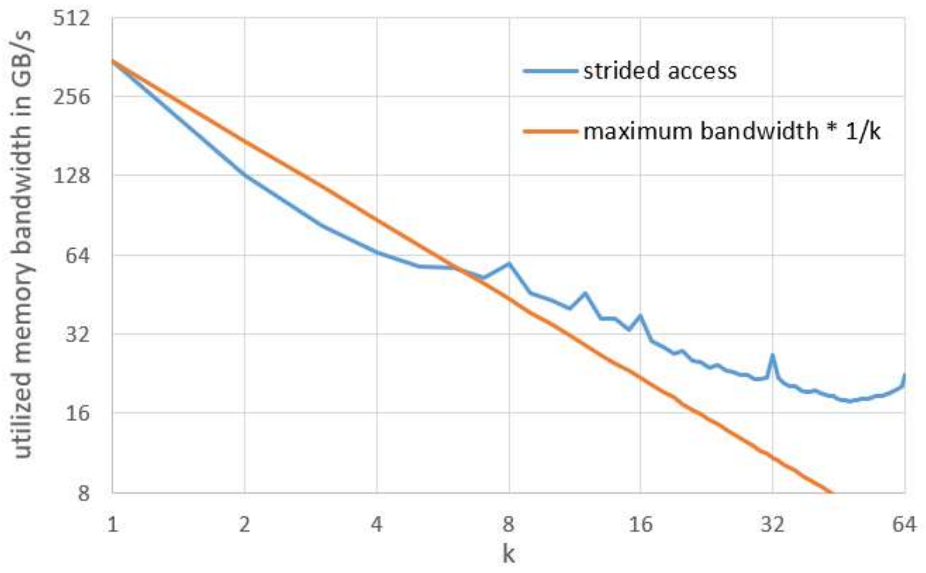
Simon Hinterseer e09925802

Exercise 03

1 Strided and offset memory access

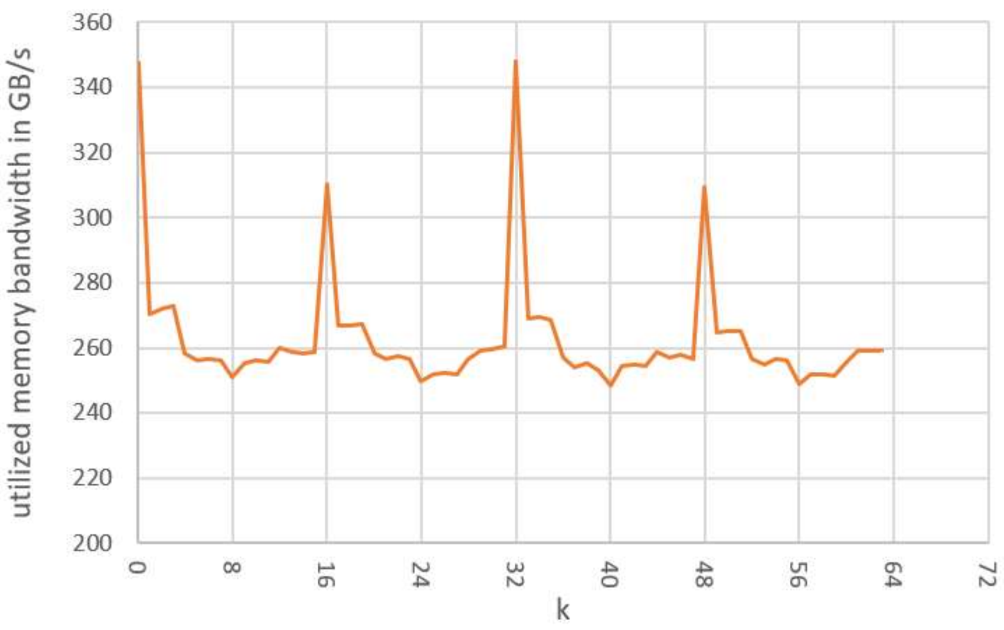
1.1 Strided memory access

When adding every k-th entry of the large vectors, it can be observed, that the utilized memory bandwidth falls roughly with $1/k$. The reduction is however not perfectly smooth. There are little bumps of increased bandwidth utilization at multiples of 16 and to a lesser extent at multiples of 8 and 4.



1.2 offset memory access

When adding all members but skipping the first k entries of the vector, the overall bandwidth usage shows peaks at multiples of 32 and to a lesser extent at multiples of 16. The differences are not as dramatic as with strided access. The minimum observed utilized bandwidth is still around 71% of the maximum.



1.3 Conclusions

First, when considering the results of strided memory access, it can be concluded, that it is better to access memory in a contiguous way than in a strided way.

Second, if accessing arrays with an offset, it is best to use and offset of 32. If that is not possible, an offset of 16 is still much better than a random offset.

2 Dense Matrix Transpose

2.1 find and fix the problems

`cuda-memcheck` finds a memory leaking problem. This is easily fixed by `free()` and `cudaFree()` statements for the initially allocated arrays.

Another problem exists concerning the correctness of the program: it will only work, if the matrix is sufficiently small and the number of threads is sufficiently large. And even then it feels a little lucky, that it works, because it relies on no thread writing to the matrix-array before all threads are finished reading from it.

This can be fixed by only iterating over the upper triangular matrix and swapping two entries while making use of a temporary double variable

```
if (row_idx < N && col_idx > row_idx){
  temp = A[row_idx * N + col_idx];
  A[row_idx * N + col_idx] = A[col_idx * N + row_idx];
  A[col_idx * N + row_idx] = temp;
}
```

and using a loop, such that every thread will keep swapping entries until the matrix is finished:

```
for(size_t i = t_idx; i < N*N; i += num_threads)
  swap_entries();
```

2.2 Fix the Problems

(see section above)

2.3 Optimize the Kernels

The assignment was interpreted in the following way:

- "opt1": make one kernel, that optimizes the implementation by dropping the inplace requirement. This kernel avoids strided memory access when reading the input matrix, but does not avoid it, when writing the output matrix.
- "opt2": make one kernel, that avoids strided memory access both when reading and writing.
- "opt2_inplace": make one kernel, that avoids strided memory access both when reading and writing. Transpose the matrix inplace.

opt1

Instead of using a temporary double to swap two entries of a matrix, every thread just grabs an entry of one matrix and writes it to the transposed position of another matrix. This causes strided memory access when writing, since it is done column wise in a matrix, that is stored row wise.

opt2

This kernel avoids strided memory access both when reading and writing by loading a $16x16$ sub-matrix, transposing it locally in a per-workgroup-shared array and writing the transposed block to the transposed position of a second matrix.

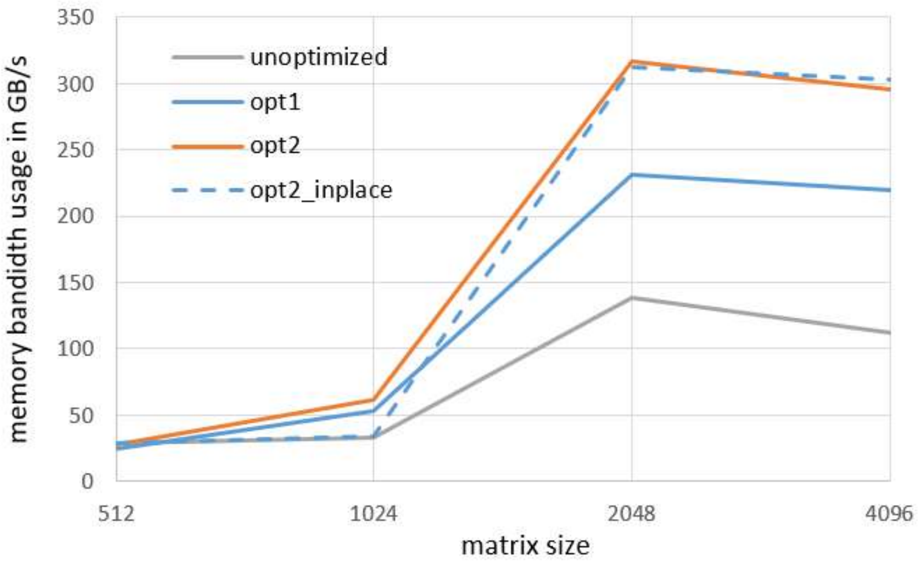
Like this, the strided memory access takes place on the local sub-matrix. The access to the matrix, that holds the result is just a row-wise copying of the local sub-matrix.

opt2_inplace

This kernel does the same as opt2, but performs the transposition inplace. For that two local per-workgroup-shared arrays are used. The kernel iterates over the upper triangular matrix and

- loads a $16x16$ sub-matrix and the sub-matrix at the transposed position to the shared arrays
- transposes both blocks locally
- writes the transposed blocks to the swapped position of the matrix

The results show, that opt2 and opt2_inplace make good usage of the available bandwidth.



3 Additional Remarks

Attached text files

Code for part 1 can be found in:

- ex03_011_sum_every_kth.cpp
- ex03_012_sum_skip_k.cpp

Code for part 2 can be found in:

- ex03_part2_handin.cpp

Execution wrapper mystery

An execution wrapper function was used, that would return the median run time after a few repetitions of the transposition. When checking for correctness, the execution with the execution wrapper ended up with an unchanged matrix for kernels, that would transpose the matrix inplace. This is expected for an even number of transpositions, but was also observed for uneven numbers. When calling the kernels directly without the use of the wrapper, the results were always correct. Also for kernels, that would write the result to a second matrix, the results were always correct.

Despite this dubious behaviour, the wrapper was still used to measure the runtimes, since the results seemed plausible.