**PROJECT 3**

Game of Life – CUDA implementation

CUDA implementation –

Program flow –

Program asks user to enter number of total elements(N\*N) in grid. This number should be perfect square.

Then user should enter number of blocks( B) , number of threads per block(T).

User can choose to run GOL in infinite loop or can enter number of generations(G).

As total elements entered by user is perfect square rowDim and colDim is sqrt(N\*N)

Random number generation for initial state of grid –

Random number generation is implemented using CPU. In built function rand() is used for the same. Each time rand() is seeded with current time.

Input grid elements are set to either 0 or 1 in random manner. 1 represents life and 0 no life.

2D array implementation –

Linear array is used to implement 2D array in row wise manner.

Data decomposition –

Program is compatible to work with any combination of B & T entered by user.

Input array is decomposed such that in each iteration B\*T elements are processed.

There are 4 cases of B\*T entered by user –

1. B\*T = N\*N

When B\*T is equal to total elements in grid(N\*N) then all elements are processed in one iteration only. No data decomposition is required as each element in grid gets a dedicated thread.

1. B\*T > N\*N

When B\*T is greater than total number of elements in grid(N\*N) then it means there are more threads available than required. In this case kernel function will use only B\*T threads using condition threadIDx < (N\*N).

Extra threads will not be used to process input elements redundantly.

No decomposition is needed as each element get dedicated thread.

1. B\*T < N\*N & N\*N % B\*T = 0

When B\*T is led than N\*N, it means that there are few threads than number of elements. So data decomposition is must.

If N\*N is perfectly divided by B\*T then there are equal chucks of data of size B\*T

These equal sized chucks are processed one by one. threadIDx is adjusted so kernel function will access right input element.

1. B\*T < N\*N & N\*N % B\*T != 0

In this case there are less threads available than needed so data decomposition is must. However no equal sized chunks of size B\*T are possible. So last partition will have less than B\*T elements and all other will get B\*T elements.

Again threadIDx is adjusted so kernel function will access right element.

Last partition gets less than B\*T elements so only enough number of threads will be used.(threadIDx < (N\*N))

Kernel routine –

It receives input array, rowDim, colDim, offset value and output array.

ThreadId is determined using blockIdx.x\*blockDim.x + threadIdx.x.

Row and col of every element in input is found using threadIdx and offset value. Offset value is useful when data is decomposed.

This row and col value maps linear input array into 2D array.

Helper function getNeighbor receives this row, col details and returns array of neighbors.

Sum of all neighbor elements is nothing but count of neighbors with life.

These rules are applied based upon sum.

Current cell has life –

sum <= 1 No life in current cell – Lonely ness

sum >= 4 No life in current cell – Over occupancy

sum = 2 or 3 Life continues at current cell

Current cell has no life-

sum = 3 Birth

otherwise No life at current cell

Helper function – getNeighbor

This function gives the array of all 8 neighbors current element has. Current element is identified using row and col passed by kernel routine.

Toroidal mesh is implemented so each element has exactly 8 elements. Based upon row and col value all 8 adjacent neighbor value(0 or 1) is passed into output array.

Elapsed time calculation –

Elapsed time in CUDA code is found using CUDA event timer. I safely ignore the printf() overhead because it was negligible compared to total execution time.

Experimental results –

Experimental results in tabular format are available in experiment.xlsx sheet

Speedup is observed for large value of N. For instance, N>=4096 it gives very good improvement. For small values of N sequential code is much affordable than CUDA program.

Bottleneck -

For large value of N, balanced combination of B & T works best.

Making either B or T equal to 1, and keeping other equal to N, gives the worst performance in most of the cases. Creating N blocks and 1 thread/block OR 1 block and N thread/block causes overhead which leads to performance degradation.

Keeping B\*T close to N gives maximum performance as it avoids data decomposition and all N elements get processed at same time. This is not possible for very large N values as total number of threads is limited. Therefore data decomposition is mandatory.

In general keeping less number of blocks and higher number of threads gives good performance for large N.

Compilation and execution instructions are provided in README.txt