5615 CUDA
Assignment 2

Cylindrical Radiator Finite Differences model
Due Fri, 2021/04/30 - 17:00

Jose Refojo

April 8, 2021

The goal of this assignment is to model the propagation of heat inside a cylindrical radiator.

1 CPU Calculation

Write C or C++ code in .h/.c or .hpp/.cpp files that allocates two floating (not double) point matrices of size n x m (both with a default value of 32 and that can be specified by passing command line arguments -n and -m, respectively). The default number of iterations will be 10, but will have to be specified with a -p number command line argument option.

The boundary conditions will be:

- in column 0, matrix[i][0] = 1.00*(float)(i+1)/(float)(n) (so the values range between 1.00/(float)(n) and 1.00)
- in column 1, matrix[i][1] = 0.80*(float)(i+1)/(float)(n) (so the values range between 0.80/(float)(n) and 0.80)

Those values will remain constant and thus columns 0 and 1 do not need to be calculated on each time step. Initial conditions will be 0 in any other position of the matrices that isn't in columns 0 or 1.

Propagation of heat happens only in rows (so there is no propagation vertically) and is directional (in the sense that water in the radiator flows mostly towards "the right"), so you must apply the following weights:

(*nextMatrix)[ui][uj]= ((1.70*(*previousMatrix)[ui][uj-2])+(1.40*(*previousMatrix)[ui][uj-1])+ (*previousMatrix)[ui][uj]+ (0.60*(*previousMatrix)[ui][uj+1])+(0.30*(*previousMatrix)[ui][uj+2])); (*nextMatrix)[ui][uj]/=(float)(5.0);

The radiator is horizontally cylindrical (and a bit of heat propagates leftwise as well) and each row (pipe) is a cycle, so, for example, to compute the positions:

- new [ui][m-2], you will need the values of old [ui][m-4],[ui][m-3],[ui][m-2],[ui][m-1] and [ui][0]
- new [ui][m-1], you will need the values of old [ui][m-3],[ui][m-2],[ui][m-1],[ui][0] and [ui][1]

Add an command line option (say, "-a"), so that, after the designated number of heat propagation time steps has been concluded, the average temperature for each row of the radiator gets calculated (this represents the thermostat that would be used to stop heating once it has reached a certain level).

2 Parallel Implementation

From the code in the host, add cuda code in .h and .cu files that implements the same model in the gpu. To make things easier, you can make the following assumptions:

- Add a "-c" flag to your code so the cpu part of the calculation can be skipped (as this is useful for debugging the gpu code and finding the ideal block size - you do not need to recalculate the cpu part every time that run the gpu one).

- The final n and m sizes are expected to be a multiple of 32 (rectangular cases, in which m != n, must still work, though) - it is recommended to start with smaller problem and block sizes (say, 5), for easier debugging.
- Feel free to use atomics, as well as shared, texture or surface memory, or any other CUDA feature (but not other libraries other than CUDA) to speed up the calculation. Feel free to check and reuse the provided source code (for example cudaEvents, atomic, sharedMemory, etc).
- You can organize the grid in whatever way you want.
- Implement the row average calculation in a different kernel, so timing can be calculated independently (and that way you get a average-reduce operation that you can use from now on).

You will need to copy back the results to the RAM; compare them with the values obtained in the CPU and report any mismatches larger than 1.E-5. Add code to track the amount of time (you can use cuda events or standard CPU timers for the gpu parts) that each one of the steps takes (compute on the cpu, allocation on the gpu, transfer to the gpu, compute on the gpu, calculation of the averages, transfer back to the ram) takes, and add a command line argument (-t) to display both the CPU and GPU timings next to each other, as well as the other steps.

3 Performance Improvement

Test the code for different sizes, use n=15360,m=15360 and p=1000 as a reference. Try as well different numbers of threads per block, and different numbers in each one of the x and y directions, if you are using a 2d grid. Calculate the speedups (CPU vs GPU) and precisions compared to the CPU versions, first for compute, and second, including as well the memory allocations and transfers.

Comment on how the new reduce that you have implemented to calculate the average temperature per row performs compared to the reduce operation that you implemented for the first assignment.

Note: In this assignment, for decent CPU code, you can expect reasonable (>6) but not particulaly large (<60) speedups (for single precision, at least!)- however, the lower that the cuda execution times are, the more marks that the assignment will receive.

4 Double Precision

Port the code to a double precision format and compare the times, speedups and mismatches, if any.

Notes

Submit a tar ball with your source code files (including a working Makefile for cuda01), speedup graphs and a writeup of what you did and any observations you have made on the behaviour and performance of your code, as well as problems that you came across while writing the assignment.

- Note: Do not forget to set up your PATH and LD_LIBRARY_PATH variables in your .bashrc, otherwise nvcc won't work! You have the instructions in the slide 65 of the pdf of the course.
- Make sure that back propagation works! (so positions [ui][m-1] and [ui][m-2] get heat from [ui][0] and [ui][1])
- Marks will be deducted for tarbombing. http://en.wikipedia.org/wiki/Tar_
- Extra marks will be given for separating the C/C++ and cuda code. You can find examples on how to do that on the "makefileCpp" and "makefileExternC" sample code.
- Remember that the code must work for non-square systems too, even when, for this assignment, we are using square ones for benchmarking. You can run use cuda-memcheck to test that, as in: cuda-memcheck ./my_exec
- When you are benchmarking the performance of your code, you can check the current load on cuda01 with the nvidia-smi command.