Exercise 06

# Inclusive and Exclusive Scan

## Exclusive Scan Implementation

Overview

An input vector and allocated space for an output vector of size 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 sections in a separate vector carries. This is illustrated in Figure 1.

A picture containing diagram

Description automatically generated

Figure – 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.

A picture containing text, clock

Description automatically generated

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.

A picture containing line chart

Description automatically generated

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.

Timeline

Description automatically generated with medium confidence

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.

Diagram

Description automatically generated

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, this 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 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 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.

## 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);

}

## 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.

## 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 entries, performs operations and writes 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 entries. Figure 6 shows, however, that there is not much potential for improvement.

Chart, line chart

Description automatically generated

Figure 6 – performance comparison for the prefix sum implementations