High Performance GEMM Challenge

Numerical Programming – Challenge exercise

The submission deadline is **Wednesday 10 December** at 23:59.

Objective

The goal of this challenge is to explore and implement **high-performance** General Matrix-Matrix Multiplication (GEMM).
Follow the *online course*:

LAFF-On Programming for High Performance

This material will help you understand the main optimization strategies for high-performance computing and how to implement gemm in C. Based on the knowledge acquired from the course, you will implement several optimized versions of GEMM.

1 Overview

You will progressively improve the performance of your GEMM implementation by applying different optimization techniques discussed in the course.

Each student will:

1. Follow the online course and understand the optimization techniques presented.
2. Implement and test the required GEMM versions described below.
3. Measure performance and generate plots of execution time and compute throughput.
4. Participate in a class-wide challenge to determine the fastest implementation.

At the end of the challenge, all implementations will be benchmarked on the same machine (of our choosing) to determine the fastest version

Prize: one pizza!

2 Mandatory Parts

2.1 Loop Ordering (Chapter 1.2)

Implement the three different loop orderings of the classical triple-loop GEMM algorithm:

$$C = A \times B + C$$

The required loop orderings are:

- `IJP` - Do it in the file `matmult_ijp.c`
- `PJI` - Do it in the file `matmult_pji.c`
- `JPI` - Do it in the file `matmult_jpi.c`

For each version:

- Measure the performance for different matrix sizes.
- Generate a plot of performance (execution time (s) and throughput (GFLOP/s)) versus matrix size.

* The code for evaluating the performance is already implemented in `main.c`, you only have to follow the instruction below in order to get the plots.

2.2 Blocked Matrix-Matrix Multiplication (Chapter 2.2)

Implement the **blocked version** of the GEMM algorithm, following the approach introduced in the course. - Do it in the file `matmult_blocked.c`

2.2.1 Play with Block Size

Experiment with different block sizes:

- Run the blocked algorithm for various block sizes.
- Identify the block size that provides optimal performance on your system.

You can store in a table the speedup (or throughput) for different values of block size and matrix dimension.

3 Competition Benchmark: Your Best Version

After completing all mandatory parts, implement your **best optimized version** of GEMM using all the techniques learned. This is your chance to compete for the fastest implementation!

- Use any advanced strategy you like, based on the online course or additional material you find.
- You are free to experiment with different techniques and compiler optimizations.
- Record performance for multiple matrix sizes (see below how to do it).

Performance Optimization Hints

Some strategies you can explore to improve your implementation are:

- **Optimal memory layout:** Ensure data is accessed in a cache-friendly manner.
- **Blocking for registers:** Use blocking techniques at both cache and register levels.
- **Micro-kernel optimization:**
  - On ARM systems, the instruction set AVX (that is used in the LAFF course) is not available. You can use static loops with:

    ```
    #pragma unroll(BLOCK_SIZE)
    ```

    and tune `BLOCK_SIZE` for implicit vectorization.
  - To push further: explore **ARM NEON** or **SVE** instructions.

4 Practical instructions:

You are provided with:

- `Makefile`,
- `main.c` - you only have to modify your Name and Surname here,
- `matmult.h`
- `matmult_jpi.c` - to be implemented,
- `matmult_ijp.c` - to be implemented,
- `matmult_pji.c` - to be implemented,
- `matmult_blocked.c` - to be implemented,
- `matmult.c` - to be implemented, $\longrightarrow$ **Your best version**

Performance analysis - plots

In order to compare the performance between `matmult_jpi.c`, `matmult_ijp.c`, `matmult_pji.c` and `matmult_blocked.c` you have to do the following two steps:

1. **Code compliation & benchmarking:** In order to compile the four versions of the code we want to compare, you can simply run in your terminal:

   ```
   make perf
   ```

   This command first builds all the optimized programs, then runs each one with different input sizes to measure performance and collects the results into a single CSV file called `perf.csv`.

2. **Performance visualisation:** After saving all the measured data in `perf.csv`, you can use the script `plot_perf.py` to visualize the performance.
   To run this code you have to set up a virtual environment as done in the Assignments:

   1. **Open a terminal and navigate to the assignment folder:**

      ```
      cd path/to/your/assignment/folder
      ```

      (Replace `path/to/your/assignment/folder` with the actual path.)

   2. **Create a virtual environment (venv):**

      ```
      python3 -m venv venv
      ```

      This will create a folder named `venv` in your assignment directory.

   3. **Activate the virtual environment:**
      On macOS/Linux:

      ```
      source venv/bin/activate
      ```

   4. **Install the required packages:** If your assignment folder contains a `requirements.txt` file, run:

      ```
      pip install -r requirements.txt
      ```

   5. **Now you can run:**

      ```
      python3 plot_perf.py
      ```

Single version test

Following the previous instructions, you compared your implementations for fixed sizes.

To run a single version of your code you have to:

1. **Compile all the versions:**
   You can compile a 'debug' version (useful for debugging and for check if your code is correct) and an 'opt' version (lower level of control, optimal for performance), to compile the two versions do:

   ```
   make debug
   ```

   or

   ```
   make opt
   ```

2. **Run the desired executable:**
   Now you have an executable version of the code for each implementation, in order to run the code with matrix size `M`, `N`, `K`, run:

   ```
   ./matmult_blocked-opt M N K output.csv
   ```

   Thanks to that you are running the blocked version with size `M`, `N`, `K` and storing the results in the file `output.csv`.

Evaluate your best version

You can compile and run your best version as discussed above, remember that you have to implement it in `matmult.c`, so you can run it using:

```
./matmult-opt M N K output.csv
```

For the challenge we will test your code on large matrix sizes, your goal is to obtain the highest possible throughput (or equivalently the smallest execution time).

Additional remarks:

- In the `Makefile`, at line 16, it is possible to set your CPU version with `-mcpu=apple-m3`. Make sure to use the correct "m" (m1, m2, m3, etc.). (You can find your version looking at 'About this Mac')
- Remember that each time you modify your code, you must compile the new version (equivalent to running one of the `make` commands) to make your changes effective.

5 Submission & presentation:

The deadline for the submission is **Wednesday 10 December at 23:59**, upload the folder with all the implementations on iCorsi.

On **Monday 15 December**, you will have the opportunity to present you work to the class (and discover the winner), in order to present you work you have to prepare a 4 slides presentation with the following structure:

1. Loop ordering,
2. Blocked version,
3. My best implemetation,
4. Advanced features.