

Università della Svizzera italiana	Institute of Computing CI

High-Performance Computing Lab

Institute of Computing

Student: INNOCENT KISOKA

Discussed with: -

Solution for Project 1

HPC Lab — Submission Instructions
(Please, notice that following instructions are mandatory:
submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
Project_number_lastname_firstname
and the file must be called:
project_number_lastname_firstname.zip
project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

In this project you will practice memory access optimization, performance-oriented programming, and OpenMP parallelization on the Rosa Cluster .

1. Rosa Warm-Up

(5 Points)

1.1. What is the module system and how do you use it?

The module system is a tool for managing software environments in a Linux-based High-Performance Computing cluster. It allows users to dynamically load and unload software packages, libraries, and compilers in their current session.[1]

1.1.1. How to use the module system?

- To list available modules: `module avail`
- To load a module : `module load gcc`
- To unload a module: `module unload gcc`
- To see which modules are currently loaded: `module list`
- To purge all loaded modules: `module purge`

1.2. What is Slurm and its intended function?

Slurm stands for Simple Linux Utility for Resource Management. It is a job scheduling system for large and small Linux clusters that manages the resources and allocates them to users for executing jobs in a queuing system.

1.3. Write a simple “Hello World” C/C++ program which prints the host name of the machine on which the program is running. function with some appropriate size.

Hello World C++ Program code that prints the hostname of the machine on which it is running

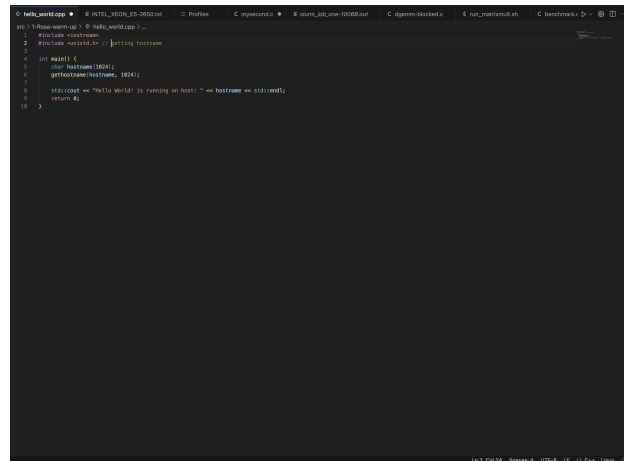
A screenshot of a code editor showing a C++ program. The program includes the <iostream> and <unistd.h> headers, uses the namespace std, and defines a main function. Inside main, it calls gethostname to get the host name and then prints it using cout. The status bar at the bottom indicates the file is 'hello_world.cpp' and the editor is 'VS Code'.

Figure 1: Hello World Program

1.4. 5. Write a batch script which runs your program on one node. The batch script should be able to target nodes with specific CPUs.

The following batch script can be used to run the Hello World program on a single node while targeting specific CPUs.

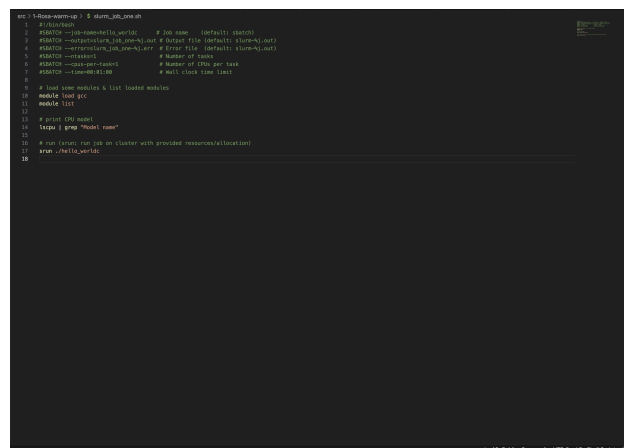
A screenshot of a Slurm batch script. The script starts with a shebang line and a comment. It then sets the job name to 'hello_world' and the output file to 'hello_world.out'. It specifies the number of nodes as 1 and the number of CPUs per node as 1. It then sets the module load path to '/usr/local/lib64' and the module list to 'gcc'. Finally, it runs the program 'hello_world' and prints the output to the screen. The status bar at the bottom indicates the file is 'hello_world.slurm' and the editor is 'VS Code'.

Figure 2: Batch script on one node

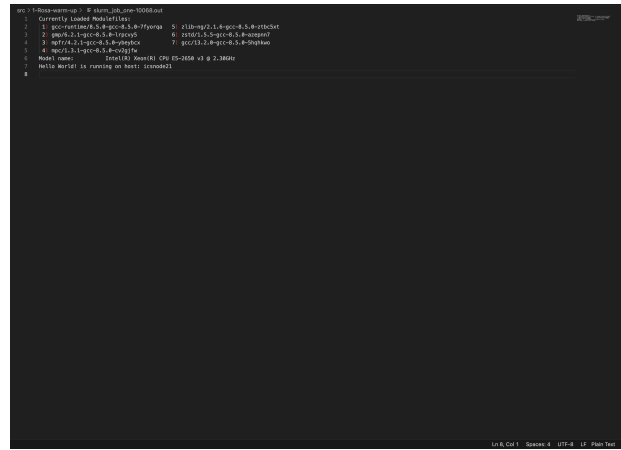


Figure 3: Slurm output on one node

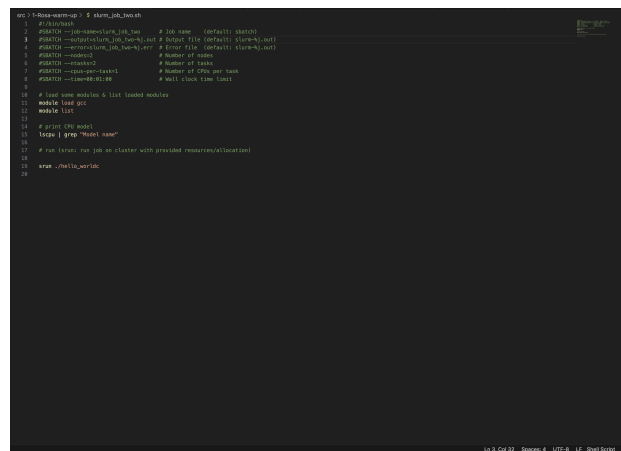


Figure 4: Batch script on two nodes

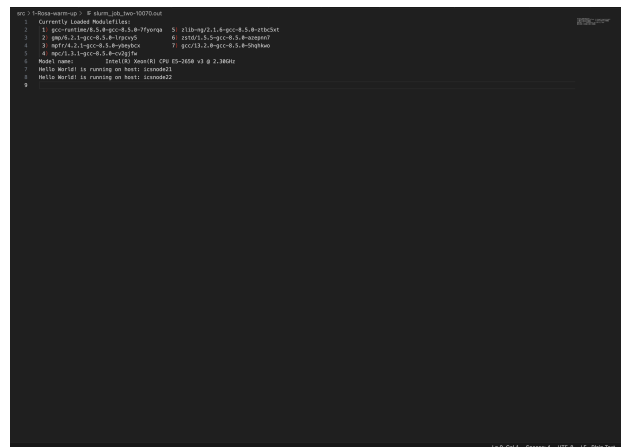


Figure 5: Slurm output on two nodes

2. Performance Characteristics

(30 Points)

2.1. Peak performance

Peak performance is the metric that quantifies the maximum rate at which a system can execute floating-point operations per second (FLOPS). It is calculated based on the hardware specifications of the CPU, including the number of cores, SIMD capability, and clock speed.[2]

- Superscalarity Factor (nSuper) is the number of floating-point operations that the floating-point unit of a core can execute in parallel within a single clock cycle.
- Fused Multiply-Add (nFMA) Factor is the ability of the to perform both a multiplication and an addition in a single instruction. this factor is 2 for that support .
- Single Instruction, Multiple Data (SIMD) Factor allows multiple data points to be processed concurrently in a single instruction.
- Clock Frequency (f) is the base clock speed of the CPU, measured in GHz that defines how fast the processor operates.

From the documentation I obtained that:

- nSuper = 2
- nFMA = 2
- SIMD factor = 4
- nCores = 10
- nSockets = 2
- nNodes = 42

Core Peak Performance (Pcore) is calculated by $nSuper \times nFMA \times nSIMD \times f = 2 \times 2 \times 4 \times 2.3 \text{ GHz} = 36.8 \text{ GFLOPS}$

CPU Peak Performance (PCPU) is calculated by $PCPU = nCores \times Pcore = 10 \times 36.8 \text{ GFLOPS} = 368 \text{ GFLOPS}$

Node Peak Performance (Pnode) is calculated by $Pnode = nSockets \times PCPU = 2 \times 368 \text{ GFLOPS} = 736 \text{ GFLOPS}$

Cluster Peak Performance (Pcluster) is calculated by

$Pcluster = nnodes \times Pnode = 42 \times 736 \text{ GFLOPS} = 30912 \text{ GFLOPS} = 30.912 \text{ TFLOPS}$

2.2. Memory Hierarchies

The key components of the memory hierarchy in the Rosa node are:

The cache memory

L1 Cache:

- L1i (Instruction Cache): Each core has its own 32 KB L1 instruction
- L1d (Data Cache): Each core also has its own 32 KB L1 data cache.

L2 Cache: Each core has a private 256 KB L2 cache.

L3 Cache: The 25 MB L3 cache is shared among all 10 cores within a package (socket).

2.2.1. Main Memory

Each node has a total of 55 GB of main memory, with 23 GB allocated to NUMA Node L0 (Socket 0) and 31 GB allocated to NUMA Node L1 (Socket 1).[3]

2.2.2. Memory Sharing Between Cores

As observed from the diagram:

L1 and L2 caches are private to each core. Each of the 20 cores in the node has its own L1i, L1d, and L2 cache. L3 cache and main memory are shared among the cores. Each package (socket) has 10 cores sharing a 25 MB L3 cache, and each socket has a shared portion of the system's main memory (23 GB for Socket 0 and 31 GB for Socket 1).

2.2.3. Graphical Representation

The provided figure 6 clearly illustrates the hierarchical structure of the memory, showing the shared L3 cache and main memory, and the private caches at the L1 and L2 levels. This figure helps in visualizing the memory distribution across different NUMA nodes and cores.

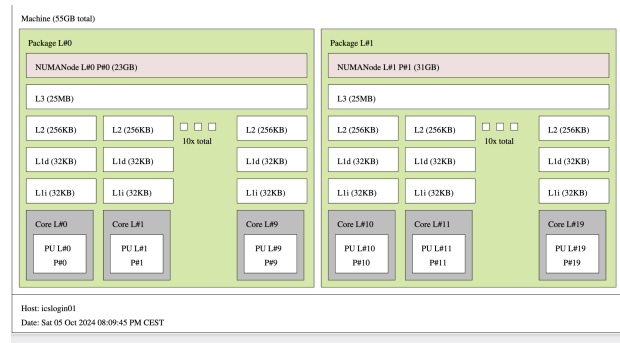


Figure 6: Memory hierarchy

2.3. Bandwidth: STREAM benchmark

The STREAM benchmark is a well-known tool used to measure memory bandwidth on various systems. It runs four main operations (kernels) which are Copy, Scale, Add, and Triad. In order to evaluate memory performance. Here is a summary of my execution of the STREAM benchmark on a single core of the Rosa system, alongside the modifications I made.

2.3.1. Modifications to mysecond.c

I made changes to the mysecond.c file to use the gettimeofday() function for more precise time measurement, but removed the timezone structure as it has been deprecated and is no longer in common use. The updated function is as follows:

```
src > 2-Performance-characteristics > 03 > C mysecond.c > get_current_time_in_seconds()
1  #include <stdio.h>
2  #include <sys/time.h>
3
4  double get_current_time_in_seconds() {
5      struct timeval tp;
6
7      int i;
8
9      //current time
10     i = gettimeofday(&tp, NULL);
11
12     // Error check for gettimeofday()
13     if (i != 0) {
14         printf("Error: gettimeofday failed\n");
15         return -1.0;
16     }
17
18     // Return time in seconds (tv_sec) and microseconds (tv_usec)
19     return (double)tp.tv_sec + (double)tp.tv_usec * 1e-6;
20 }
21
22 int main() {
23     double current_time = get_current_time_in_seconds();
24     printf("Current time: %f seconds\n", current_time);
25     return 0;
26 }
27
28
29
```

Figure 7: modified mysecond.c code

2.3.2. Results

After running the STREAM benchmark on the Rosa node, the output was as follows:

- **Copy** operation had the highest bandwidth at **19199.6 MB/s**.
- **Scale, Add, and Triad** operations were consistent, with bandwidths around **11241-12302 MB/s**.

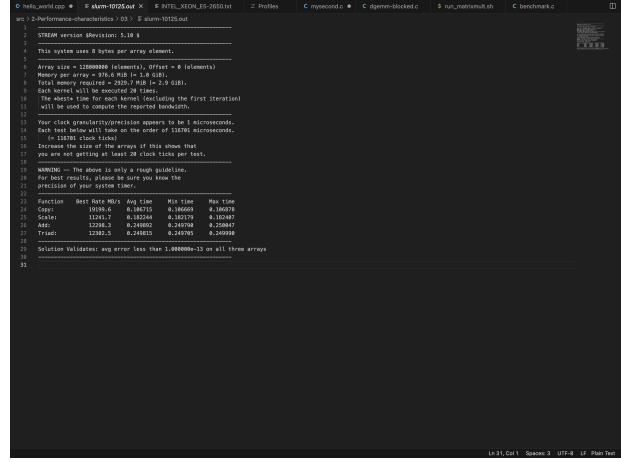


Figure 8: STREAM benchmark output

- The results show that the memory system on the Rosa node has excellent throughput, especially for simple data transfer tasks like Copy.
- The Scale, Add, and Triad operations exhibit consistent bandwidth rates, demonstrating that the memory system sustains a high level of performance across different types of computational tasks.

2.3.3. 2.4 Performance model: A simple roofline model

The roofline model consists of two main regions:

1. **Memory-Bound Region:** This is defined by the available memory bandwidth of the system, with performance limited by how fast data can be moved to and from memory.
2. **Compute-Bound Region:** This is where performance is limited by the computational power of the processor, capped at the theoretical peak performance.

The point where these two regions intersect is called the **ridge point**.^[4]

The roofline model for a single core of the Rosa nodes is visualized in the plot below above.

- **Y-axis (Performance in GFLOP/s):** This axis represents the attainable performance in floating-point operations per second.
- **X-axis (Operational Intensity in Flops/Byte):** This axis represents the number of floating-point operations performed per byte of data moved.

The plot shows:

- A sloped line in the memory-bound region where performance increases linearly with operational intensity.

- A horizontal line in the compute-bound region, capped at the theoretical peak performance of **36.8 GFLOPs/s**.

At an operational intensity of approximately **1.92 Flops/Byte** where there is a transition between the memory-bound and compute-bound regions.

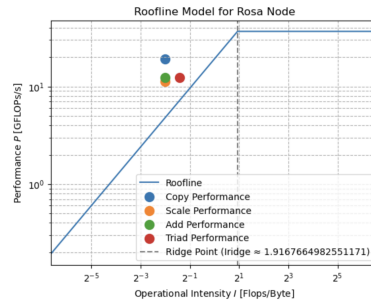


Figure 9: Roofline model

3. Optimize Square Matrix-Matrix Multiplication (50 Points)

3.1. Optimizations Used or Attempted

The main optimization introduced was **blocking** in the `dgemm-blocked.c` code. The purpose of blocking is to better utilize cache by dividing the matrix into smaller sub-blocks, ensuring that frequently used elements remain in the cache for as long as possible during multiplication. This reduces cache misses and improves memory locality, particularly when working with large matrices that don't fit in cache.[5]

Tuning the block size was another optimization strategy. The goal was to find an optimal block size that maximizes performance without creating too much overhead from small block computations.

3.2. Results of the Optimizations

As seen from the performance graph, the **blocked DGEMM** implementation performed noticeably better than the **naive DGEMM**. The **blocked DGEMM** approach resulted in more stable performance as matrix size increased, especially for matrices larger than 400 x 400. However, the performance was still far behind the **Reference BLAS DGEMM (Intel MKL)**, which maintained much higher performance across all matrix sizes.

Blocked DGEMM achieved up to around **2 GFLOPs/s**, while naive DGEMM was consistently below **1 GFLOPs/s**. On the other hand, the Reference BLAS DGEMM peaked at over **10 GFLOPs/s**.

3.3. Odd Behavior (Dips in Performance)

Several performance dips, especially in the blocked DGEMM line, can be explained by **cache contention** and **sub-optimal block sizes** for specific matrix dimensions. If the block size chosen is too small or doesn't align well with cache sizes, the blocked implementation can suffer from higher overhead due to repeated memory access for small sub-blocks.

3.4. Comments on the `dgemm-blocked.c` Implementation and Visualizations

The `dgemm-blocked.c` implementation, after incorporating blocking, showed clear performance improvements on both local machines and the cluster. The graph visualizing performance on a **single-core Intel Xeon E5-2650** shows that blocking provided a tangible benefit, particularly

for larger matrices. However, the performance still lagged behind the highly-optimized **Intel MKL** implementation.

On the local machine, blocked DGEMM performed similarly, but the cluster offered better absolute performance due to its architecture and larger cache size, which can handle blocked matrices more efficiently.

3.5. Performance Metrics:

- **Naive DGEMM** (purple line) shows lower performance overall, with fluctuations in the GFLOPs/s as matrix size increases.
- **Blocked DGEMM** (green line) performs better than the naive approach, with improved stability in performance as matrix size increases, especially for larger matrices.
- **BLAS DGEMM** (blue stars) clearly outperforms both the naive and blocked implementations across all matrix sizes, maintaining consistently higher GFLOPs/s.

3.5.1. Observations:

- The **Naive DGEMM** implementation struggles as the matrix size increases, never exceeding 1 GFLOPs/s.
- **Blocked DGEMM** offers significant improvements over the naive version, especially for matrix sizes of 800 or more.
- BLAS DGEMM maintains performance near or above 10 GFLOPs/s for most matrix sizes and scales well with increasing matrix size.

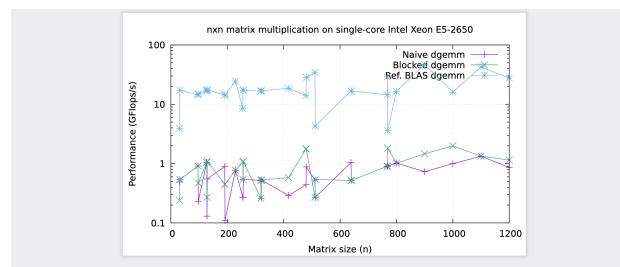


Figure 10: Timing

Therefore, the introduction of blocking in `dgemm-blocked.c` improved performance compared to the naive implementation but did not reach the efficiency of professional libraries like Intel MKL.

4. Quality of the Report

(15 Points)

References

- [1] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- [2] Intel® Xeon® Processor E5-2650 (20M Cache, 2.00 GHz, 8.00 GT/s Intel® QPI) - Product Specifications — Intel — intel.com. <https://www.intel.com/content/www/us/en/products/sku/64590/intel-xeon-processor-e52650-20m-cache-2-00-ghz-8-00-gts-intel-qpi/specifications.html>. [Accessed 09-10-2024].

- [3] Intel® 64 and IA-32 Architectures Optimization Reference Manual Volume 1 — intel.com. <https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html>. [Accessed 09-10-2024].
- [4] Samuel Williams, A Waterman, and D Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore. *ACM Communications*, page 16, 2009.
- [5] Kazushige Goto and Robert A van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):1–25, 2008.