Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing, Fall 2024
Lecturer: Dr. A. Eftekhari, Prof. O. Schenk
Assistants: M. Lechekhab, D. Vega
D. Santarsiero, J. Palumbo, I. Ecevit

Project 1 – Performance Characteristics, Memory Hierarchies and Matrix-Matrix Multiplications

Due date: 9 October 2024 at 23:59 (See iCorsi for updates)

The first part of this project will introduce you to using the Rosa cluster and collecting some performance characteristics. The second part of the project will be about the optimization of general matrix multiplication. Both project parts will be single-threaded. Parallel programming will be the subject of the forthcoming projects.

1. Rosa warm-up [5 points]

Review the Rosa documentation and the Slurm tutorial to gain an understanding of its features and usage. Answer briefly the following questions in your report:

1. What is the module system and how do you use it?
2. What is Slurm and its intended function?
3. Write a simple "Hello World" `C/C++` program which prints the host name of the machine on which the program is running.
4. Write a batch script which runs your program on one node. The batch script should be able to target nodes with specific CPUs with different memories. You can obtain the information on available nodes using the command `sinfo`. **Hint**: Slurm hast the option `--constraint` to select some nodes with specific features.
5. Write another batch script which runs your program on two nodes. Check that the Slurm output file (`inputs/slurm-*.out` by default) contains two different host names.

Include the source code, batch scripts, and Slurm output files in your submission (see the notes at the end of the document).

2. Performance characteristics [30 points]

In the realm of HPC, understanding the performance limits of computing systems, particularly in relation to the memory system and floating-point operations, is crucial for designing efficient algorithms and optimizing their implementations on (a particular) hardware. The performance characteristics and concept of lightspeed estimates comes into play when considering the physical constraints that limit data transfer rates and computation speeds within a computing system. These constraints are not only dictated by the raw processing power of the CPU but also by the latency and throughput of the memory hierarchy. It is essential to grasp these concepts in order to appreciate the theoretical and practical limits of computing, including how data locality and memory bandwidth impact the achievable performance. This knowledge is fundamental in pushing the boundaries of what is computationally possible while being mindful of the inherent practical limitations posed by physics and current technology.
However, determining the performance characteristics of computing systems is often a complex and nuanced task that requires a multi-faceted approach. Manufacturer documentation provides a baseline of theoretical capabilities and specifications, but real-world performance can deviate significantly from these idealized scenarios due to a myriad

of factors. Web resources, including insights from HPC centers, offer valuable empirical data and analyses that reflect the performance characteristics. Yet another source of information is experimentation in the form of low-level benchmarking. Thus, a combination of scrutinizing manufacturer documentation, leveraging the wealth of information available from HPC centers and other web resources, and conducting methodical benchmarking experiments is essential for a comprehensive understanding of computing system performance.
The goal of this task is to collect such performance characteristics and lightspeed estimates. As a refresher (or a concise introduction), we *strongly* encourage you to read the article and appendix [1], and Chapters One and Three of the book [2][1].

2.1. Peak performance

In this task, we ask you to compute the core, CPU, node and cluster peak performance for the Rosa nodes. Please detail your computation and all sources in your report. In the following, we provide some context and give a step-by-step guide to complete and report the task.
In scientific computing, the focus often lies on floating-point (FP) data, typically utilizing *double precision*. The rate at which the CPU's floating-point unit (FPU) can produce results for multiplication and addition operations is quantified in terms of *floating-point operations per second* (Flops/s, or simply FLOPS). The maximum rate at which a system is capable of executing Flops/s is the so-called (theoretical[2]) *peak (FP) performance*. The peak performance P_{core} of a single core is computed as follows

$$P_{\text{core}} = n_{\text{super}} \times n_{\text{FMA}} \times n_{\text{SIMD}} \times f$$

where

- n_{super} is the superscalarity factor: the number of FP operations the FPU of a core can execute in parallel within a single clock cycle.
- n_{FMA} is the fused multiply-add factor: $n_{\text{FMA}} = 2$ if the FPU supports the FMA instruction (enabling it to perform a multiplication and an addition in a single operation), or $n_{\text{FMA}} = 1$ if the FPU does not support FMA (requiring separate instructions for multiplication and addition).
- n_{SIMD} is the SIMD (Single Instruction, Multiple Data) factor: the number of doubles the FPU can process concurrently with a single SIMD instruction (i.e., the width of the SIMD registers in units of doubles).
- f is the (base) clock frequency (modern multi-core CPUs may dynamically increase their clock frequency to make use of Thermal Design Power (TDP) more efficiently if fewer cores are active).

The peak performance P_{CPU} of a CPU is then

$$P_{\text{CPU}} = n_{\text{cores}} \times P_{\text{core}},$$

where n_{cores} is the number of *physical* cores[3]. The peak performance P_{node} of a node with n_{sockets} identical CPUs

$$P_{\text{node}} = n_{\text{sockets}} \times P_{\text{CPU}}.$$

Finally, the peak performance P_{cluster} of a cluster consisting of n_{nodes} identical nodes is

$$P_{\text{cluster}} = n_{\text{nodes}} \times P_{\text{node}}.$$

[1] To gain access, you have to be within the USI network, e.g., by using the VPN.
[2] There is a distinction between theoretical and measured peak performance, but we will ignore it for the time being (see here)
[3] As opposed to *logical* cores often present on modern CPUs with *threading* capabilities, such as Hyper-threading (HT) for Intel CPU or Simultaneous Multithreading (SMT) for AMD.

Let's compute the peak performance of the Euler III (2016-2022) nodes as an example. The system comprised 1215 nodes (i.e., $n_{\text{nodes}} = 1215$), each equipped with a single Intel Xeon E3-1585Lv5 CPU (i.e., $n_{\text{sockets}} = 1$). According to the Euler webpage, which conveniently provides detailed specifications, each of these CPUs has four cores (i.e., $n_{\text{cores}} = 4$). Additionally, the link to Intel's ARK website offers further technical details, revealing that the CPU operates at a base clock frequency of $f = 3.00$ GHz and supports AVX2 SIMD instructions with 256-bit wide vector registers. This setup allows for processing four 64-bit double-precision FP numbers simultaneously, leading to $n_{\text{SIMD}} = 4$.
The remaining factors to consider for calculating peak performance are the superscalarity and FMA factors. However, these details are not as readily available and typically require a deep dive into technical documents provided by the CPU manufacturer or from technology review sites and academic research. In this specific instance, valuable information is found in Figure 2-9 of Intel's Optimization Reference Manual Volume 1 (around p. 77 in the PDF). This figure indicates that Ports 0 and 1 each can perform one vector FMA operation (i.e., $n_{\text{FMA}} = 2$), establishing the superscalarity factor $n_{\text{super}} = 2$. Hence, we can compute the (theoretical) peak performance of the Euler III nodes

$$\begin{aligned} P_{\text{core}} &= 2 \times 2 \times 4 \times 3 \text{ GHz} = 48 \text{ GFlops/s}, \\ P_{\text{CPU}} &= 4 \times P_{\text{core}} = 192 \text{ GFlops/s}, \\ P_{\text{node}} &= 1 \times P_{\text{CPU}} = 192 \text{ GFlops/s}, \\ P_{\text{Euler III}} &= 233'280 \text{ GFlops/s} = 233.28 \text{ TFlops/s} \end{aligned}$$

Here are some useful hints for the task:

- Rosa cluster reveals that each node has two Intel Xeon E5-2650 CPUs and the Intel's ARK website provides general specifications. From Intel's specifications you get that the CPU is part of the Intel Xeon E5 collection, from which you deduce that it is based on fourth-generation core microarchitecture (codenamed "Haswell").
- The cpu-world.com website will provide you with the SIMD factor.
- The uops.info website will provide you with the superscalarity factor. Instead, you could consult Intel's documentation for this factor. However, this approach is slightly more complex and can be avoided.
- This Intel's Optimization Reference Manual will then provide you with the FMA factor.
- Remember that ThroughPut (TP) is defined by how many cycles it takes to execute one instruction (i.e., Cycles Per Instruction (CPI)).

In general (beyond this project and course), the following resources are valuable:

- Intel's software and optimization manuals: The easiest way to find them is to search for "Software Developer's Manual" or "Optimization Reference Manual" in your favorite web search engine.
- AMD's software and optimization manuals: The easiest way to find them is to search for "AMD documentation hub" in your favorite web search engine. Once on the hub, search for "Software optimization guide" specific to the CPU model you are interested.
- Intel Intrinsics Guide
- Agner Fog's software optimization resources: `https://www.agner.org/optimize/`
- Latency and throughput, listing: `https://uops.info/`.
- `https://en.wikichip.org`: Can be a useful resource (despite the ads).

2.2. Memory Hierarchies

2.2.1. Cache and main memory size

The next task is to identify the parameters of the memory hierarchy on a node of the Rosa cluster . Follow the step-by-step guide provided below and detail your results in your project report.
To guide you, we show how this can be achieved for a Rosa login node. A useful tool to determine the CPU architecture is

```
[user@icslogin]$ lscpu
```

From this we obtain the CPU model (Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz) featuring ten cores and some basic information on the memory hierarchy.
To obtain the total available main memory, one option is to look at the

```
[user@icslogin]$ cat /proc/meminfo
```

In summary, we have collected so far the information listed in Table 1. Since this is a multi-core CPU, it is valuable to obtain more information on how the memory hierarchy is organized and shared among the cores. A (possible; there are others) tool for this is provided by `hwloc`. In particular, the following commands will give us the desired information:

```
[user@icslogin]$ module load hwloc
[user@icslogin]$ hwloc-ls
```

From the output, we conclude that each of the ten cores has its own L1 instruction (L1i) and data (L1d) caches as well as an L2 cache. The L3 cache, along with the main memory, are shared among all cores. This observation has been noted in the caption of Table 1 to provide a complete overview. A graphical representation of the memory hierarchy is displayed in Fig. 1. To obtain such a figure, first ask `hwloc-ls` to output in the "fig" format

```
[user@icslogin]$ hwloc-ls --whole-system --no-io -f --of fig XEON\_E5-2650.fig
```

You can open the file using Xfig on your machine, you can convert `XEON_E5-2650.fig` by copying it on your local machine and converting it to a PDF file:

```
$ scp username@rosa.usi.ch:~/XEON_E5-2650.fig /your/local/folder
$ fig2dev XEON_E5-2650.fig XEON_E5-2650.pdf
```

For more information on these commands and their options, please have a look at there man pages (e.g., `man lscpu`).

Main memory	23 GB
L3 cache	25 MB
L2 cache	256 KB
L1 cache	32 KB

Table 1: Memory hierarchy of a Rosa login node with an Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz. Each core has its own L1 and L2 cache, while L3 is shared among all the ten cores.

2.3. Bandwidth: STREAM benchmark

Now that we have an overview of the peak performance, as well as the topology and size of the memory system, another key performance characteristic to consider is the speed of the memory system. The speed of the memory system is quantitatively measured in terms of its bandwidth, which indicates the amount of data that can be transferred within a specific time frame. McCalpin's STREAM benchmark [3] is a widely recognized tool used for measuring the memory bandwidth of a CPU. It consists of four operations or kernel — Copy, Scale, Add, and Triad — that evaluate

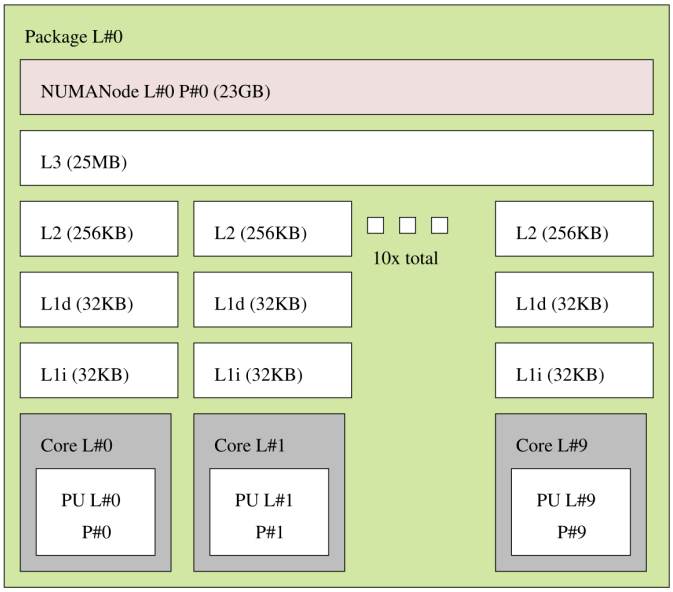

Figure 1: Schematic of a Rosa login node with an Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz.

the sustainable memory bandwidth and the corresponding computation rate for simple vector kernels (see, e.g, [2, Sec. 3.1.2]).
In this task, we ask you to run the STREAM benchmark on the Rosa nodes in order to measure the single-core bandwidth. Find below a step-by-step guide:

1. Download the STREAM Benchmark: it can be obtained from the official website in the source code directory. You will need `stream.c` and `mysecond.c`.

2. In `stream.c`, you will find precise instructions on how to compile and run the benchmark. It is particularly important to tune the sizes of the arrays (used in the Copy, Scale, Add, and Triad kernels) to match the cache sizes of the system of interest: Each array must be at least four times the size of the last cache level. This is set by the `STREAM_ARRAY_SIZE` preprocessor macro and two examples are given in the `stream.c`. For the CPU studied in the example of 2.2.1, we would then

```
module load gcc                                    # load a compiler (gcc/13.2.0)
gcc -O3 -march=native -DSTREAM_TYPE=double -DSTREAM_ARRAY_SIZE=128000000 /
-DNTIMES=20 stream.c -o stream_c.exe                # compile
sbatch --mem-per-cpu=4G --wrap "./stream_c.exe" # submit to queue and run
```

3. The output (`slurm-jobid.out`) produced:

```
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           19158.8     0.106950     0.106896     0.107101
Scale:          11228.1     0.182465     0.182399     0.182589
Add:            12278.5     0.250293     0.250193     0.250484
Triad:          12285.6     0.250150     0.250048     0.250300
-------------------------------------------------------------
```

   We observe that the bandwidths resulting from the Scale, Add, and Triad kernels are roughly consistent, while the bandwidth for the Copy kernel appears to be substantially higher. There may be several explanations for this discrepancy, but they are beyond the scope of this project (and course). For a rough estimate, we can assume a maximum bandwidth $b_{\text{STREAM}} = 12$ GB/s.

Of course, you will need to adapt some these steps to the task. In your report, follow a similar style to the above description. Include source code (including build files), batch scripts, and Slurm output files in your submission.
For further context and advice regarding the STREAM benchmark, we recommend looking at the official website.

2.4. Performance model: A simple roofline model

The *roofline* model is a visual representation that serves as a powerful tool for evaluating the performance potential of computing systems, especially in the context of high-performance computing [1]. It plots achievable performance (in GFlops/s) against operational intensity (in Flops/Byte), delineating the maximum performance given by the system's peak performance and the memory bandwidth constraints. The "roofline" itself consists of two main segments: the *memory-bound* region, where performance is limited by the system's memory bandwidth, and the *compute-bound* region, where performance is capped by the peak computational power of the processor. The frontier between both regimes is called the *ridge point*. This model allows developers and researchers to identify whether their algorithms are memory-bound or compute-bound and to understand the performance implications of hardware limitations, guiding optimizations towards achieving maximum efficiency within the given hardware constraints.
A simple (or sometimes called naive) roofline model for a hypothetical system combining Subsections 2.1 and 2.2.1 is displayed in Fig. 2. We observe that the ridge point is roughly at an operational intensity of $I_{\text{ridge}} \approx 4$. Hence, a given kernel or application with an operational intensity below I_{ridge} is memory-bound. Similarly, a given kernel or application with an operational intensity above I_{ridge} is compute-bound.
Your tasks are:

1. With your performance data collected in the previous tasks, create a roofline model for a single core of the Rosa nodes.
2. Visualize your roofline models in a plot similar to Fig. 2.
3. At what operational intensity is a kernel or application memory/compute-bound?

Include plotting scripts in your submission.

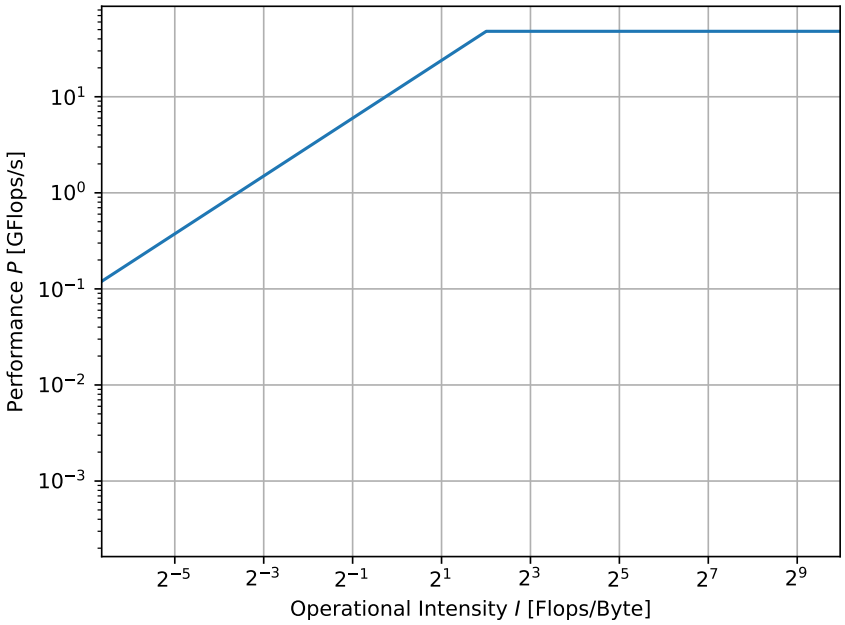

Figure 2: Simple (or naive) roofline model for a single core of a hypothetical CPU.

3. Optimize Square Matrix-Matrix Multiplication [50 points]

Problem statement

Your second task in this section[4] is to write an optimized matrix multiplication function on the Rosa computer. We will give you a generic matrix multiplication code (also called matmul or dgemm), and it will be your job to tune our code to run efficiently on the ICS processors.
Write an optimized single-threaded matrix multiply kernel. This will run on only one core.

Matrix multiplication

Matrix multiplication is the basic building block in many scientific computations; since it is an $\mathcal{O}(n^3)$ algorithm, these codes often spend a lot of their time in matrix multiplication. However, the arithmetic complexity is not the limiting factor on modern architectures. The actual performance of the algorithm is also influenced by the memory transfers. We will illustrate the effect with a common technique for improving cache performance, called blocking. Please refer to the additional material on the course webpage, titled *Motivation for Improving Matrix Multiplication* or in the book. Since we want to write fast programs, we must take the architecture into account. The most naive code to multiply matrices is short, simple, and very slow:

```
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C[i,j] = C[i,j] + A[i,k] * B[k,j]
    end
  end
```

[4]This document is originally based on a project from Professor Katherine A. Yelick from the Computer Science Department at the University of Berkeleyhttp://www.cs.berkeley.edu/ yelick/

```
end
```

Instead, we want to implement the algorithm that is aware of the memory hierarchy and tries to minimize the number of references to the slow memory (please refer to the "Motivation for Improving Matrix Multiplication" document provided on the HPC course web page `mini-project-info.pdf` for more detailed explanation):

```
for i=1 to n/s
   for j=1 to n/s
      Load C_{i,j} into fast memory
      for k=1 to n/s
         Load A_{i,k} into fast memory
         Load B_{k,j} into fast memory
         NaiveMM (A_{i,k}, B_{k,j},  C_{i,j}) using only fast memory
      end for
      Store C_{i,j} into slow memory
   end for
end for
```

Starter Code

Download the starter code: Directory matmul
The directory contains starter code for the matrix multiply. It contain the following source files:

- `dgemm-blocked.c` – A simple blocked implementation of matrix multiply. It is your job to optimize the square_dgemm() function in this file.
- `dgemm-blas.c` – A wrapper which calls the vendor's optimized BLAS implementation of matrix multiply (here, MKL).
- `dgemm-naive.c` – For illustrative purposes, a naive implementation of matrix multiplication using three nested loops.
- `benchmark.c` – A driver program that runs your code. You will not modify this file, except perhaps to change the MAX_SPEED constant if you wish to test on another computer (more about this below).
- `Makefile` – A simple makefile to build the executables.
- `run_matrixmult.sh` – Script that executes all three executables and produces log files (`*.data`) that contain the performance logs.
- `plot_matrixmult.sh` – Script that plots the data in the performance logs and produces a figure showing the results.

Running our Code

The starter code should work out of the box. To get started, we recommend you to log into the Rosa cluster and download the first part of the assignment. This will look something like the following:

```
[user@icslogin]$ cd 3-Optimize-Matrix-Matrix-Mult/
[user@icslogin]$ ls
benchmark.c  dgemm-blas.c  dgemm-blocked.c  dgemm-naive.c  Makefile
run_matrixmult.sh  timing_basic_dgemm.data  timing.gp
```

Next let's build the code.

```
[user@icslogin]$ module load gcc intel-oneapi-mkl
[user@icslogin]$ make
```

We now have three binaries: benchmark-blas, benchmark-blocked, and benchmark-naive. The easiest way to run the code is to submit a batch job. We have already provided batch files which will launch jobs for each matrix multiply version using one core:

```
[user@icslogin]$ sbatch run_matrixmult.sh
```

or

```
[user@icslogin]$ sbatch --reservation=hpc-wednesday run_matrixmult.sh
```

Please note that you need to adjust the reservation flag if running this code during class hours.
Our jobs are now submitted to the Rosa cluster 's job queue. We can now check on the status of our submitted jobs using a few different commands.

```
[user@icslogin]$ squeue
   JOBID     USER PARTITION     NAME ST   START_TIME     END_TIME TIME_LEFT  NODES
 1173224  user         slim matrixmu  R     14:47:17     15:17:17     29:50      1
 NODELIST(REASON)  PRIORITY
 icsnode17       707
```

When our job is finished, we'll find new files in our directory containing the output of our program. For example, we will find the files matrixmult-xxx.out and matrixmult-xxx.err . The first file contains the standard output of our program, and the second file contains the standard error. Additionally, the performance data are stored in *.data files. The script also generates `timing.ps` file, a visual representation of the results. You can copy the `timing.ps` file to your laptop and open it with your favorite PDF file viewer.

Interactive Session

You may find it useful to launch an interactive session when developing your code. This lets you compile and run code interactively on a compute node that you have reserved. In addition, running interactively lets you use the special interactive queue, which means you'll receive you allocation quicker. The `benchmark.c` file generates matrices of a number of different sizes and benchmarks the performance. It outputs the performance in FLOPS and in a percentage of theoretical peak attained. Your job is to get your matrix multiply's performance as close to the theoretical peak as possible.

Theoretical Peak

Our benchmark reports numbers as a percentage of theoretical peak. Here, we show you how we calculate the theoretical peak of the Rosa cluster 's Haswell processors. If you'd like to run the assignment on your own processor, you should follow this process to arrive at the theoretical peak of your own machine, and then replace the MAX_SPEED constant in benchmark.c with the theoretical peak of your machine.

One Core on the Rosa cluster

One core has a clock rate of 2.30 GHz, so it can issue 2.3 billion instructions per second. Haswell processors also have a 256-bit vector width, meaning each instruction can operate on 8 32-bit data elements at a time. Furthermore, the Haswell microarchitecture includes a fused multiply-add (FMA) instruction, which means 2 floating point operations can be performed in a single instruction. So, the theoretical peak of the Rosa cluster 's Haswell node is

- 2.3 GHz* 8-element vector * 2 ops in an FMA = 36.8 GFlops/s

Note that the matrices are stored in C style row-major order. However, the BLAS library expects matrices stored in column-major order. When we provide a matrix stored in row-wise ordering to the BLAS, the library will interpret it as its transpose. Knowing this, we can use an identity $B^T A^T = (AB)^T$ and provide matrices A and B to BLAS in

rowwise storage, swap the order when calling `dgemm` and expect the transpose of the result, $(AB)^T$. But the result is returned again in column-wise storage, so if we interpret it in rowwise storage, we obtain the desired result AB. Have a look at `dgemm-blas.c` to see how the A and B are passed to `dgemm`. Also, your program will actually be doing a multiply and add operation $C := C + A \cdot B$. Look at the code in `dgemm-naive.c` or study the `dgemm` signature if you find this confusing. The driver program supports result validation (enabled by default). So during the run of `benchmark-blocked` binary compiled from the `square_dgemm` code you wrote, the result correctness will be automatically checked for different matrix sizes.

3.1. Optimizing the matrix multiplication

Now, it's time to optimize!

- The `dgemm-blocked.c` contains the naive implementation of the square matrix multiply. Modify the code so that it performs blocking. Test your code and tune block sizes to obtain the best performance.
- Compare performance of your implementation to the Intel MKL by compiling and running the driver program and visualizing the performance results.

4. Quality of the Report [15 Points]

Each project will have 100 points (out of 15 point will be given to the general written quality of the report).

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.) and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi .

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - All the source codes of your solutions.
  - Build files and scripts. If you have modified the provided build files or scripts, make sure they still build the sources an run correctly. We will use them to grade your submission.
  - project_number_lastname_firstname.pdf, your write-up with your name.
  - Follow the provided guidelines for the report.
- Submit your .tgz through iCorsi .

Additional resources:

You may find useful the "Motivation for Improving Matrix Multiplication" document provided on the HPC course web page (`project1-info.pdf`). Please always use the project template which is available on the iCorsi webpage for your submission.

Code of Conduct and Policy

- Do not use or otherwise access any on-line source or service other than the iCorsi system for your submission. In particular, you may not consult sites such as GitHub Co-Pilot or ChatGPT.
- You must acknowledge any code you obtain from any source, including examples in the documentation or course material. Use code comments to acknowledge sources.
- Your code must compile with a standard-configuration C/C++ compiler.

References

[1] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[2] Georg Hager and Gerhard Wellein. Introduction to High Performance Computing for Scientists and Engineers. Chapman & Hall/CRC Computational Science, CRC Press, 2010. ISBN: 978-1-4398-1193-1, DOI: 10.1201/ebk1439811924.

[3] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 2, December 1995. `https://www.cs.virginia.edu/~mccalpin/papers/balance/`.

[4] Intel MKL – https://software.intel.com/en-us/intel-mkl

[5] Hager, G. and Wellein, G. Introduction to High Performance Computing for Scientists and Engineers. CRC Press, Inc. 2010. ISBN 9781439811924.

[6] OpenMP Tutorial - LLNL Computation: https://computing.llnl.gov/tutorials/openMP/