and different number of threads per block. I want to see that your code gives a correct result regardless of whether it runs with one block and many threads, or several blocks.

Difrrent combinations of problem sizes and threads were preformed in this task and the changes were only applied to .qsub file and is located in 1D folder in global subfolder.
This bash script uses "bc" to perform the division required for DT.

So for evaluating different combinations of threads using a kernel with Global memory I used a fixed problem size with properties being Nx = 100,000 DT = 8E-06 X_LENGTH = 10 T_FINAL = 0.2

In this case different combinations of thread and blocks were used stating from 16 threads and moving up to 1024 as multiples of 16 and they all preformed currectly.

what does this plot show exactly? I assume it is water height, but you need to label your axis

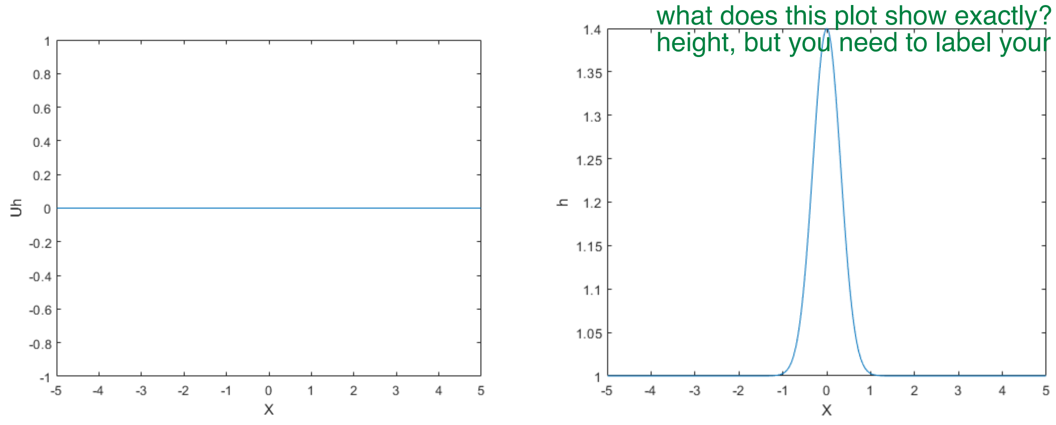


Figure 1) the initial condition of shallow water equation

likewise here, what is this figure showing?

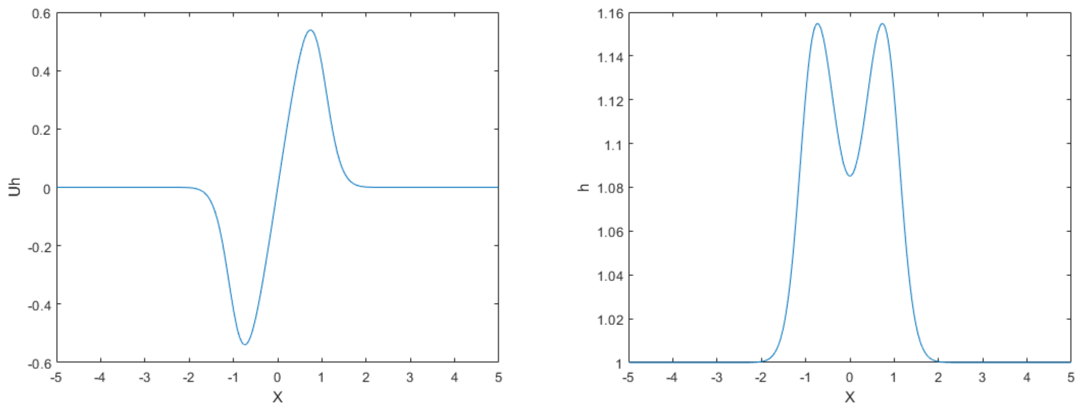


Figure 2) the final condition of shallow water equation

1.2. On a plot, show how the performance of the code (in terms of run time of the kernel) depends on the number of threads per block. Is there an optimal setting for this problem? Make sure you choose large problem size (N at least 100000).

The results were quite unexpected as the optimum number of Threads for this Kernel is not actually a product of 32 and anywhere between 608 and 672 threads has the best performance. It also appears that we cross a threshold after 32 threads meaning that any number of thread greater than 32 and blow 1024 would perform similarly with minor differences in speed. This further indicates that the ???

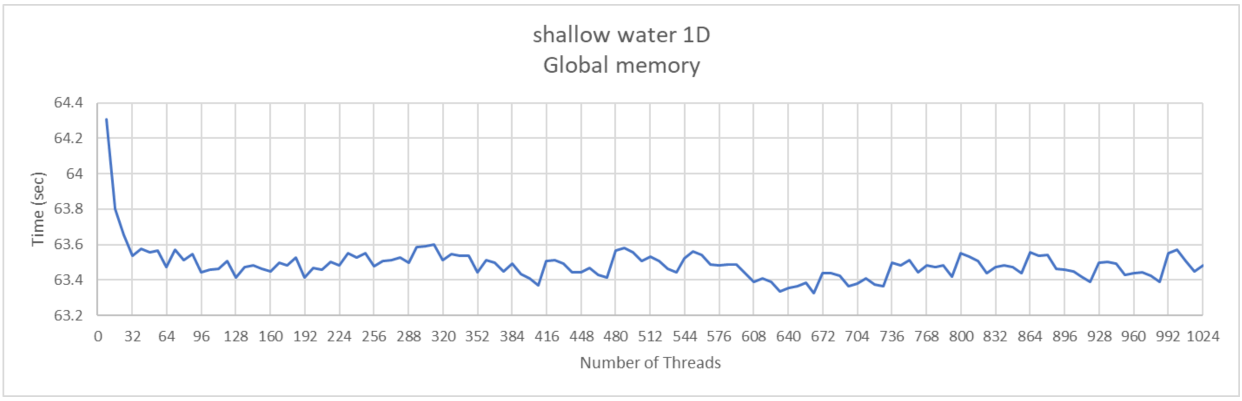


Figure 3) Shallow water 1D with global memory implementation

1.3. **(ME 571)** Repeat the exercise 1.2 with the version of the kernel where you use shared memory instead of global memory. Is there an improvement in performance? Why or why not?

For evaluating different combinations of threads using a kernel with local memory I used a fixed problem size with properties being Nx = 1,000,000 DT = 8E-07 X_LENGTH = 10 T_FINAL = 0.2.

This is 10 times greater for both Nx and Dt compared to the global memory implementation, and as shown in figures 3 and 4, although the problem is bigger the time it took to compute was half, which means using local memory extremely boosts the performance.

On the other hand, it can be seen from Figure 4 that the choice of number of threads doesn't have that much impact on speed of solving this Kernel. The mid ranges appear to be optimum whereas, the 32 threads and 900-1024 are slow.

Better comparison would be to use the same problem size for both global and shared memory, otherwise you are not comparing apples to apples.

if you want to compare two graphs, it is best to plot them on one figure

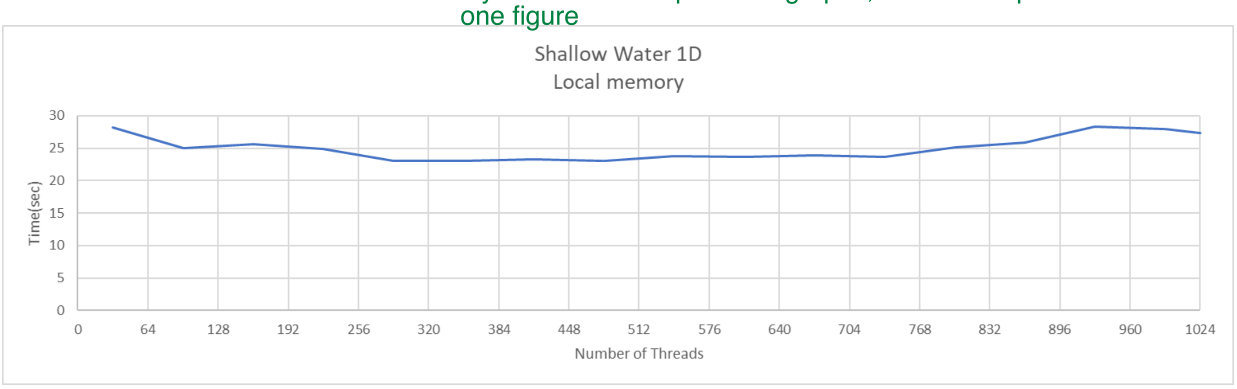


Figure 4) Shallow water 1D with local memory implementation

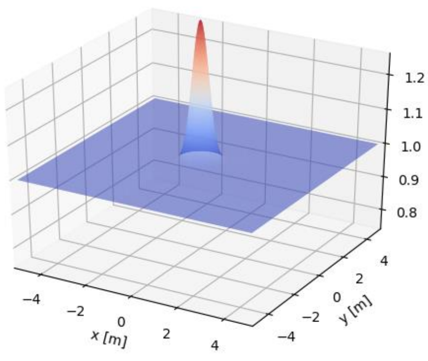
The use of local memory on derivative (dudx.cu) fie did not improved the speed of computation significantly, most likely because there is less computation compared to shallow water 1D.

18 pts - you need to work on your data presentation, the discussions are good!

Part 2: Comparison of the 2D shallow water equations in CUDA and MPI (20 pts)

2.1. Set up an experiment where you compare two versions of the shallow water code. I want you to pick the most optimal configuration in terms of number of cores you have found in HW3 for the MPI code. Run this configuration for a range of problem sizes (starting at 400x400 up to 8000 x 8000, pick at least four problem sizes in that range), keeping in mind that increasing resolution means decreasing time-step (dt) accordingly. Make sure you keep the domain size and final time constant. Run this test on R2, keeping in mind that it has more available cores per node, and many more nodes, but also is a research machine with many users actively running their computations, so wait times can be long.

The results of the shallow water 2D on R2:



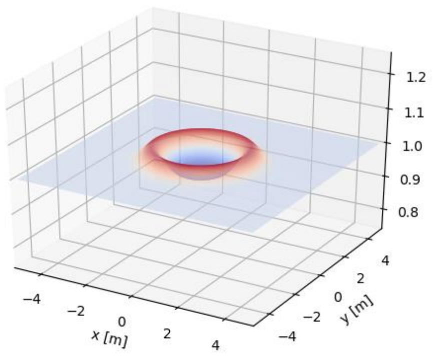


Figure 5) The left most figure is the initial wave at t=0s and the most right figure is the final wave at t=0.2s.

you need to work on your plots, use Python or Matlab maybe?

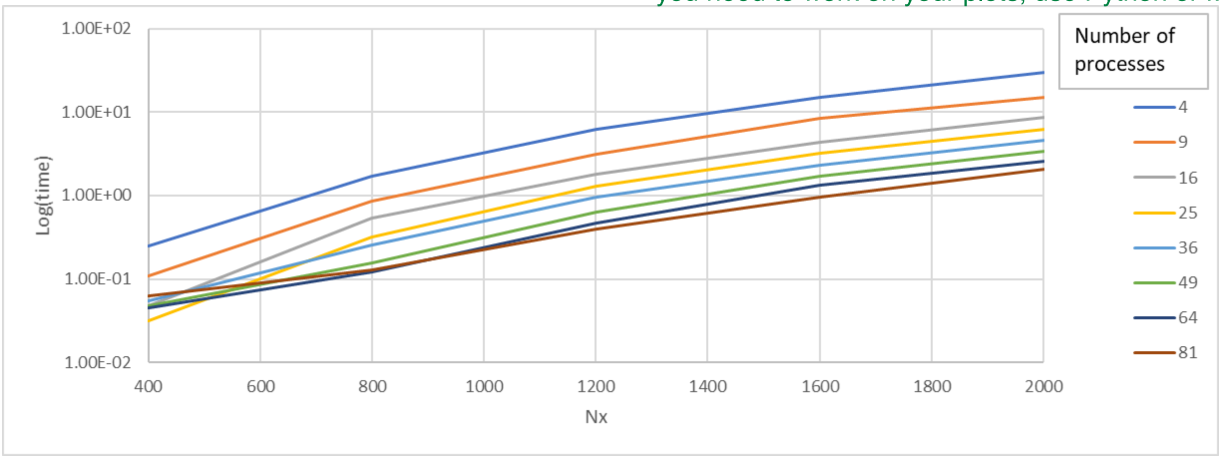
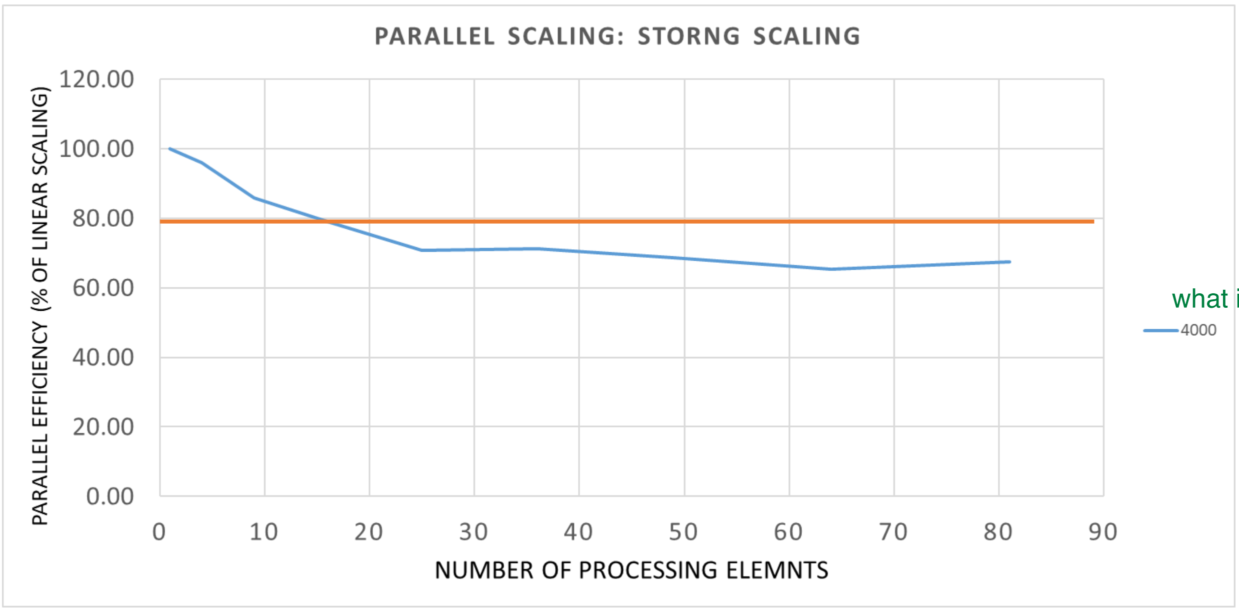


Figure 6) Log of time versus Nx for different number of processes on R2

Strong scaling



what is 4000?

Figure 7) The Strong scaling in R2

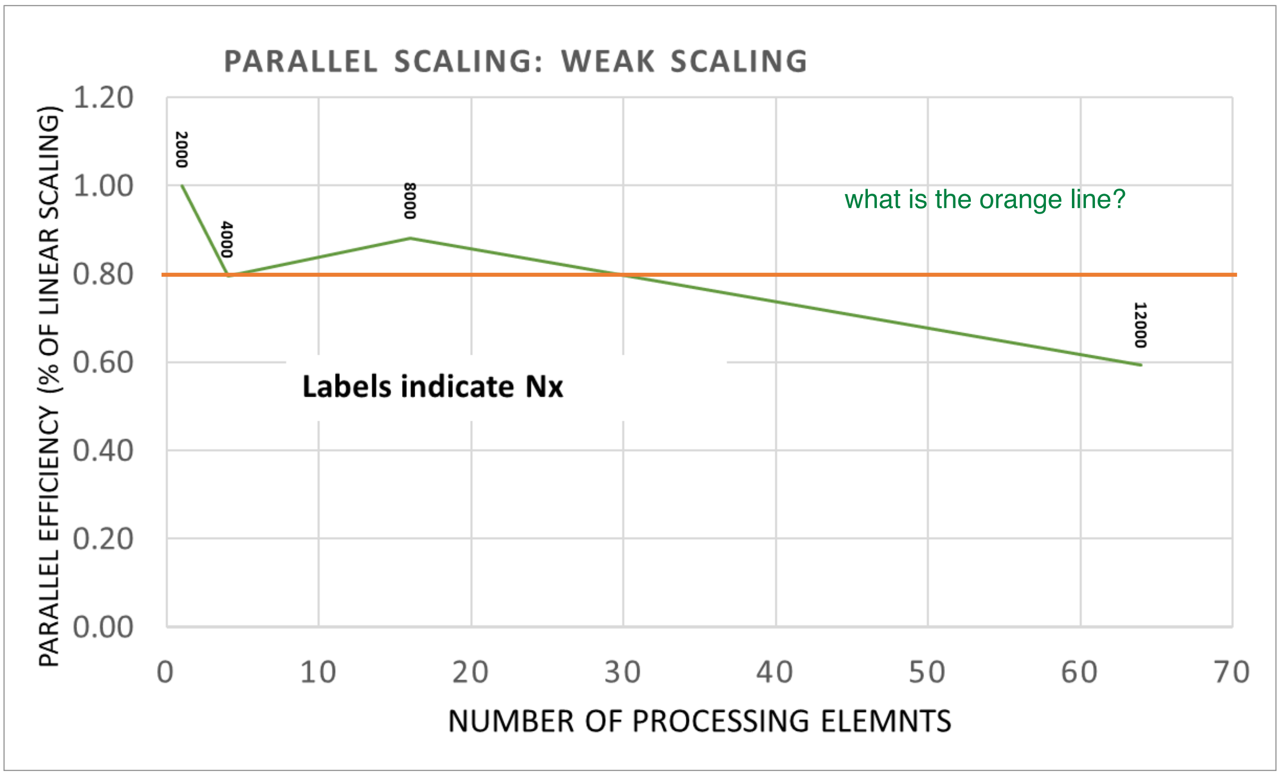


Figure 8) The Week scaling in R2

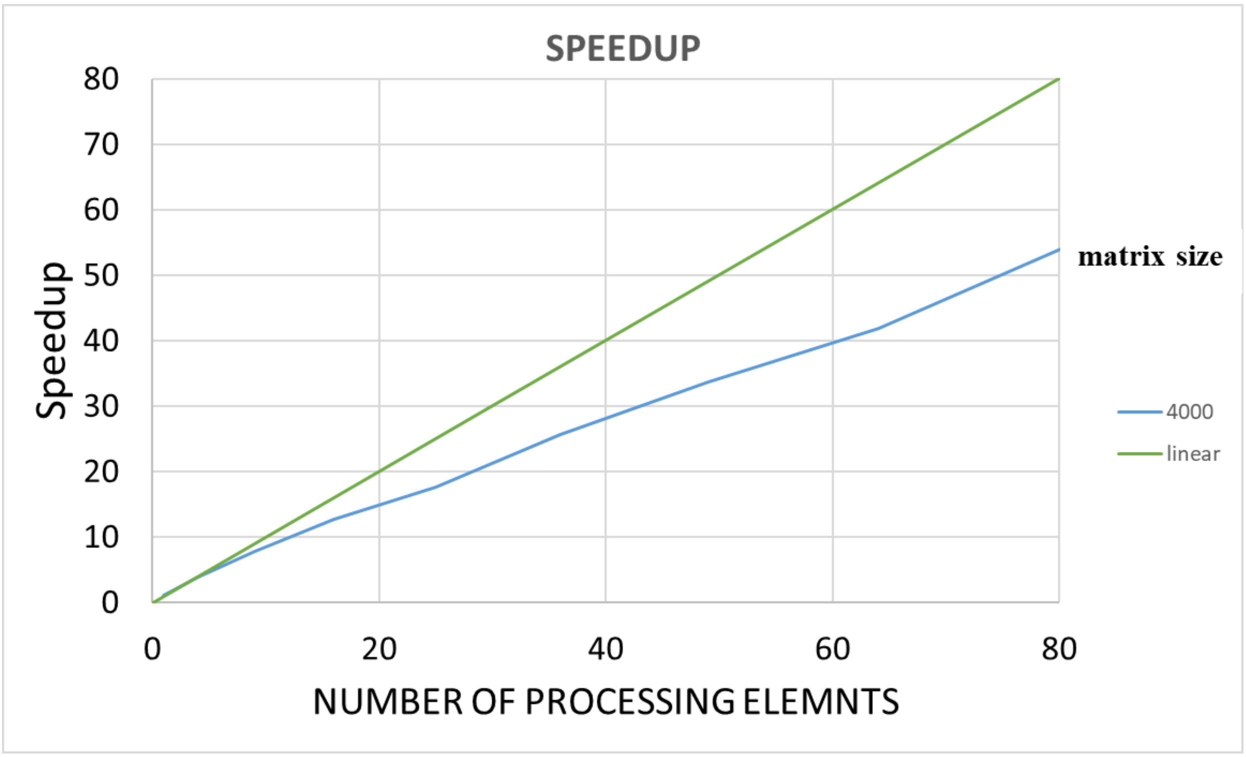


Figure 9) The speedup in R2

The results of the shallow water 2D on R1:

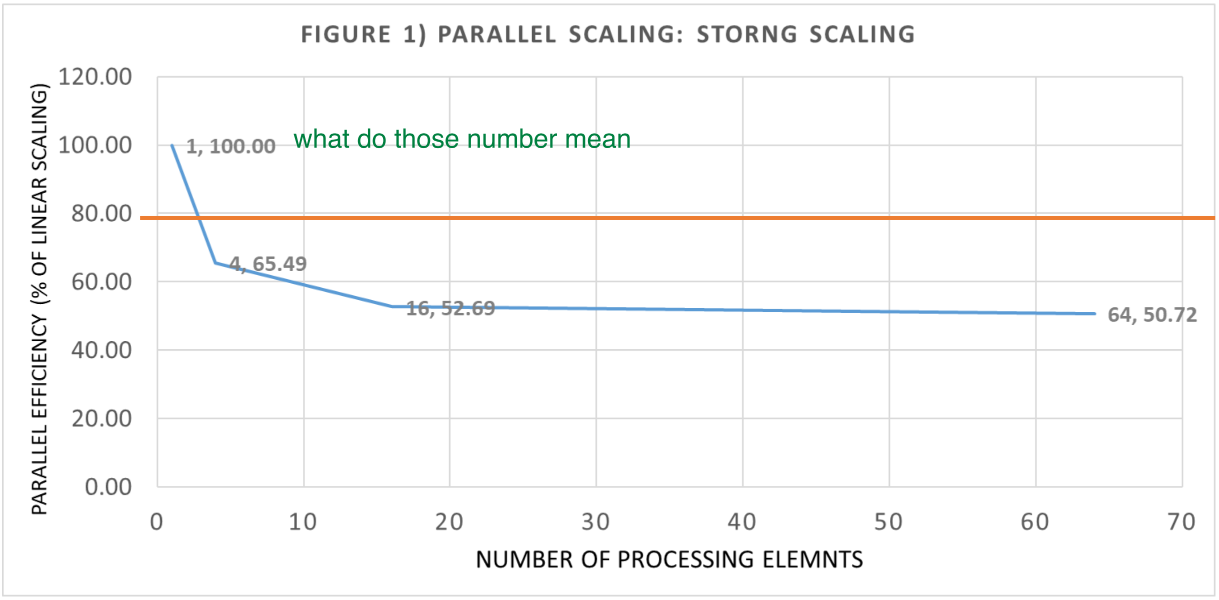


Figure 10) The Strong scaling in R1

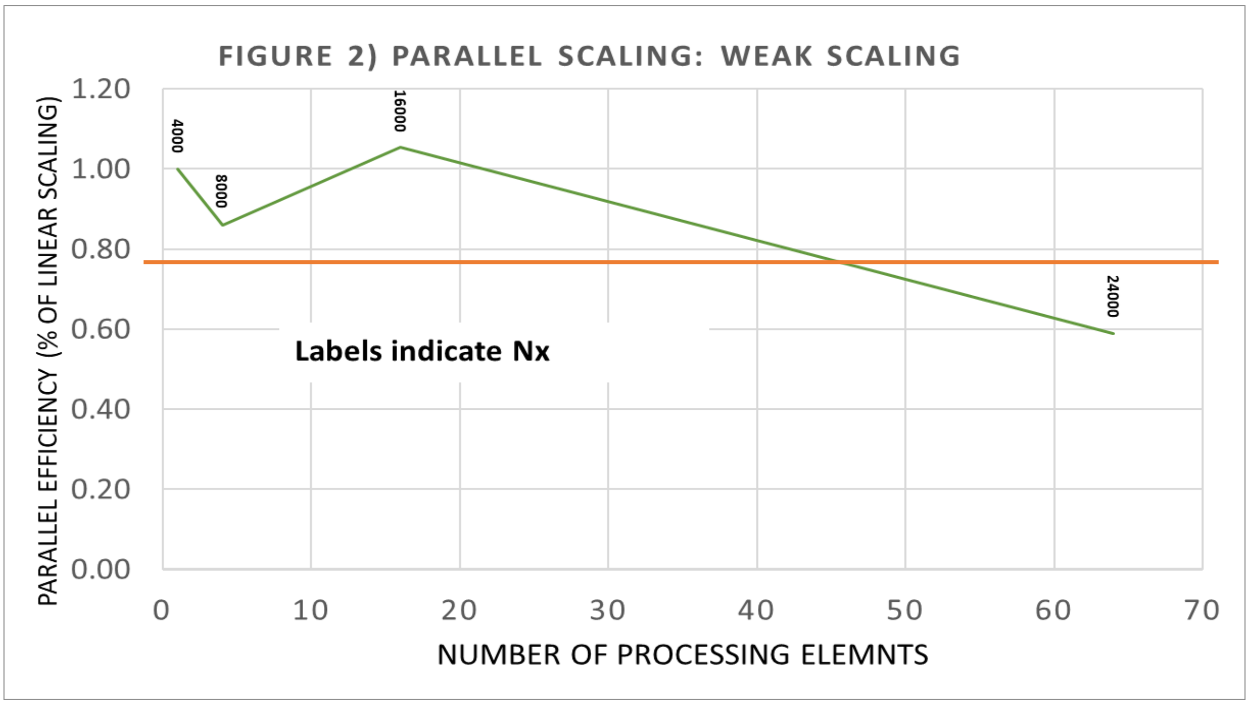


Figure 11) The week scaling in R1

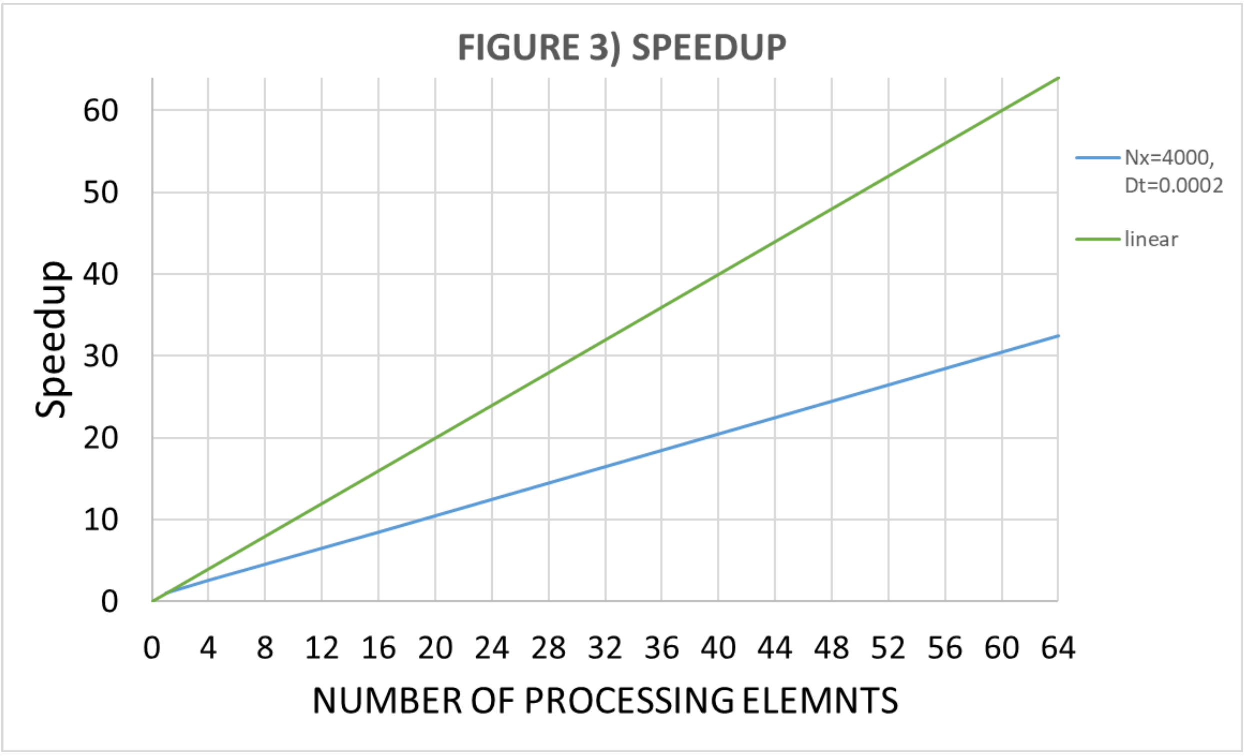


Figure 12) The speedup in R1

Based on figure 6 at very small problem sizes 16 number of cores has the best speed but after that the 64 and 81 number of possessors are the fastest.

The strong scaling results are much improved in R2 as we obtain more than 80% efficiency up to 9 cores and this was not the case in R1 as we never achieved and efficiency higher than 80% (4 cores had 65% efficiency)

The week scaling has not improved on R2 and surprisingly a bump still exists at 16 cores! This was also observed in R1 and it appears that 16 cores for week scaling of this code is the most efficient one.

Finally, the speedup has improved in R2 compared to R1 and is almost twice the order of magnitude. Results, indicate that week scaling efficiency is not different on R1 and R2 but the strong scaling is improved on R2 suggesting that maybe stronger CPUs have greater impact on strong scaling efficiency.

I would say that better network would improve scaling, faster processor with the same slow network would probably make the scaling worse - less compute time + the same communication overhead = worse speedup

2.2. Using the same problem configurations (nx, dt, x_length, t_final), run those test using a GPU node on R2 and record times. You will have to decide on your choice of THREADS_PER_BLOCK_X and THREADS_PER_BLOCK_Y variables, which decide how many threads per block in each direction to use.

Remember that the maximum number of threads per block (total) is 1024, so product of THREADS_PER_BLOCK_X and THREADS_PER_BLOCK_Y cannot exceed that number. Compare both results on a plot (time to solution vs problem size). Which code gets to solution faster? Is the same code always the fastest one for all problem sizes? Try to explain your result.

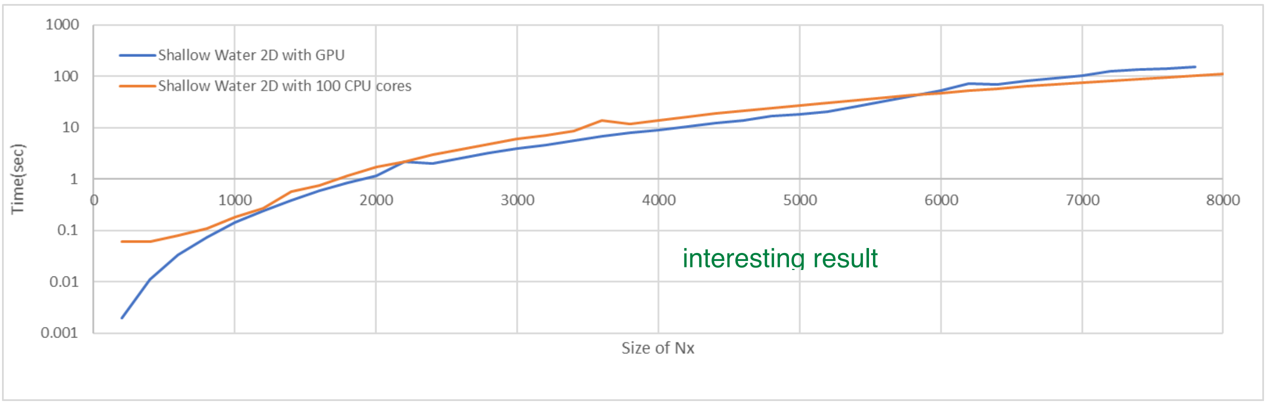
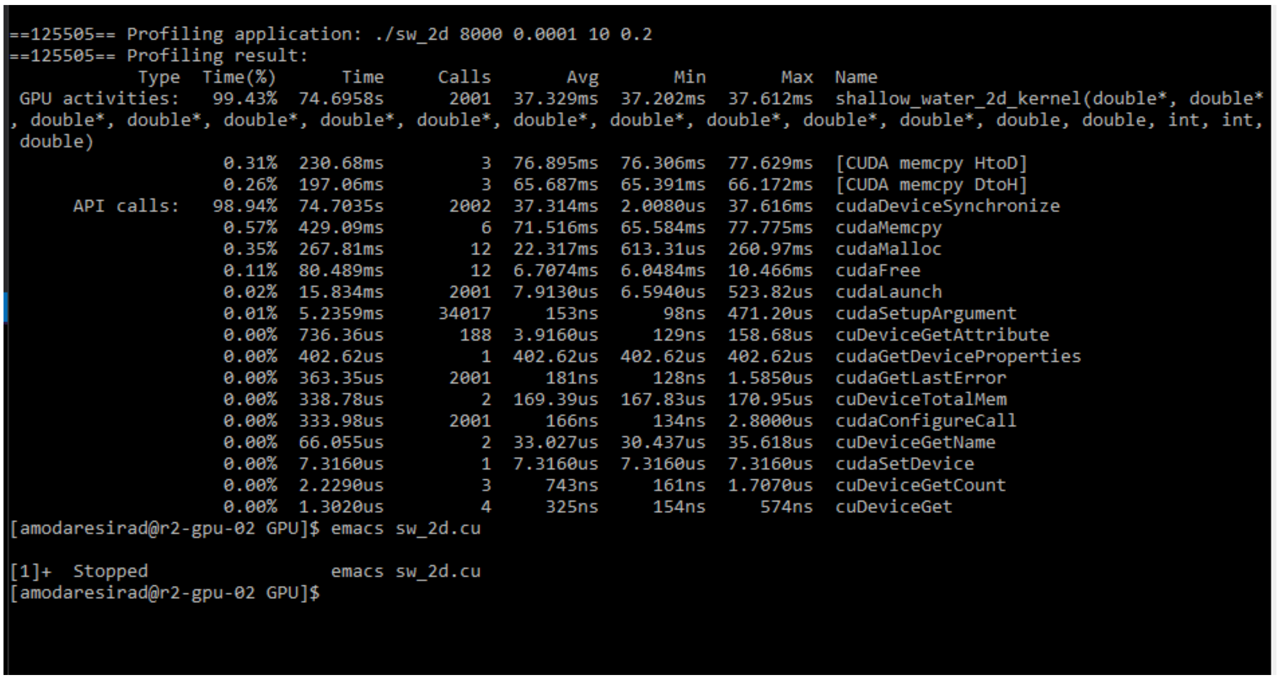


Figure 13) Comparing the speed of 81 cores of CPU vs GPU

is it 100 or 81?

By comparing the results of the time required to run shallow water 2D on different problem sizes, it is evident that GPU out preforms the speed of 100 CPU processors in all problem sizes smaller than Nx = 6000. This difference tend to be minimized as the problem size gets larger and larger. Finally, after the problem size on 6000 the CPU wins as it takes less time to complete the computation. This could possibly be related to the excessive amount of time spent on synchronizing threads and as problem gets bigger we need more time to do this synchronization and the computation it self is actually much more faster.

2.3. (ME 571) Use nvprof to run your CUDA code (nvprof ./sw_2d <arguments>). What conclusions do you draw from the profile you get? Where does the code spend majority of time? Compare the time spent on moving data to and from device with time spent in the kernel - based on this, explain whether this is code is using GPU efficiently or not.



```
==125505== Profiling application: ./sw_2d 8000 0.0001 10 0.2
==125505== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.43%  74.6958s      2001  37.329ms  37.202ms  37.612ms  shallow_water_2d_kernel(double*, double*
, double*, double*, double*, double*, double*, double*, double*, double*, double*, double*, double, double, int, int,
 double)
                    0.31%  230.68ms         3  76.895ms  76.306ms  77.629ms  [CUDA memcpy HtoD]
                    0.26%  197.06ms         3  65.687ms  65.391ms  66.172ms  [CUDA memcpy DtoH]
      API calls:   98.94%  74.7035s      2002  37.314ms  2.0080us  37.616ms  cudaDeviceSynchronize
                    0.57%  429.09ms         6  71.516ms  65.584ms  77.775ms  cudaMemcpy
                    0.35%  267.81ms        12  22.317ms  613.31us  260.97ms  cudaMalloc
                    0.11%  80.489ms        12  6.7074ms  6.0484ms  10.466ms  cudaFree
                    0.02%  15.834ms      2001  7.9130us  6.5940us  523.82us  cudaLaunch
                    0.01%  5.2359ms     34017     153ns      98ns  471.20us  cudaSetupArgument
                    0.00%  736.36us       188  3.9160us     129ns  158.68us  cuDeviceGetAttribute
                    0.00%  402.62us         1  402.62us  402.62us  402.62us  cudaGetDeviceProperties
                    0.00%  363.35us      2001     181ns     128ns  1.5850us  cudaGetLastError
                    0.00%  338.78us         2  169.39us  167.83us  170.95us  cuDeviceTotalMem
                    0.00%  333.98us      2001     166ns     134ns  2.8000us  cudaConfigureCall
                    0.00%  66.055us         2  33.027us  30.437us  35.618us  cuDeviceGetName
                    0.00%  7.3160us         1  7.3160us  7.3160us  7.3160us  cudaSetDevice
                    0.00%  2.2290us         3     743ns     161ns  1.7070us  cuDeviceGetCount
                    0.00%  1.3020us         4     325ns     154ns     574ns  cuDeviceGet
[amodaresirad@r2-gpu-02 GPU]$ emacs sw_2d.cu

[1]+  Stopped                 emacs sw_2d.cu
[amodaresirad@r2-gpu-02 GPU]$
```

Figure 14) Comparing the speed of 81 cores of CPU vs GPU

Figure 14 shows that the 2D shallow water actually kills the GPU power by using the synchronization that was called 2000 times and this is 99% of the total time of computation with GPU. Possibly a better way would be to mask some other computations in between synchronizations or to write the Kernel in a way that it minimizes the number of times cudaDeviceSynchornize is called.

Look at first three lines of GPU activities - the kernel uses 99.43% time, while moving data the rest - this is actually pretty good. I am not sure what it means that synchronization takes 99% time, as this breakdown does not show you time spent on actual computations - only calls to CUDA APIs.

2.4. (extra credit) Based on what you have learned in class, do you have any recommendations on how to improve the performance of the CUDA code? 3 pts

Two improvements can be done to increase the speed and to make sure that we don't run into any problems in very large problem sizes

1- Using local memory: as shown in question 1 using local memory would have a much more faster data transfer and therefore it can boost the speed tremondesly. A similar result was observed when comparing the speed boost in derivative and shallow water 1D, where due to having more computations in CUDA, shallow water 1D gained a lot of speed as local memory was used to store the output of computations in GPU. Therefore, it is expected that shallow water 2D would even benefit more from this. good conclusion!

2- Using Stride algorithm: this algorithm although may not change the speed of computations it definitely will improve the stability of Kernel when dealing with large problems, as it uses the free threads to compute again. true

3- Masking some computation in between thread synchronization: Possibly a better way would be to mask some other computations in between synchronizations or to write the Kernel in a way that it minimizes the number of times cudaDeviceSynchornize is called.

yes, synchronization is never good, but sometimes necessary. The trick is to synchronize only when completely necessary - all very good conclusions!