Exercise 07

# Atomic Add, Warp Shuffles and Thread Configuration

The code for this part can be found in **ex07\_1.cpp**.

The task is to calculate dot products in four ways

* No use of atomicAdd(). Copy partial result from each work group to the CPU, sum there.
* No use of atomicAdd(), sum partial result from each work group on the GPU in a second kernel.
* Using atomicAdd() once for each work group.
* Using atomicAdd() once for each warp

## Implementation

The following kernels were made:

* gpu\_dotp\_shared() … this computes the dot product and makes a reduction of the final sum using a block-level shared array. This will end up with an array holding a partial result, that needs a final summation step.
* gpu\_final\_add() … this expects to be called in a configuration with a single block. It will perform the final summation step of the above kernel. For this, a block-level shared array is used.
  + If the above kernel was called with this configuration <<<GRID\_SIZE, BLOCK\_SIZE>>>
  + then this kernel should be called with this configuration <<<1, GRID\_SIZE>>>

This works well as long as GRID\_SIZE stays low enough. For the configuration (N+255)/256 blocks with 256 threads each this led to problems once the number of blocks exceeded 10000. In order to allow benchmarking, the size of the shared array was limited. This would produce false results for the dot product but should not have noticeable consequences for the execution time.

* gpu\_dotp\_atomic\_warp() … This computes the dot product and makes a reduction of the final sum using warp shuffles. One thread per warp will add the partial result to the final memory location using atomicAdd().
* gpu\_dotp\_atomic\_shared() … This is a lot like the above function, but uses a block-level shared array for the reduction rather than warp shuffles.

The kernels are called by four subprograms:

* final\_sum\_on\_cpu() … this calls gpu\_dotp\_shared(), moves the partial result to the host and computes the final sum there.
* final\_sum\_on\_gpu() … this calls gpu\_dotp\_shared() and gpu\_final\_add() and finally moves the final result to the host.
* atomic\_add\_per\_workgroup() … this calls gpu\_dotp\_atomic\_shared() and moves the final result to the host.
* atomic\_add\_per\_warp() … this calls gpu\_dotp\_atomic\_warp() and moves the final result to the host.

These subprograms are then timed by the execution\_wrapper() function, which will execute each program 11 times and return the median execution time.

## Performance Comparisons

The subprograms were executed for vector sizes ranging between and for the thread configurations 128x128, 256x256, 512x512, (N+255)/256 x 256

Chart, line chart

Description automatically generated

Figure 1 – Performance comparison with configuration 128x128

Chart, line chart

Description automatically generated

Figure 2 – Performance comparison with configuration 256x256

Chart, line chart

Description automatically generated

Figure 3 – Performance comparison with configuration 512x512

Chart, line chart

Description automatically generated

Figure 4 – Performance comparison with configuration (N+255)/256 x 256

A few things should be noticed when considering these results.

* For large vectors, for all four programs very similar execution times were observed.
* For small vectors, where the overhead of calling a kernel is an issue, the program, that called two kernels was a little slower than the others.
* The fastest program was for small vectors the kernel, that used warp shuffles and one atomicAdd() operation per warp. It has to be noted however, that the time for initialization of the memory location of the final result, was not taken into account here. The fact, that this strategy was faster for smaller vectors but not for larger ones motivates the assumption, that there is some overhead, that this strategy avoids. This might be the time it takes to create the block-level shared array, that is not needed, when warp-shuffles are used.
* When comparing the thread configurations, the performance differences are quite significant. The optimum was at 256x256. This is quite surprising, since in earlier comparisons it was found that the performance becomes optimal once a certain number of threads is exceeded.

# Dot Product with OpenCL

The code for this part can be found in **ex07\_2.ccp**.

## Implementation

Kernel

Based on the provided OpenCL code for vector addition, a program for computing a dot product was implemented.

It uses a block-level shared array to perform a reduction step and returns an array of length equal to the grid size. The final sum over this array is computed on the host using std::accumulate().

One difference to CUDA programming is, that since the kernel is stored as a string literal, it is not possible to work with precompiler statements like #define GRID\_SIZE 256, which would be very convenient in such a situation, because it allows to define grid size and block size in a single spot in the program. Instead, the integer literal *256* shows up a second time (first time is the kernel configuration in the host program) in definition of the kernel string:

\_\_local double shared\_dotp[256];

This inconvenience and potential source of errors could possibly be avoided   
with either string manipulation done by the host program (maybe something like find\_and\_replace(kernelstring,“BLOCK\_SIZE”,”256”)) or the use of constexpr. Neither of these possibilities was tried though.

Apart from that and different naming of instructions, the implementation is exactly the same as it would be in CUDA.

Host Program

The platform is selected using a switching Boolean compute\_on\_gpu, that, if true, overwrites the my\_platform variable with platform\_ids[1], which corresponds to the platform with the GPU.

For different vector sizes the allocation, repeated execution of the kernel (to get median time) and deallocation is done. Results are printed to the console.

One difference to CUDA here is, that no obvious way was found to use a convenient execution wrapper, that returns the median execution time for a passed function. Instead, this loop was explicitly written in the main() function.

Another difference is that kernel arguments must be declared using clSetKernelArg().

Finally, there is a nice touch in comparison to CUDA, that in OpenCL it is explicitly stated, that a kernel is not called, but enqueued in list of tasks for the device. This is more intuitive than the CUDA syntax, that makes it look like the kernel is called like a function, which would suggest, that this call only returns, once the kernel has finished execution. In OpenCL it is more obvious, that another statement is required, that will make the program wait for the device to finish.

## Performance Comparison

For performance comparison the results from section 1 with the 256x256 configuration were used. Since the OpenCL kernel uses a block-level shared array for reduction and computes the final sum on the host. Therefore, the for the comparison the results of final\_sum\_on\_cpu() were used. The results are displayed in Figure 5.

Chart, line chart

Description automatically generated

Figure 5 – Performance comparison CUDA / OpenCL

It is remarkable how close the execution times for large vectors are between CUDA and OpenCL. For smaller vectors, OpenCL shows a somewhat better execution time. The reason for this was not found. OpenCL execution on the platform using the CPU was slower by a factor of 50-150 which is not surprising.