Computational Science on Many-Core Architectures Exercise 5

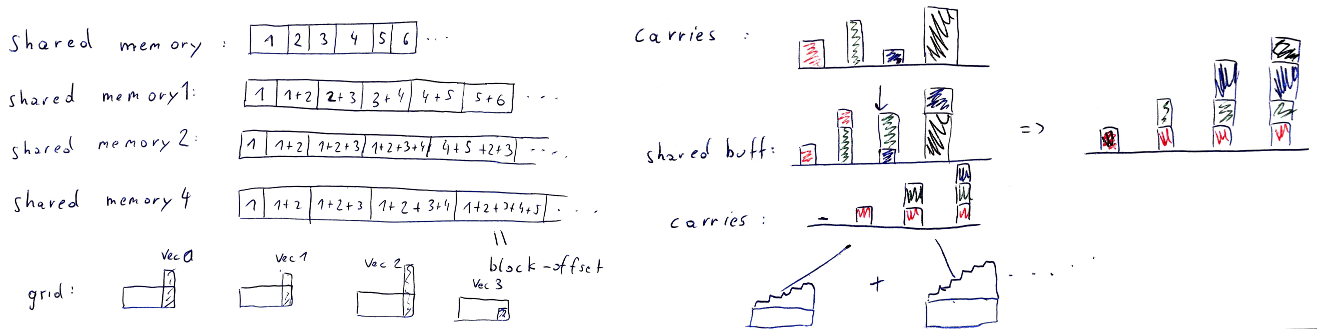
Example 1 Inclusive and Exclusive Scan (4 Points)

a)

```
10 __global__ void scan_kernel_1(double const *X,
11                               double *Y,
12                               int N,
13                               double *carries)
14 {
15   __shared__ double shared_buffer[256];
16   double my_value;
17
18   unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
19   unsigned int block_start = work_per_thread * blockDim.x *  blockIdx.x;
20   unsigned int block_stop  = work_per_thread * blockDim.x * (blockIdx.x + 1);
21   unsigned int block_offset = 0;
22
23   // run scan on each section
24   for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
25   {
26     // load data:
27     my_value = (i < N) ? X[i] : 0;
28
29     // inclusive scan in shared buffer:
30     for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
31     {
32       __syncthreads();
33       shared_buffer[threadIdx.x] = my_value;
34       __syncthreads();
35       if (threadIdx.x >= stride)
36         my_value += shared_buffer[threadIdx.x - stride];
37     }
38     __syncthreads();
39     shared_buffer[threadIdx.x] = my_value;
40     __syncthreads();
41
42     // exclusive scan requires us to write a zero value at the beginning of each block
43     my_value = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
44
45     // write to output array
46     if (i < N)
47       Y[i] = block_offset + my_value;
48
49     block_offset += shared_buffer[blockDim.x-1];
50   }
51
52   // write carry:
53   if (threadIdx.x == 0)
54     carries[blockIdx.x] = block_offset;
55
```

For the first kernel "kernel_1" and simulate it with 4 blocks and 6 threads per block.
Begin with line 24: the for loop iterates over the values X which belong to the block.
At the end of the for loop → write the temporary result of the scan into a vector Y and the offset stored in block_offset. At the end of the kernel every block contains it scanned value and the vector for the next step.

```
58 // exclusive-scan of carries
59 __global__ void scan_kernel_2(double *carries)
60 {
61   __shared__ double shared_buffer[256];
62
63   // load data:
64   double my_carry = carries[threadIdx.x];
65
66   // exclusive scan in shared buffer:
67
68   for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
69   {
70     __syncthreads();
71     shared_buffer[threadIdx.x] = my_carry;
72     __syncthreads();
73     if (threadIdx.x >= stride)
74       my_carry += shared_buffer[threadIdx.x - stride];
75   }
76   __syncthreads();
77   shared_buffer[threadIdx.x] = my_carry;
78   __syncthreads();
79
80   // write to output array
81   carries[threadIdx.x] = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
82 }
83
84 __global__ void scan_kernel_3(double *Y, int N,
85                               double const *carries)
86 {
87   unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
88   unsigned int block_start = work_per_thread * blockDim.x *  blockIdx.x;
89   unsigned int block_stop  = work_per_thread * blockDim.x * (blockIdx.x + 1);
90
91   __shared__ double shared_offset;
92
93   if (threadIdx.x == 0)
94     shared_offset = carries[blockIdx.x];
95
96   __syncthreads();
97
98   // add offset to each element in the block:
99   for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
100    if (i < N)
101      Y[i] += shared_offset;
102 }
```



b)

Listing 1: kernel for inclusive_scan)

```
__global__ void Inclusive(double *Y, int N, const double *X)
{
    for (int i = blockDim.x * blockIdx.x + threadIdx.x; i < N-1;
    i += gridDim.x * blockDim.x)
    {
        Y[i] = Y[i+1];
    }
    if (blockDim.x * blockIdx.x + threadIdx.x == 0)
        // First step: Scan within each thread group and write carries
    scan_kernel_1<<<num_blocks, threads_per_block>>>(input, output, N, carries);

    // Second step: Compute offset for each thread group
    (exclusive scan for each thread group)
    scan_kernel_2<<<1, num_blocks>>>(carries);

    // Third step: Offset each thread group accordingly
    scan_kernel_3<<<num_blocks, threads_per_block>>>(output, N, carries);

    // Make inclusive
    makeInclusive<<<num_blocks, threads_per_block>>>(output, N, input);

    cudaFree(carries);
    {
        Y[N-1] += X[N-1];
    }
}

void exclusive_scan(double const * input, double* output, int N)
{
    int num_blocks = 512;
    int threads_per_block = 512;

    double *carries;
    cudaMalloc(&carries, sizeof(double) * num_blocks);

    // First step: Scan within each thread group and write carries
```

```
    scan_kernel_1<<<num_blocks, threads_per_block>>>(input, output, N, carries);

    // Second step: Compute offset for each thread group
    (exclusive scan for each thread group)
    scan_kernel_2<<<1, num_blocks>>>(carries);

    // Third step: Offset each thread group accordingly
    scan_kernel_3<<<num_blocks, threads_per_block>>>(output, N, carries);

    // Make inclusive
    makeInclusive<<<num_blocks, threads_per_block>>>(output, N, input);

    cudaFree(carries);
}
```

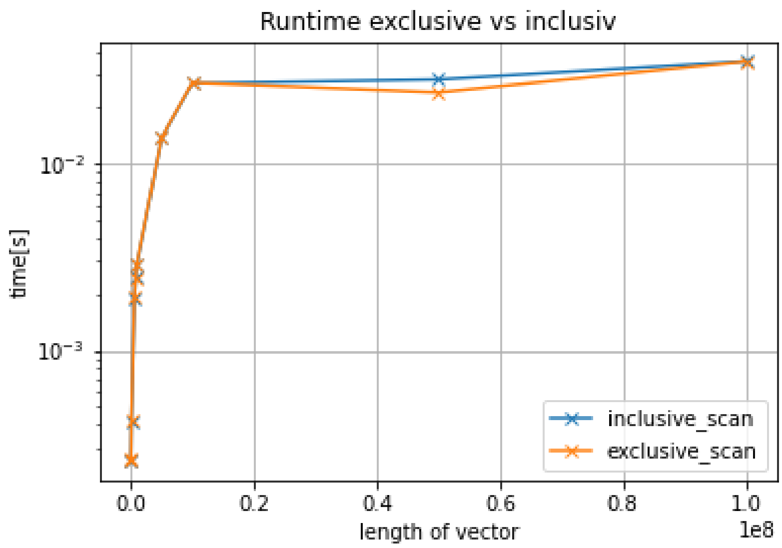
c)

Only nessesary to remove the folloing code snipped:

Listing 2: kernel for inclusive_scan)

```
// exclusive scan requires us to write a zero value
at the beginning of each block
my_value = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
```

d)



There is bassicaly no differences.

2 Poisson equation (5 Points)

First I have to change all row_offsets, indices to an int datatype because there was an error with the matching datatype for the residual_norm function in "poisson2D.hpp".

a)

Listing 3: kernel for serching for zeros)

```
__global__ void count_nonzero_entries(int* row_offsets, int N, int M) {
    for(int row = blockDim.x * blockIdx.x + threadIdx.x; row < N * M;
     row += gridDim.x * blockDim.x) {
        int nnz_for_this_node = 1;
        int i = row / N;
        int j = row % N;

        if(i > 0) nnz_for_this_node += 1;
        if(j > 0) nnz_for_this_node += 1;
        if(i < N-1) nnz_for_this_node += 1;
        if(j < M-1) nnz_for_this_node += 1;

        row_offsets[row] = nnz_for_this_node;
    }
}
```

b)

Again changed the double const *input to a int const * input.

Listing 4: given exclusive scan)

```
void exclusive_scan(int const * input,
int         * output, int N)
{
    int num_blocks = 512;
    int threads_per_block = 512;

    int *carries;
    cudaMalloc(&carries, sizeof(int) * num_blocks);

    // First step: Scan within each thread group and write carries
    scan_kernel_1<<<num_blocks, threads_per_block>>>(input, output, N, carries);

    // Second step: Compute offset for each thread group (exclusive scan for eac
    scan_kernel_2<<<1, num_blocks>>>(carries);

    // Third step: Offset each thread group accordingly
    scan_kernel_3<<<num_blocks, threads_per_block>>>(output, N, carries);

    cudaFree(carries);
}
```

c)

Listing 5: assemble the matrix A)

```
__global__ void assemble_A_GPU(double* values, int* columns, int* row_offsets, i
{
    for(int row = blockDim.x * blockIdx.x + threadIdx.x;
    row < N*M; row += gridDim.x * blockDim.x)
    {
        int i = row / N;
        int j = row % N;
        int counter = 0;

        if ( i > 0)
        {
            values[(int)row_offsets[row] + counter] = -1;
            columns[(int)row_offsets[row] + counter] = (i-1)*N+j;
            counter++;
        }

        if ( j > 0)
        {
            values[(int)row_offsets[row] + counter] = -1;
            columns[(int)row_offsets[row] + counter] = i*N+(j-1);
            counter++;
        }

        values[(int)row_offsets[row] + counter] = 4;
        columns[(int)row_offsets[row] + counter] = i*N+j;

        counter++;

        if ( j < M-1)
        {
            values[(int)row_offsets[row] + counter] = -1;
            columns[(int)row_offsets[row] + counter] = i*N+(j+1);
            counter++;
        }
        if ( i < N-1)
        {
            values[(int)row_offsets[row] + counter] = -1;
            columns[(int)row_offsets[row] + counter] = (i+1)*N+j;
            counter++;
        }
    }
}
```

d)

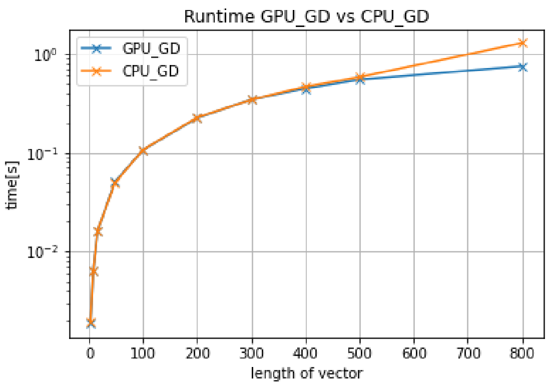
Just adapt the GD program that you gave us and changed the names from the row_offsets.
Code is in the appendix in the mail.

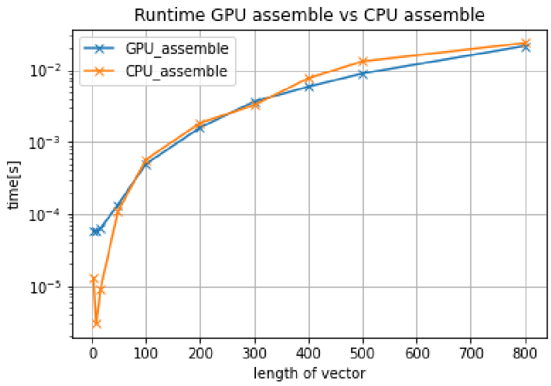
e)

Simply replace the conditions for the loop in the assembling kernel with this.
Code is in the appendix in the mail.

Listing 6: kernel for serching for zeros)

```
for(int row = 0; row < N*M; row++)
```





In total the differences are not that much. Maybe for higher dimensions of the matrix A but I got an SEGMENTATION FAULT at a certain point of the benchmark $N > 800$. I tried to change int into a long int but then I got also some errors. I think the difference should be much higher for bigger dimensions of A.

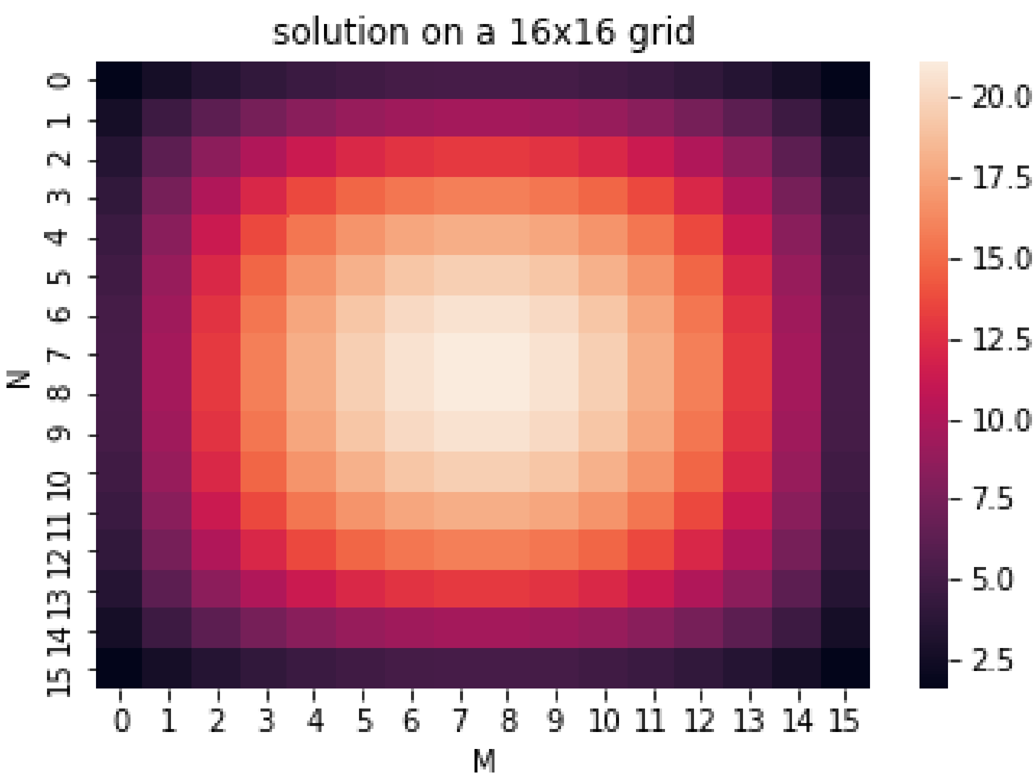
f) Bonus point

Very ineffective way to get the results for the plotting. The two for loops outputs the first line of the solution matrix.

Listing 7: kernel for serching for zeros)

```
int ck = 0;
for (int i = 0; i < N/max; i++)
{
    for (int j = 0; j < N/max; j++)
    {
        std::cout << solution[ck] << "," << std::endl;
        ck = ck + 1;
    }
    std::cout << " " << std::endl;
    std::cout << " " << std::endl;
    std::cout << " " << std::endl;
    std::cout << " " << std::endl;
    std::cout << " " << std::endl;
}
```

I copied the output by hand in python and plot it with the function "heat_map". The dimension of the solution is 16×16 so in total 256 unknowns. The result looks nice. It is obviously the solution to the poisson equation.



The color bar on the side represents the amplitude of the solution. Where N and M are the cordinates of the solution matrix. In our case is the unite square given in the matrix assemble function.