Exercise 05

# Multiple Dot Products

The code for this part can be found in the attached file **ex05\_1.cpp**.

The task is to calculate dot products for .

This is done by repeatedly calling a custom kernel, that calculates 8 dot products at a time. The kernel does this very much in the same way, as it was done in the previous exercise. The only difference is, that instead of having a single thread local register, that holds the intermediate result, now there is an array of eight such registers. The first reduction stage is done by using warp shuffles and the second one by using atomicAdd().

Looking at the resulting execution times, it is well visible, that up until a vector size of the execution time is dominated by overheads. Only for larger vectors of size a linear dependency between vector size and execution time can be observed. It must also be noted, that a dip in execution time can be observed between a vector size of and . Some kind of performance optimization seems to take place beyond a certain vector size.

Chart, line chart

Description automatically generated

Figure 1 – comparison of custom kernels for different values of K

Comparing the custom kernel with calls to cublasDdot(), it can be noted, that the custom kernel is typically between 1.8 and 2.4 times as fast. This is to be expected, since calculating the dot product has a low arithmetic intensity and is therefore likely to be memory bound. The custom kernel only has to load 9 vectors to calculate 8 dot products, while cublasDot() has to load 16 vectors for the same task, which is roughly twice as much. In addition to that, there is some overhead because a larger number of kernels must be launched. Finally, the comparison is not completely fair, because cublasDot() gets handed a pointer to host memory for the result, so the transfer of the result from device memory is included here, while for the custom kernel all results were transferred after measuring the execution time. Figure 2 shows the comparison.

Chart, line chart

Description automatically generated

Figure 2 – comparison of custom kernels and the cuBLAS function

General Values of K

If cannot be assumed to be a multiple of 8, several approaches are possible. Ideally the bulk of the work is done by calling a kernel, that calculates 8 dot products at a time times and processing the remaining vectors in a single function call. This can be done by making a kernel, that gets the number of remaining vectors passed as a parameter. In the kernel found in **ex05\_1.cpp** it is possible to just replace every integer literal 8 by the number of vectors, that are to be processed. When looping over calls of these kernels, one might use something like this:

// do the bulk of the work

**for**(**int** i = 0; i < **int**(K/8); i++)

kernel\_dop8(x, y[i\*8], other arguments);

// process the remaining vectors

kernel\_dotp\_various(x, y[8\***int**(K/8)], K % 8, other arguments);

Of course, it would also be possible to just process all vectors in the second kernel. For large numbers of K, this could be inconvenient, because the space for all vectors needs to be allocated on the device. Also the potential speedup when increasing the size of the processed batch of vectors beyond 8 is capped at .