VT Valsid Tech Pvt Ltd

CUDA Programming

By: Anirudh N

- **CUDA program that implements a simple cellular automaton, specifically Conway's Game of Life, in parallel using CUDA.**

- **It showcases parallel computation on a 2D grid, which is a typical use case for CUDA.**

CODE :

```
#include <cuda_runtime.h>

#include <stdio.h>

#define N 16           // Define the grid size

#define BLOCK_SIZE 16

__device__ int countNeighbors(int* grid, int x, int y, int width, int height) {

  int count = 0;

  for (int dx = -1; dx <= 1; ++dx) {

    for (int dy = -1; dy <= 1; ++dy) {

      if (dx == 0 && dy == 0) continue;

      int nx = (x + dx + width) % width;

      int ny = (y + dy + height) % height;

      count += grid[ny * width + nx];

    }
    }
    return count;
}

__global__ void gameOfLifeKernel(int* currentGrid, int* nextGrid, int width, int height) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < width && y < height) {
        int neighbors = countNeighbors(currentGrid, x, y, width, height);
        int cell = currentGrid[y * width + x];
        if (cell == 1 && (neighbors < 2 || neighbors > 3)) {
            nextGrid[y * width + x] = 0;
        } else if (cell == 0 && neighbors == 3) {
            nextGrid[y * width + x] = 1;
        } else {
            nextGrid[y * width + x] = cell;
        }
    }
}

void printGrid(int* grid, int width, int height) {
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            printf(grid[y * width + x] ? "█" : " ");
        }
        printf("\n");
    }
}

int main() {
    int gridSize = N * N * sizeof(int);
    int* h_currentGrid = (int*)malloc(gridSize);
    int* h_nextGrid = (int*)malloc(gridSize);

    // Initialize the grid with a simple pattern
```

```
for (int i = 0; i < N * N; ++i) {
    h_currentGrid[i] = rand() % 2;
}

int* d_currentGrid;
int* d_nextGrid;
cudaMalloc((void**)&d_currentGrid, gridSize);
cudaMalloc((void**)&d_nextGrid, gridSize);

cudaMemcpy(d_currentGrid, h_currentGrid, gridSize, cudaMemcpyHostToDevice);

dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 numBlocks((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (N + BLOCK_SIZE - 1) / BLOCK_SIZE);

for (int i = 0; i < 100; ++i) {
    gameOfLifeKernel<<<numBlocks, threadsPerBlock>>>(d_currentGrid, d_nextGrid, N, N);
    cudaDeviceSynchronize();

    cudaMemcpy(h_nextGrid, d_nextGrid, gridSize, cudaMemcpyDeviceToHost);
    printGrid(h_nextGrid, N, N);

    // Swap grids
    int* temp = d_currentGrid;
    d_currentGrid = d_nextGrid;
    d_nextGrid = temp;

    printf("\n\n");
    usleep(100000);  // Sleep for 100ms
}

cudaFree(d_currentGrid);
cudaFree(d_nextGrid);
free(h_currentGrid);
free(h_nextGrid);
```

```
    return 0;
}
```

OUTPUT:

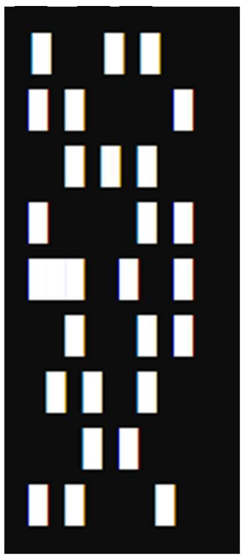
1. **Compile:**

```
nvcc -o game_of_life game_of_life.cu
```

2.Run:

```
./game_of_life
```

3.Result:



NOTE : Each iteration will produce a new grid, and you will see the cells changing according to the rules of the Game of Life. The actual output will vary depending on the initial random configuration of the grid.