Nesma Abd El-Halim Salah El-Din Galal
Muhammad Salah El-Okda
Rana Mohammed Safwat
Yasmin Mohammed

HPC Project Report

Steady State Relaxation

Submitted to

Eng. Dina El-Sayed

Abstract

Steady State Relaxation is a project for calculating the regular change in a matrix in which every cell is affected by its neighbors horizontally & vertically. This algorithm is used in this problem in assuming an isolated room initialed of 0 degree & suddenly opened 4 thermal sources from the 4 edges; we need to calculate the effect of these temperatures on the room in different iterations.

Algorithm

Steady state relaxation is a project that identifies different places temperature in an isolated room in different phases (iterations). There are 4 temperature sources can affect this room. The room size, the 4 temperature sources & the number of iterations are ALL inputs from the user. The output of our code is a file containing all the output matrices of the iterations number the user entered at the beginning.

Steps

1- We asked the user to enter all the variables in the system.
2- We initialized 2 matrices with zeros (a room & a copy of it).
   This copy is saved every iteration to get sure that the calculations of the iteration are for sure made from the previous iteration's result to avoid threads racing.
3- Iterations are dependable so they can't be run in parallel, but each cell in the single iteration is independent on the other cells of the same iteration; so they are calculated in parallel.
4- We realized that we have 8 special cases; 4 vertices, & 4 edges. We had put those cases in consideration.
5- Last but not least; we had to output the iteration's matrix.

We had faced a challenge in using OpenMP for the nested loops; but we got over it by the secret of " omp_set_nested(1);", but it wasn't an issue in CppAmp.

Architecture

1- Normal C++ for sequential code.
2- OpenMP for parallel code on the CPU.
3- CppAmp for parallel code on the GPU.

Runtime Measurement

All timings are measured in (mSec).

#Threads is the matrix size in each case.

Speedup= Architecture_timing – seq_timing

Efficiency= Architecture_timing / seq_timing %

PS: As the efficiency decreases in the positive range as the output seems better.

			CPU			GPU		
Sequential time	#iterations	#Threads	Parallel Time	Speed up	Efficiency	Parallel Time	Speedup	Efficiency
30	2	3*3=9	13	17	43.33	24	6	80
34	2	10*10=100	18	16	47.05	571	-541	1679.4
26851	1	500*500=250k	24719	2132	92	80069	-53218	298.2

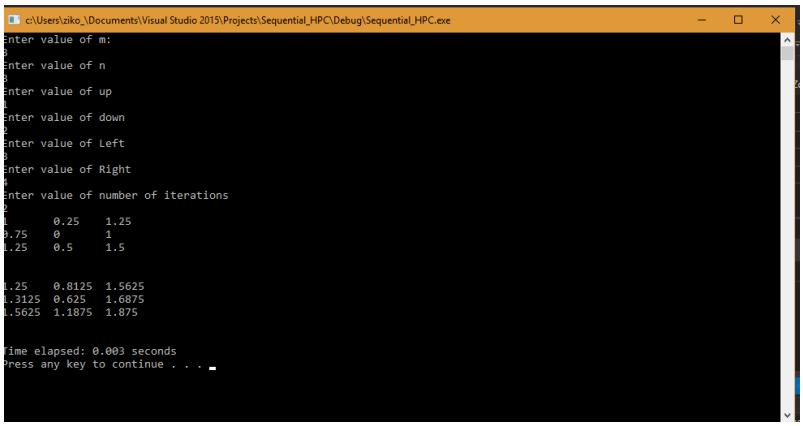
Those timings are calculated as a sample only. If the matrix size was changed, they would change but the efficiency won't suffer a great change as it is a percentage of Parallel_time/Seq_time.

The GPU efficiency increases by increasing the matrix size (number of threads used). This proves that the GPU cores are well utilized when used on a great number of undependable funcions.

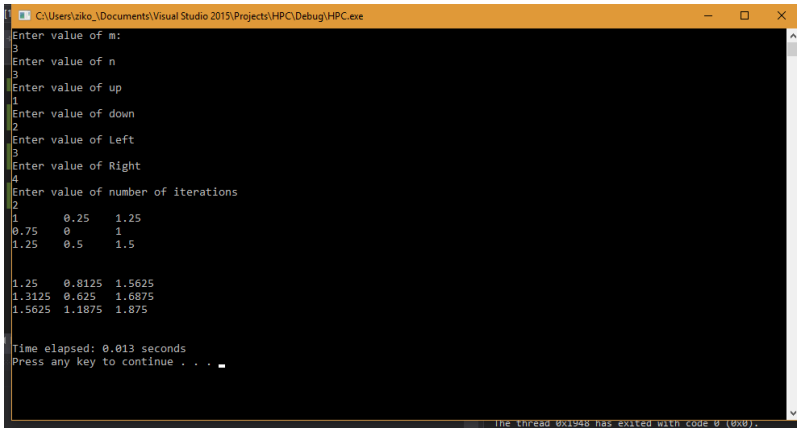
Those RED results are not acceptable at all.

CASE 1:

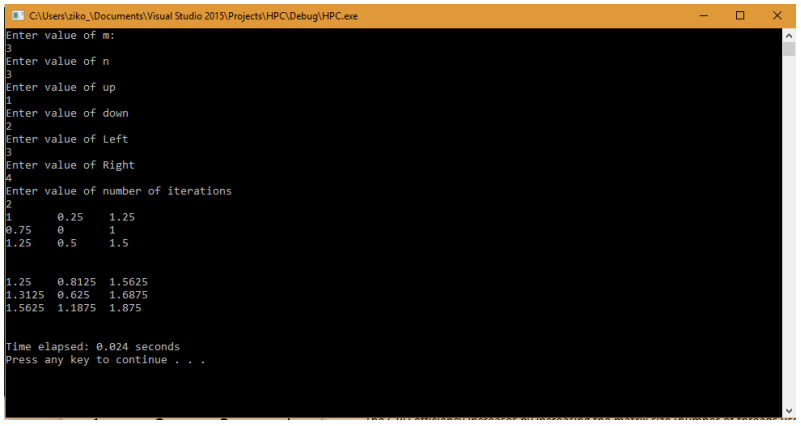
Sequential



CPUs

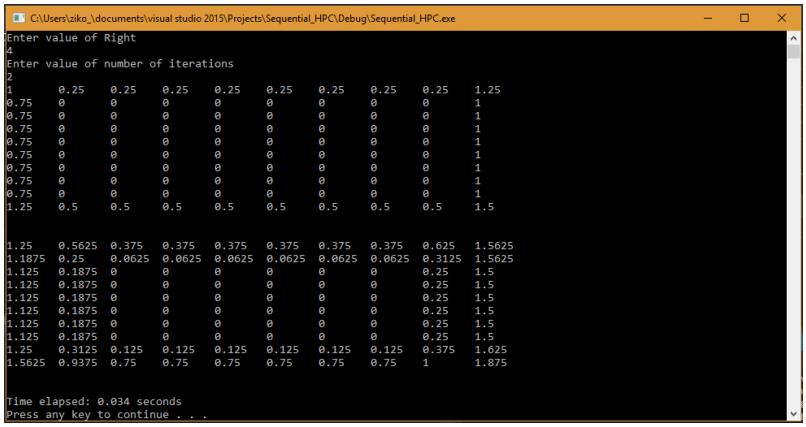


GPU

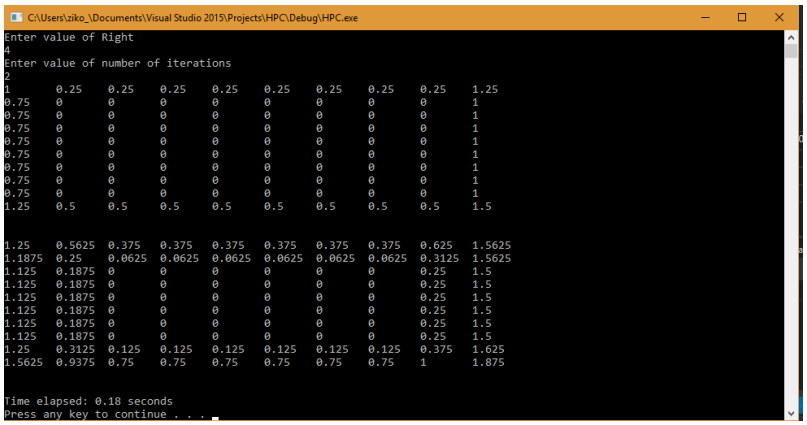


CASE 2:

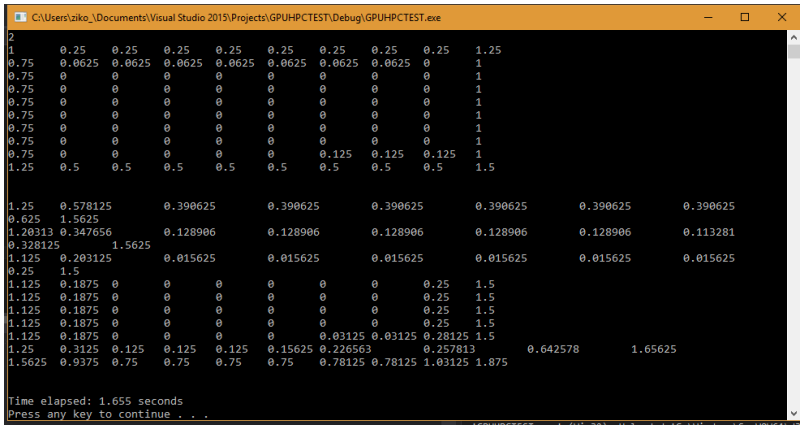
Sequential



CPU

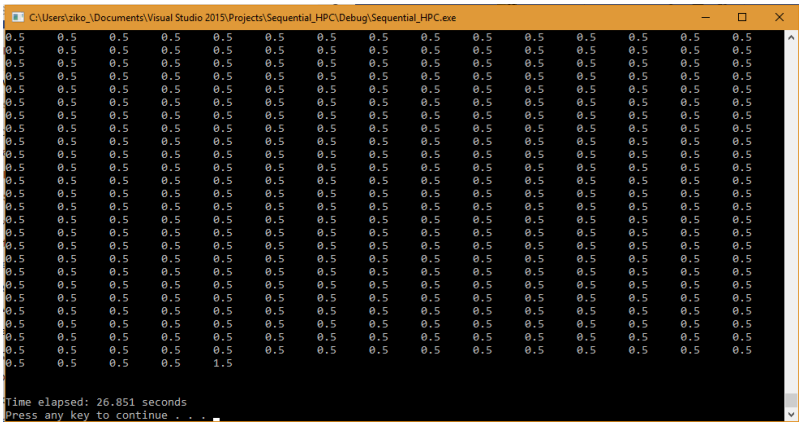


GPU

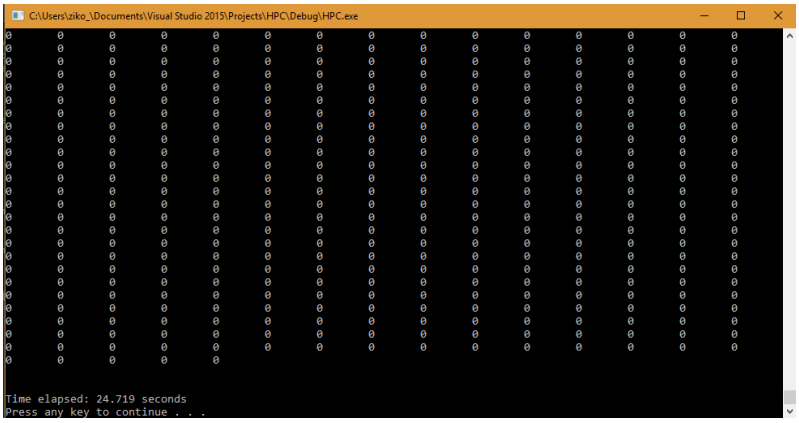


CASE 3:

Sequential



CPU



GPU