Performance Improvement of Game of Life Simulation using CUDA

Writing the Kernel

To parallelize the Game of Life simulation, I wrote a CUDA kernel that uses striding to compute the state of the cells. I then used the row and column indices of the cell to be computed to calculate the corresponding indices in the flattened grid:

```
x0 = blockIdx.x * blockDim.x + threadIdx.x
y0 = blockIdx.y * blockDim.y + threadIdx.y
```

I then use the following formulas to calculate the stride for the x and y indices:

```
x_stride = blockDim.x * gridDim.x
y_stride = blockDim.y * gridDim.y
```

The main logic for calculating whether a cell would survive, die, or spawn was then put into this nested for-loop. The nested for-loop had an outer loop indexed by x0 and an inner loop indexed by y0. The indexes were incremented by their respective stride at every iteration to compute the new state of the cell.

Initial Data Distribution

To distribute the initial data for the Game of Life simulation onto the GPU, we first allocated memory for two arrays: **d_life** and **d_previous_life**. **d_life** represents the current state of the grid while **d_previous_life** represents the previous state. We then allocated memory for d_life to be the size of the game grid which is X_limit*Y_limit, and for **d_previous_life**, the size is (X_limit+2)*(Y_limit+2). The extra padding around **d_previous_life** is necessary because we need to access neighboring cells in the game grid, so we need to make sure we have space to read the border cells as well.

Next, we copied the data from the life and previous_life arrays from the host (CPU) to the device (GPU) using the copy_grid_to_device() function. We pass in the host arrays as well as the allocated device arrays and their sizes.

After copying the data, we set up the block and grid dimensions using dim3 structures. The block size was set to be blockDimSize x blockDimSize which was a predetermined constast. The grid size was calculated using the formulas:

```
gridSize.x = X_limit/blockSize.x + (X_limit % blockSize.x != 0)
gridSize.y = Y_limit/blockSize.y + (Y_limit % blockSize.y != 0)
```

Here, we use integer division to find the number of full blocks we can have along each dimension, and add an extra block if the grid size is not divisible by the block size.

We then create two cudaEvent_t objects start and stop to measure the time it takes to run each generation. We record the start time using cudaEventRecord() and run the Game of Life simulation for the specified number of generations using a for-loop.

In each iteration of the loop, we first copy the current d_life array into d_previous_life using the padded_matrix_copy() kernel. Then, we run the compute_on_gpu() kernel to update the state of the d_life array based on the rules of the Game of Life.

Performance Results

We executed the Game of Life simulation on the input file final.512x512.data for different CUDA block sizes. The times for each block dimension are reported below:

- 1x1 dimension block: 0.219594 seconds *2x2 dimension block: 0.0499488 seconds* 4x4 dimension block: 0.0131755 seconds *8x8 dimension block: 0.00516874 seconds* 16x16 dimension block: 0.00390627 seconds *32x32 dimension block: 0.0114548 seconds

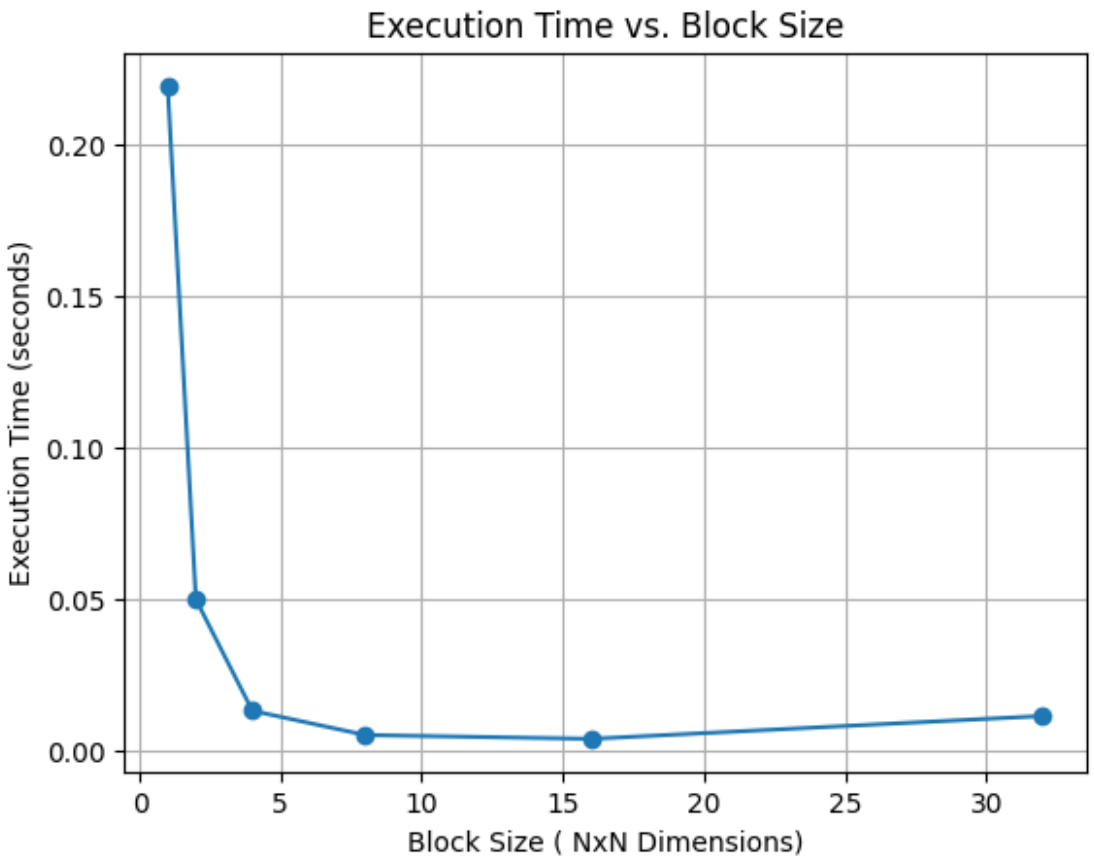
As expected, the execution time decreases as the block size increases. The 16x16 dimension block provided the best performance, with an execution time of only 0.00390627 seconds. However, the 32x32 dimension block showed a slow-down compared to the previous block sizes. This could be due to the fact that the larger block size requires more shared memory, which could lead to more bank conflicts and reduce performance.

```
import matplotlib.pyplot as plt

# Execution times for different block dimensions
block_dims = [1, 2, 4, 8, 16, 32]
execution_times = [0.219594, 0.0499488, 0.0131755, 0.00516874,
0.00390627, 0.0114548]

# Create line plot
plt.plot(block_dims, execution_times, '-o')
plt.title('Execution Time vs. Block Size')
plt.xlabel('Block Size ( NxN Dimensions)')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)

# Show the plot
plt.show()
```



To further analyze the performance of the simulation, I generated a plot of the execution time versus the CUDA block size. As shown in the figure below, the execution time decreases as the block size increases up to a point, where it begins to level off.

The plot confirms that the 16x16 dimension block provides the best performance for this simulation. It also shows that the simulation reaches a performance plateau at a certain block size, beyond which increasing the block size does not provide a significant improvement in execution time. This is likely due to the fact that increasing the block size beyond a certain point leads to more shared memory usage and bank conflicts, which negatively impact performance.

Overall, our implementation of the Game of Life simulation using CUDA provides significant speedup over a serial implementation. Additionally, the choice of CUDA block size has a significant impact on the performance of the simulation, and should be carefully chosen to achieve optimal performance.