Project4 Writeup

In the project we were given serial compute function and was asked to implement compute_on_gpu function which uses gpu for implementing given serial compute function.

Data distribution is done by first setting grid size with input, and blocksize with set blockDimsize of choice. Then, calling cuda function with <<gridsize,blocksize >> divides distributes data. We index the data by finding start of block row with x0 = blockIdx.x * blockDim.x + threadIdx.x;
and start of column of each block with y0 = blockIdx.y * blockDim.y + threadIdx.y;

The compute kernal was written by using given serial function. Much of it was similar to the serial function except the iterator variable for the for loops were initialized by x0 = blockIdx.x * blockDim.x + threadIdx.x; for outer loop that loops thorugh x-axis and y0 = blockIdx.y * blockDim.y + threadIdx.y for the inner loop that loops thorugh column.
We do this since we have divided data into blocks and x0 and y0 gives start of each blocks row and column index. Since data is shared across blocks, no extra measure needed to be done and was very similar to serial compute.
One more thing that needs to be done was implement striding which will consider cases where there are more cells than thread avialable. We stride over x-axis by blockDim.x * gridDim.x. updating outer loops iterator variable by adding the stride value each iteration. This way, we stride over grid in terms of x-axis by length of block at each iteration and loop terminates when iterator varialbe is >= grid x-axis length making it so that we check every cell possible x-axis wise. We do same for Y axis in the inner loop striding over blockDim.y * gridDim.y adding to the iterator variable of inner loop each iteration. This results in striding over y-axis wise by the hiehgt of the block and since loop terminates when iterator variable >= grid hieght, we then check every possible cell y-axis wise.

Performance Results

Below is a peformance result of running the program for 500 generation on life.512x512.data on differing block sizes. Grid sizes were adjusted accordingly as below:
gridSizeX = X_Limit/blockDimSize +(1 if X_Limit%blockDimSize != 0, else 0)
gridSizeY = Y_Limit/blockDimSize +(1 if Y_Limit%blockDimSize != 0, else 0)
Grid Sizes:
Block dim size 2 = 256x256

Block dim size 4 = 128x128

Block dim size 8 = 64x64

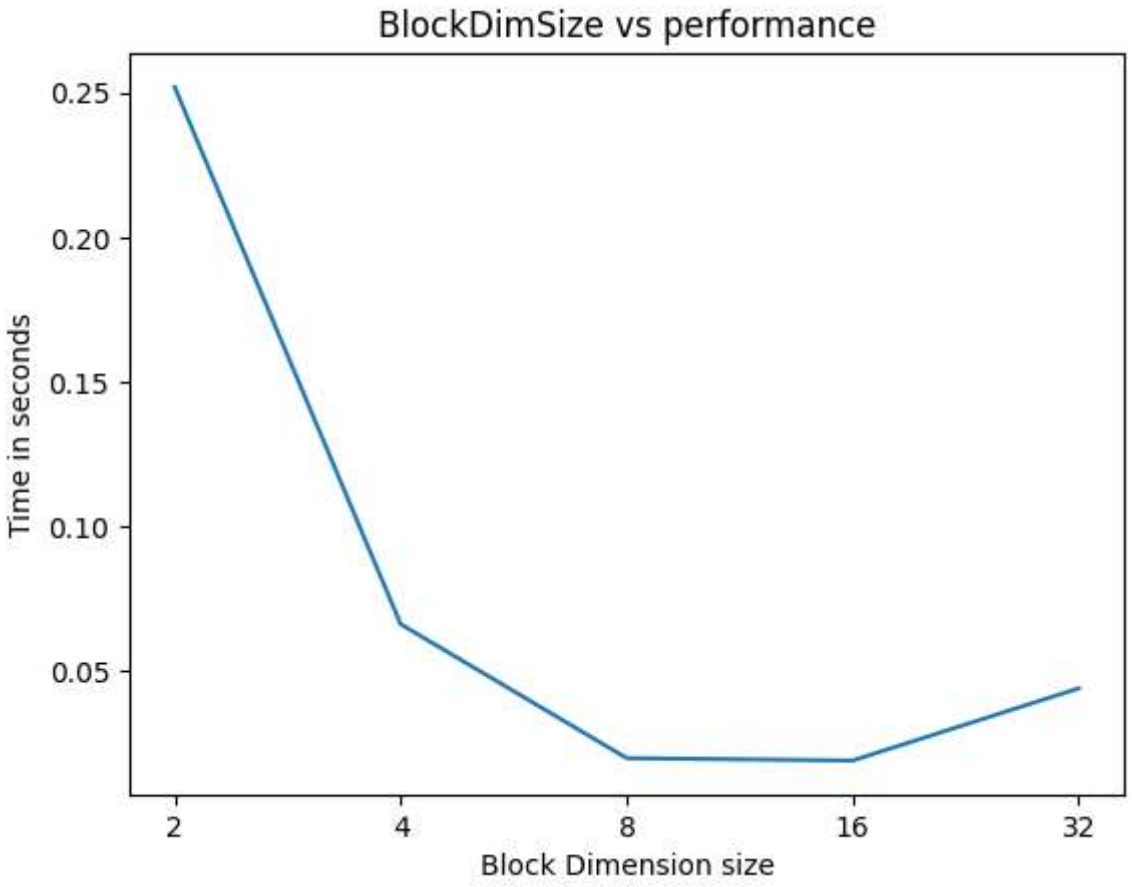
Block dim size 16 = 32x32

Block dim size 32 = 16x16

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(['2','4','8','16','32'])
y = np.array([0.251926,0.0662176,0.0198878,0.0190638,0.0440331])

plt.plot(x,y)
plt.title("BlockDimSize vs performance")
plt.xlabel("Block Dimension size")
plt.ylabel("Time in seconds")
plt.xticks(x)
plt.show()
```



Performance behaved as expected until dimension size of 16. I expected that as block dimension increases, performance will increase since block is threads that reside on the same SM; bigger block dimension means more thread can be excecuted at once per SM. However at 32, I saw unexpected behavior. There was increase in run time compared to dim size of 16. One hypothesis I could think of for reason beeing is that due to blocksize beeing large enough, GPU can't run that many threads at a time due to limitation of hardware.