Simon Hinterseer e09925802

Exercise 06

1 Inclusive and Exclusive Scan

The code for this part can be found in the attached file **ex06_1.cpp**.

1.1 Exclusive Scan Implementation

Overview

An input vector and allocated space for an output vector of size N are given. The output vector will end up holding the exclusive prefix sum of the input vector.

For that a function on the host will consecutively call three kernels:

The first kernel `scan_kernel_1()` will divide the input vector in as many sections as there are thread blocks. It will then compute the exclusive prefix sum for each section and store it in the corresponding section of the output vector. It will also store the sum over all but the last section in a separate vector `carries`. This is illustrated in Figure 1.

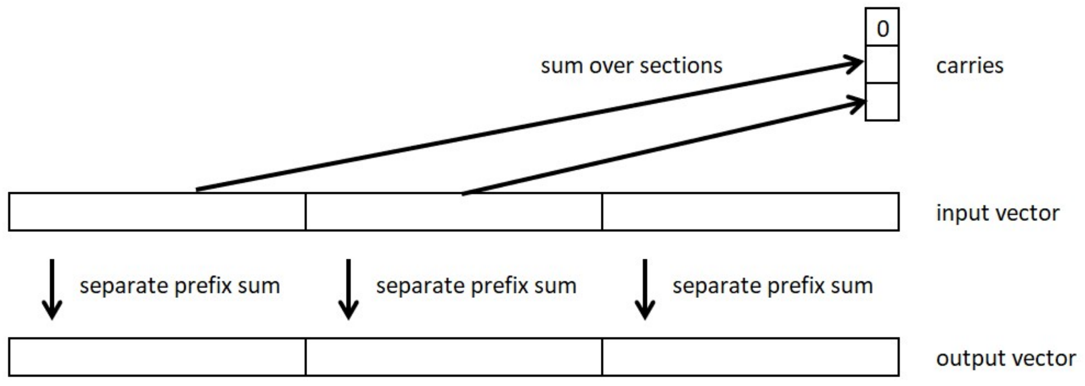

Figure 1 – kernel 1

The second kernel `scan_kernel_2()` will perform an exclusive scan over the `carries` vector. This is done inplace, therefore the `carries` vector afterwards holds for each section of the input vector the sum of all elements of all previous sections.

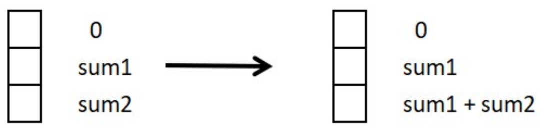

Figure 2 – kernel 2

The third and final kernel `scan_kernel_3()` will for each section of the output vector add the corresponding value, found in the `carries` vector, to all entries in the section. This will finalize the prefix sum operation.

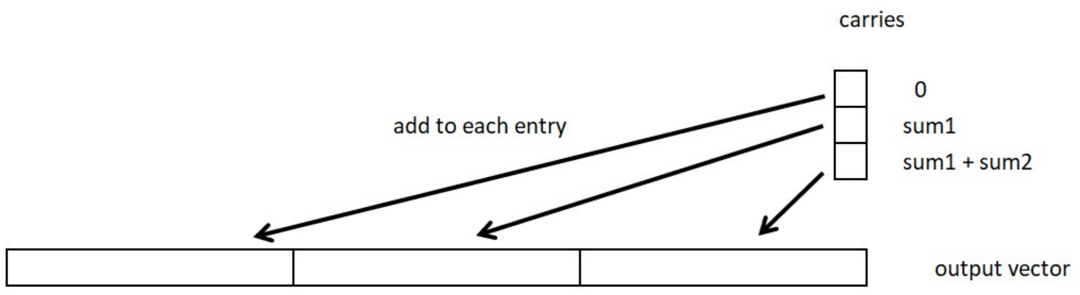

Figure 3 – kernel 3

First Kernel

The first kernel will divide the input vector in as many subsections as there are blocks. Each block will then only operate on its own section. The section will furthermore be divided in subsections of a size, equal to the block size. This is illustrated in Figure 4.

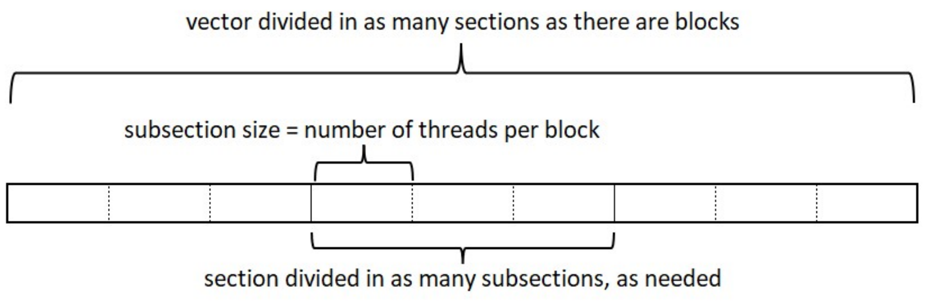
vector divided in as many sections as there are blocks
subsection size = number of threads per block
section divided in as many subsections, as needed

Figure 4 – subdivision of the input vector

For each subsection, the kernel will then compute the prefix sum and store it in the corresponding section of the output vector.

The prefix sum is performed through iterative strided addition operations. Figure 5 shows an example how one entry ends up holding the sum of all previous entries including its own value.

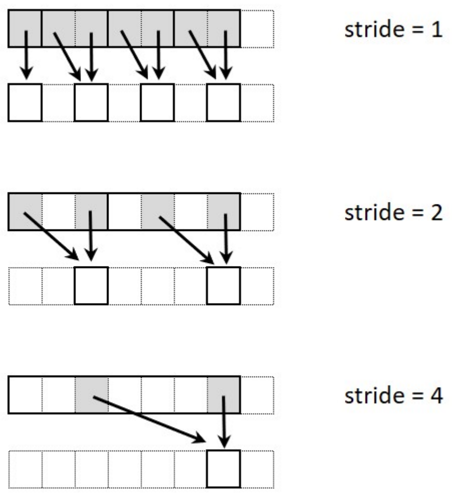

Figure 5 – inclusive prefix sum algorithm for entry 7 in a vector of size 8

Kernel 1 will do, what is illustrated in Figure 5 consecutively on each subdivision (remember subdivisions from Figure 4). It will also take into account, the sum of all previously processed subdivisions, by accumulating a sum in the thread local variable `block_offset`. Adding this to each value results in an inclusive prefix sum, that is stored in the block-level shared array `shared_buffer`. In order to end up with an exclusive sum rather than an inclusive one, `shared_buffer` is padded with a single zero value in the beginning and written to the output vector. Also the final value of `block_offset`, that now holds the sum over the whole section, is written to the corresponding position of `carries`.

Kernel 2 and Kernel 3

Kernel 2 and kernel 3 are quite straight forward:

- Kernel 2 performs an inclusive prefix sum on the `carries` array. This is done by a single block that has as many threads as the original grid had blocks. This means, that there is one thread for each entry of `carries`, which simplifies the task.
- Kernel 3 adds entries of `carries` to each entry in the corresponding section of the output vector.

1.2 Inclusive Scan (Reusing Exclusive Scan)

This is done by simply adding the input vector to the output vector after performing an exclusive scan.

```
void inclusive_scan1(const double *device_input, double *device_output, int N)
{
  // get exclusive scan
  exclusive_scan(device_input, device_output, N);
  // add input to output
  device_excl_to_incl<<<GRID_SIZE, BLOCK_SIZE>>>(device_input, device_output, N);
}
```

1.3 Inclusive Scan (Modifying Exclusive Scan)

As described in the corresponding section, kernel 1 performs a shift in the block-level shared array shared_buffer in order to make the prefix sum exclusive. By omitting this shift, the kernel computes an inclusive prefix sum.

1.4 Performance Comparison

All three versions of computing the prefix sums performed very similarly. It can be seen, however, that the first implementation of inclusive scan, that reused the exclusive scan, was a little bit slower, than the second one. This is not surprising, since it calls another kernel, that reads $2N$ entries, performs N operations and writes N entries. It might have been faster to shift the entries of the output vector and only calculate the last entry. This way, it would have been sufficient to only read $N + 1$ entries. Figure 6 shows, however, that there is not much potential for improvement.

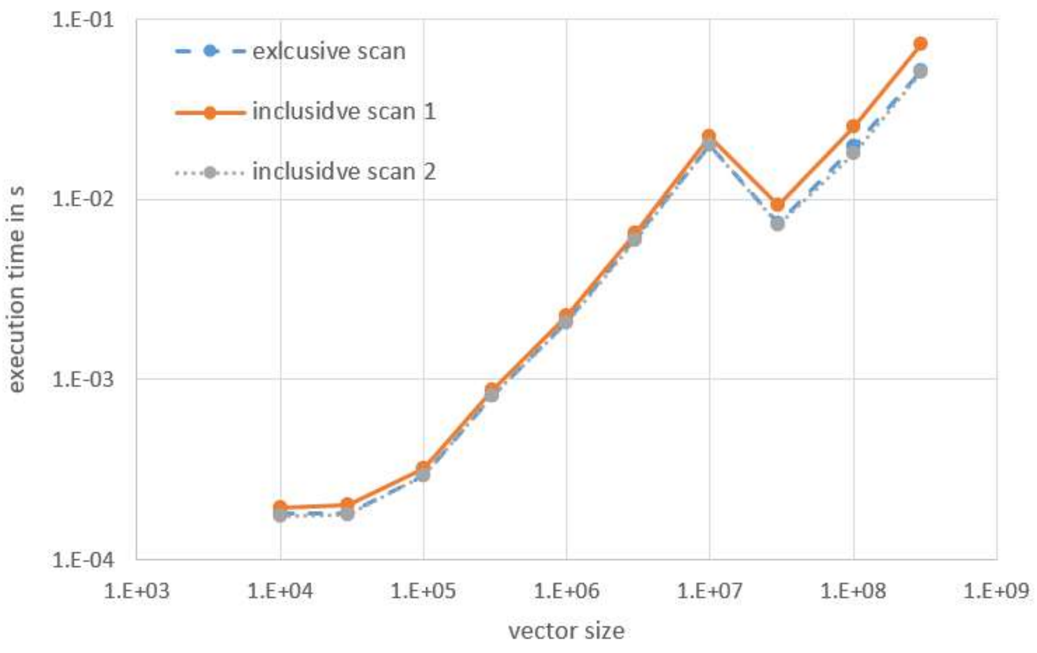

Figure 6 – performance comparison for the prefix sum implementations

2 Finite Differences on the GPU

The code for this part can be found in the attached file **ex06_2.cpp**.

The Poisson equation with homogenous boundary conditions is to be solved using the Finite Differences method. For this, the domain is discretized in an $N \times M$ grid. This leads to a linear system of equations $Ax = b$ with a symmetric positive definite matrix A. This will be solved using the Conjugate Gradients method.

The matrix A is of dimensions $N \cdot M \times N \cdot M$. The row with index $(i \cdot M + j)$ represents the equation for the element (i, j) in the discretized domain. The matrix will be assembled in CSR format.

2.1 System Assembly

This is done in the host function `my_generate_system()`. It will take the dimensions N, and M as well as the three pointers to pre allocated arrays on the device for storing the CSR matrix. Below the steps for assembling the three arrays are listed. These steps are then described in the sections below.

- `device_csr_rowoffsets`: count nonzero entries, exclusive prefix sum, add final entry.
- `device_csr_colindices`: see kernel `device_assembleA()`.
- `device_csr_values`: see kernel `device_assembleA()`

Counting Nonzero Entries

This is done in the kernel void `device_num_nonzero_entries()`.

Parallelization: The rows of the matrix will be processed in parallel.

In order to determine the number of nonzero elements of the row with index $(i \cdot M + j)$ the positions of grid point (i, j) is analysed:

- if point (i, j) is in a corner, the row has 3 nonzero entries.
- if point (i, j) is at an edge (but not in a corner), the row has 4 nonzero entries.
- if point (i, j) is an inner point, the row has 5 nonzero entries.

Exclusive Prefix Sum

The array holding the number of nonzero entries for each row is processed by the exclusive scan program from Part 1.

Add final entry

In the beginning of the kernel `device_assembleA()`, the thread with index zero will write the last entry of `device_csr_rowoffset`.

Assemble Matrix Kernel

Parallelization: The rows of the matrix will be processed in parallel.

After finalizing `device_csr_rowoffsets`, the kernel `device_assembleA()` will go through the $N \cdot M$ rows of the matrix. For the row with index $row = (i \cdot M + j)$, it will add entries to the arrays `device_csr_colindices` and `device_csr_values` in accordance to the position of the point (i, j) in the discretized domain. This is done by performing the following steps in the order, that is noted here:

- if point (i, j) is not on the southern border, account for a southern neighbour:
  add $row - M$ a to `device_csr_colindices` and -1 `to device_csr_values`.
- if point (i, j) is not on the western border, account for a western neighbour:
  add $row - 1$ a to `device_csr_colindices` and -1 `to device_csr_values`.
- all points need to take themselves into account:
  add row a to `device_csr_colindices` and 4 `to device_csr_values`.
- if point (i, j) is not on the eastern border, account for an eastern neighbour:
  add $row + 1$ a to `device_csr_colindices` and -1 `to device_csr_values`.
- if point (i, j) is not on the northern border, account for a northern neighbour:
  add $row + M$ a to `device_csr_colindices` and -1 `to device_csr_values`.

2.2 Conjugate Gradients and Performance Analysis

The function `solve_system()` from the previous exercise is adapted so that it is able to both benchmark the two ways of assembling the system matrix as well as running the CG solver. This is controlled via `bool bm_sa` that is passed from `main()` to `solve_system()`.

The time measurement for the system assembling benchmark is done by using the `execution_wrapper()` function to get the median time of 7 repetitions.

The CG solver has its own time measurement, which is used to compare the execution time for system assembly with the time for computation of a solution. This kind of time measurement does not rely on repeated execution and might be less accurate. It still serves the purpose, which is to assess, whether performing the system assembly on the GPU results in a noteworthy reduction in overall computation time.

For this benchmark, square shaped domains with side lengths of 30, 60, 120, 240, 480 and 960 have been used. The result is shown in Figure 7.

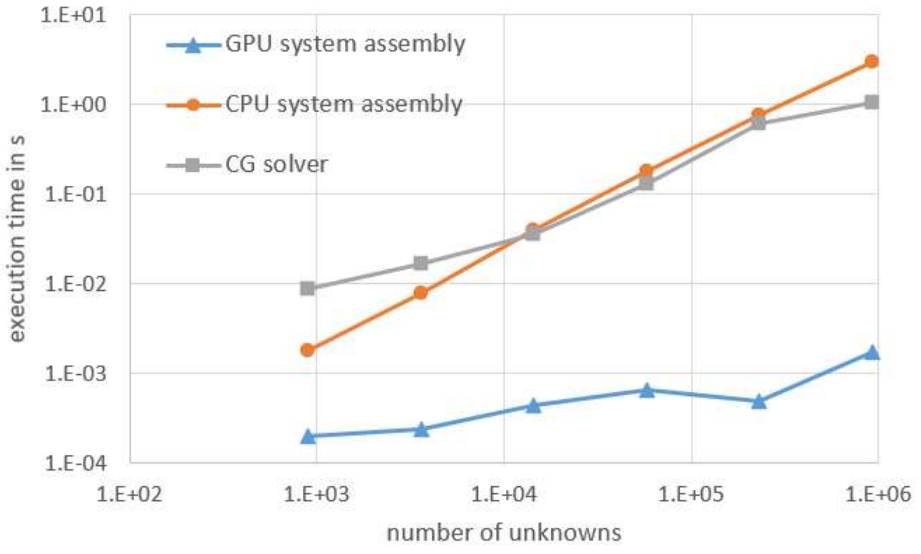

Figure 7 – performance analysis for system assembly and CG solver

It can be seen, that the system assembly on the GPU is much faster than on the CPU. The speedup ranges from 20-fold for smaller systems to over 200-fold for larger systems. It can also be seen, that the assembly of the system can take a significant portion of the overall computation time. For larger systems, the system assembly on the CPU takes longer than the CG solution. Therefore, it can be concluded, that unless the same CSR matrix is used for many runs of the CG solver, the performance improvement gained by assembling the matrix on the GPU makes a very significant difference.

3 Bonus – Visualization

The additional `bool bonus` was added to `main()` and passed to `solve_system()`. If set to true, the solution vector will be printed to the console.

This will be processed by the python script **ex06_bonus.py** which is attached. It will recreate a two-dimensional field out of the solution vector and use `axes.contourf()` from the Matplotlib library to draw a heatmap of the values.

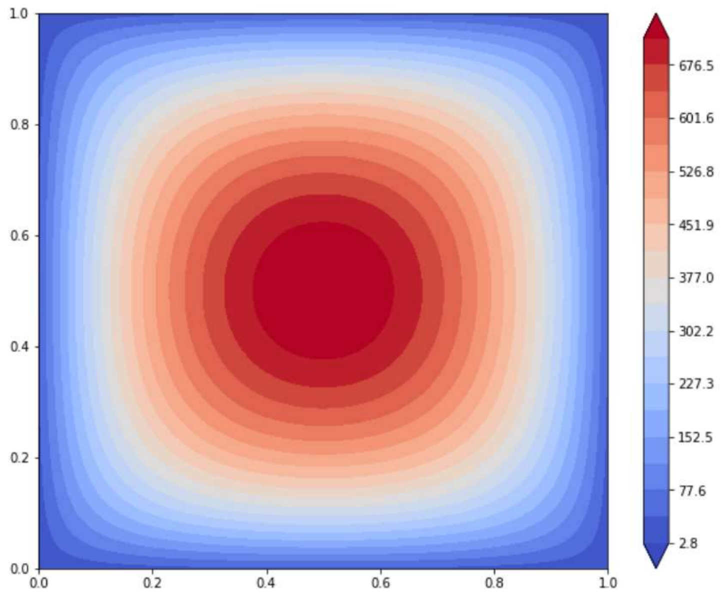

Figure 8 – Visualisation of the solution for a 100x100 grid

This is what is to be expected from the homogenous Poisson equation, with a constant heating term (right hand side is constant 1).