Simon Hinterseer e09925802

Exercise 07

Contents

1 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

1.1 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.

1.2 Performance Comparisons

The subprograms were executed for vector sizes ranging between 10^3 and 10^7 for the thread configurations 128x128, 256x256, 512x512, (N+255)/256 x 256

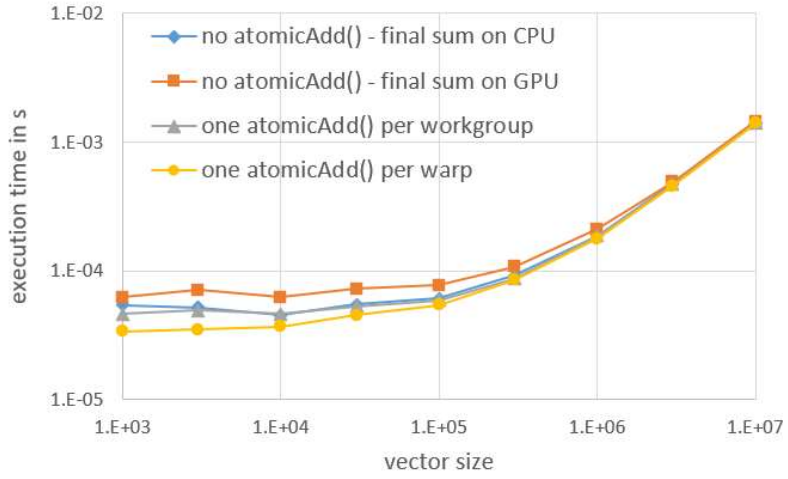

Figure 1 – Performance comparison with configuration 128x128

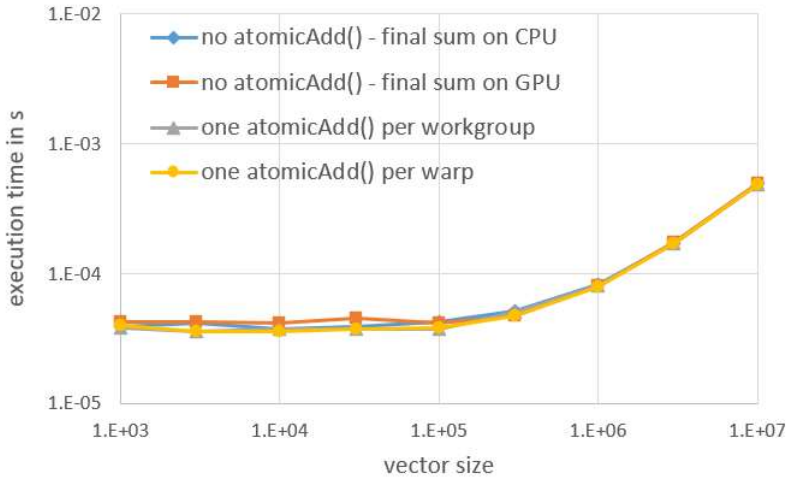

Figure 2 – Performance comparison with configuration 256x256

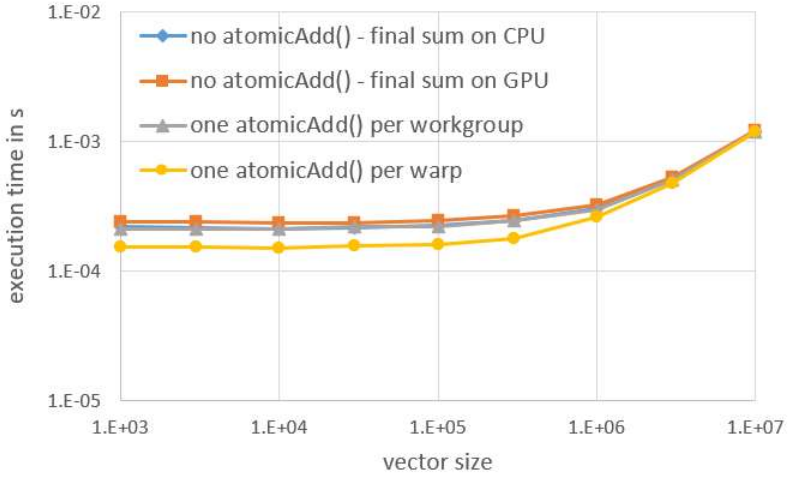

Figure 3 – Performance comparison with configuration 512x512

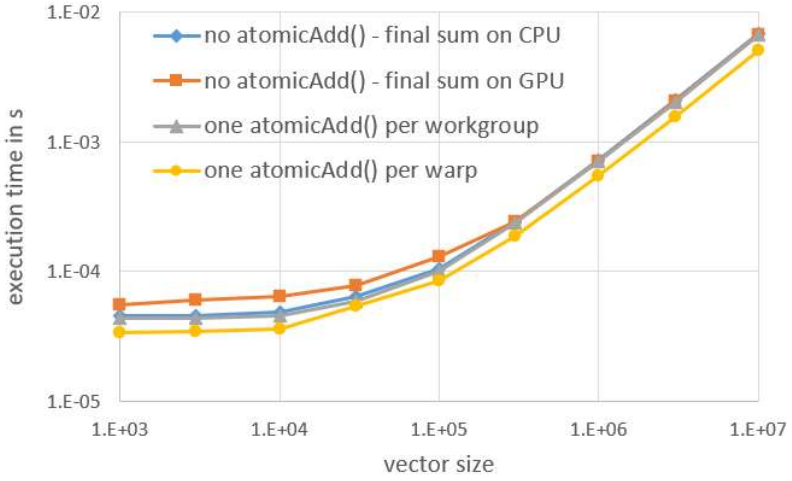

Figure 4 – Performance comparison with configuration (N+255)/256 x 256

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 for small vectors was 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.
- The fact, that the OpenCL version (see section 2) and the 256x256 version of `final_sum_on_cpu()` perform so similarly, suggests the assumption, that something went wrong with the other configurations here. This is peculiar since the only thing, that changed was the precompiler statements `#define GRID_SIZE [number]` and `#define BLOC_SIZE [number]`. Only in the configuration (N+255)/256x256 more changes were made.

2 Dot Product with OpenCL

The code for this part can be found in **ex07_2.ccp**.

2.1 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 number of work groups. 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) inside 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 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: In OpenCL it is explicitly stated, that a kernel is not called, but enqueued in list of tasks for the device. This seems 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.

2.2 Performance Comparison

For performance comparison the results from section 1 with the 256x256 configuration were used. 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.

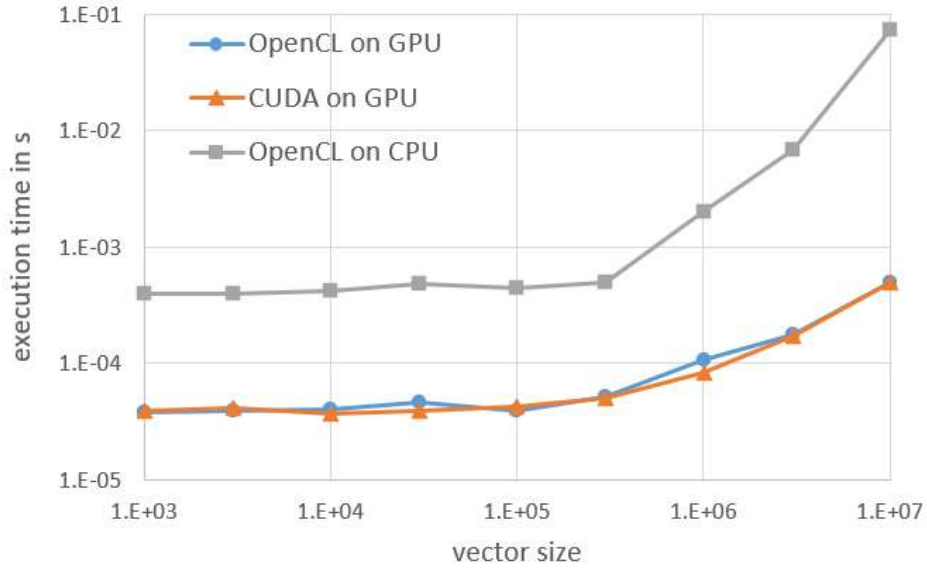

Figure 5 – Performance comparison CUDA / OpenCL

It is remarkable how close the execution times are between CUDA and OpenCL. OpenCL execution on the platform using the CPU was slower by a factor of 10-150. This is expected for large vectors. For small vectors however, a CPU version of the dot product should be able to outperform the GPU versions. It can be observed here, that this is not the case when using OpenCL.

3 OpenCL Sparse Matrix-Vector Product Kernels

The code for this part can be found in **ex07_3.cpp**.

The task is to implement two OpenCL kernels for sparse matrix vector multiplication:

- one kernel, that uses one thread per row
- one kernel, that uses a whole work group per row

These kernels will then be compared in terms of execution times when it comes to two kinds of matrices:

- few nonzero values: matrices with 3 to 5 nonzero values per row
- many nonzero values: matrices with 200 to 2000 nonzero values per row

3.1 Implementation

Program

- `main()`: here several general parameters are defined:
    - the kernel string (to use one thread or one work group per row)
    - the platform on which to run the kernel (GPU/CPU)
    - matrix generator (few or many nonzero entries)
    - The problem sizes for the benchmark

  Then the `benchmark_matvec()` function is launched once for each of the chosen problem sizes.
- `benchmark_matvec()`: this performs all OpenCL related tasks to allow the execution of the kernel. It will launch the passed kernel several times and print the median execution time to the console.

Kernel 1

The first kernel is referred to by the string `ocl_sparse_matvec1`. It is a very generic parallelisation of the provided sparse matrix vector multiplication code. The only thing, that was changed, was the header of the for loop to make it use several threads:

```
for (size_t i = get_global_id(0); i < N; i += get_global_size(0))
```

Kernel 2

The second kernel is referred to by the string `ocl_sparse_matvec2`. It uses a whole work group per matrix row, by changing the outer for-loop header like this:

```
for (size_t i = get_group_id(0); i < N; i += get_num_groups(0))
```

The inner for loop has been changed, to make use of the threads inside the work group:

```
for (size_t j = csr_rowoffsets[i] + get_local_id(0);
     j < csr_rowoffsets[i+1];
     j += get_local_size(0))
```

Since the thread local register variable `double value` accumulates over the course of the inner for-loop, a block-level shared array is used:

```
__local double shared_value[256];
```

outside the inner for-loop, the array is initialized:

```
shared_value[get_local_id(0)] = 0;
```

inside the inner for-loop, the array is populated with numbers, that need to be summed up:

```
shared_value[get_local_id(0)] += csr_values[j] * vector[csr_colindices[j]]
```

After the nested for-loops, a reduction is performed and the value of the first entry of the array is written to the result vector. This is much like the reduction in section 2.

3.2 Performance Comparison

Since the generation of the matrices takes a large amount of time, only matrices with 900-14000 rows were examined. The configuration was 256 work groups with 256 threads per work group.

Few nonzero entries

When it comes to matrices with few nonzero entries, the kernel, that uses one thread per row, is a lot faster. This is expected, since there are only up to 5 numbers per row, that are to be accounted for. This leaves most threads without work, if a whole work group is used per row.

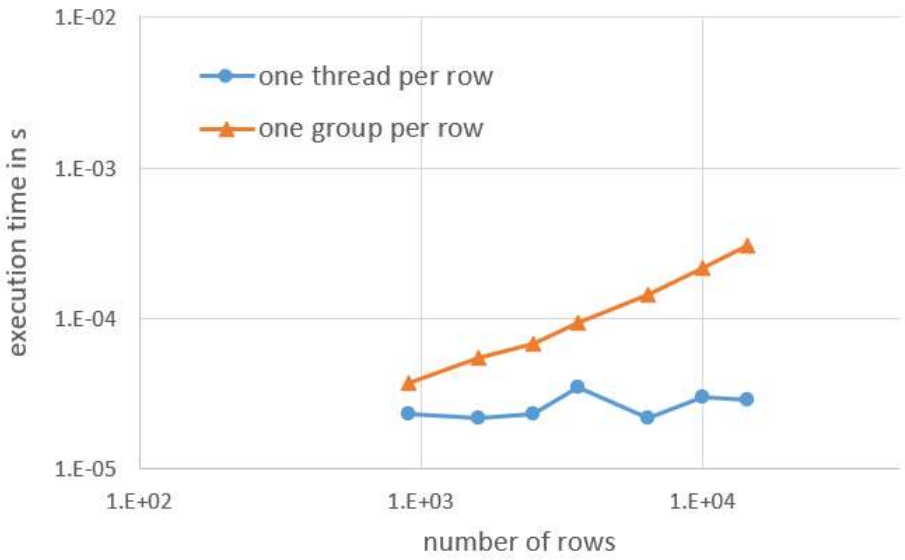

Figure 6 – Performance comparison for matrices with few nonzero entries

It is also noticeable, that the execution time does not depend on the problem size when using one thread per row in this configuration. This is not surprising on the one hand, considering that the number of threads was $256 \cdot 256 = 65{,}536$ while the number of rows was at most $14{,}400$. On the other hand, there should be only about $3{,}600$ cores on the RTX 1080. Therefore, there were enough rows to keep the cores busy, one might think.

Many nonzero entries

When it comes to matrices with 200 to 2000 nonzero entries per row, the kernel, that uses a whole work group per row, performed noticeably better. There does not seem to be as easy of an explanation for this, as there was for the few nonzero entry matrices. It is possible, that memory locality has something to do with it. Threads that work on the same row will access entries of arrays, that are close to each other. This might result in a better cache hit ratio.

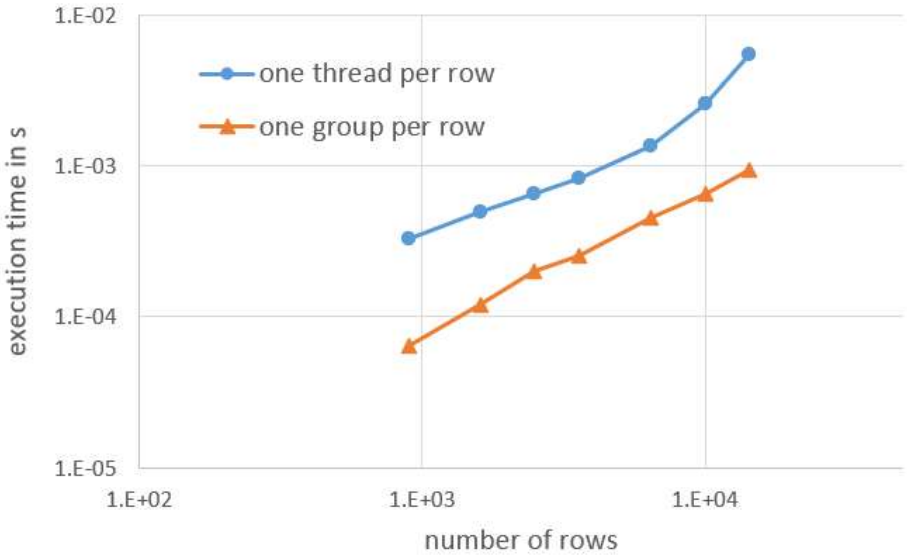

Figure 7 – Performance comparison for matrices with many nonzero entries