CUDA

Part 3

Compute **U**nified **D**evice **A**rchitecture

Adam Simpson
Aug. 2011

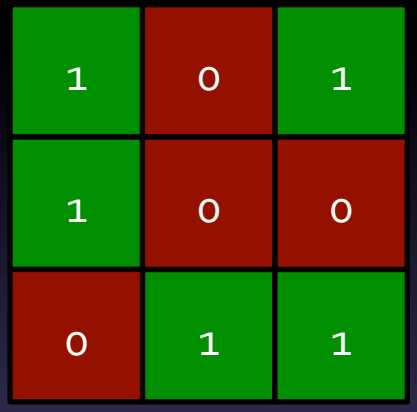
Game of Life

- Create 2D square grid of cells
- Each cell has two possible states: alive or dead
- Cell interacts with 8 nearest neighbors
- One game tick: Apply game rules to all cells simultaneously
- Use resulting cell grid to further play the game
- Use periodic boundary conditions

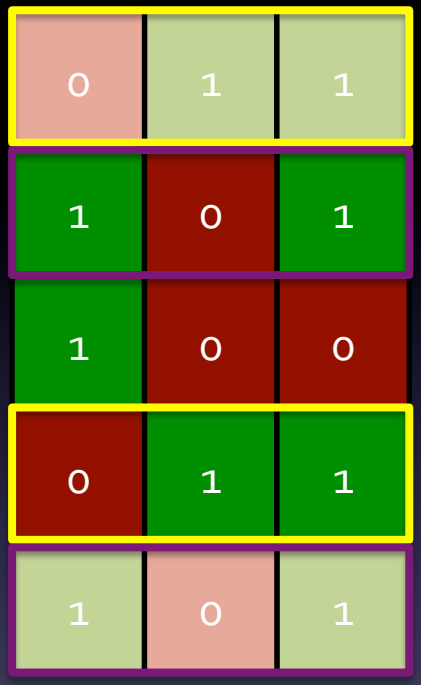
Game of Life: Rules

- A live cell with less than two live neighbors dies
- A live cell with two or three live neighbors lives
- A live cell with more than three live neighbors dies
- A dead will with 3 live neighbors becomes live

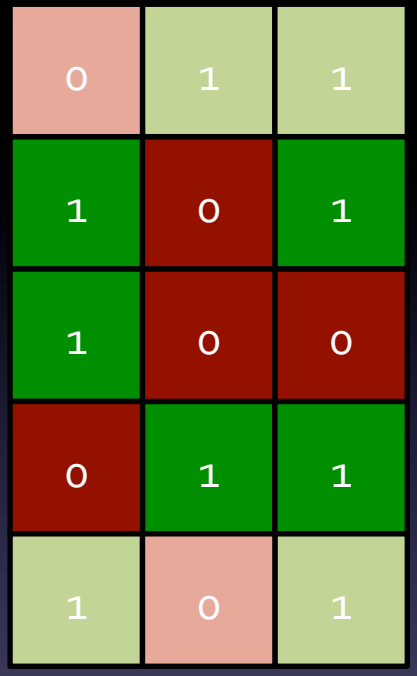
Game of Life



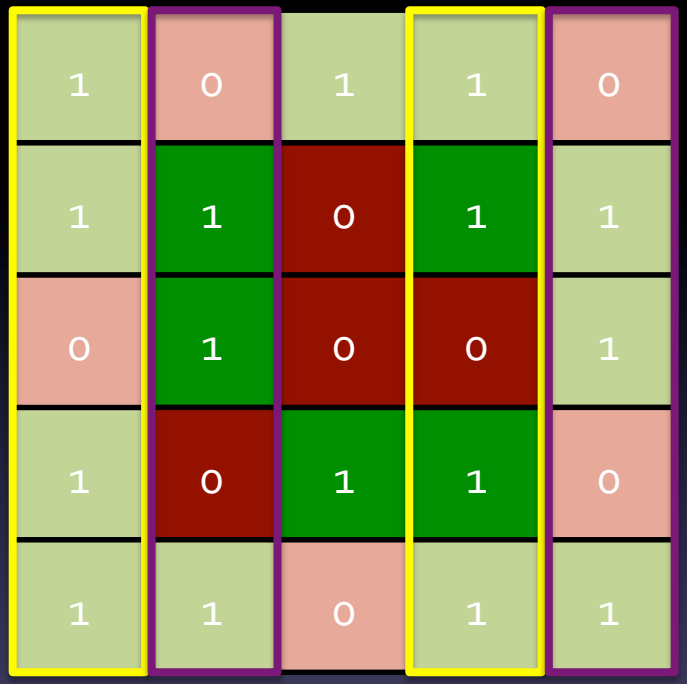
Game of Life



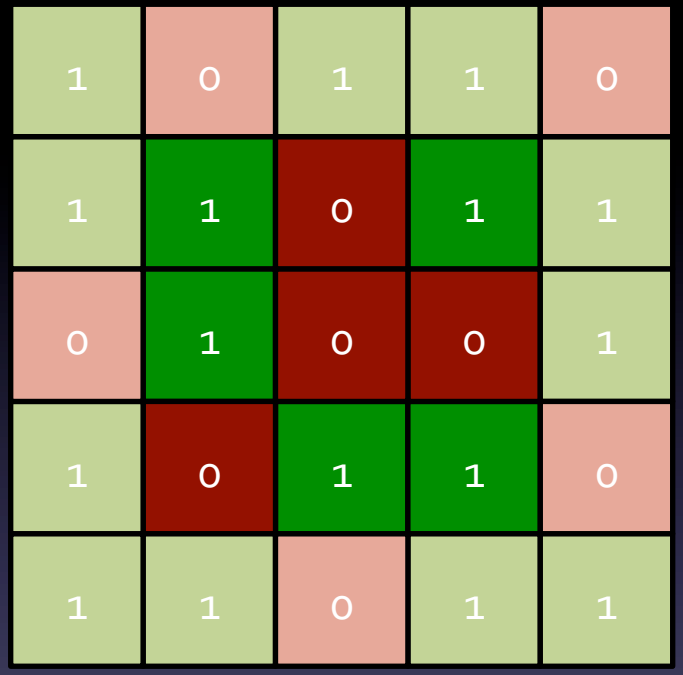
Game of Life



Game of Life



Game of Life



GOL CPU

```
// Main game loop
for (iter = 0; iter<maxIter; iter++) {
    // Left-Right columns
    for (i = 1; i<=dim; i++) {
        grid[i][0] = grid[i][dim]; // Copy first real column to right ghost column
        grid[i][dim+1] = grid[i][1]; // Copy last real column to left ghost column
    }
    // Top-Bottom rows
    for (j = 0; j<=dim+1; j++) {
        grid[0][j] = grid[dim][j]; // Copy first real row to bottom ghost row
        grid[dim+1][j] = grid[1][j]; // Copy last real row to top ghost row
    }

    // Now we loop over all cells and determine their fate
    for (i = 1; i<=dim; i++) {
        for (j = 1; j<=dim; j++) {
            // Get the number of neighbors for a given grid point
            int numNeighbors = grid[i+1][j] + grid[i-1][j]     //upper lower
                             + grid[i][j+1] + grid[i][j-1]     //right left
                             + grid[i+1][j+1] + grid[i-1][j-1] //diagonals
                             + grid[i-1][j+1] + grid[i+1][j-1];

            // The game rules
            if (grid[i][j] == 1 && numNeighbors < 2)
                newGrid[i][j] = 0;
            else if (grid[i][j] == 1 && (numNeighbors == 2 || numNeighbors == 3))
                newGrid[i][j] = 1;
            else if (grid[i][j] == 1 && numNeighbors > 3)
                newGrid[i][j] = 0;
            else if (grid[i][j] == 0 && numNeighbors == 3)
                newGrid[i][j] = 1;
            else
                newGrid[i][j] = grid[i][j];
        }
    }

    // Done with one step so we swap our grids
    int **tmpGrid = grid;
    grid = newGrid;
    newGrid = tmpGrid;
}// End main game loop
```

CUDA GOL Host

```
// Main game loop
for (iter = 0; iter<maxIter; iter++) {
    ghostRows<<<cpyGridRowsGridSize, cpyBlockSize>>>(dim, d_grid);
    ghostCols<<<cpyGridColsGridSize, cpyBlockSize>>>(dim, d_grid);
    GOL<<<gridSize, blockSize>>>(dim, d_grid, d_newGrid);

    // Swap our grids
    int *d_tmpGrid = d_grid;
    d_grid = d_newGrid;
    d_newGrid = d_tmpGrid;
}//iter loop
```

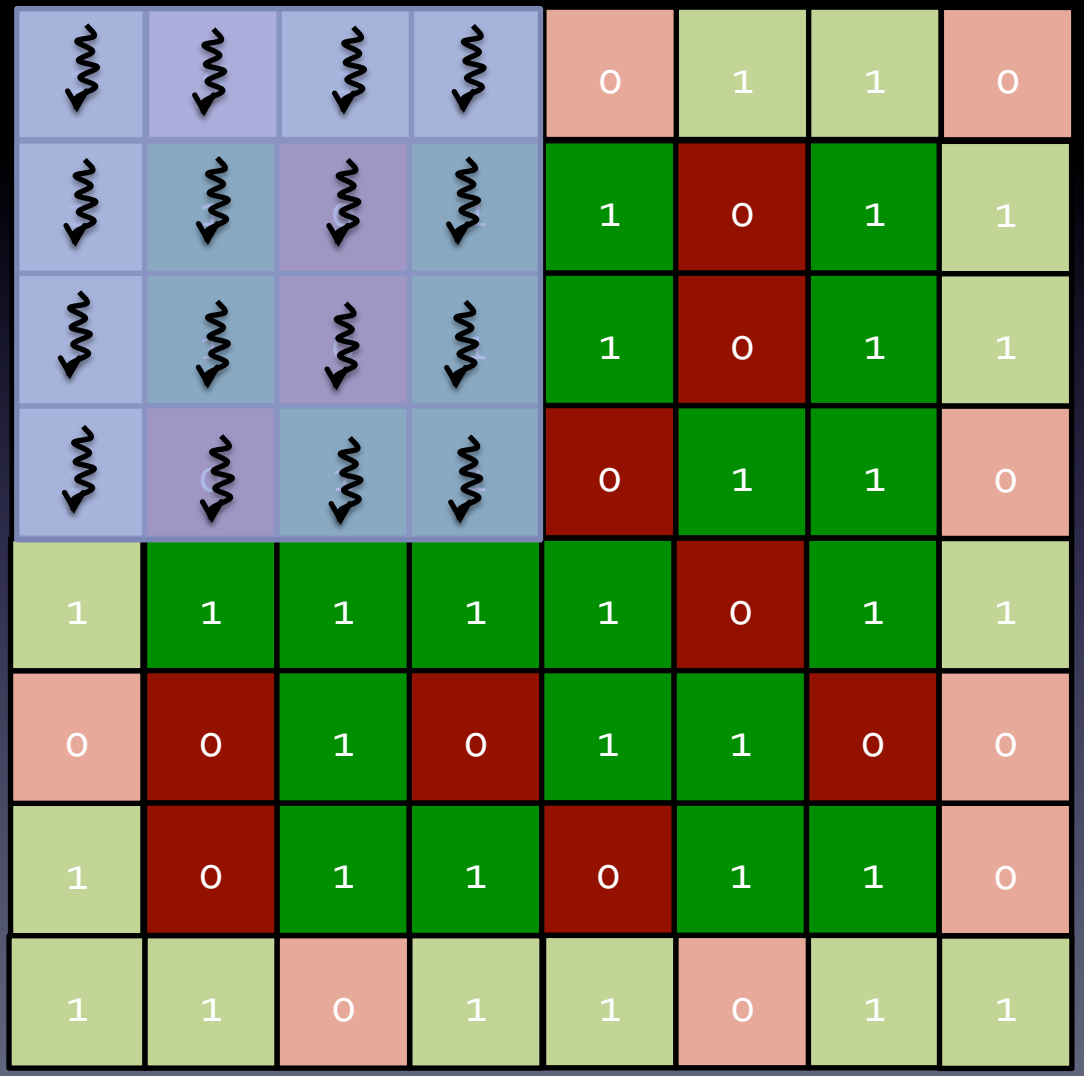
Memory Transfer

```
__global__ void ghostRows(int dim, int *grid)
{
    // We want id ∈ [1,dim]
    int id = blockDim.x * blockIdx.x + threadIdx.x + 1;

    if (id <= dim)
    {
        grid[(dim+2)*(dim+1)+id] = grid[(dim+2)+id]; //Copy first real row to bottom ghost row
        grid[id] = grid[(dim+2)*dim + id]; //Copy last real row to top ghost row
    }
}

__global__ void ghostCols(int dim, int *grid)
{
    // We want id ∈ [0,dim+1]
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    if (id <= dim+1)
    {
        grid[id*(dim+2)+dim+1] = grid[id*(dim+2)+1]; //Copy first real column to right most ghost column
        grid[id*(dim+2)] = grid[id*(dim+2) + dim]; //Copy last real column to left most ghost column
    }
}
```

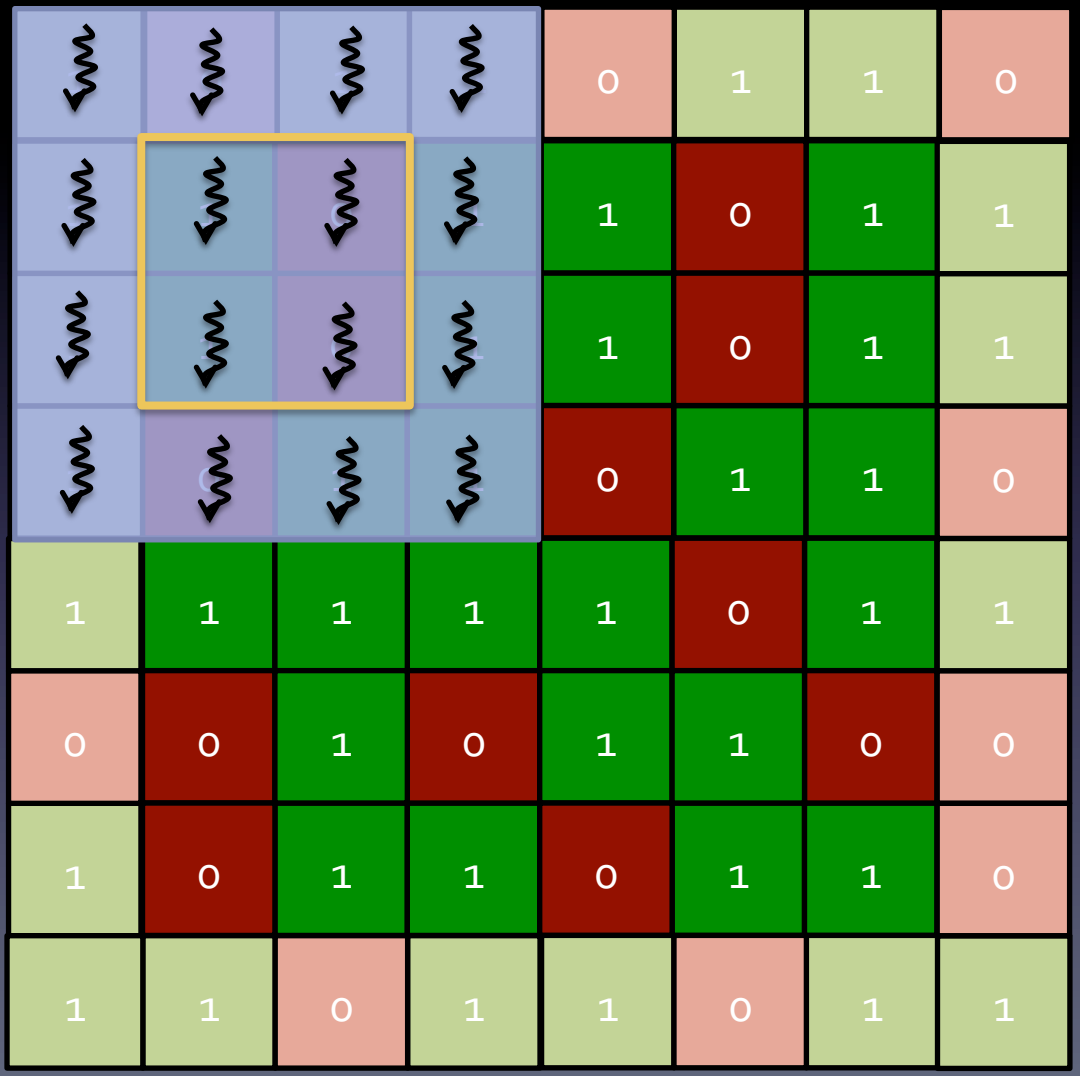
Global GOL Kernel

```
__global__ void GOL(int dim, int *grid, int *newGrid)
{
    // We want id ∈ [1,dim]
    int iy = blockDim.y * blockIdx.y + threadIdx.y + 1;
    int ix = blockDim.x * blockIdx.x + threadIdx.x + 1;
    int id = iy * (dim+2) + ix;

    int numNeighbors;

    if (iy <= dim && ix <= dim) {

        // Get the number of neighbors for a given grid point
        int numNeighbors = grid[id+(dim+2)] + grid[id-(dim+2)] //upper lower
                         + grid[id+1] + grid[id-1]             //right left
                         + grid[id+(dim+3)] + grid[id-(dim+3)] //diagonals
                         + grid[id-(dim+1)] + grid[id+(dim+1)];

        int cell = grid[id];
        // Here we have explicitly all of the game rules
        if (cell == 1 && numNeighbors < 2)
            newGrid[id] = 0;
        else if (cell == 1 && (numNeighbors == 2 || numNeighbors == 3))
            newGrid[id] = 1;
        else if (cell == 1 && numNeighbors > 3)
            newGrid[id] = 0;
        else if (cell == 0 && numNeighbors == 3)
            newGrid[id] = 1;
        else
            newGrid[id] = cell;
    }
}
```

Game of Life

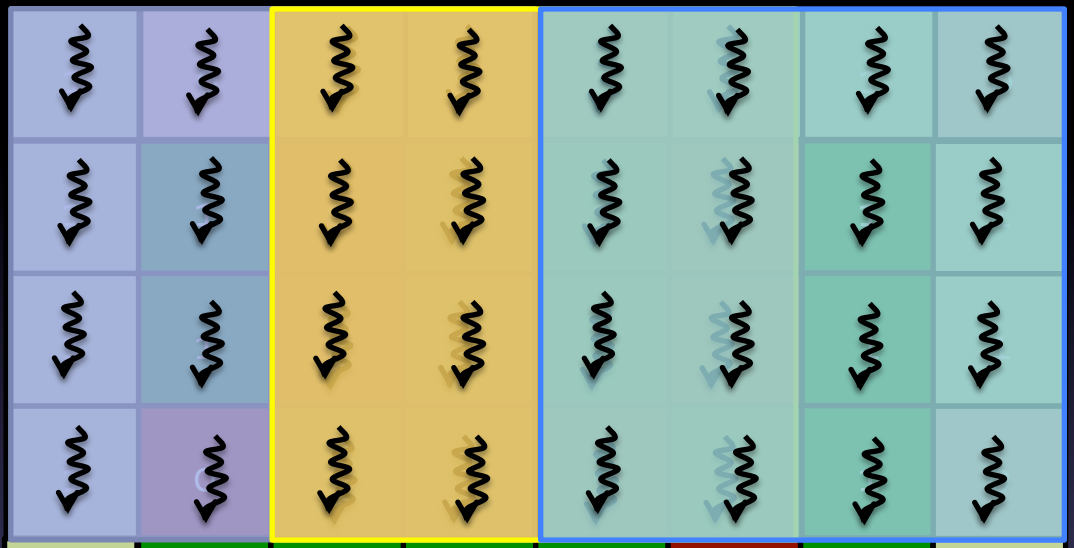


Game of Life



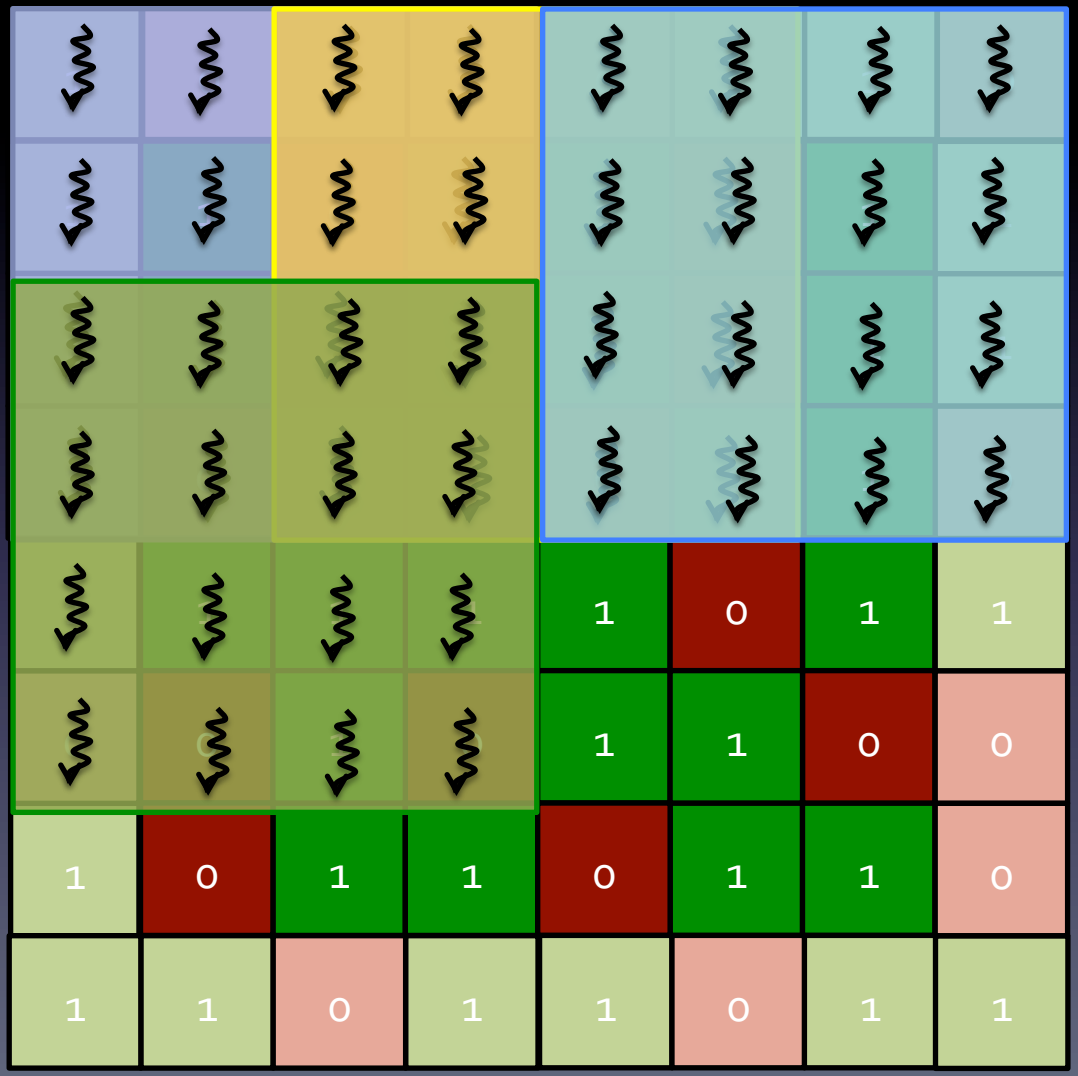
Game of Life

Game of Life

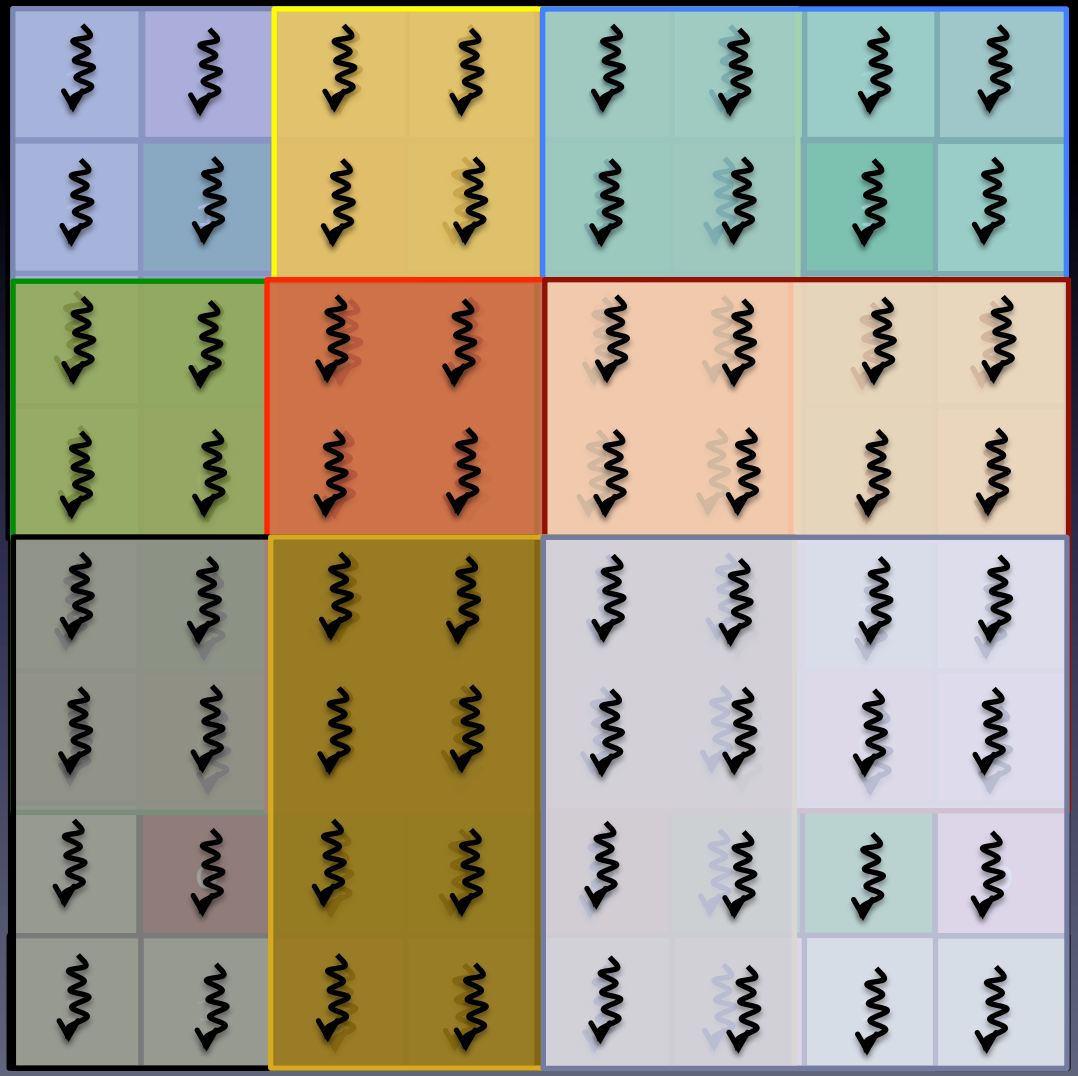


1	1	1	1	1	0	1	1
0	0	1	0	1	1	0	0
1	0	1	1	0	1	1	0
1	1	0	1	1	0	1	1

Game of Life



Game of Life



Shared GOL Kernel

```
__global__ void GOL(int dim, int *grid, int *newGrid)
{
        int iy = (blockDim.y -2) * blockIdx.y + threadIdx.y;
        int ix = (blockDim.x -2) * blockIdx.x + threadIdx.x;
        int id = iy * (dim+2) + ix;

        int i = threadIdx.y;
        int j = threadIdx.x;
        int numNeighbors;

       __shared__ int s_grid[BLOCK_SIZE_y][BLOCK_SIZE_x];

       if (ix <= dim+1 && iy <= dim+1)
           s_grid[i][j] = grid[id];

       //Sync all threads in block
       __syncthreads();

       if (iy <= dim && ix <= dim) {
           if(i != 0 && i !=blockDim.y-1 && j != 0 && j !=blockDim.x-1) {
               // Get the number of neighbors for a given grid point
               numNeighbors = s_grid[i+1][j] + s_grid[i-1][j] //upper lower
                            + s_grid[i][j+1] + s_grid[i][j-1] //right left
                            + s_grid[i+1][j+1] + s_grid[i-1][j-1] //diagonals
                            + s_grid[i-1][j+1] + s_grid[i+1][j-1];

               int cell = s_grid[i][j];

               if (cell == 1 && numNeighbors < 2)
                   newGrid[id] = 0;
               else if (cell == 1 && (numNeighbors == 2 || numNeighbors == 3))
                   newGrid[id] = 1;
               else if (cell == 1 && numNeighbors > 3)
                   newGrid[id] = 0;
               else if (cell == 0 && numNeighbors == 3)
                   newGrid[id] = 1;
               else
                   newGrid[id] = cell;
           }
       }
}
```

Warps

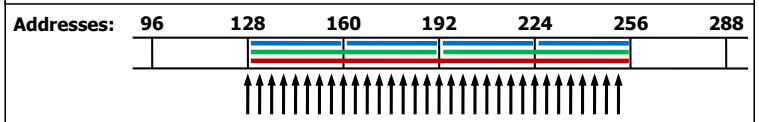
- 32 consecutive threads
- Instructions dealt at warp level
- Blocks should be multiple of warp size
- Warps on each SM depends on resource usage
- All warps of a given block stay on a single SM

Global Coalescing

- 4 consecutive bytes by each thread in warp: single 128B transaction
- >4 bytes broken down into 128B transactions
- 2D arrays: row should be multiple of warp size

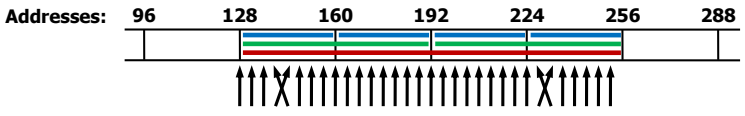
Global Coalescing

Aligned and sequential



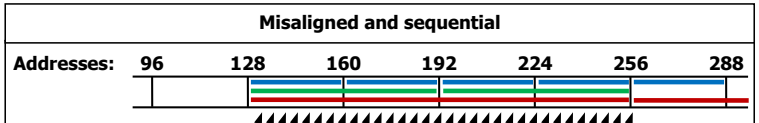
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Aligned and non-sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Misaligned and sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

CUDA C Programing guide

2D Global Arrays

```
int width = 64, height = 64;
float *d_a
size_t pitch;

cudaMallocPitch(&d_a, &pitch, width*sizeof(float), height);
cudaMemcpy2D(d_a, pitch, h_a, pitch, width, height, cudaMemcpyDeviceToHost);

__global__ void kern(float *d_a, size_t pitch, int width, int height) {
    // d_a[10][10]
    float* row = (float*)(d_a + 10 * pitch);
    float  elm = row[10];

}
```