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 pproject1.pdfperformance 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 experproject1.pdfiments 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 [10], and Chapters One and Three of the book [3][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 2. 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 2 to provide a complete overview. A graphical representation of the memory hierarchy is displayed in Fig. 3. 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 [7] 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

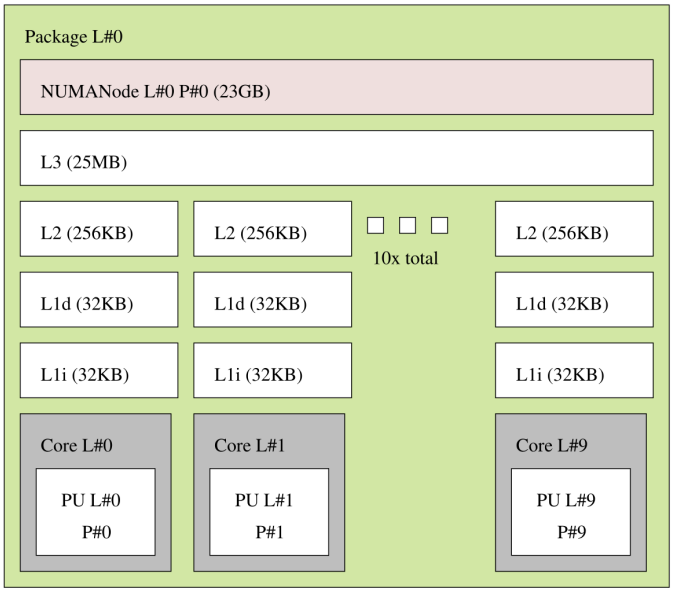

Figure 1: Schematic of a Rosa login node with an Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz.

the sustainable memory bandwidth and the corresponding computation rate for simple vector kernels (see, e.g, [3, 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.

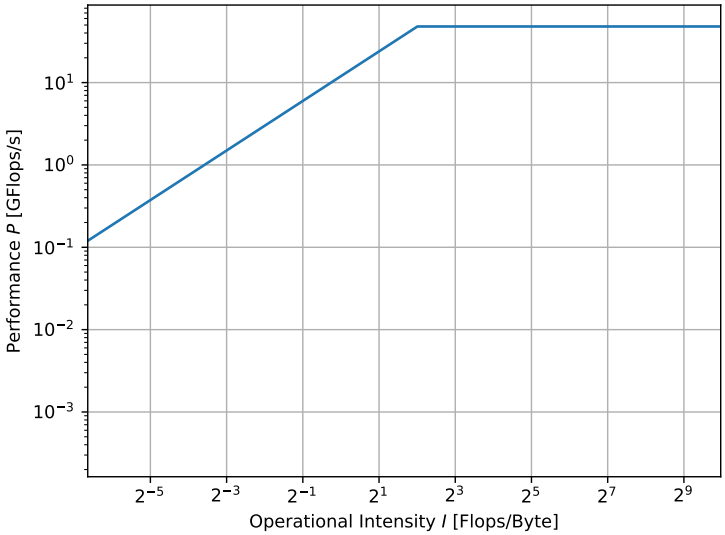

Figure 2: Simple (or naive) roofline model for a single core of a hypothetical CPU.

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 [10]. 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.

3. Matrix multiplication optimization[4] [50 points]

The first part of this project has introduced you how to use the Rosa cluster and you collected some performance characteristics in the second part. Your next task in this project 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 Rosa processors. Write an optimized single-threaded matrix multiply kernel. This will run on only one core.

3.1. Motivation

Linear algebra is a cornerstone operation in computational science and holds a particularly pivotal role in HPC. In particular, we are going to focus on matrix multiplication. This mathematical operation underpins a vast array of scientific and engineering disciplines, facilitating the modeling of complex systems, simulations of physical phenomena, and the processing of large datasets. In HPC environments, the efficiency and scalability of matrix multiplication directly influence the performance of algorithms (used in computational sciences, engineering, machine learning, and beyond). This functieonality is offered by BLAS (Basic Linear Algebra Subprograms), a specification encompassing a suite of low-level routines designed for executing common linear algebra operations efficiently.
Each manufacturer typically supplies a BLAS library (e.g., Intel's MKL library, AMD's AOCL, ...), carefully hand-optimized for its machines. It includes a large number of routines, not just matrix multiplication, although matrix multiplication is one of the most important, because it is used as a benchmark to compare the speed of computers (e.g., LINPACK Benchmark[5] used in the Top 500[6] list of the fastest supercomputers). In addition to manufacturer supplied libraries, there are also a number of open source projects. For instance, there are the ATLAS[7], GotoBLAS[8], OpenBLAS[9], and BLIS[10] (the 2023 recipient of the James H. Wilkinson Prize for Numerical Software). The list is not meant to be exhaustive.
The performance difference between naive and library implementations can be drastic. Figure 3 shows the performance in GFlops/s of n-by-n matrix multiplication (on a single core of an AMD EPYC 7763 CPU) as a function of matrix size n. It compares the highly optimized Intel MKL and OpenBLAS libraries with a naive implementation. As apparent from the figure, the naive implementation is sub-optimal (to say the least). As it turns out, matrix multiplication, commonly referred to as `DGEMM` (Double precision GEneral Matrix Multiplication), has a growing operational intensity (with the matrix size n). Therefore it can be (asymptotically) pushed into the compute bound regime (e.g., roofline model). However, achieving this requires specialized, hardware-dependent optimization techniques. The goal of this project task is to familiarize you with some basic techniques and offer pointers to literature for more advanced techniques.
However, achieving this requires very specialized, hardware dependent, optimization techniques. It is the goal of this project task to familiarize you with some basic techniques and provide you some pointers to the literature for more advanced techniques in the next paragraph.
The anatomy of a HPC matrix multiplication is described in the article [2]. The BLIS framework is described by Zee et al. [11] and Low et al. [5] propose an analytical performance model to determine certain parameters. The latter article gives a nice overview of the anatomy of a fast `DGEMM`. The recent article by Alaejos et al. [1] provides templates to write micro-kernels using vector intrinsics. Last but not least, we highly recommend the very comprehensive

[4] This task is based on a previous HPC Lab for CSE at ETH Zurich by Prof. Olaf Schenk from the Institute of Computing at the Faculty of Informatics at Universita della Svizzera Italiana (USI), which was itself originally based on a tutorial from Prof. Katherine A. Yelick from the Computer Science Department at the University of Berkeley (`http://www.cs.berkeley.edu/~yelick/`).
[5] `https://www.top500.org/project/linpack/`
[6] `https://www.top500.org/`
[7] `https://math-atlas.sourceforge.net/`
[8] `https://en.wikipedia.org/wiki/GotoBLAS`
[9] `https://www.openblas.net/`
[10] `https://github.com/flame/blis`

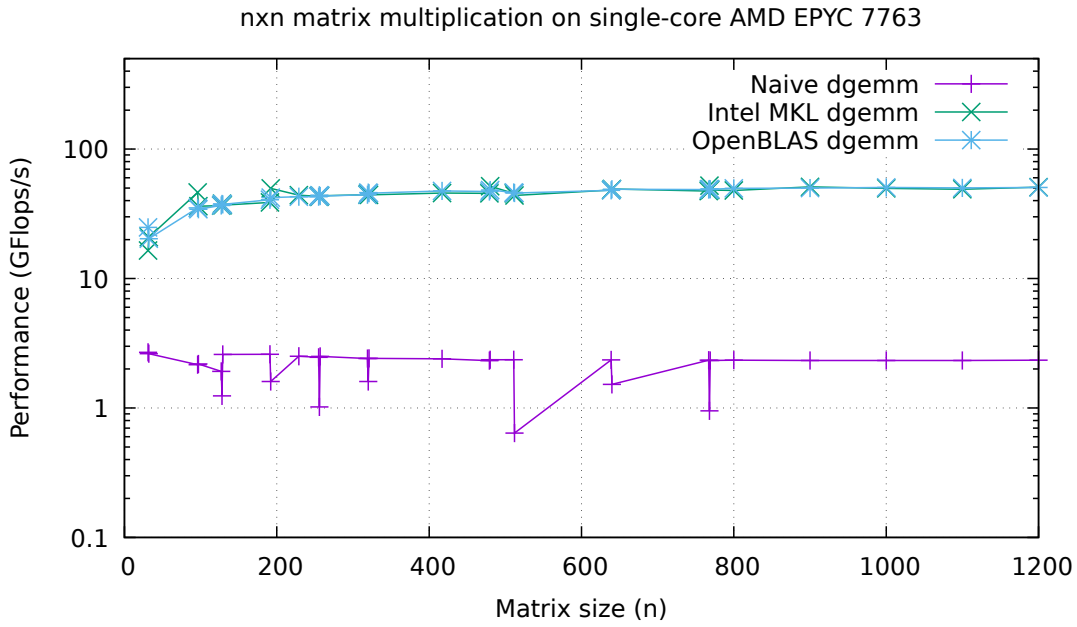

Figure 3: Serial performance of matrix multiplication on the AMD EPYC 7763.

online course "LAFF-On Programming for High Performance" [9]. On a less scientific level, you might also find the newspaper article [6] interesting.

3.2. Importance of data access optimization

Among the various factors that can limit performance in HPC, data access stands out as the most critical. This is due to the inherent imbalance in modern CPUs, where there is a significant discrepancy between their theoretical peak performance and the available memory bandwidth. Therefore, any attempt at optimization should initially focus on minimizing data traffic, or, if that proves to be impractical, strive to make data transfer as efficient as possible. If you haven't already, we strongly recommend to read Chapter 3 of the book [3].

3.2.1. Modeling memory hierarchies

Real memory hierarchies are very complicated, so modeling them carefully enough to predict the performance of an algorithm is hard. To make progress, we will use the following simplified model:

1. There are only **two levels** in the memory hierarchy, **fast** (e.g., registers, caches) and **slow** (e.g., main memory).
2. All the input data starts in the slow level, and the output must eventually be written back to the slow level.
3. The size of the fast memory is M, which is large but much smaller than the slow memory. In particular, the input of large problems will not fit in the fast memory simultaneously.
4. The programmer explicitly controls which data moves between the two memory levels and when.
   Remark: It might seem incorrect to suggest that programmers can control data movement between main memory and cache, as this process is typically automated by the hardware. However, we know how the hardware works: it moves data to the cache precisely when the user first tries to load it into a register to perform arithmetic, and puts it back in main memory when the cache is too full to get the next requested word. This means that we can write programs as if we controlled the cache explicitly by doing arithmetic operations in different

orders. Thus, two programs that do the same arithmetic in different orders may run at very different speeds, because one reduces data movement between the main memory and cache.

5. Arithmetic (and logic) can only be done on data residing in the fast memory and data movement and computation cannot overlap. Each arithmetic operation takes time t_a. Moving a word of data from one memory to the other takes time $t_m \gg t_a$. Hence, if a program performs m memory moves and a arithmetic operations, the total time it takes is $T = a \cdot t_a + m \cdot t_m$, where it may be that $m \cdot t_m \gg a \cdot t_a$.

With this model, let us get a **lower bound** on the speed of any algorithm for a problem that has m_i inputs and m_o outputs, and does a arithmetic operations. The only difference between algorithms that we permit is the order in which the arithmetic operations are executed. According to our model, the run-time taken will be at least

$$T = a \cdot t_a + (m_i + m_o) \cdot t_m,$$

no matter which clever order we do the arithmetic in.
For example, suppose the problem is to take two input arrays of n numbers each, $a[i]$ and $b[i]$ for $i = 1, .., n$, and produce two output arrays s and p, where $s[i] = a[i] + b[i]$ and $p[i] = a[i] \cdot b[i]$. Then the number of arithmetic operations is $a = 2n$, and the number of memory moves is $m_i = 2n$ and $m_o = 2n$. Thus, a lower bound on the run-time for any algorithm for this problem is $T = 2nt_a + 4nt_m$. Let us look at two different algorithms for this problem, and analyze which will be faster. The two algorithms are listed in Table 2. According to our model, we assume that a and b are initially in the slow memory, and $M \ll n$, so a and b cannot fit in the fast memory, and we have indicated when loads from and stores to slow memory occur. The point is that Algorithm 1 loads a and b twice,

Algorithm 1

```
for i = 1, n
  Load a[i], b[i] into fast memory
  s[i] = a[i] + b[i]
  Store s[i] into slow memory
end for

for i = 1, n
  Load a[i], b[i] into fast memory
  p[i] = a[i] * b[i]
  Store p[i] into slow memory
end for
```

Algorithm 2

```
for i = 1, n
  Load a[i], b[i] into fast memory
  s[i] = a[i] + b[i]
  Store s[i] into slow memory
  p[i] = a[i] * b[i]
  Store p[i] into slow memory
end for
```

Table 2: Two algorithms to compute the sum and product of the arrays.

whereas Algorithm 2 only loads them once (remember, there is not enough room in the fast memory to keep all the $a[i]$ and $b[i]$ for a second pass to compute $p[i]$). As per our simple model, Algorithm 1 takes run-time $T_1 = 2nt_a + 6nt_m$, whereas Algorithm 2 takes only $T_2 = 2nt_a + 4nt_m$, the minimum possible. Thus, for $t_a \ll t_m$, Algorithm 1 takes 50% longer to run than Algorithm 2. In the case of matrix multiplication, we will observe an even more dramatic difference.

3.2.2. Minimizing slow memory access in matrix multiplication

We will only consider different ways to implement n-by-n matrix multiplication $C = C + A \cdot B$ using $2n^3$ operations (i.e., we will not consider Strassen's method [8] and variants thereof). Thus, the only difference between algorithms will depend on the number of loads and stores to the slow memory. We also assume that the slow memory is large

enough to contain our three n-by-n matrices A, B and C, but the fast memory is too small for this. Otherwise, if the fast memory were large enough to contain A, B, and C simultaneously, then our algorithm would simply be:

```
Load A, B and C from slow into fast memory

method dgemm_fastmem(A, B, C)

  Compute $C = C + A \cdot B$ entirely in fast memory

end method dgemm_fastmem

Store the result $C$ back to slow memory
```

The number of slow memory accesses for this algorithm is $4n^2$ ($m_i = 3n^2$ loads of A, B, and C into fast memory, and $m_o = n^2$ stores of C to slow memory), yielding a run-time of $T = 2n^3 t_a + 4n^2 t_m$. Clearly, no algorithm doing $2n^3$ arithmetic operations can be faster. At the extreme where the fast memory is very small (M = 1), then there will be at least 1 memory reference per operand for each arithmetic operation involving entries of A and B, for a run-time of at least $T = 2n^3 t_a + (2n^3 + 2n^2)t_m$.

So, when the size of fast memory M satisfies $1 \ll M \ll 3n^2$, what is the fastest algorithm?

This is the practical question for large matrices, since real caches have thousands of entries. As we just saw, the worst case run-time is $2n^3 t_a + (2n^3 + 2n^2)t_m$, and the best we can hope for is $2n^3 t_a + 4n^2 t_m$, which would be almost $n/2$ times faster for $t_m \gg t_a$.
We begin by analyzing the simplest matrix multiplication algorithm, which we repeat below, including descriptions of when data moves between the slow and the fast memories. Remember that A, B, and C all start in the slow memory, and that the result C must be finally stored in the slow memory.

```
Naive matrix multiplication $C = C + A \cdot B$

method dgemm_naive(A, B, C)

  for i = 1, n
    for j = 1, n
      Load c_{i,j} into fast memory
      for k = 1, n
        Load a_{i,k} into fast memory
        Load b_{k,j} into fast memory
        c_{i,j} = c_{i,j} + a_{i,k} * b_{k,j}
      end for
      Store c_{i,j} into slow memory
    end for
  end for

end method dgemm_naive
```

Let m_{naive} denote the number of **slow memory references** in this naive algorithm. Then

$$\begin{aligned} m_{\text{naive}} &= n^3 && \dots \text{ for loading each entry of A } n \text{ times} \\ &+ n^3 && \dots \text{ for loading each entry of B } n \text{ times} \\ &+ 2n^2 && \dots \text{ for loading and storing each entry of C once} \\ &= 2n^3 + 2n^2, \end{aligned}$$

or about as many slow memory references as arithmetic operations. Thus, the run-time is $T_{\text{naive}} = 2n^3 t_a + (2n^3 + 2n^2)t_m$, the worst possible.

Let us analyse an improved algorithm called **blocked matrix multiplication** (sometimes it is called **tiled** or **panelled** instead of **blocked**). The n-by-n matrix A is divided into smaller s-by-s sub-matrices or blocks A_{ij}, where s is a parameter called the block size that we will specify later. We assume s divides n for simplicity. Matrices B and C are similarly partitioned. Then, we can think of A as an n/s-by-n/s block matrix, where each entry $A_{i,j}$ is an s-by-s block. The inner loop $C_{i,j} = C_{i,j} + A_{i,k} \cdot B_{k,j}$ now runs for $k = 1$ to s, and represents an s-by-s matrix multiplication and addition. The algorithm becomes:

```
1  Blocked matrix multiplication $C = C + A \cdot B$
2
3  method dgemm_blocked(A, B, C)
4
5    for i = 1, n/s
6      for j = 1, n/s
7        Load C_{i,j} block into fast memory
8        for k = 1, n/s
9          Load A_{i,k} block into fast memory
10         Load B_{k,j} block into fast memory
11         dgemm_fastmem(A_{i,k}, B_{k,j}, C_{i,j})
12       end for
13       Store C_{i,j} into slow memory
14     end for
15   end for
16
17 end method dgemm_blocked
```

The inner loop `dgemm_fastmem`$(A_{i,k}, B_{k,j}, C_{i,j})$ has all its data residing in the fast memory, and so causes no slow memory traffic at all. Redoing the count of slow memory references yields

$$\begin{aligned} m_{\text{blocked}} = m_{\text{blocked}}(s) &= (n/s)^3 \cdot s^2 && \dots \text{ for loading each block } A_{i,k} \ (n/s)^3 \text{ times} \\ &+ (n/s)^3 \cdot s^2 && \dots \text{ for loading each block } B_{k,j} \ (n/s)^3 \text{ times} \\ &+ 2(n/s)^2 \cdot s^2 && \dots \text{ for loading and storing each block } C_{i,j} \text{ once} \\ &= 2n^3/s + 2n^2. \end{aligned}$$

Comparing $m_{\text{naive}} = 2n^3 + 2n^2$ and $m_{\text{blocked}} = 2n^3/s + 2n^2$, it is obvious that we want to pick s as large as possible to make $m_{\text{blocked}}(s)$ as small as possible. But how big can we pick s? The largest possible value is obviously $s = n$, which corresponds to loading all of A, B and C into the fast memory, which we cannot do. So, s depends on the size M of the fast memory, and the constraint it must satisfy is that the three s-by-s blocks $A_{i,k}, B_{k,j}$, and $C_{i,j}$ must simultaneously fit in the fast memory, which implies $3s^2 \leq M \implies s \leq \sqrt{M/3}$. Therefore, the largest value

$s = \sqrt{M/3}$ yields

$$m_{\text{blocked}}(\sqrt{M/3}) = \sqrt{12}\frac{n^3}{\sqrt{M}} + 2n^2.$$

In other words, for large matrices (large n) we decrease the number of slow memory references, the most expensive operation, by a factor $\mathcal{O}(M)$. This is attractive, because it says that cache (fast memory) helps, and the larger the cache the better.
In summary, the running time for this algorithm is

$$T_{\text{blocked}} = 2n^3 \cdot t_a + (\sqrt{12}\frac{n^3}{\sqrt{M}} \cdot t_m).$$

There is a theorem, which we will not prove, that says that up to constant factors, we cannot do fewer slow memory references than this (while doing the usual $2n^3$ arithmetic operations):

Theorem (Hong + Kung, 1981, 13th Symposium on the Theory of Computing [4]): Any implementation of matrix multiplication using $2n^3$ arithmetic operations performs at least $\mathcal{O}(n^3/\sqrt{M})$ slow memory references.

In practice, this technique is very important to get matrix multiplication to run as fast as possible. But careful attention must also be paid to other details of the instruction set, arithmetic units, and so on. If there are more levels of memory hierarchy (two levels of cache), then one might use this technique recursively, dividing s-by-s blocks into yet smaller blocks to exploit the next level of memory hierarchy. However, this goes beyond the scope of this project and course.

3.3. Optimizing square matrix multiplication

Your task is now to write a blocked matrix multiplication (as outlined above) and optimize it for a single core on the Rosa Cluster.

Skeleton code

Download the skeleton code provided on the iCorsi page. It contains the following:

- `dgemm-naive.c` – For illustrative purposes, a naive implementation of matrix multiplication using three nested loops.
- `dgemm-blas.c` – A wrapper which calls an optimized BLAS implementation of matrix multiplication (OpenBLAS).
- `dgemm-blocked.c` – A skeleton for a blocked implementation of matrix multiplication. It is your task to implement and optimize the `square_dgemm` function in this file.
- `benchmark.c` – A driver program that generates matrices of a number of different sizes and benchmarks the performance. It outputs the performance in GFlops/s and in a percentage of theoretical peak performance attained. You should not need to modify this file, except perhaps to change the peak performance constant `MAX_SPEED` if you wish to test on another computer.
- `Makefile` – A simple makefile to build the executables for the benchmarking of the naive, optimized BLAS and your blocked implementation.
- `run_matrixmult.sh` – A job script that executes all three executables and produces log files (`*.data`) that contain the performance logs. It also plots the data in the performance logs and produces `timing.pdf` showing the results.

Familiarize yourself with the code and the used conventions. In particular, note that we use the *column major order* storage scheme to conform with BLAS' Fortran origins.

Running the code

The skeleton code should run out of the box and a file listing should look as:

```
[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
```

We will use the GNU Compiler Collection and the OpenBLAS implementation, which can be loaded on Rosa as follows:

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

Building the code

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

generates the three executables `benchmark-naive`, `benchmark-blas` and `benchmark-blocked`. 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
Submitted batch job 49100417
```

or

```
[user@icslogin]$ sbatch --reservation=hpc-wednesday run_matrixmult.sh
Submitted batch job 49100415
```

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 PART  NAME      ST  START_TIME END_TIME TIME_LEFT  NODES
 1173224  user slim  matrixmu  R   14:47:17   15:17:17 29:50      1
```

When our job is finished, you will find new files in the directory containing the output of the benchmark programs. For example, we will find the files `matrixmult-jobid.out` and `matrixmult-jobid.err`. The first file contains the standard output and the second file contains the standard error. Additionally, the performance data are stored in `*.data` files and `timing.pdf` is a plot of the performance.

Optimizing square matrix multiplication [50 points]

- **Implementation**: Implement blocking for square matrix multiplication in `dgemm_blocked.c`.
- **Optimization**: Optimize your code to maximize its performance. Consider various strategies, such as compiler options, tuning data access patterns (including block size adjustments), using `#pragma` directives, vectorization, loop unrolling, and more.
- **Documentation and Analysis**: Document the used or attempted optimizations in your report, including performance graphs and, if applicable, tables. Provide a detailed description of your experimental setup. Compare your optimized implementation to the OpenBLAS library.
- Employ any advanced techniques from library implementations as described in the references provided in the motivation 3.1, including any references therein or other relevant resource. However, remember to accurately cite all your sources.

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 .

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] Guillermo Alaejos, Adrian Castello, Hector Martinez, Pedro Alonso-Jorda, Francisco D. Igual, and Enrique S. Quintana-Orti. Micro-kernels for portable and efficient matrix multiplication in deep learning. *The Journal of Supercomputing*, 79(7):8124–8147, May 2023. doi:10.1007/s11227-022-05003-3.

[2] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):1–25, May 2008. doi:10.1145/1356052.1356053.

[3] Georg Hager and Gerhard Wellein. Introduction to high performance computing for scientists and engineers. *Chapman & Hall/CRC Computational Science*, July 2010. URL: http://dx.doi.org/10.1201/EBK1439811924, doi:10.1201/ebk1439811924.

[4] Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 326–333, New York, NY, USA, May 1981. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/800076.802486, doi:10.1145/800076.802486.

[5] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Transactions on Mathematical Software*, 43(2):12:1–12:18, August 2016. URL: https://dl.acm.org/doi/10.1145/2925987, doi:10.1145/2925987.

[6] John Markoff. Writing the Fastest Code, by Hand, for Fun: A Human Computer Keeps Speeding Up Chips. *The New York Times*, 2005. [Online; accessed 24-February-2024]. URL: https://www.nytimes.com/2005/11/28/technology/writing-the-fastest-code-by-hand-for-fun-a-human-computer-keeps.html?smid=url-share.

[7] 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. URL: https://www.cs.virginia.edu/~mccalpin/papers/balance/.

[8] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, aug 1969. doi:10.1007/bf02165411.

[9] Robert van de Geijn, Margaret Myers, and Devangi Parikh. LAFF - On Programming for High Performance. https://www.cs.utexas.edu/users/flame/laff/pfhp/, November 2021. [Online; accessed 24-February-2024].

[10] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65, April 2009. doi:10.1145/1498765.1498785.

[11] Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. Van De Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. The BLIS framework. *ACM Transactions on Mathematical Software*, 42(2):1–19, jun 2016. doi:10.1145/2755561.