Calcul Numérique et accélération Matérielle (CNM)

Prof Marina Zapater
Assistant: Mehdi Akeddar

Session 1 - Matrix multiplication optimization

lab01 (03.10.2024)

Objectives of the laboratory

The objectives of Lab01 are to exercise the content regarding acceleration using CPU and SIMD. Lab01 is structured in three sessions. This manual refers to "Session 1" and is devoted to optimizing the performance of matrix multiplication by taking into consideration the cache memory hierarchy.

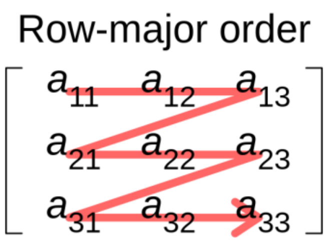
Laboratory validation and grading

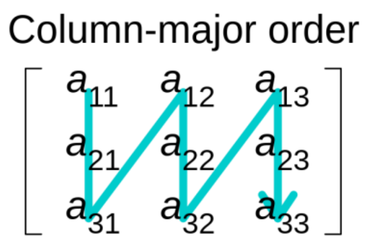
- We will grade this laboratory based on the work we ask to be delivered.
  The parts in *light blue* are what you are expected to reply in the readme file.
- We expect (at least) three different commits to the git repository, one per session, clearly indicating the session submission (for example: git commit -m "submission session 1 lab01")
- This commit should be done at the latest the night before the next lab session.
- Therefore, you have a full week to submit the lab.

Stage 1 – Understanding matrix memory layout and matrix multiplication

Since main memory is linear (1D) and matrices are 2D structures, a matrix can be stored in memory in two different ways: row major and column major:

- In row major order elements on each row are adjacent in each memory block
- In column major order, elements on each column are adjacent in each memory block.


Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}


Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

In this example, we have two square matrices *N* × *N* (N=3) in both ways to store 2D arrays in linear storage

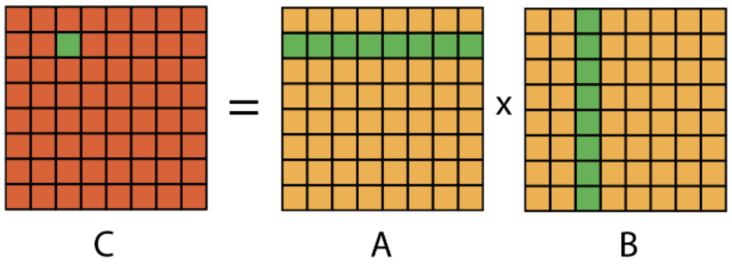
- In the row major order matrix (R), element (i, j) is stored at *R[i * N + k]*
- In the column major order (C), element (i, j) is stored at *C[i + k * N]*

Matrix multiplication is a binary operation. If A is an *m* × *n* matrix and B is an *n* × *p* matrix, then the matrix product C = AB is defined to be an *m* × *p* matrix such that

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix} \mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

Here we have a graphical representation of matrix multiplication (in this case, between square matrices).



$$\mathbf{C[i,j] = A[i,:] \cdot B[:,j]}$$

Stage 2 – Source template and compiler flags

For this laboratory, we provide source code with a basic implementation of square matrix multiplication and then some templates so that you can write your own code. We also provide the basics to measure performance.

Compile the source code using the following command.

```
gcc -O0 -o main main.c
```

Flag "-O" controls the optimization level performed by the compiler. Here we use "–O0" to disable most of the optimizations. You can read more about optimizations in the official documentation.

The generated program accepts two arguments. The first one, required, defines the size of the matrices (we will use square matrices, so only one dimension is needed) and the second one, if present, indicates the tile size.

```
# 100x100 matrix multiplication

./main 100

# 100x100 tiling matrix multiplication with tile size 5x5

./main 100 5
```

The output is the time (in seconds) of the matrix multiplication operation.

Stage 3 –Implementing general matrix multiplication

We provided a simple implementation of matrix multiplication for the specific case when we have square matrices. In this case, the dimensions are the same for all the matrices; this makes the naïve multiplication method easier to understand.

We ask you to implement the general matrix multiplication method (not only square matrices) based on the code of the naive implementation of square matrix multiplications. Use the function already declared in the code.

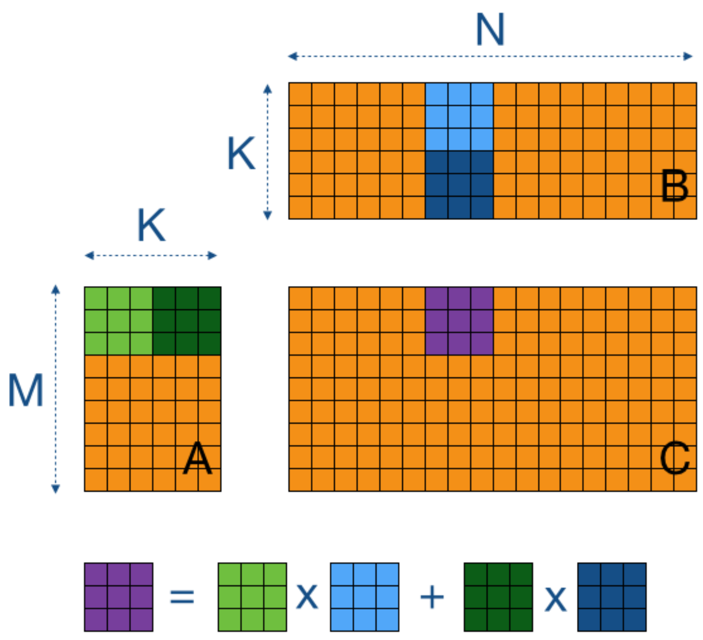
Stage 4 – Measuring naïve matrix multiplication performance

Once you have implemented the general matrix multiplication method, we will measure its performance based on matrix size. We will measure the time it takes to perform matrix multiplication for different matrix sizes and analyse how different matrix sizes affects performance.

We ask you to measure the time it takes to perform naive matrix multiplication for different matrix sizes. Call the generated program with different matrix sizes. Analyse the results based on matrix size and cache (use big enough matrices to measure the case where they do not fit in the cache l1d and l2).

Stage 5 –Implementing tile matrix multiplication

The main reason the naïve implementation (without any optimization) does not perform so well is because cache and data locality. One of the optimizations proposed is to break down the matrix multiplication problem into smaller matrix multiplication to optimize data locality and cache.



We ask you to implement the tiling matrix multiplication based on what you have already seen in the matrix multiplication. Use the function already declared in the code.

Stage 6 – Measuring tiling matrix multiplication performance

Once you have implemented the tiling matrix multiplication method, we will measure its performance based on matrix size and tile size. We will measure the time it takes to perform matrix multiplication for different matrix sizes and tile sizes and analyse how different matrix sizes and tile sizes affects performance.

We ask you to measure the time it takes to perform tiling matrix multiplication for different matrix sizes and tile sizes. Call the generated program with different matrix and tile sizes. Analyse the results based on matrix size and cache size and characteristics.

Stage 7 – Perf monitoring tool:

The **perf** tool, available on the command line via the “perf” command, is a powerful and versatile performance analysis tool available in the Linux ecosystem. It is part of the "perf tools" suite, which is a collection of utilities for performance monitoring, profiling, and tracing. **perf** works by reading a set of hardware registers named “performance counters” that are embedded in modern processors. **perf** can be used used to collect various types of performance events, such as hardware event statistics (misses/hits to the different levels of the cache hierarchy, branches, number of instructions), function call traces, and more. It allows you to gather detailed performance data about your applications and system.
This information is crucial for understanding how your software is performing and identifying performance bottlenecks or issues. **Have a look at the** documentation **to perform the next task.**

Launching perf monitoring:

As you will see in the documentation we linked above, to gather performance statistics for the program you want run, you can use “perf stat”.

```
perf stat -e EVENTS_YOU_WANT_TO_MONITOR ./program-to-monitor
```

You need to provide **perf stat** with a comma-separated list of the events you want to record, as well as the name of the program that you want to monitor.

Run the perf command on the 2 executables (Naïve and Tile implementation). ***Record the L1 cache loads, the L1 cache loads misses,*** *and the total task time. To accomplish this task, you will need to look into the “perf stat” documentation. We recommend that you list the events available, and that you pick the relevant ones you want to record.*

Before you leave…

Make sure you have all the requests *in light blue* of this manual, that you have implemented all the functions required and filled-in the answers in the *README* file in the repo. Then commit all files to your own cnm24 repository.

We expect (at least) one commit for this session, clearly indicating the session submission (for example: git commit -m “submission session 1 lab01”)

This commit should be done at the latest the day before the next lab session at 23:59h.