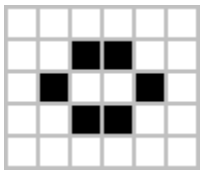
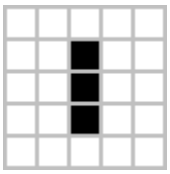
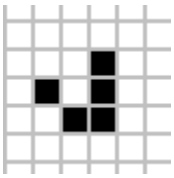
IT477 Project: Conway's Game of Life



Beehive - Still Life



Blinker - Oscillator



Glider - Spaceship

Team Members:

1. Devarshi Raval, 201601005
2. Monil Soni, 201601049
3. Geet Patel, 201601139
4. Bhavik Mehta, 201601223

Context

The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. The game is a zero player game, meaning that it's evolution is determined by its initial state, requiring no further input.

The universe is an infinite orthogonal grid of cells in which a cell can either be dead(unpopulated) or alive(populated). The state of a cell is determined on the basis of states of its eight neighbors by the following rules:

1. Any live cell with fewer than two live neighbours dies, as if by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The code for this will be simulated using ticks - a discrete moment when one generation of the universe moves to the next generation. Births and Deaths occur simultaneously during the tick. All the plots generated are using the time average of 100 generations.

Serial Approach

Inputs: First generation of the universe in NxN size, number of ticks(k)
Output: Universe after k generations
Algorithm: Loop over the matrix and for each cell check the above four conditions. Then store the result in a different matrix to avoid interfering with the current computation.

In order to tackle control divergence at an early stage, we replace the four if-else statements into just one if-else block with the following condition:

```
aliveNeighbours == 3 || (aliveNeighbours == 2 && grid[x*N + y])
```

Time complexity for the serial code is O(N*N*8). Even for grids of smaller size such as N = 10^5, the total complexity becomes O(8*10^{10}) which is very slow. Moreover, 9 memory reads will be required per cell. Of course, this will be optimized by cache but still 9*10^{10} memory reads in the crude sense is large enough.

Parallel Approach 1 - Global Memory Implementation

CPU Configuration:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    1
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 62
Model name:            Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
Stepping:              4
CPU MHz:               2458.862
CPU max MHz:           2500.0000
CPU min MHz:           1200.0000
BogoMIPS:              3999.68
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
NUMA node0 CPU(s):     0-7
NUMA node1 CPU(s):     8-15
```

GPU Configuration:

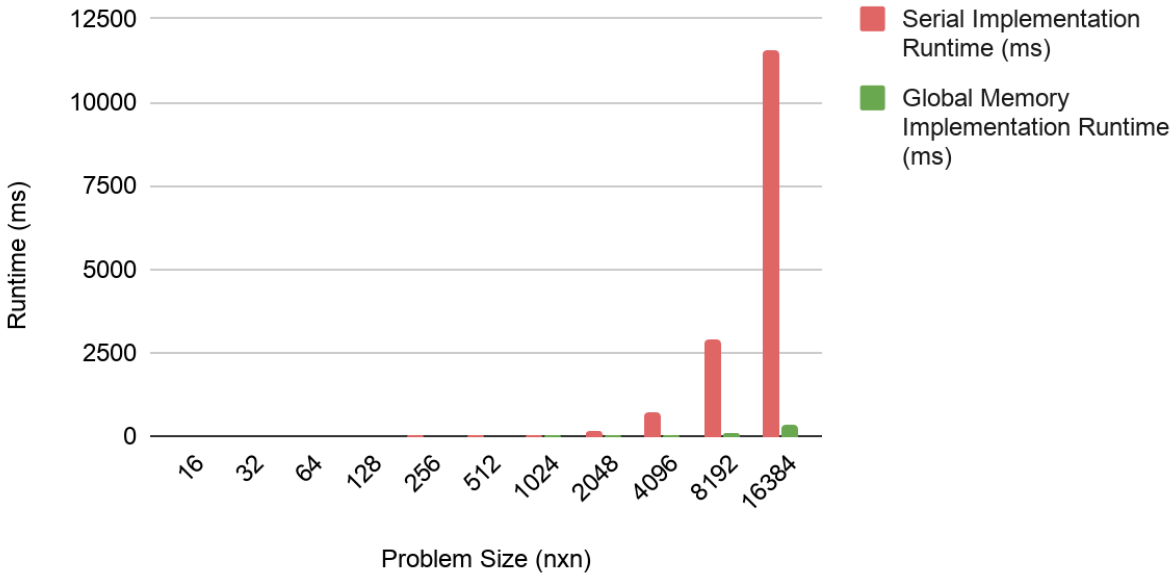
```
Device 0:
     Name: Tesla K40c
     Compute Capability 3.5
     Multiprocessor Count: 15
     Total Global Memory (GB): 11.17
     Total Constant Memory (kB): 64
```

```
Shared Memory per Block (kB): 48
Registers per Block: 65536
Max Threads per Block: 1024
Max Threads per Multiprocessor: 2048
Warp Size: 32
Max Thread Dim: 1024,1024,64
Max Grid Dim: 2147483647,65535,65535
Memory Pitch: 2147483647
L2 Cache Size: 1572864
Clock Rate (kHz): 745000
Memory Clock Rate (kHz): 3004000
Memory Bus Width (bits): 384
```

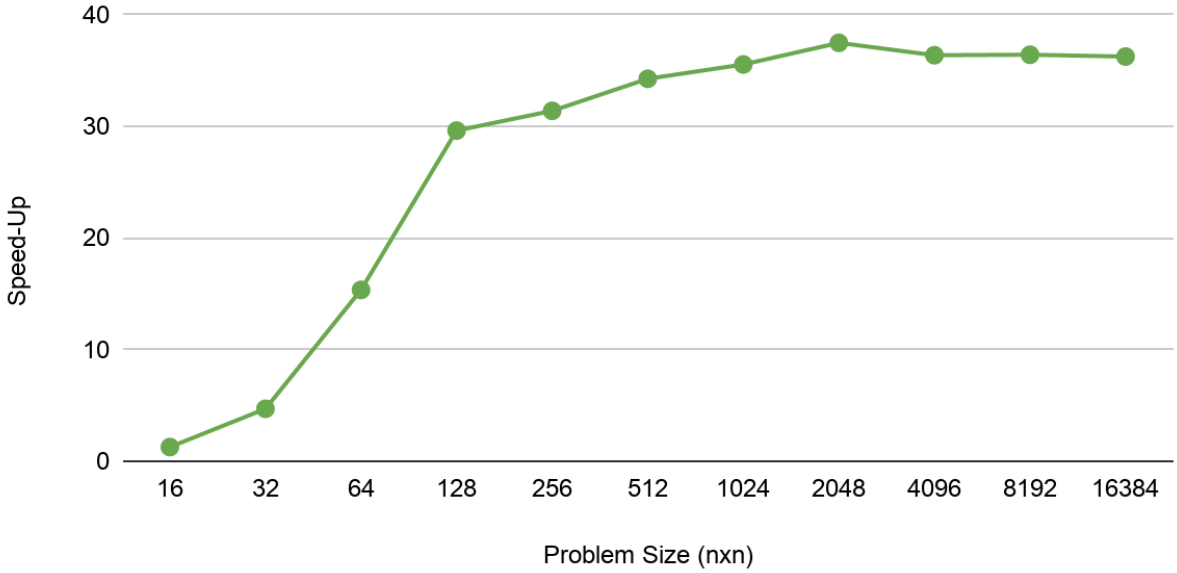
In the parallel code, each thread is responsible for the state of one cell. In each tick, all the threads compute the next stage of their respective cells and then the data is copied to global memory for the computation for the next generation.

While parallel approach improves the throughput and consequently granted speedup, it still has an issue. The CGMA ratio for this is 10/9, since for the computation of one cell, 9 global memory accesses are required and 10 floating point operations are carried out. Moreover, every cell is loaded 9 times in memory. 1 for the time when it is operated on and 8 for when that cell is a neighbour sell of the others. We can reduce the number of global memory accesses by using shared memory to our advantage.

Runtime (ms): Serial Implementation vs Global Memory Implementation



Speed-Up: Serial Implementation vs Global Memory Implementation



Parallel Approach 2 - Shared Memory Implementation

Theory of Implementation

As with matrix multiplication, we let each thread bring in the corresponding cell from global memory with the exception of the border cells bringing its neighbor cells also. So for a block of size bxb, we were earlier making b*b*9 memory accesses and now we will be making (b+2)*(b+2) global memory accesses only.

Using shared memory instead of global memory significantly improves throughput and speedup. The CGMA ratio calculation for shared memory is done for tile. Let the tile size be ***t***.
Then for each cell there are 10 floating operations as discussed in global memory implementation and the number of global memory accesses will be $t^2 + 4t + 4$ for the whole tile. As there are t^2 cells in a tile,

$$\text{CGMA ratio} = \frac{10t^2}{t^2+4t+4}$$

The CGMA ratio is approximately 10 for large t, which is a significant improvement from the global memory implementation.

Practical Considerations

In a GPU with major compute capability 3,

a) Maximum Blocks per SM: 8
b) Maximum Threads per SM: 2048
c) Maximum Threads per Block: 1024

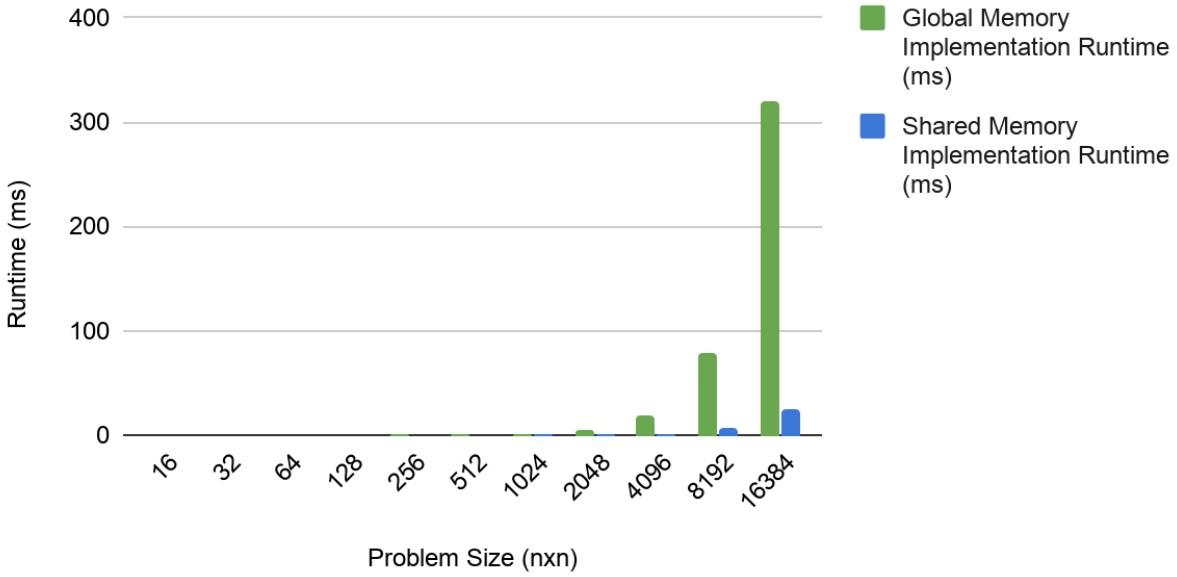
Given these stats, it can be concluded that when 16x16 = 256 threads are launched per block, the number of blocks per SM is 8, and when the number of threads per block is increased, let's say 512, the number of blocks per SM will reduce to 4 to keep the number of threads per SM fixed to 2048, its maximum. So, there will be no performance difference in launching 256 threads per block or 512 or 1024.That is the reason why the number of threads per block is kept constant at 16x16 = 256.

Since the block size is 16x16, and it is the same size as the shared memory, but shared memory will also have two extra rows and columns of data, to accommodate calculations of boundary elements, so the exact size of the shared memory tile will be (1+16+1)x(1+16+1) = 18x18.

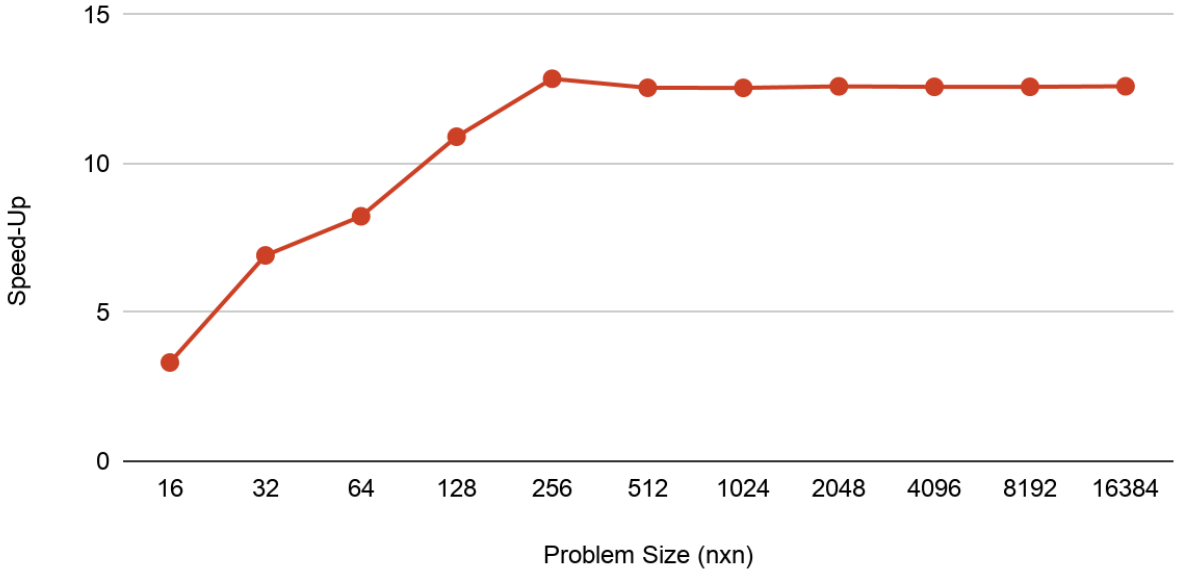
Statistics

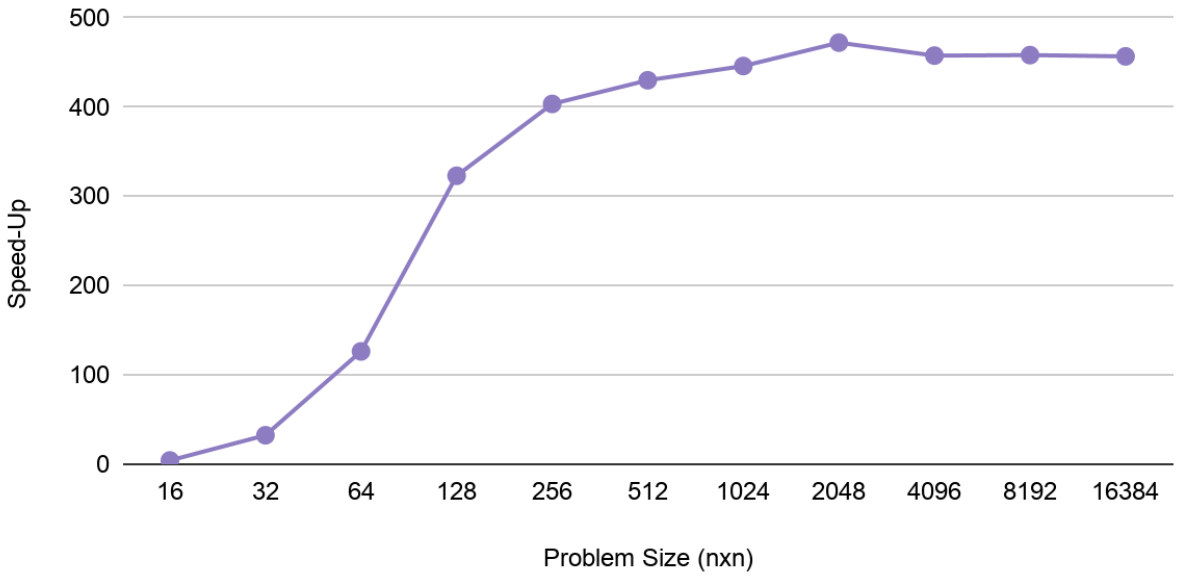
Size (nxn)	Serial Implementation Runtime (ms)	Global Memory Implementation Runtime (ms)	Shared Memory Implementation Runtime (ms)	Speed-Up: Serial vs Global	Speed-Up: Global vs Shared	Overall Speed-Up: Serial vs Shared
16	0.0093	0.0073	0.0022	1.27	3.31	4.22
32	0.039	0.0083	0.0012	4.69	6.91	32.5
64	0.1643	0.0107	0.0013	15.35	8.23	126.38
128	0.6785	0.0229	0.0021	29.62	10.9	323.09
256	2.7442	0.0874	0.0068	31.39	12.85	403.55
512	11.0527	0.3225	0.0257	34.27	12.54	430.06
1024	44.5584	1.2533	0.0999	35.55	12.54	446.03
2048	187.4284	4.9995	0.3969	37.48	12.59	472.23
4096	725.2211	19.9334	1.5845	36.38	12.58	457.69
8192	2902.0156	79.669	6.3327	36.42	12.58	458.25
16384	11574.7578	319.2195	25.336	36.25	12.59	456.85

Runtime (ms): Global Memory Implementation vs Shared Memory Implementation



Speed-Up: Global Memory Implementation vs Shared Memory Implementation




Overall Speed-Up: Serial Implementation vs Shared Memory Implementation
Speed-Up
500
400
300
200
100
0
16
32
64
128
256
512
1024
2048
4096
8192
16384
Problem Size (nxn)