MSIAM: High Performance Computing

Spring 2024

Lab sheet #1: Welcome to UGAPerformance

07.02.2024

Contents

Situation

Welcome on board! You recently started at your new employer *UGAPerformance* which provides HPC consultation and services for companies which like to exploit the top performance of their computers.

The company *LOWPerformance* provided some source code which they like to have working and optimized. This source code already includes certain features to measure the passed time, to start benchmarks, etc. It's important to stick to this source code to ensure the reusability of your developments for LOWPerformance.

Assignment **First steps with the code:** Your first task is to get the code running on the provided computer and have a look at it for at least 15 minutes before further proceeding to these assignments.

Assignment **Presentation:** LOWPerformance just wants to see whether you're able to do this work and to discuss further steps with you in **two weeks** once you got some results. For this meeting, you should **prepare a short presentation** with the results. This is not a formal meeting, so feel free to copy-paste the output (e.g., screenshots) of the benchmarks directly to the presentation slides. Also, include a **brief explanation of the results**.

1) Matrix initialization

LOWPerformance wants to make sure that your code optimizations are indeed doing the right computations since HPC optimizations might introduce errors. Therefore, having an analytical solution of the resulting outcome of matrix-matrix multiplications would be good to validate our implementation.

We could, e.g., set up the matrix "A" and "B" in a way that the rows of "A" are orthogonal to the rows of "B". Similar to the Fourier modes, you can use the trigonometric functions

$$A_{i,j} = \cos\left(\frac{(j+\frac{1}{2})(i+\frac{1}{2})}{N}\pi\right) \tag{1}$$

$$B_{i,j} = \cos\left(\frac{(i+\frac{1}{2})(j+\frac{1}{2})}{N}\pi\right)\cdot\frac{2}{N} \tag{2}$$

where N is the number of columns and rows of the square matrices. Using these values to setup the matrix entries, we (should) get

$$AB = I$$

with I the identity matrix.

Assignment Implement this matrix setup in the functions `matrix_setup_A` and `matrix_setup_B`.

If you did anything wrong, other validation tests (which test for the identity matrix) might point out issues about this in one of the following assignments.

2) Matrix norm

There are different matrix norms which are all quite interesting from a mathematical point of view. However, in this course we are not interested to really use them for investigating numerical properties, but solely to study performance impacts of computing them.

Assignment Implement two different variants to compute a simple summation over all elements of the matrix "A".

- The first variant should have the innermost loop across the rows of the matrix (`kernel__matrix_sum_rowwise`).
- The second variant should have the innermost loop across the columns of the matrix (`kernel__matrix_sum_colwise`).

Compare the performance of both implementations for different problem sizes of matrices and explain the performance impact you observe.

There is a script `./run_01_matrix_norm_novec.sh` available which you can execute to get some benchmark results for both variants.

Comment There are a variety of `./run_*.sh` scripts made available. Each one of them starts the building process and in case of success, starts executing benchmarks for different problem sizes and variants. Also, they automatically collect benchmarks results in a .csv file which will be generated automatically.

3) Matrix multiplication

This assignment is about developing the fundamental matrix multiplication code which will be investigated also over the next worksheets. We will investigate a dense matrix-matrix multiplication. Here, two matrices A and B will be multiplied, resulting in C so that

$$C+ = AB.$$

In the most straight-forward way, the detailed algorithm is given as follows:

Algorithm 1: Matrices product with "ijk" ordering of loops

Data: Matrices A and B
Result: Matrix C

```
1 for i ← 1 to n do
2     for j ← 1 to p do
3         sum = 0
4         for k ← 1 to m do
5             sum += A_ik B_kj
6         end
7         C_ij += sum
8     end
9 end
```


This follows the "ijk" ordering of loops: the outer loop using index "i", the next one "j" and the inner one "k".

Assignment Implement the "ijk" ordering to multiply two matrices and use the given validation features to make sure that the implementation is correct.
The corresponding function is called `kernel__matrix_matrix_mul_simple_ijk`.

There is a script `./run_02a_mmul_simple_ijk.sh` available to test the implementation with the "ijk" ordering. Executing this script should also output some information about how many floating point operations per second are executed for which matrix size.

Assignment Step-by-step, also implement all other variants of different loop orderings (there are 6 variants in total). Once you implemented all different orderings, you can test them with the script `./run_02b_mmul_simple_all.sh`. Given the overall results, start interpreting them: How do you account for results being different just by swapping the order of the loops?

4) Aliasing & restrict statements

(See also lecture slides) Given a function of the form

```
void foo(double *a, double *b)
```

requires the compiler to be careful if updating one of the elements pointed to. E.g., `a` might be identical to `b`. Assuming that some data in `a` has been already read, updating values addressed indirectly by `b` would required to reload the values pointed to by `a`.

The `__restrict__` keyword can be used with the GNU compiler to annotate your function arguments to tell the compiler that there's no such aliasing.

Assignment Implement the kernel `kernel__matrix_matrix_mul_restricted_ikj` where you use the `__restrict__` keywords. Measure the performance with the `./run_03_mmul_restrict.sh` script.