Simon Hinterseer e09925802

Exercise 04

1 Dot Product with Warp Shuffles

The code for this part can be found in the attached file **ex04_1_dotproduct.cpp**.

Four numbers are to be calculated w.r.t. a vector:

- the sum of all entries,
- the sum of the absolute value of all entries (1-norm),
- the sum of the square of all entries (squared 2-norm),
- the number of zero entries.

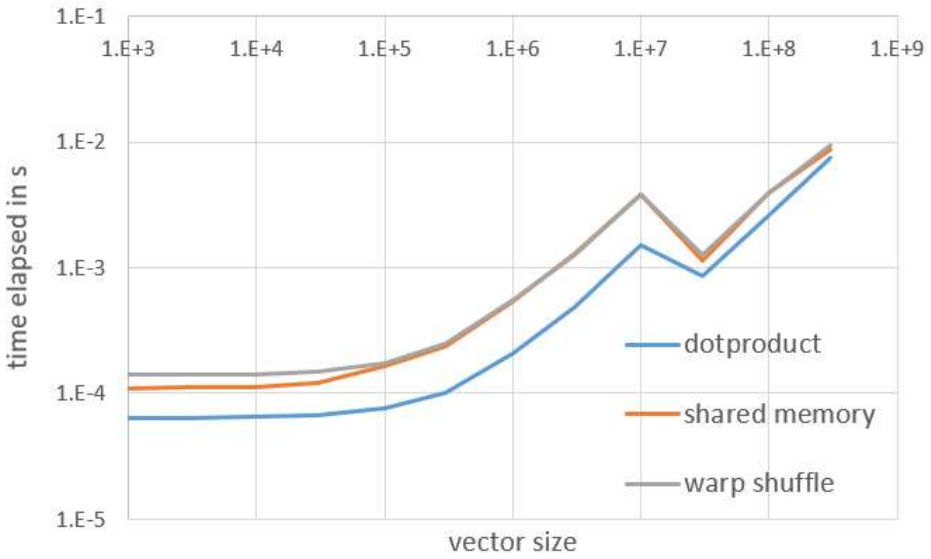
Three kernels are compared in this section

a) one kernel calculates the four numbers using shared memory

b) one kernel calculates the four numbers using warp shuffles

c) one kernel calculates just the dot product using shared memory

Result: Kernels (a) and (b) performed almost equally. Kernel (c) was between 2.2 and 1.3 times quicker. So calculating the four numbers in separate CUBLAS kernels, that have to access the main GPU memory four times, is expected to be slower.



The fact that kernel (b) (warp shuffles) is not faster than kernel (a) (shared memory) is likely due to the calculation time being dominated by fetching data from the main GPU memory. The reduction seems to be fast in comparison to data-fetching one way or the other.

There is a peculiar kink, at a vector size of $3 \cdot 10^7$, that takes significantly shorter to process than a vector of size 10^6. I do not understand, why this is.

Description of the warp shuffle kernel

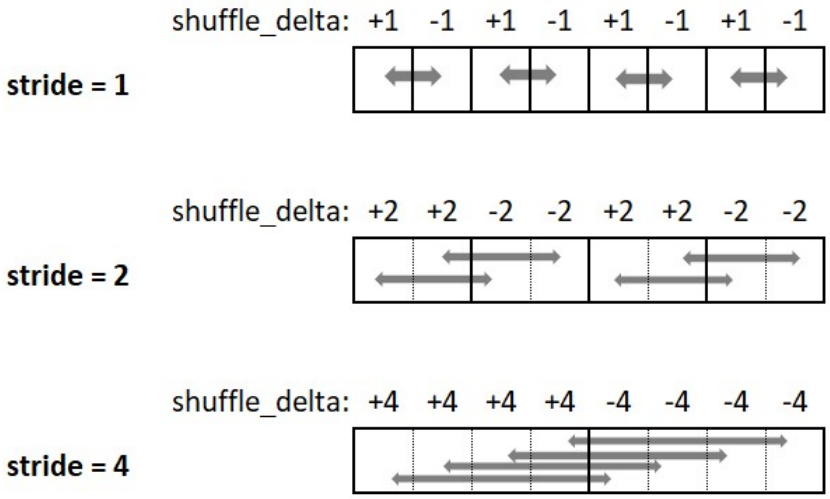
the warp shuffle performs a reduction in a way similarly to exercise 01.

Here is the warp shuffle reduction in code:

```
// at this point, the vector has been traversed and the threads locally hold sums of vector
// entries

// now the reduction inside the warp
int shuffle_delta;
int lane = threadIdx.x % warpSize;
for(unsigned int stride = 1; stride <= warpSize/2; stride *= 2){

	// if you are lower half -> get value from upper half and vice versa
	if ((lane % (2*stride)) < stride)
		shuffle_delta = stride;
	else
		shuffle_delta = (-1)*stride;

	__syncwarp();
	thread_sum += __shfl_down_sync(-1, thread_sum, shuffle_delta);
	thread_abssum += __shfl_down_sync(-1, thread_abssum, shuffle_delta);
	thread_squaresum += __shfl_down_sync(-1, thread_squaresum, shuffle_delta);
	thread_numzeros += __shfl_down_sync(-1, thread_numzeros, shuffle_delta);
}
// next up will be a loop of atomicAdd() to reduce the sums to a single number
```

like this, each thread ends up with the full sum in their local register. Therefore, it does not matter, which thread of the warp ends up executing the `atomicAdd()` function.

Here is a graphical representation of the reduction strategy (depicted for a warp size of 8):



Remark: warning thrown

This use of `__shfl_down_sync()` causes this warning to be thrown twice per line, where it is used:

warning: integer conversion resulted in a change of sign

This is, because the function expects an `unsigned` value for `shuffle_delta`. This implementation yields correct results, though.

Remark about transferring result values

- Trying to transfer the result values with a single call to `cudaMemcopy` led to unexplainable behaviour: Using a `struct Result`, that held the four results led to some complications.
- Using an array of length four caused very strange behaviour: The first result always ended up in the first three entries of the array. Therefore results two and three were lost. It might have been possible to use an array of length 6 and write the other three results to positions [2, 3 and 5] but this was not tried.

In the end, four separate double pointers were used for transferring the result. This is of course not ideal because of transfer latency.

2 Conjugate Gradients

2.1 Implementation

The full implementation is in the attached file **ex04_2_conj_grad.cpp**.

Kernels

Five kernels were implemented

- scalar product with warp shuffles (`gpu_dotp_wshuffle`)
  For this an adapted version of the kernel from the first part of the assignment was used.
- sparse matrix vector product (`gpu_csr_matvec_product_par`)
  For this the provided non-parallel kernel was parallelized. This was done by simply dividing the outer for-loop amongst the threads.
- three kernels for vector operations (`gpu_line6and7, gpu_line8, gpu_line12`)
  a very generic parallelization of the vector operations was used.

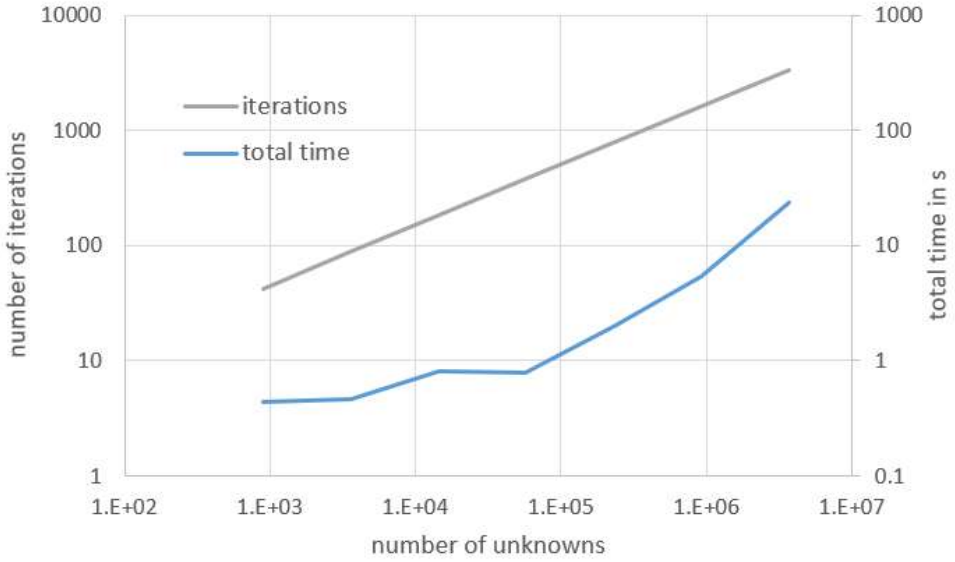
Conjugate Gradient Function

Before the iteration loop all objects, that will be worked on, are allocated on the device and initial values are copied over, where necessary.

During the loop only the squared residual norm $\langle r_i, r_i \rangle$ has to be copied to the host to test the stopping condition of the iteration. In the given implementation also the values for α and β are copied to allow the same printing to the console, that is done in the provided non-parallel version. This is a luxury, that would likely be omitted in a streamlined program, but was used here, because it did not noticeably affect the performance for large systems. No data is moved from the host to the device during the iteration loop.

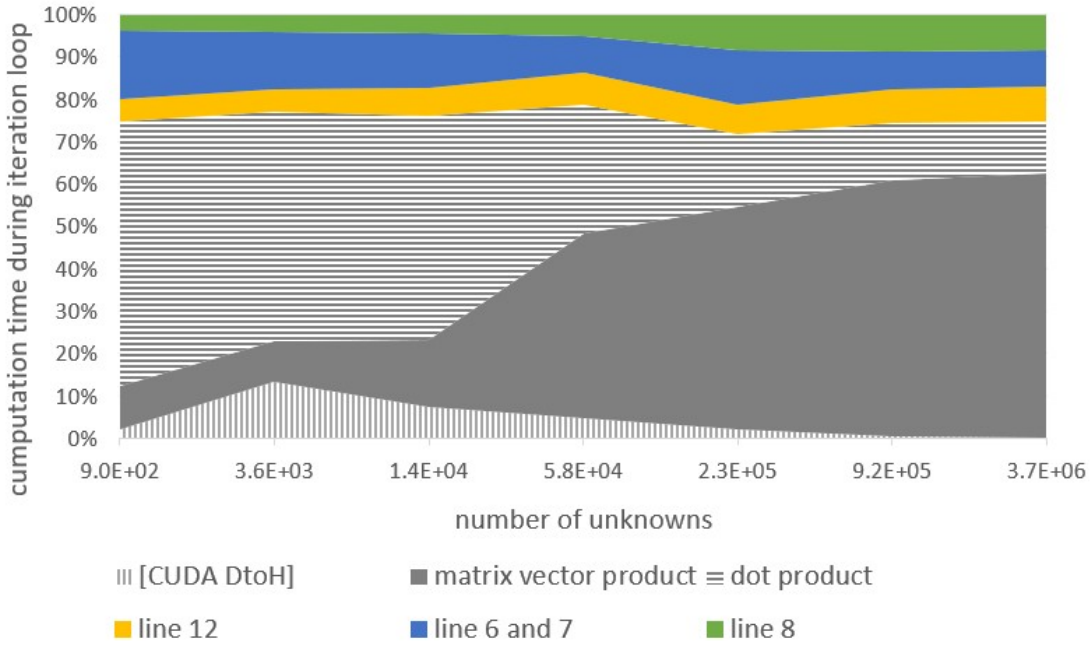
2.2 Performance Analysis

The following figure shows the time and number of iterations needed until convergence was reached for different system sizes:



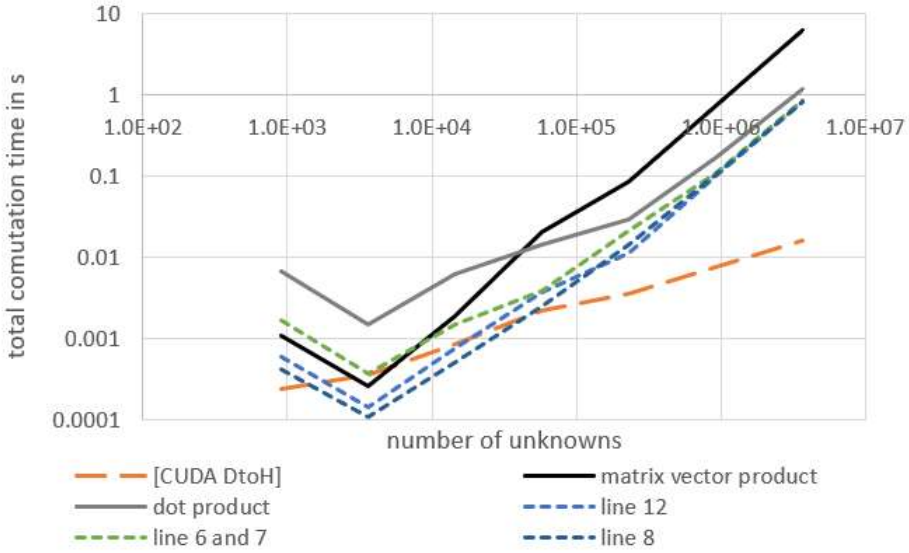
Analysis on a Per-Kernel Basis

It can be seen, that as the number of unknowns increases, the computation time during the iteration loop gets more and more dominated by the sparse matrix vector product. It can also be seen that the time for transfer of single numbers to the host becomes negligible. It can be concluded, that for large systems the optimization of the sparse matrix vector product would potentially yield the highest speedup.



It must be noted, that for 14,400 unknowns (grid size = 120) no convergence was observed, when running the program with active profiler. It did converge for all grid sizes, when the profiler was deactivated.

Here is another figure, that shows the total computation time on a per kernel basis.



Memory Performance Model

A simple memory performance model has been developed based on latency and bandwidth. The bandwidth was estimated using the fastest data transfer, that was observed, which was about $117\ GB/s$. For this, a data transfer of $(5+1+1)\cdot 8\cdot N$ Bytes per call of the `gpu_csr_matvec_product_par()` function was assumed (5 for the matrix, 1 for the vector and 1 for the result). For the latencies two values were used:

- latency1: the observed average execution for the smallest system ($23\mu s$)
- latency2: the shortest observed average execution time, which occurred for a system with 3600 unknowns ($2.9\mu s$)

The following figure shows the execution times compared to the memory models.