

# Лекция 12

# Технология CUDA

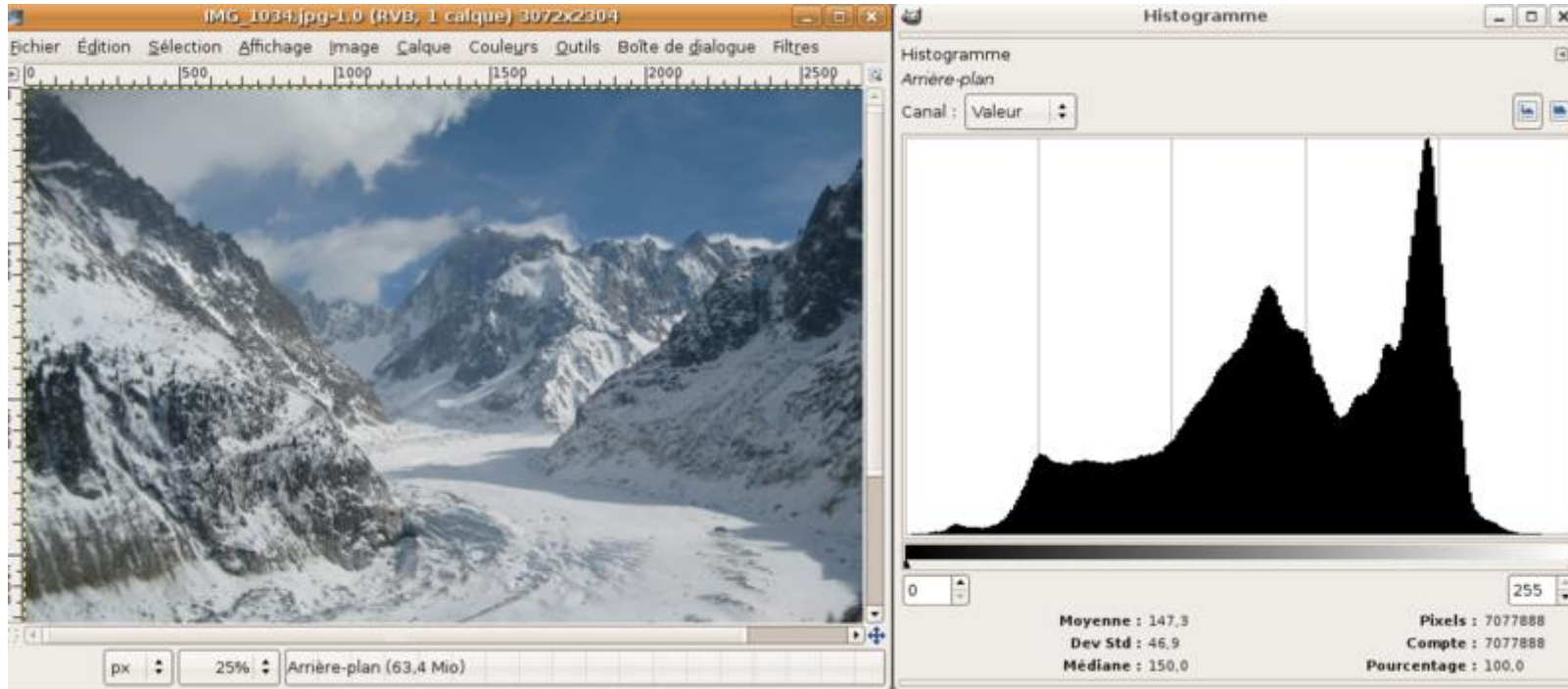
**Курносов Михаил Георгиевич**

E-mail: [mkurnosov@gmail.com](mailto:mkurnosov@gmail.com)

WWW: [www.mkurnosov.net](http://www.mkurnosov.net)

Курс «Высокопроизводительные вычислительные системы»  
Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)  
Осенний семестр, 2015

# Гистограмма цветов изображения



- Задано изображение – двумерный массив целых чисел из интервала  $[0..255]$
- Необходимо построить гистограмму, таблицу  $hist[0..255]$ , элемент  $hist[i]$  которой равен числу пикселей с цветом  $i$

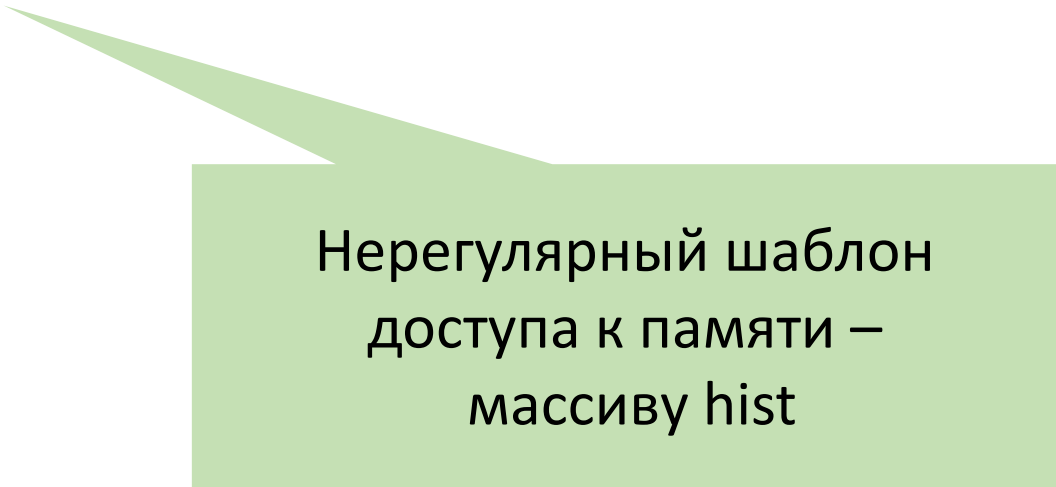
# Гистограмма цветов изображения (CPU)

```
// Выделение памяти под изображение
size_t size = sizeof(uint8_t) * width * height;
uint8_t *image = (uint8_t *)malloc(size);
if (image == NULL) {
    fprintf(stderr, "Allocation error.\n");
    exit(EXIT_FAILURE);
}
// Инициализация изображения
srand(0);
for (size_t i = 0; i < size; i++)
    image[i] = (rand() / (double)RAND_MAX) * 255;

// Инициализация гистограммы
int hist[256];
memset(hist, 0, sizeof(*hist) * 256);
```

# Гистограмма цветов изображения (CPU)

```
void hist_host(uint8_t *image, int width, int height,  
               int *hist)  
{  
    for (int i = 0; i < height; i++)  
        for (int j = 0; j < width; j++)  
            hist[image[i * width + j]]++;  
}
```



Нерегулярный шаблон  
доступа к памяти –  
массиву hist

# Гистограмма цветов изображения (GPU)

Каждый поток обрабатывает один пиксель изображения

```
// Гистограмма в памяти GPU
```

```
int *d_hist = NULL;
```

```
cudaMalloc((void **)&d_hist, sizeof(*d_hist) * 256);
```

```
cudaMemset(d_hist, 0, sizeof(*d_hist) * 256);
```

```
uint8_t *d_image = NULL;
```

```
cudaMalloc((void **)&d_image, size);
```

```
cudaMemcpy(d_image, image, size, cudaMemcpyHostToDevice);
```

```
int threadsPerBlock = 1024;
```

```
int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;
```

```
hist_gpu<<<blocksPerGrid, threadsPerBlock>>>(d_image, width,  
                                              height, d_hist);
```

```
cudaMemcpy(hist, d_hist, sizeof(*d_hist) * 256, cudaMemcpyDeviceToHost);
```

# Гистограмма цветов изображения (GPU)

Каждый поток обрабатывает один пиксель изображения

```
__global__ void hist_gpu(uint8_t *image, int width, int height,
                          int *hist)
{
    size_t size = width * height;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size) {
        hist[image[i]]++;
    }
}
```

# Гистограмма цветов изображения: тест

```
$ ./hist
```

```
Sum (CPU) = 104857600.000000
```

```
CUDA kernel launch with 102400 blocks of 1024 threads
```

```
Sum (GPU) = 3928061.000000
```

```
CPU version (sec.): 0.052969
```

```
GPU version (sec.): 0.068165
```

```
Memory ops. (sec.): 0.000019
```

```
Speedup: 0.78
```

```
Speedup (with mem ops.): 0.78
```

```
$ ./hist
```

```
Sum (CPU) = 104857600.000000
```

```
CUDA kernel launch with 102400 blocks of 1024 threads
```

```
Sum (GPU) = 3931478.000000
```

```
CPU version (sec.): 0.052965
```

```
GPU version (sec.): 0.068171
```

```
Memory ops. (sec.): 0.000020
```

```
Speedup: 0.78
```

```
Speedup (with mem ops.): 0.78
```

# Гистограмма цветов изображения (GPU)

Каждый поток обрабатывает один пиксель изображения

```
__global__ void hist_gpu(uint8_t *image, int width, int height,
                        int *hist)
{
    size_t size = width * height;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size) {
        hist[image[i]]++;
    }
}
```

Потоки одновременно читают и записывают данные  
в одни и те же ячейки таблицы hist[0..255] – data race



# CUDA Atomic Functions

- **Атомарная функция** (CUDA atomic function) – функция, выполняющая операцию *read-modify-write* (RMW) над 32 или 64 битным значением в глобальной или разделяемой памяти GPU
- `int atomicAdd(int* address, int val);`
- `unsigned long long int atomicAdd(unsigned long long int* address,`  
▪ `unsigned long long int val);`
- `float atomicAdd(float* address, float val);   // compute capability >= 2.0`
- `int atomicSub(int* address, int val);`
- `int atomicMin(int* address, int val);`
- `int atomicAnd(int* address, int val);`
- `int atomicXor(int* address, int val);`
- `int atomicCAS(int* address, int compare, int val);`
- `...`

# atomicAdd для double (CAS-based)

```
__device__ double atomicAdd(double* address, double val)
{
    unsigned long long int* addr = (unsigned long long int*)address;
    unsigned long long int old = *addr, assumed;

    do {
        assumed = old;
        old = atomicCAS(addr, assumed,
                        __double_as_longlong(val +
                        __longlong_as_double(assumed)));
    } while (assumed != old);

    return __longlong_as_double(old);
}
```

**int atomicCAS(int\* address, int compare, int val);**

1. Загружает в old значение по адресу address
2. Записывает обратно значение:  
(old == compare) ? val : old
3. Возвращает old

# Гистограмма цветов изображения (GPU)

```
__global__ void hist_gpu_atomic(uint8_t *image, int width,
                                int height, int *hist)
{
    size_t size = width * height;
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < size) {
        atomicAdd(&hist[image[i]], 1);
    }
}
```

# Гистограмма цветов изображения (GPU)

```
__global__ void hist_gpu_atomic(uint8_t *image, int width,  
                                int height, int *hist)  
{  
    size_t size = width * height;  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
  
    while (i < size) {  
        atomicAdd(&hist[image[i]], 1);  
        i += stride;  
    }  
}
```

Если пикселей больше  
числа потоков

# Редукция (reduction)

- **Задан** массив  $v[0..n - 1]$  из  $n$  элементов и ассоциативная операция  $\otimes$
- **Необходимо** вычислить результаты редукции

$$r = v[0] \otimes v[1] \otimes \dots \otimes v[n - 1]$$

- **Пример:**

$$v[0..5] = [5, 8, 3, 12, 1, 7]$$

$$r = (((((5 + 8) + 3) + 12) + 1) + 7) = 36$$

# Последовательная версия (float)

```
void reduce_cpu(float *v, int n, float *sum)
{
    float s = 0.0;
    for (int i = 0; i < n; i++)
        s += v[i];
    *sum = s;
}

int n = 10000;
for (size_t i = 0; i < n; i++)
    v[i] = i + 1.0;

reduce_cpu(v, n, &sum);
```

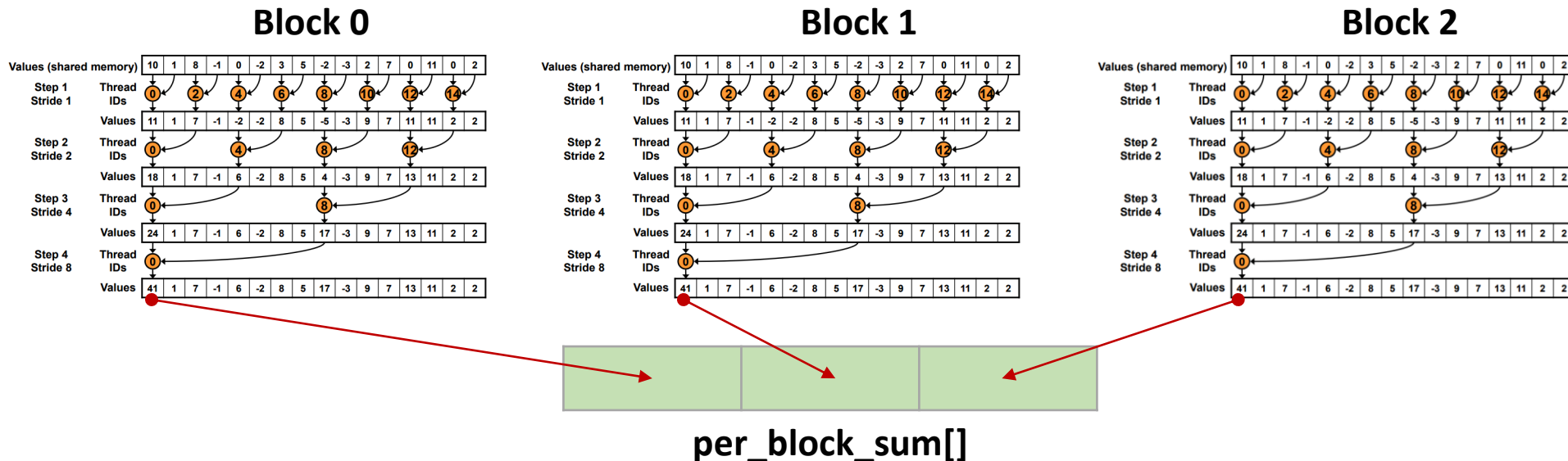
- **Floating-point Summation** // <http://www.drdobbs.com/floating-point-summation/184403224>
- David Goldberg. **What Every Computer Scientist Should Know About Floating-Point Arithmetic** // ACM Computing Surveys, Vol. 23, #1, March 1991, pp. 5-48.

```
# Real sum = 50005000
Sum (CPU) = 50002896.000000, err = 2104.000000
```

# Последовательная версия (int)

```
void reduce_cpu(int *v, int n, int *sum)
{
    int s = 0.0;
    for (int i = 0; i < n; i++)
        s += v[i];
    *sum = s;
}
```

# Параллельная редукция v1

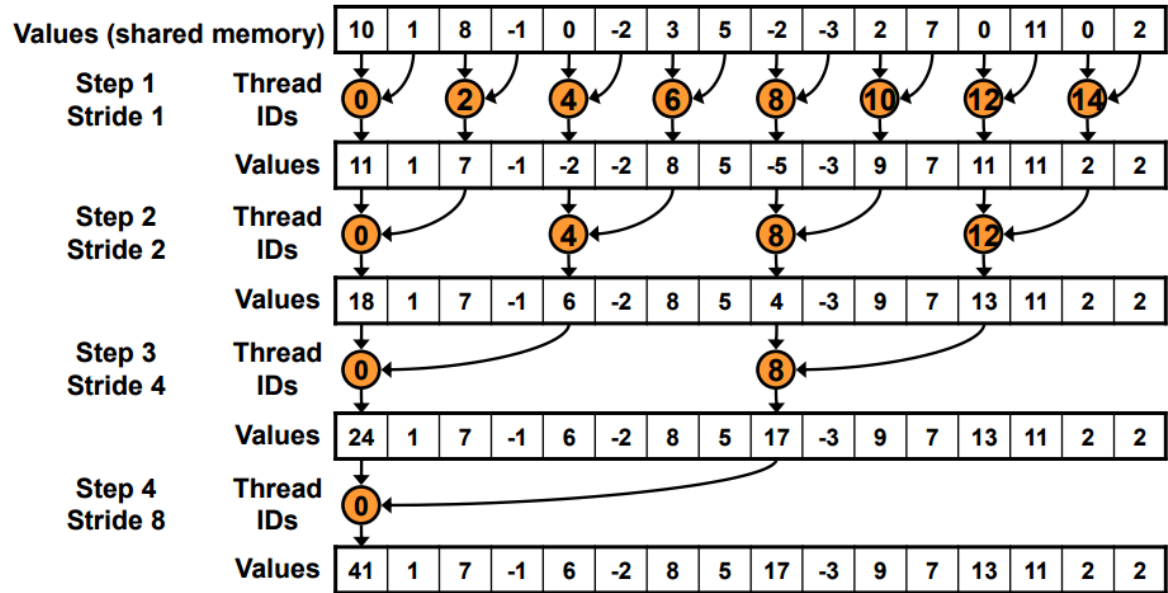


- Каждый блок потоков обрабатывает часть массива  $v[0..n - 1]$
- Поток 0 каждого блока сохраняет результат редукции в массив `per_block_sum[0..blocks - 1]`
- Для выполнения редукции массива `per_block_sum[]` нужен еще один шаг на GPU или CPU:

```
cudaMemcpy(sums, per_block_sum, sizeof(int) * blocks, cudaMemcpyDeviceToHost);  
int sum_gpu = 0;  
for (int i = 0; i < blocks; i++)  
    sum_gpu += sums[i];
```



# Параллельная редукция v1



```
for (int s = 1; s < blockDim.x; s *= 2) {  
    if (tid % (2 * s) == 0)  
        sdata[tid] += sdata[tid + s];  
}
```

- Каждый поток по своему номеру tid определяет с кем ему взаимодействовать
- На шаге 1 каждый 2-й поток взаимодействует с соседом справа на расстоянии  $s = 1$
- На шаге 2 каждый 4-й поток взаимодействует с соседом справа на расстоянии  $s = 2$
- На шаге 3 каждый 8-й поток взаимодействует с соседом справа на расстоянии  $s = 4$
- ...
- Всего шагов  $O(\log n)$

# Параллельная редукция v1

```
const int block_size = 1024;
const int n = 4 * (1 << 20);

int main()
{
    size_t size = sizeof(int) * n;
    int *v = (int *)malloc(size);
    for (size_t i = 0; i < n; i++)
        v[i] = i + 1;

    int sum;
    reduce_cpu(v, n, &sum);

    /* Allocate on device */
    int threads_per_block = block_size;
    int blocks = (n + threads_per_block - 1) / threads_per_block;
    int *dv, *per_block_sum;
    int *sums = (int *)malloc(sizeof(int) * blocks);
    tmem = -wtime();
    cudaMalloc((void **)&per_block_sum, sizeof(int) * blocks);
    cudaMalloc((void **)&dv, size);
    cudaMemcpy(dv, v, size, cudaMemcpyHostToDevice);
    tmem += wtime();
}
```

# Параллельная редукция v1

```
/* Compute per block sum: stage 1 */
tgpu = -wtime();
reduce_per_block<<<blocks, threads_per_block>>>(dv, n, per_block_sum);
cudaDeviceSynchronize();
tgpu += wtime();

tmem = -wtime();
cudaMemcpy(sums, per_block_sum, sizeof(int) * blocks, cudaMemcpyDeviceToHost);
tmem += wtime();

/* Compute block sum: stage 2 */
tgpu -= wtime();
int sum_gpu = 0;
for (int i = 0; i < blocks; i++)
    sum_gpu += sums[i];
tgpu += wtime();

printf("CPU version (sec.): %.6f\n", tcpu);
printf("GPU version (sec.): %.6f\n", tgpu);
printf("GPU bandwidth (GiB/s): %.2f\n", 1.0e-9 * size / (tgpu + tmem));
printf("Speedup: %.2f\n", tcpu / tgpu);
printf("Speedup (with mem ops.): %.2f\n", tcpu / (tgpu + tmem));
```

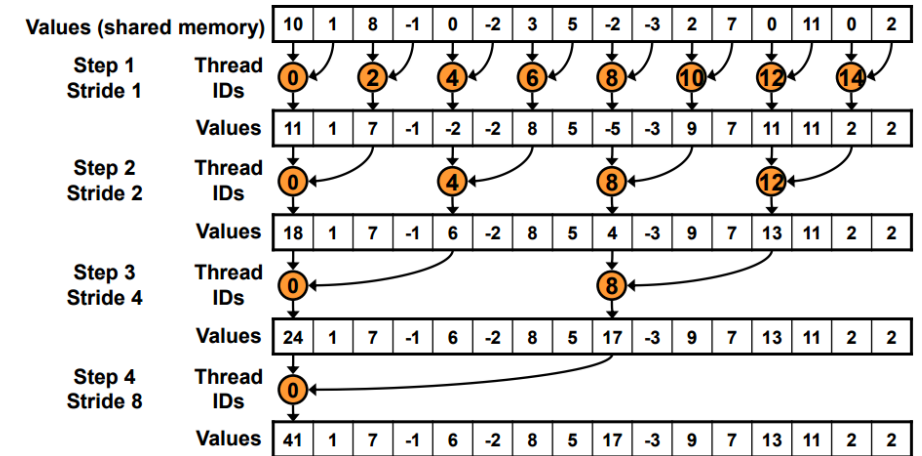
# Параллельная редукция v1

```
__global__ void reduce_per_block(int *v, int n, int *per_block_sum)
{
    __shared__ int sdata[block_size];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) {
        sdata[tid] = v[i];
        __syncthreads();

        for (int s = 1; s < blockDim.x; s *= 2) {
            if (tid % (2 * s) == 0)
                sdata[tid] += sdata[tid + s];
            __syncthreads();
        }

        if (tid == 0)
            per_block_sum[blockIdx.x] = sdata[0];
    }
}
```



```
CUDA kernel launch with 4096 blocks of 1024 threads
Sum (CPU) = 2097152
Sum (GPU) = 2097152
CPU version (sec.): 0.005956
GPU version (sec.): 0.002198
GPU bandwidth (GiB/s): 7.56
Speedup: 2.71
Speedup (with mem ops.): 2.68
```

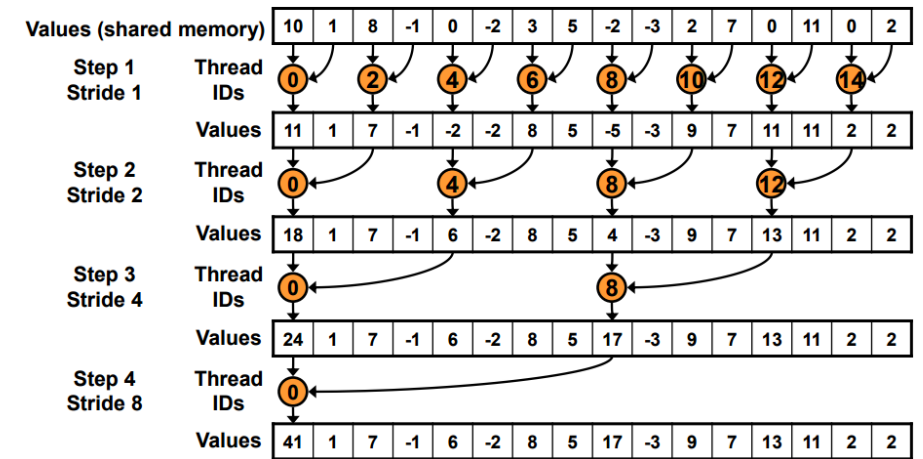
# Параллельная редукция v1

```
__global__ void reduce_per_block(int *v, int n, int *per_block_sum)
{
    __shared__ int sdata[block_size];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) {
        sdata[tid] = v[i];
        __syncthreads();

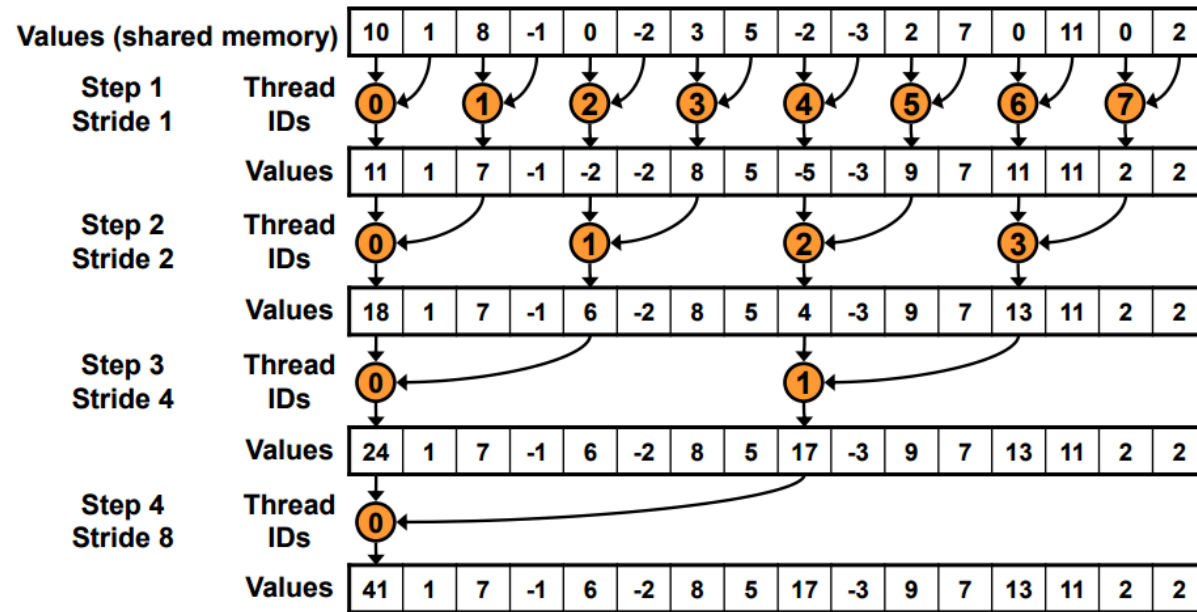
        for (int s = 1; s < blockDim.x; s *= 2) {
            if (tid % (2 * s) == 0)
                sdata[tid] += sdata[tid + s];
            __syncthreads();
        }

        if (tid == 0)
            per_block_sum[blockIdx.x] = sdata[0];
    }
}
```



Результат условия зависит от номера потока – часть потоков деактивируется (control flow divergence)

# Параллельная редукция v2



**Control flow divergence!**

```
for (int s = 1; s < blockDim.x; s *= 2) {
    if (tid % (2 * s) == 0)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}
```



```
for (int s = 1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x)
        sdata[index] += sdata[index + s];
    __syncthreads();
}
```

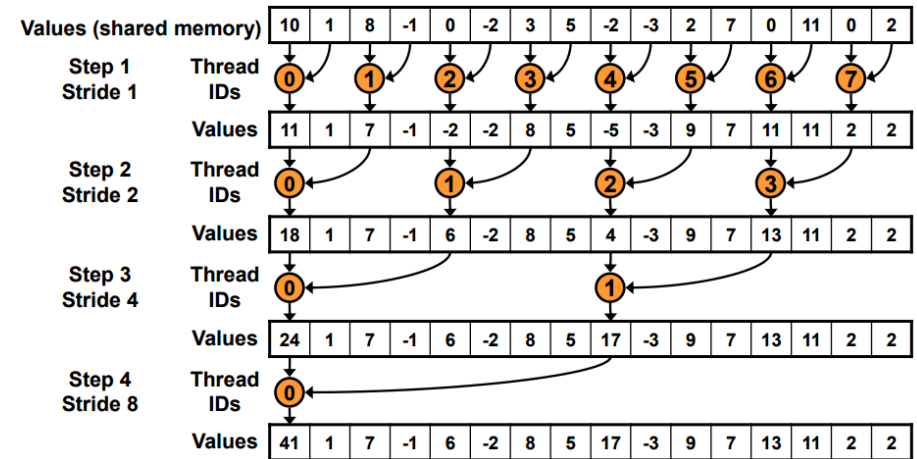
# Параллельная редукция v2

```
__global__ void reduce_per_block(int *v, int n, int *per_block_sum)
{
    __shared__ int sdata[block_size];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) {
        sdata[tid] = v[i];
        __syncthreads();

        for (int s = 1; s < blockDim.x; s *= 2) {
            int index = 2 * s * tid;

            if (index < blockDim.x)
                sdata[index] += sdata[index + s];
            __syncthreads();
        }
        if (tid == 0)
            per_block_sum[blockIdx.x] = sdata[0];
    }
}
```



```
CUDA kernel launch with 4096 blocks of 1024 threads
Sum (CPU) = 2097152
Sum (GPU) = 2097152
CPU version (sec.): 0.006481
GPU version (sec.): 0.001968
GPU bandwidth (GiB/s): 8.44
Speedup: 3.29
Speedup (with mem ops.): 3.26
```

# Параллельная редукция v2

```
__global__ void reduce_per_block(int *v, int n, int *per_block_sum)
{
    __shared__ int sdata[block_size];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) {
        sdata[tid] = v[i];
        __syncthreads();

        for (int s = 1; s < blockDim.x; s *= 2) {
            int index = 2 * s * tid;

            if (index < blockDim.x)
                sdata[index] += sdata[index + s];
            __syncthreads();
        }
        if (tid == 0)
            per_block_sum[blockIdx.x] = sdata[0];
    }
}
```

- Обращение к разделяемой памяти выполняется через 32 параллельно функционирующих банка
- Каждый банк 4 байта (настраивается)
- Номер банка = адрес % 32
- Одновременный доступ к одному банку сериализуется!

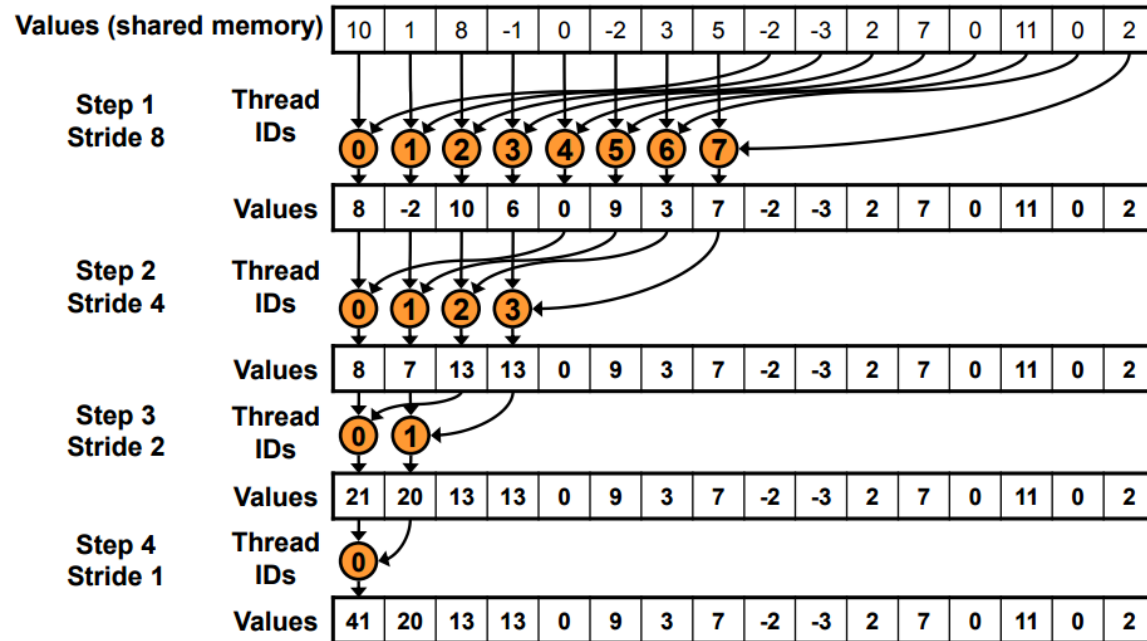
TID	S = 1	S = 2	S = 4
0	0-1 // banks 0, 1	0-2 // banks 0, 1	0-4 // banks 0, 1
1	2-3 // banks 2, 3	4-6 // banks 2, 3	
2	4-5 // banks 4, 5	8-10 // banks 4, 5	
3	6-7 // banks 6, 7	12-14 // banks 6, 7	24-28 // banks 24, 28
4	8-9 // banks 8, 9	16-18 // banks 8, 9	32-36 // banks 0, 2
...			
8		32-34 // banks 0, 2	
9		36-38 // banks 4, 6	



# Shared memory bank conflicts



# Параллельная редукция v3



## Bank conflicts

```
for (int s = 1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x)
        sdata[index] += sdata[index + s];
    __syncthreads();
}
```



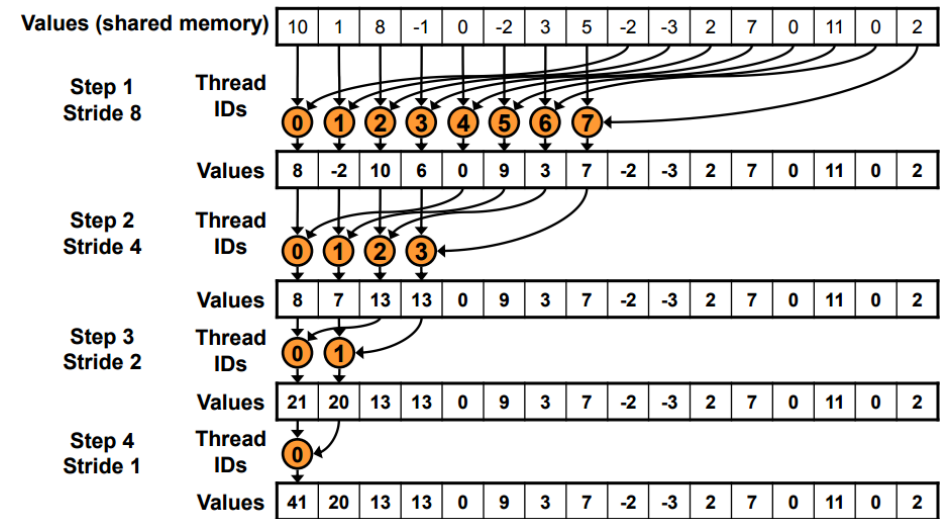
```
for (int s = blockDim.x / 2; s > 0; s >>= 1) {
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}
```

# Параллельная редукция v3

```
__global__ void reduce_per_block(int *v, int n, int *per_block_sum)
{
    __shared__ int sdata[block_size];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) {
        sdata[tid] = v[i];
        __syncthreads();

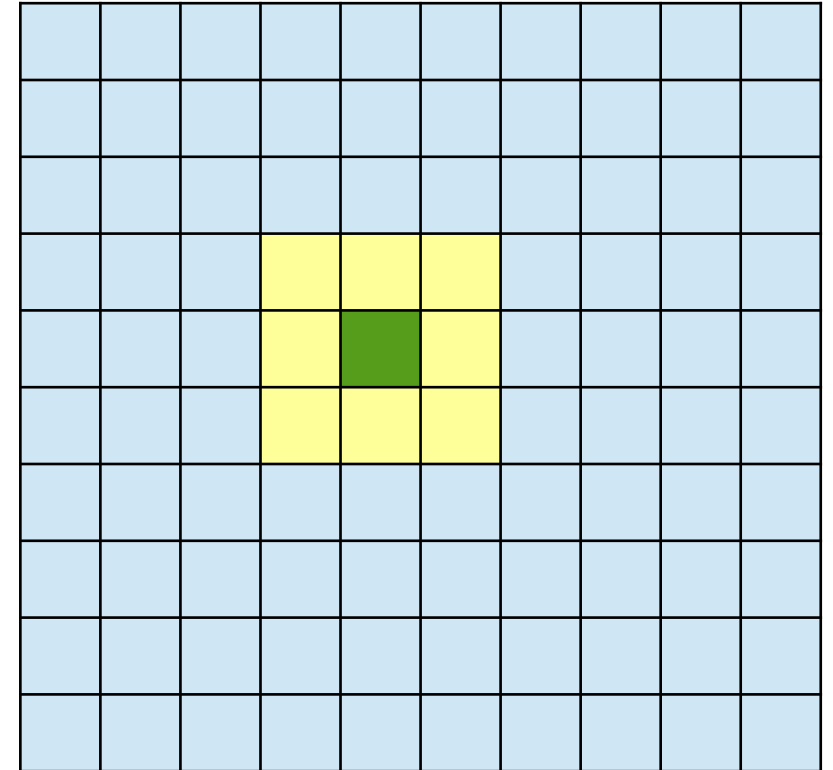
        for (int s = blockDim.x / 2; s > 0; s >>= 1) {
            if (tid < s)
                sdata[tid] += sdata[tid + s];
            __syncthreads();
        }
        if (tid == 0)
            per_block_sum[blockIdx.x] = sdata[0];
    }
}
```



```
CUDA kernel launch with 4096 blocks of 1024 threads
Sum (CPU) = 2097152
Sum (GPU) = 2097152
CPU version (sec.): 0.004661
GPU version (sec.): 0.001163
GPU bandwidth (GiB/s): 14.17
Speedup: 4.01
Speedup (with mem ops.): 3.94
```

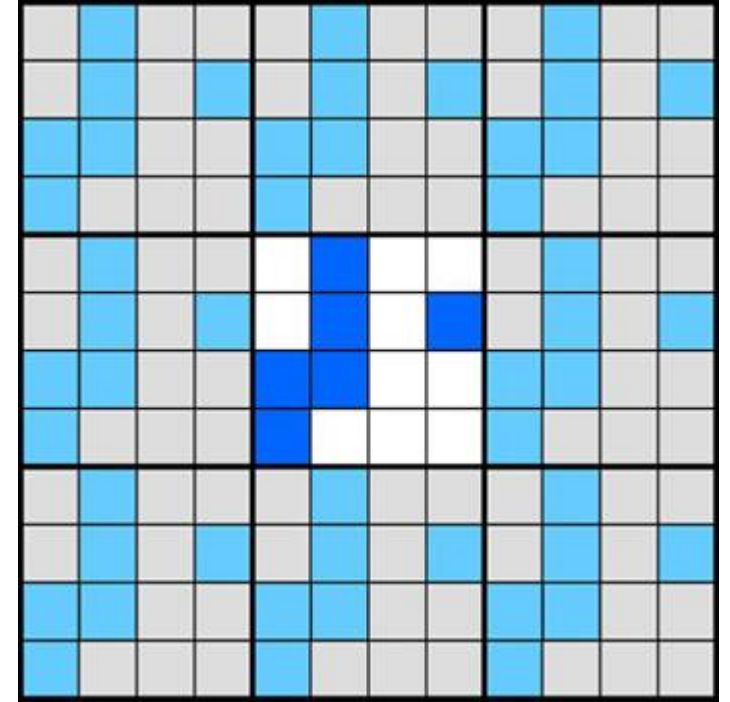
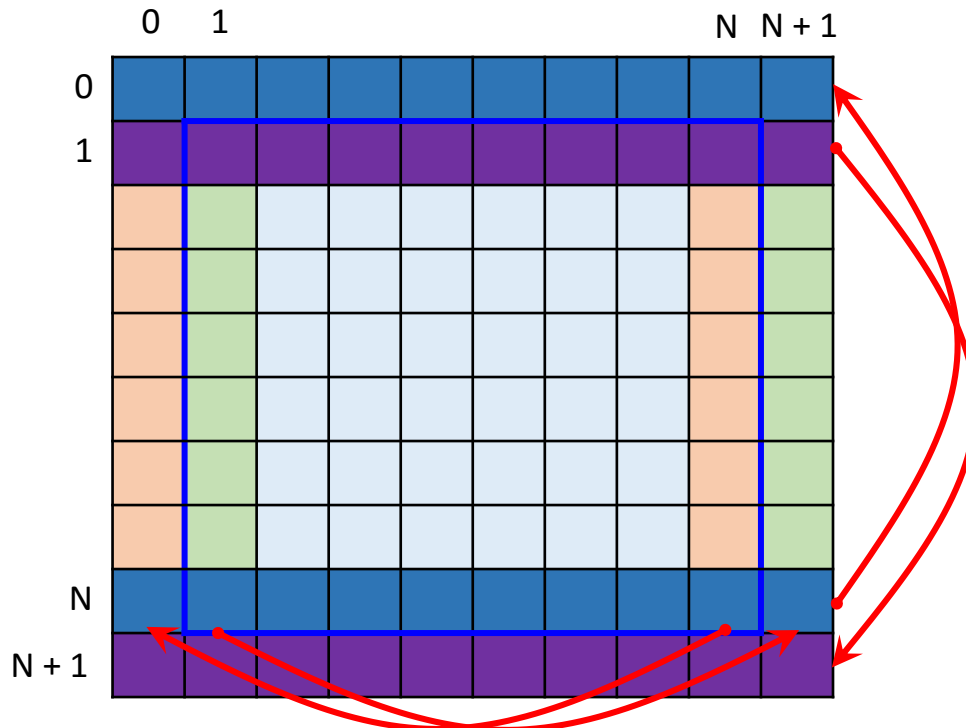
# Conway's Game of Life

- **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



# 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 теньевые 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)

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

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

```
typedef uint8_t cell_t;
```

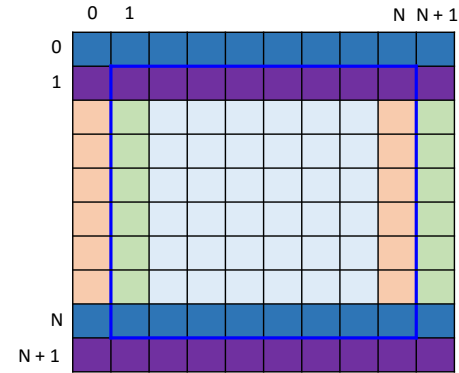
```
int main(int argc, char* argv[])  
{
```

```
    // Grid with periodic boundary conditions (ghost cells)
```

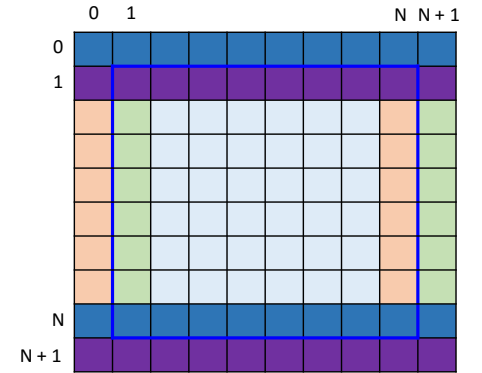
```
    size_t ncells = (N + 2) * (N + 2);  
    size_t size = sizeof(cell_t) * ncells;  
    cell_t *grid = malloc(size);  
    cell_t *newgrid = malloc(size);
```

```
    // Initial population
```

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



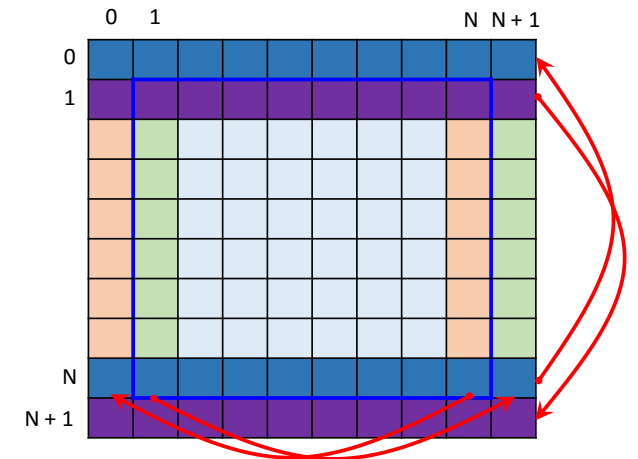
`grid[N + 2][N + 2]`



`newgrid[N + 2][N + 2]`

## Последовательная реализация (продолжение)

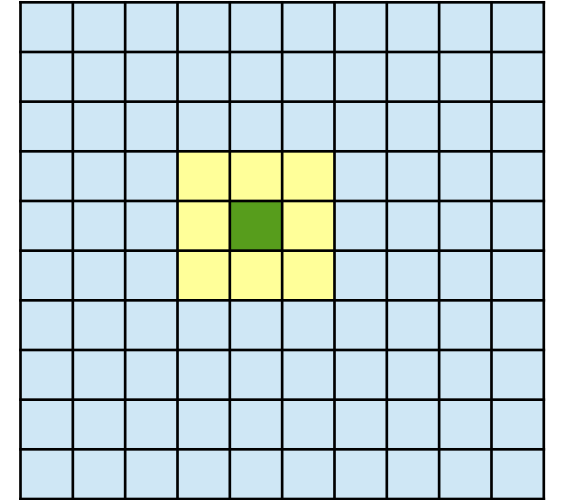
```
double t = wtime();
int iter;
for (iter = 0; iter < ITERS_MAX; iter++) {
    // Copy ghost columns
    for (int i = 1; i <= N; i++) {
        grid[IND(i, 0)] = grid[IND(i, N)];    // left ghost column
        grid[IND(i, N + 1)] = grid[IND(i, 1)]; // right ghost column
    }
    // Copy ghost rows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

```
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)

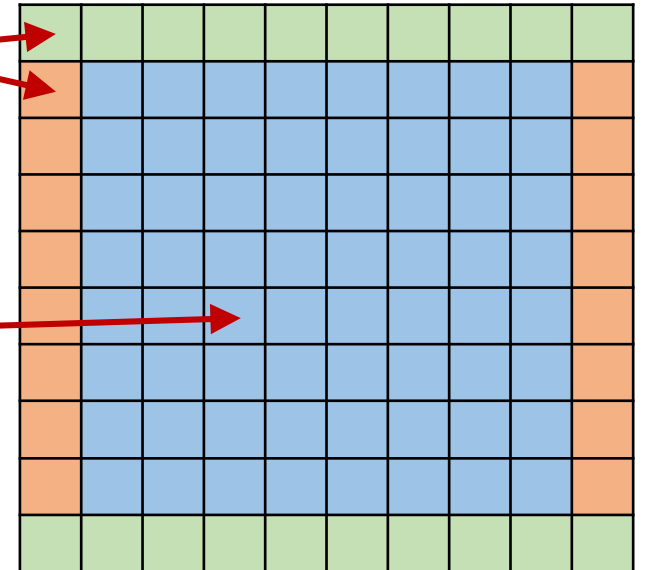
```
cell_t *d_grid, *d_newgrid;  
double tmem = -wtime();  
cudaMalloc((void **)&d_grid, size);  
cudaMalloc((void **)&d_newgrid, size);  
cudaMemcpy(d_grid, grid, size, cudaMemcpyHostToDevice);  
tmem += wtime();
```

```
// 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);

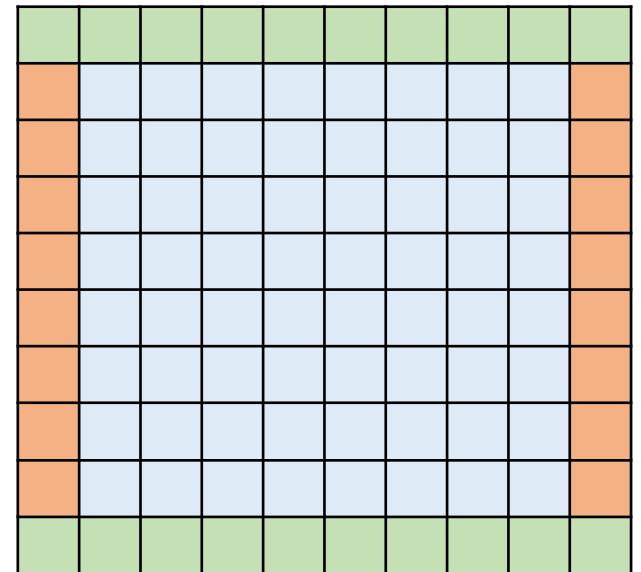
    // Swap grids
    cell_t *p = d_grid; d_grid = d_newgrid; d_newgrid = p;
}
cudaDeviceSynchronize();
t = wtime() - t;

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)] +  
                     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 (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

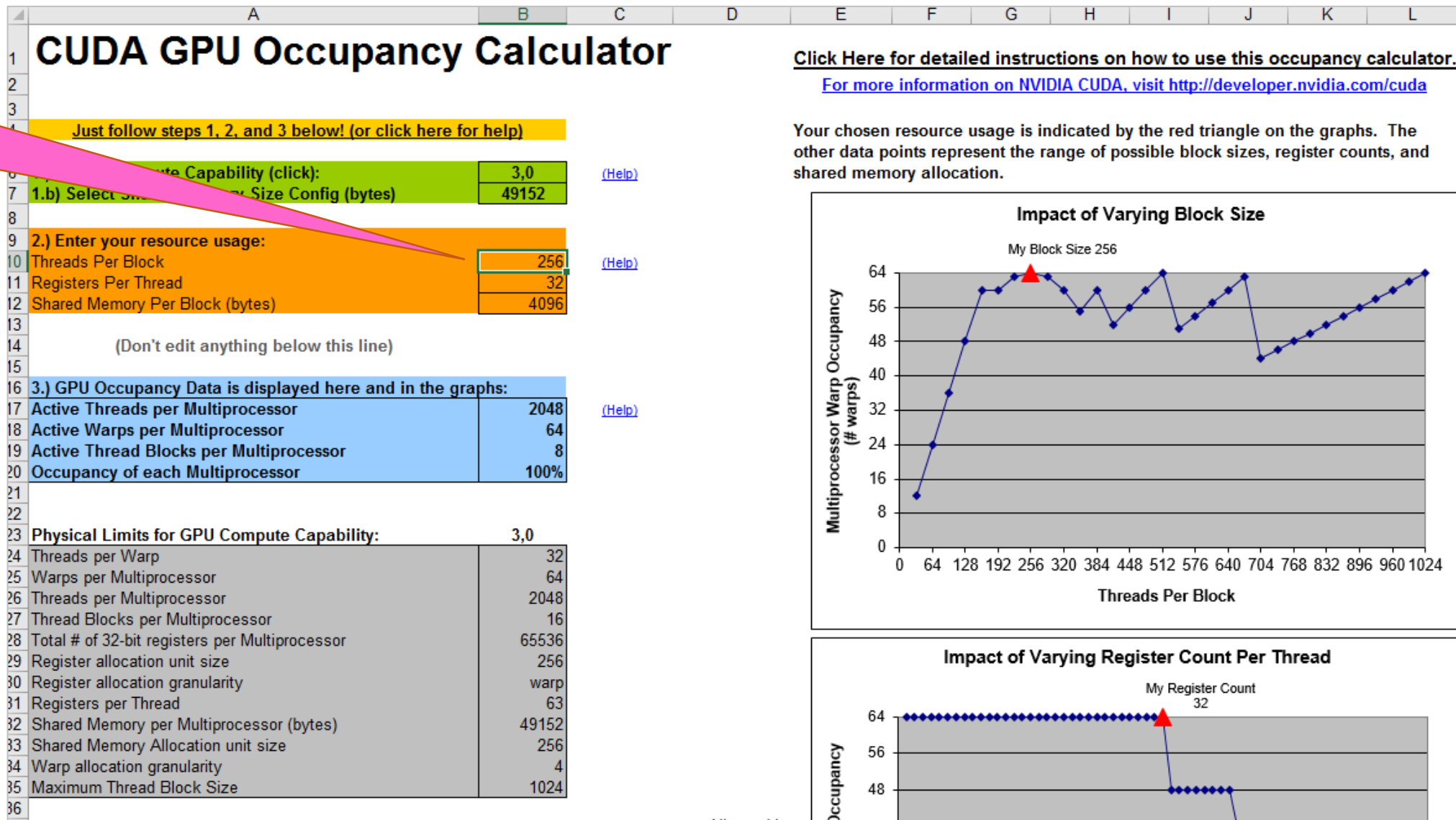
Total time (sec.): 0.453355

Speedup 5.0

# CUDA GPU Occupancy Calculator

[http://developer.download.nvidia.com/compute/cuda/CUDA\\_Occupancy\\_calculator.xls](http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls)

В программе  
блоки  
1d: 1x16  
2d: 16x16



## 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 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);  
    // ...  
}
```

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**