Семинар 18
Технология CUDA
Conway's Game of Life

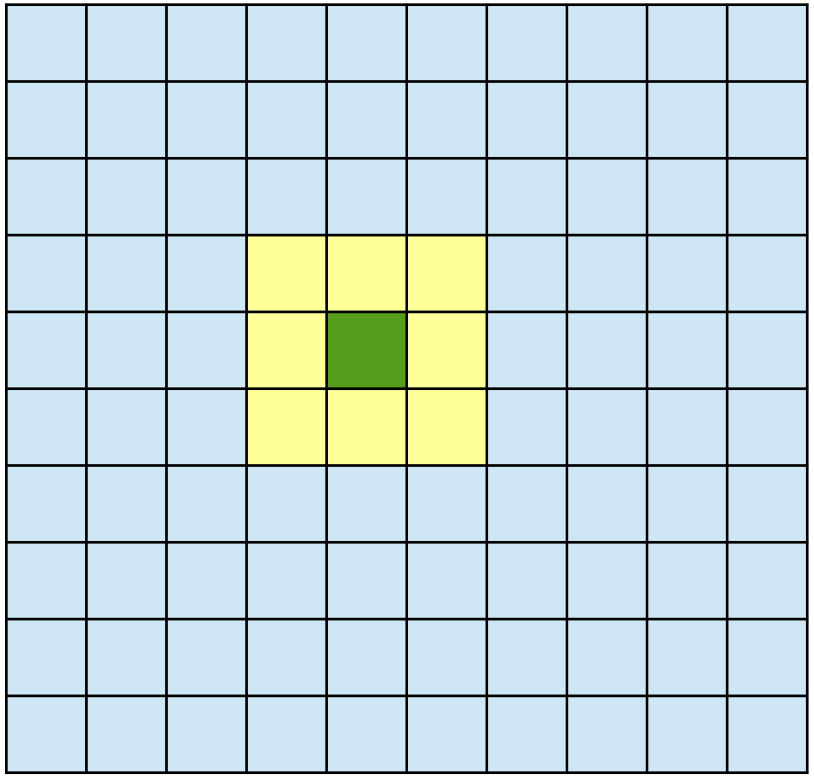
Михаил Курносов

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Цикл семинаров «Основы параллельного программирования»
Институт физики полупроводников им. А. В. Ржанова СО РАН
Новосибирск, 2015

Conway's Game of Life

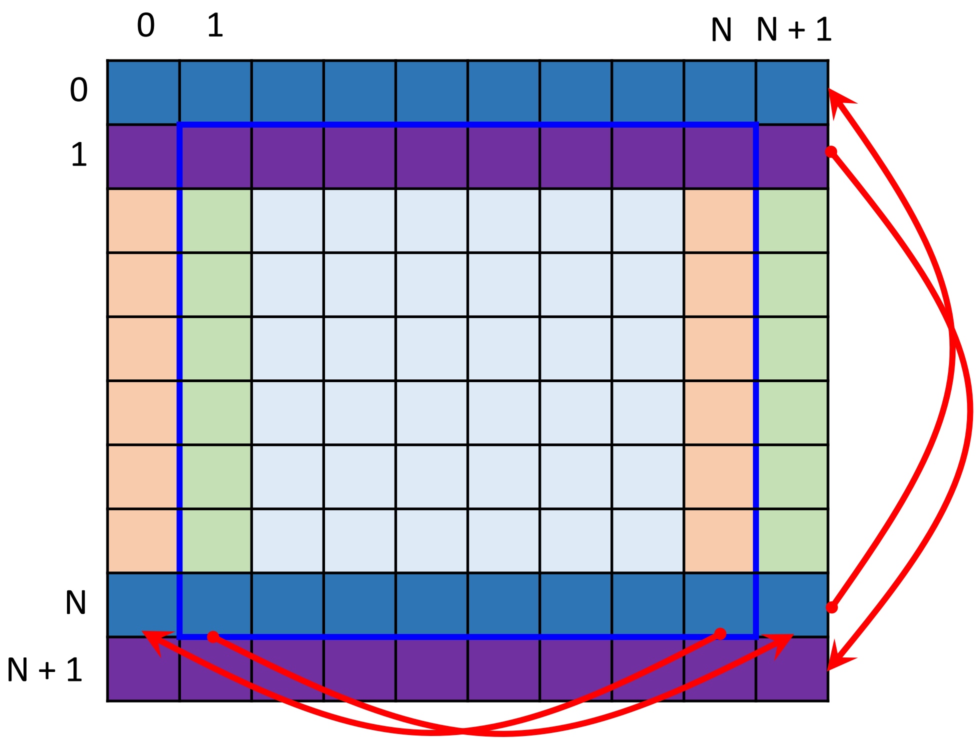
- **Игра «Жизнь»** (Game of Life, Дж. Конвей, 1970)
- Игровое поле — размеченная на клетки плоскость
- Каждая клетка может находиться в двух состояниях: «живая» и «мёртвая», и имеет восемь соседей
- Распределение живых клеток в начале игры называется первым поколением. Каждое следующее поколение рассчитывается на основе предыдущего:
  1) в мертвой клетке, рядом с которой три живые клетки, зарождается жизнь
  2) если у живой клетки есть две или три живые соседки, то эта клетка продолжает жить; в противном случае (соседей < 2 или > 3) клетка умирает

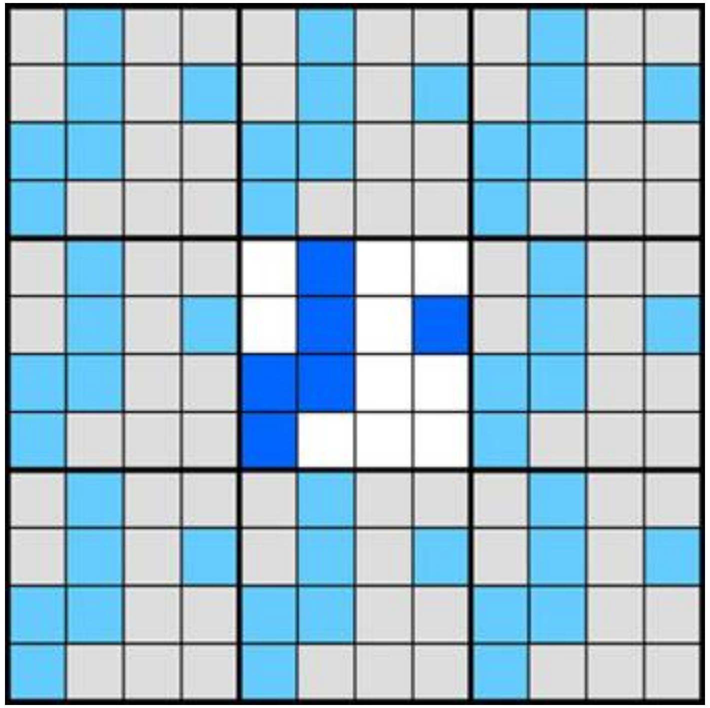


http://en.wikipedia.org/wiki/Conway's_Game_of_Life

Периодические граничные условия (periodic boundary conditions)

- **Как вычислять состояния граничных ячеек (ячеек слева, справа, снизу, сверху может не существовать)?**
- Одно из решений – **периодические граничные условия (periodic boundary conditions)**
- Игровое поле бесконечно продолжается по всем направлениям
- В массиве требуется хранить теневые ячейки (ghost cells, shadow cells)

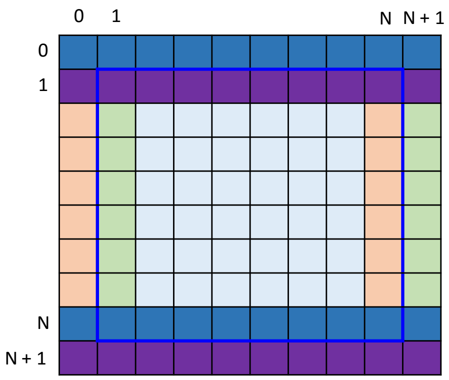




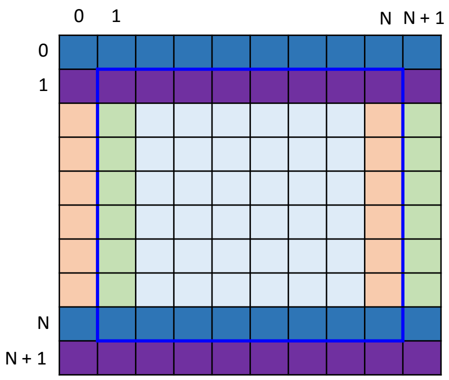
https://www.pdc.kth.se/education/tutorials/summer-school/mpi-exercises/mpi-lab-1-program-structure-and-point-to-point-communication-in-mpi/background-for-the-game-of-life

Последовательная реализация (1_gol/gol.c)

```c
#define IND(i, j) ((i) * (N + 2) + (j))

enum {
    N = 1024,
    ITERS_MAX = 1 << 10
};

typedef uint8_t cell_t;

int main(int argc, char* argv[])
{
    // Grid with periodic boundary conditions (ghost cells)
    size_t ncells = (N + 2) * (N + 2);
    size_t size = sizeof(cell_t) * ncells;
    cell_t *grid = malloc(size);
    cell_t *newgrid = malloc(size);

    // Initial population
    srand(0);
    for (int i = 1; i <= N; i++)
        for (int j = 1; j <= N; j++)
            grid[IND(i, j)] = rand() % 2;
```



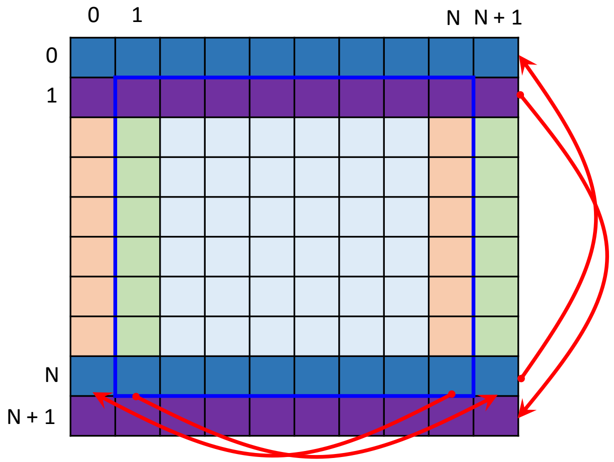
grid[N + 2][N + 2]



newgrid[N + 2][N + 2]

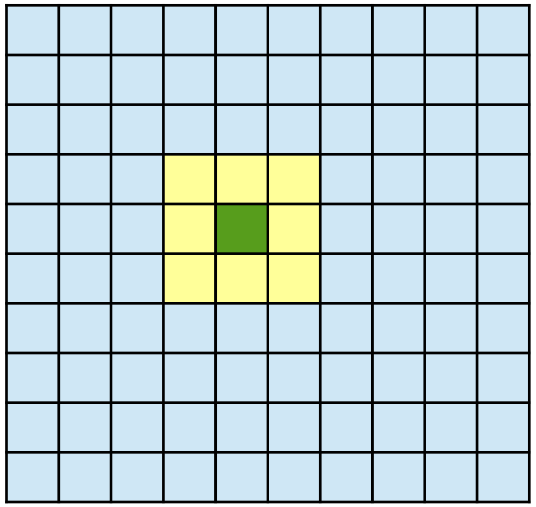
Последовательная реализация (продолжение)

```
double t = wtime();
int iter;
for (iter = 0; iter < ITERS_MAX; iter++) {
    // Copy ghost columns
    for (int i = 1; i <= N; i++) {
        grid[IND(i, 0)] = grid[IND(i, N)];      // left ghost column
        grid[IND(i, N + 1)] = grid[IND(i, 1)];  // right ghost column
    }
    // Copy ghost rows
    for (int i = 0; i <= N + 1; i++) {
        grid[IND(0, i)] = grid[IND(N, i)];      // top ghost row
        grid[IND(N + 1, i)] = grid[IND(1, i)];  // bottom ghost row
    }
```



Последовательная реализация (продолжение)

```
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            int nneibs = grid[IND(i + 1, j)] + grid[IND(i - 1, j)] +
                         grid[IND(i, j + 1)] + grid[IND(i, j - 1)] +
                         grid[IND(i + 1, j + 1)] + grid[IND(i - 1, j - 1)] +
                         grid[IND(i - 1, j + 1)] + grid[IND(i + 1, j - 1)];

            cell_t state = grid[IND(i, j)];
            cell_t newstate = state;
            if (state == 1 && nneibs < 2)
                newstate = 0;
            else if (state == 1 && (nneibs == 2 || nneibs == 3))
                newstate = 1;
            else if (state == 1 && nneibs > 3)
                newstate = 0;
            else if (state == 0 && nneibs == 3)
                newstate = 1;
            newgrid[IND(i, j)] = newstate;
        }
    }
    cell_t *p = grid; grid = newgrid; newgrid = p;
}
t = wtime() - t;
```



Последовательная реализация (окончание)

```c
    size_t total = 0;
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++)
            total += grid[IND(i, j)];
    }
    printf("Game of Life: N = %d, iterations = %d\n", N, iter);
    printf("Total alive cells: %lu\n", total);
    printf("Iters per sec.: %.2f\n", iter / t);
    printf("Total time (sec.): %.6f\n", t);

    free(grid);
    free(newgrid);
    return 0;
}
```

Cluster Oak / cngpu1 (Intel Core i5-3320M)

```
Game of Life: N = 1024, iterations = 1024
Total alive cells: 47026
Iters per sec.: 280.05
Total time (sec.): 3.656496
```

Модификация последовательной реализации (2_gol_state)

```
int states[2][9] = {
     {0, 0, 0, 1, 0, 0, 0, 0, 0},  /* New states for a dead cell */
     {0, 0, 1, 1, 0, 0, 0, 0, 0}   /* New states for an alive cell */
 };

 for (int i = 1; i <= N; i++) {
     for (int j = 1; j <= N; j++) {
         int nneibs = grid[IND(i + 1, j)] + grid[IND(i - 1, j)] +
                      grid[IND(i, j + 1)] + grid[IND(i, j - 1)] +
                      grid[IND(i + 1, j + 1)] + grid[IND(i - 1, j - 1)] +
                      grid[IND(i - 1, j + 1)] + grid[IND(i + 1, j - 1)];

         cell_t state = grid[IND(i, j)];
         newgrid[IND(i, j)] = states[state][nneibs];
     }
 }
```

Cluster Oak / cngpu1 (Intel Core i5-3320M)

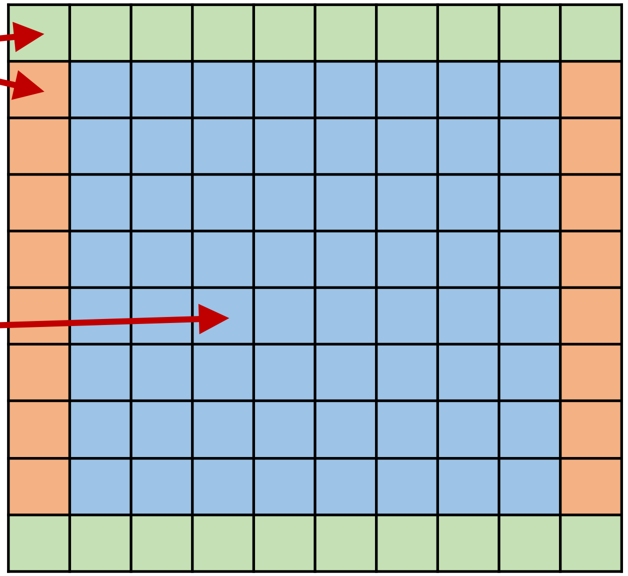
```
Game of Life: N = 1024, iterations = 1024
Total alive cells: 47026
Iters per sec.: 448.07
Total time (sec.): 2.285372      Speedup x1.6
```

Реализация на CUDA (2_gol_cuda)

```
#define IND(i, j) ((i) * (N + 2) + (j))
enum {
    N = 1024,
    ITERS_MAX = 1 << 10,
    BLOCK_SIZE = 16
};

typedef uint8_t cell_t;

int main(int argc, char* argv[])
{
    // Grid with periodic boundary conditions (ghost cells)
    size_t ncells = (N + 2) * (N + 2);
    size_t size = sizeof(cell_t) * ncells;
    cell_t *grid = (cell_t *)malloc(size);

    // Initial population
    srand(0);
    for (int i = 1; i <= N; i++)
        for (int j = 1; j <= N; j++)
            grid[IND(i, j)] = rand() % 2;
```

Реализация на CUDA (2_gol_cuda)

```
cell_t *d_grid, *d_newgrid;
double tmem = -wtime();
cudaMalloc((void **)&d_grid, size);
cudaMalloc((void **)&d_newgrid, size);
cudaMemcpy(d_grid, grid, size, cudaMemcpyHostToDevice);
tmem += wtime();

// 1d drids for copying ghost cells
dim3 block(BLOCK_SIZE, 1, 1);
dim3 cols_grid((N + block.x - 1) / block.x, 1, 1);
dim3 rows_grid((N + 2 + block.x - 1) / block.x, 1, 1);

// 2d grid for updating cells: one thread per cell
dim3 block2d(BLOCK_SIZE, BLOCK_SIZE, 1);
int nblocks = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
dim3 grid2d(nblocks, nblocks, 1);
```

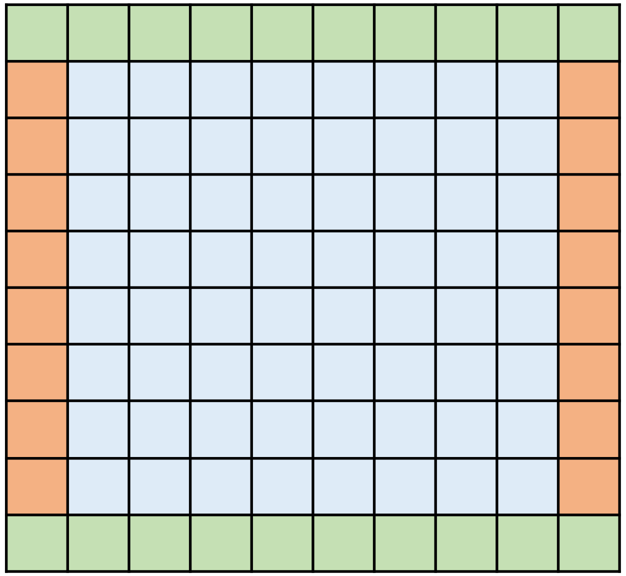


Реализация на CUDA (2_gol_cuda)

```
double t = wtime();
int iter = 0;
for (iter = 0; iter < ITERS_MAX; iter++) {
    // Copy ghost cells: 1d grid for rows, 1d grid for columns
    copy_ghost_cols<<<cols_grid, block>>>(d_grid, N);
    copy_ghost_rows<<<rows_grid, block>>>(d_grid, N);

    // Update cells: 2d grid
    update_cells<<<grid2d, block2d>>>(d_grid, d_newgrid, N);

    // Swap grids
    cell_t *p = d_grid; d_grid = d_newgrid; d_newgrid = p;
}
cudaDeviceSynchronize();
t = wtime() - t;

tmem -= wtime();
cudaMemcpy(grid, d_grid, size, cudaMemcpyDeviceToHost);
tmem += wtime();
```

Реализация на CUDA (2_gol_cuda)

```
__global__ void copy_ghost_rows(cell_t *grid, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i <= n + 1) {
        // Bottom ghost row: [N + 1][0..N + 1] <== [1][0..N + 1]
        grid[IND(N + 1, i)] = grid[IND(1, i)];
        // Top ghost row: [0][0..N + 1] <== [N][0..N + 1]
        grid[IND(0, i)] = grid[IND(N, i)];
    }
}

__global__ void copy_ghost_cols(cell_t *grid, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x + 1;
    if (i <= n) {
        // Right ghost column: [1..N][N + 1] <== [1..N][1]
        grid[IND(i, N + 1)] = grid[IND(i, 1)];
        // Left ghost column: [1..N][1] <== [1..N][N]
        grid[IND(i, 0)] = grid[IND(i, N)];
    }
}
```



Реализация на CUDA (2_gol_cuda)

```
__global__ void update_cells(cell_t *grid, cell_t *newgrid, int n)
{
    int i = blockIdx.y * blockDim.y + threadIdx.y + 1;
    int j = blockIdx.x * blockDim.x + threadIdx.x + 1;

    if (i <= n && j <= n) {
        int states[2][9] = {
            {0, 0, 0, 1, 0, 0, 0, 0, 0},  /* New states for a dead cell */
            {0, 0, 1, 1, 0, 0, 0, 0, 0}   /* New states for an alive cell */
        };

        int nneibs = grid[IND(i + 1, j)] + grid[IND(i - 1, j)] +
                     grid[IND(i, j + 1)] + grid[IND(i, j - 1)] +
                     grid[IND(i + 1, j + 1)] + grid[IND(i - 1, j - 1)] +
                     grid[IND(i - 1, j + 1)] + grid[IND(i + 1, j - 1)];

        cell_t state = grid[IND(i, j)];
        newgrid[IND(i, j)] = states[state][nneibs];
    }
}
```

Реализация на CUDA (2_gol_cuda)

```
    size_t total = 0;
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++)
            total += grid[IND(i, j)];
    }
    printf("Game of Life: N = %d, iterations = %d\n", N, iter);
    printf("Total alive cells: %lu\n", total);
    printf("Iterations time (sec.): %.6f\n", t);
    printf("GPU memory ops. time (sec.): %.6f\n", tmem);
    printf("Iters per sec.: %.2f\n", iter / t);
    printf("Total time (sec.): %.6f\n", t + tmem);

    free(grid);
    cudaFree(d_grid);
    cudaFree(d_newgrid);
    return 0;
}
```

Cluster Oak / cngpu1 (GeForce GTX 680)

```
Game of Life: N = 1024, iterations = 1024
Total alive cells: 47026
Iterations time (sec.): 0.911321
GPU memory ops. time (sec.): 0.238265
Iters per sec.: 1123.64
Total time (sec.): 1.149586
```

Speedup 1.99

Реализация на CUDA v2 (2_gol_cuda_constant)

```
__constant__ int states[2][9] = {
    {0, 0, 0, 1, 0, 0, 0, 0, 0},  /* New states for a dead cell */
    {0, 0, 1, 1, 0, 0, 0, 0, 0}   /* New states for an alive cell */
};

__global__ void update_cells(cell_t *grid, cell_t *newgrid, int n)
{
    int i = blockIdx.y * blockDim.y + threadIdx.y + 1;
    int j = blockIdx.x * blockDim.x + threadIdx.x + 1;

    if (i <= n && j <= n) {
        int nneibs = grid[IND(i + 1, j)] + grid[IND(i - 1, j)] +
                     grid[IND(i, j + 1)] + grid[IND(i, j - 1)] +
                     grid[IND(i + 1, j + 1)] + grid[IND(i - 1, j - 1)] +
                     grid[IND(i - 1, j + 1)] + grid[IND(i + 1, j - 1)];

        cell_t state = grid[IND(i, j)];
        newgrid[IND(i, j)] = states[state][nneibs];
    }
}
```

Реализация на CUDA v2 (2_gol_cuda_constant)

```
__constant__ int states[2][9] = {
    {0, 0, 0, 1, 0, 0, 0, 0, 0},  /* New states for a dead cell */
    {0, 0, 1, 1, 0, 0, 0, 0, 0}   /* New states for an alive cell */
};

__global__ void update_cells(cell_t *grid, cell_t *newgrid, int n)
{
    int i = blockIdx.y * blockDim.y + threadIdx.y + 1;
    int j = blockIdx.x * blockDim.x + threadIdx.x + 1;

    if (i <= n && j <= n) {
        int nneibs = grid[IND(i + 1, j)] + grid[IND(i - 1, j)] +
                     grid[IND(i, j + 1)] + grid[IND(i, j - 1)] +
                     grid[IND(i + 1, j + 1)] +
                     grid[IND(i - 1, j + 1)] +

        cell_t state = grid[IND(i, j)];
        newgrid[IND(i, j)] = states[state][nnei
    }
}
```

Cluster Oak / cngpu1 (GeForce GTX 680)

```
Game of Life: N = 1024, iterations = 1024
Total alive cells: 47026
Iterations time (sec.): 0.221800
GPU memory ops. time (sec.): 0.231555
Iters per sec.: 4616.77
Total time (sec.): 0.453355     Speedup 5.0
```

CUDA GPU Occupancy Calculator

http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

В программе блоки
1d: 1x16
2d: 16x16

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

...te Capability (click):	3,0	(Help)
1.b) Select Sh... Size Config (bytes)	49152	
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	32	
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

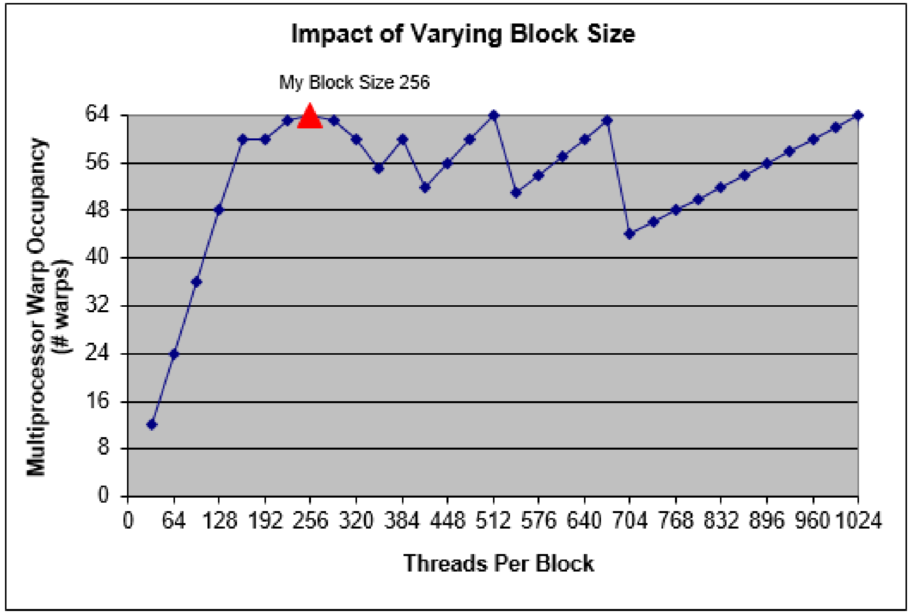
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	8	
Occupancy of each Multiprocessor	100%	

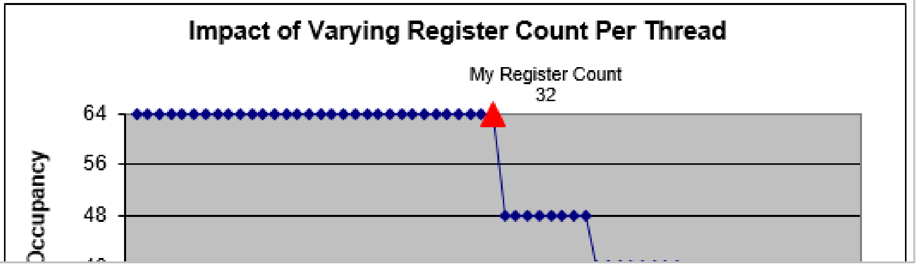
Physical Limits for GPU Compute Capability:	**3,0**
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	256
Warp allocation granularity	4
Maximum Thread Block Size	1024

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.





Реализация на CUDA v2 (2_gol_cuda_occupancy)

```
enum {
    BLOCK_1D_SIZE = 1024, BLOCK_2D_SIZE = 32
};

int main(int argc, char* argv[])
{
    // ...
    // 1d grids for copying ghost cells
    dim3 block(BLOCK_1D_SIZE, 1, 1);
    dim3 cols_grid((N + block.x - 1) / block.x, 1, 1);
    dim3 rows_grid((N + 2 + block.x - 1) / block.x, 1, 1);

    // 2d grid for updating cells: one thread per cell
    dim3 block2d(BLOCK_2D_SIZE, BLOCK_2D_SIZE, 1);
    int nblocks = (N + BLOCK_2D_SIZE - 1) / BLOCK_2D_SIZE;
    dim3 grid2d(nblocks, nblocks, 1);
    // ...
}
```

Реализация на CUDA v2 (2_gol_cuda_occupancy)

```
enum {
    BLOCK_1D_SIZE = 1024, BLOCK_2D_SIZE = 32
};

int main(int argc, char* argv[])
{
    // ...
    // 1d grids for copying ghost cells
    dim3 block(BLOCK_1D_SIZE, 1, 1);
    dim3 cols_grid((N + block.x - 1) / block.x, 1, 1);
    dim3 rows_grid((N + 2 + block.x - 1) / block.x, 1, 1);

    // 2d grid for updating cells: one th
    dim3 block2d(BLOCK_2D_SIZE, BLOCK_2D
    int nblocks = (N + BLOCK_2D_SIZE - 1
    dim3 grid2d(nblocks, nblocks, 1);
    // ...
}
```

Cluster Oak / cngpu1 (GeForce GTX 680)

```
Game of Life: N = 1024, iterations = 1024
Total alive cells: 47026
Iterations time (sec.): 0.169622
GPU memory ops. time (sec.): 0.238457
Iters per sec.: 6036.95
Total time (sec.): 0.408079      Speedup 5.7
```

Задание

- Реализовать версию с использованием разделяемой памяти (__shared__):
  каждый 2d-блок потоков загружает в shared-массив свои и пограничные ячейки
- Шаблон программы находится в каталоге _task_gol_cuda_shared
- В программе имеется логическая ошибка – результаты работы программы не совпадают
  с результатами работы последовательной версии (по значению “total alive cells”)