Republic of Tunisia
Ministry of Higher Education, Scientific Research
and Information and Communication
Technologies
Sfax University
National Engineering School of Sfax

Computer Engineering & Applied Mathematics
Department

GRADUATION PROJECT
Serial N°: 2025 /
DIMA-045

GRADUATION PROJECT REPORT

presented at

National Engineering School of Sfax
(Computer Engineering & Applied Mathematics Department)

in order to obtain the

National Engineering Diploma in Computer Science

by

Mohamed Karim BEZINE

HPC Software Stack Build, Benchmarking and Automation

Defended on 13/09/2025 in front of the committee composed of

Mr. Riadh BEN HALIMA	President
Ms. Mariem LAHAMI	Reviewer
Mr. Bechir ZALILA	Academic supervisor
Ms. Rahma SMAOUI	Industrial supervisor

Dedication

*To My **Family**, My **Parents**, and My **Grandparents**,*
Your unwavering support and encouragement have been a constant source of strength. Thank you for believing in me and guiding me through every challenge.

*To My **Professors and Mentors**,*
I extend my deepest gratitude for your invaluable guidance and expertise. Your mentorship has profoundly shaped my academic and personal growth.

*To My **Friends**,*
Your steadfast support and encouragement have been instrumental throughout this journey. I deeply appreciate your friendship and loyalty.With sincere appreciation.

Mohamed Karim BEZINE

Acknowledgement

First and foremost, I thank God Almighty for granting me the strength, health, and determination to complete this work. To Him, I owe all that I have achieved.

I wish to express my deepest appreciation to my supervisor, **Mr. Bechir ZALILA**, for his unwavering patience, availability, and invaluable guidance. His expertise and willingness to engage with even the smallest details of this project transformed my ideas into coherent results.

Special recognition goes to **Ms. Rahma SMAOUI** and **Ms. Rahma KRICHEN**. Their practical insights, constant encouragement, and ability to bridge academic theory with real-world challenges were pivotal to the success of this work.

To the entire **ReDX** team: I will forever cherish the warm welcome you extended to me and the countless hours you spent sharing knowledge, troubleshooting obstacles, and offering wisdom that no classroom could replicate. Your generosity in helping me adapt to your work environment made this experience unforgettable.

I would also like to express my heartfelt gratitude to the **Toubkal** and **UM6P** team, especially **Mr. Robert BASMADJIAN**, **Mr. Othmane CHAKIR**, and **Mr. Charaf Eddine ZABAKH**. Your support, technical assistance, and dedication provided me with the resources and environment needed to successfully carry out my experiments. The collaboration and valuable exchanges with you enriched my understanding of high-performance computing and greatly contributed to the progress of this work.

Finally, to all my teachers at the National Engineering School of Sfax, the rigorous education and intellectual curiosity you instilled in me formed the foundation of this project. Your dedication to excellence continues to shape engineers who can tackle tomorrow's challenges.

Contents

List of Figures

List of Tables

List of Acronyms

AI Artificial Intelligence

API Application Programming Interface

CI/CD Continious Integration/Continious Deployment

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DevOps Development Operations

DRAM Dynamic Random Access Memory

DRMS Distributed Resource Management System

ELK Elasticsearch Logstash Kibana

ENIS National Engineering School of Sfax

FFTW Fastest Fourier Transform in the West

FLOPS Floating Point Operations Per Second

GDDR Graphics Double Data Rate

GPU Graphics Processing Unit

HBM High Bandwidth Memory

HDD Hard Disk Drive

HPC High Performance Computing

HPL High Performance Computing Linpack

Hz Hertz

MDS Metadata Server

MDT Metadata Target

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

MPI Message Passing Interface

NCCL NVIDIA Collective Communications Library

NUMA Non Uniform Memory Access

NVHPC NVIDIA HPC SDK

OpenACC Open Accelerators

OpenMP Open Multi-Processing

OSSs Object Storage Server

OST Object Storage Target

PCIe Peripheral Component Interconnect Express

QE Quantum ESPRESSO

RAID Redundant Array of Independent Disks

RAM Random Access Memory

SIMD Single Instruction Multiple Data

SISD Single Instruction Single Data

SM Streaming Multiprocessor

SRAM Video Random Access Memory

SRAM Shared Random Access Memory

SSD Solid State Drive

SSH Secure Shell

TOTP Time-based One-Time Password

VASP Vienna Ab initio Simulation Package

VM Virtual Machine

General introduction

High Performance Computing benchmarking is essential for evaluating the efficiency and capabilities of supercomputers and large-scale computing clusters. It involves running standardized workloads that simulate real scientific or industrial applications to assess how effectively a system handles demanding computational tasks such as simulations, data analytics, or machine learning.

HPC systems support modern scientific research, engineering simulations, and data-intensive applications. These systems are inherently complex and heterogeneous, spanning multiple layers from the physical infrastructure to the software stack provided to users. Even minor modifications at any layer can significantly affect system performance and stability.

Ensuring HPC reliability and efficiency requires validating system health after maintenance and continuously monitoring performance in production. Early detection of anomalies preserves service quality, maximizes productivity, and fosters innovation across disciplines.

This report presents a project conducted on the Toubkal Supercomputer, focusing on benchmarking, software environment setup, automation, and monitoring. The work involved compiling scientific applications from source, deploying them on the Toubkal cluster, executing performance benchmarks, and implementing automated workflows and dashboards for real-time monitoring and analysis. These efforts collectively contribute to maintaining Toubkal as a robust and high-performing platform for scientific research.

This report documents a project conducted on the **Toubkal Supercomputer**, focusing on benchmarking, software environment preparation, automation, and monitoring. The project involves building scientific applications from source, executing performance benchmarks, and setting up automated workflows and dashboards for real-time system monitoring and analysis.

The report is organized as follows:

- **Chapter 1: Project Foundations** — Introduces the context, problem statement, motivation, and methodology.
- **Chapter 2: Theoritcal Background** — Summarizes key HPC concepts, GPU architecture, programming models, and the main components of HPC clusters.
- **Chapter 3: Toubkal Supercomputer Architecture & HPC Software Stack Build** — Details the whole architecture of the working environment as well as the whole build and deployment phases of scientific applications.
- **Chapter 4: HPC Benchmarking** — Presents the state-of-the-art benchmarks used to evaluate Toubkal's performance.
- **Chapter 5: HPC Automation & Monitoring** — Discusses the automation of test execution using Jenkins and ReFrame, as well as visualization and real-time monitoring using Kibana, Prometheus, and Grafana.

Chapter

1 Project Foundations

1.1 *Introduction*

In this chapter, we establish the foundational elements of our project. We begin by presenting the general context that motivates our work and highlights its relevance. Next, we introduce the host organization, emphasizing its role and the environment in which the project took place. We then provide an overview of the project, outlining the problem we aimed to solve and the solution we proposed. Lastly, we describe the project management methodology we adopted to plan, organize, and carry out the different phases of the project.

1.2 *General Context*

High Performance Computing plays a vital role in solving complex scientific, engineering, and data-intensive problems. By leveraging the massive parallelism of modern supercomputers, researchers can perform simulations, data analyses, and computations that are otherwise impractical. Over the past decades, HPC systems have evolved significantly in scale and complexity, integrating thousands of compute cores, high-throughput interconnects, and scalable storage solutions [1].

As these systems grow in complexity and heterogeneity, ensuring consistent and efficient performance has become a major challenge. Bottlenecks can occur at any level (computation,

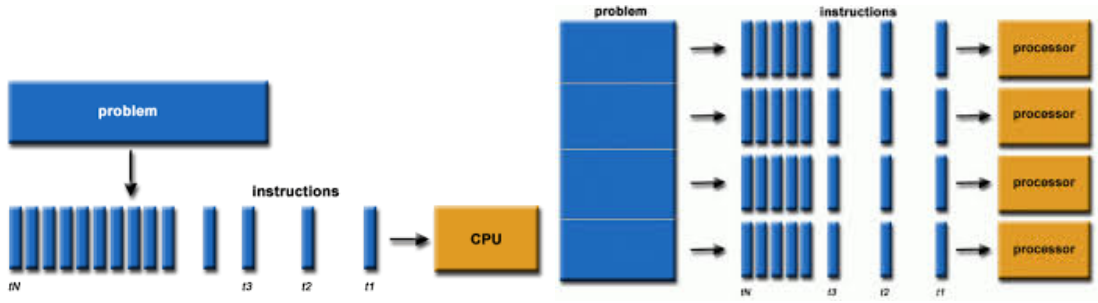
memory, interconnects, storage, ...) making performance monitoring and tuning indispensable. Benchmarking and observability techniques are therefore essential to validate system behavior, detect regressions, and guide infrastructure improvements.

1.2.1 High Performance Computing

High-Performance Computing relies on the parallel execution of demanding computational tasks. In such systems, large workloads are broken down into smaller units that are distributed across multiple resources to be processed at the same time. Thanks to this parallelism, HPC clusters are able to handle extensive computations more quickly and efficiently than conventional computing models.

1.2.2 Parallel Programming

Parallel computing, also known as parallel programming, involves dividing complex computational problems into smaller sub-problems that are processed at the same time by multiple processors. These processors exchange information through shared memory, and their partial results are merged using appropriate algorithms. Compared to serial computing, where a single processor executes tasks one after another, parallel computing offers a substantial gain in speed and efficiency. Figure 1.1 below highlight the distinction between these two approaches.



(a) *Serial* **(b)** *Parallel*

Figure 1.1. *Serial vs. Parallel Programming [2]*

1.2.3 Importance of Parallel Programming

High performance parallel computers have become essential tools across nearly all scientific and industrial fields. They are particularly necessary when addressing problems that are too complex, costly, or risky to tackle directly, for example, in weather prediction, pharmaceutical research, or simulations of aircraft accidents.

Modern real-world challenges are often massive in both scale and complexity. In computational simulations, the number of particles involved can be extremely large, and running the models may take months or even years. Weather forecasting illustrates this well: the atmosphere is divided into a three-dimensional grid, where each grid point represents local conditions such as temperature, pressure, humidity, and wind properties. The evolution of these points is governed by physical and fluid dynamics equations.

Two main factors contribute to the complexity of this task. First, improving accuracy requires refining the grid, which drastically increases the number of equations to be solved. Second, since each grid point's state depends on its neighbors, the system leads to highly nonlinear partial differential equations that can only be handled efficiently by supercomputers.

1.2.4 Challenges Faced in HPC systems

HPC environments are inherently complex, combining diverse hardware configurations, evolving software stacks, and demanding scientific workloads. Maintaining optimal performance, reliability, and usability in such settings is a persistent challenge. Over time, system performance may degrade due to hardware aging, inefficient resource utilization, or misconfigurations. Frequent software or hardware updates can also introduce regressions or incompatibilities. Without proactive monitoring and validation, these issues often go undetected until they significantly impact research productivity.

Another challenge lies in managing software environments. HPC systems are typically shared by many users, each with different project requirements that may depend on specific tools, versions, or configurations. Some applications require the latest software versions, while others depend on older, well-tested releases. Without a rigorous system to manage

this complexity, software installations risk becoming inconsistent, disrupting other users' workflows, or introducing errors during execution.

1.2.5 CPU Overview

The CPU is the core element of a computer, often described as its "control center." Also known as the main or central processor, it consists of complex electronic circuits responsible for running the operating system and applications. Its role is to decode, process, and execute instructions coming from both hardware and software.

In practice, the CPU carries out arithmetic and logical operations, along with other tasks, to convert raw input data into meaningful output information.

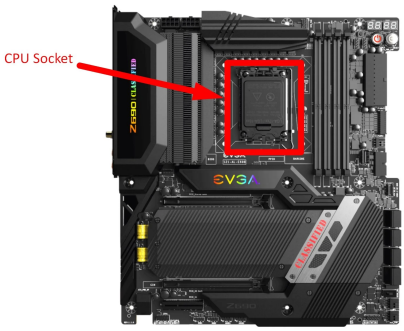


Figure 1.2. *CPU Socket in MotherBoard [3]*

1.2.6 GPU Overview

The Graphics Processing Unit is a specialized electronic circuit designed to carry out mathematical computations at very high speeds. Tasks such as graphics rendering, machine learning, or video editing often involve applying identical mathematical operations across large datasets. Thanks to its architecture, the GPU can process many data elements simultaneously, making it highly efficient for workloads that demand intensive computation.

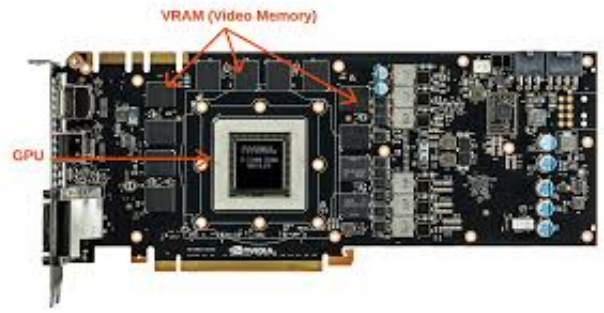


Figure 1.3. *GPU Chip*

1.2.7 Fundamentel difference between CPU and GPU

A CPU and a GPU are both essential components for data processing, but they are architecturally and functionally distinct. The CPU is designed with a few powerful cores optimized for sequential task execution and general-purpose computing. It handles complex logic and decision-making tasks efficiently. In contrast, the GPU consists of thousands of smaller, lightweight cores engineered for parallel processing. This architecture makes GPUs exceptionally well-suited for workloads that require simultaneous execution of many simple operations, such as image rendering, simulations, and training deep learning models. While CPUs excel in versatility and control-intensive tasks, GPUs provide superior performance in data-parallel and compute-intensive applications.

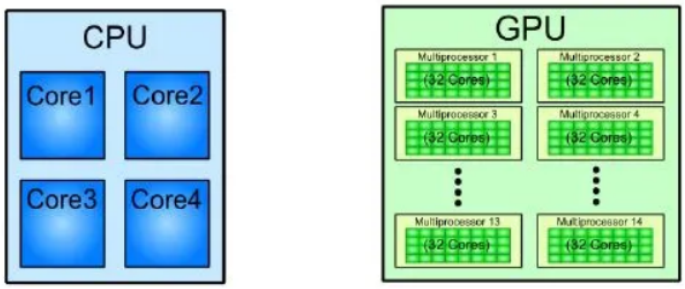


Figure 1.4. *CPU/GPU Architecture Comparison [4]*

1.3 *Host Organization*

The end-of-studies project was carried out at **ReDX Technologies**[5], a cutting-edge company founded in Tunisia in 2020. Based in Sfax, ReDX is the first company in the region to specialize exclusively in High-Performance Computing (HPC) and cloud solutions. It was established with a clear strategic vision to drive digital transformation through advanced computational technologies.

ReDX, which stands for Revolutionary Digital Transformation Technologies, focuses on empowering businesses, academic institutions, and research organizations to overcome complex computational challenges. By leveraging HPC, Artificial Intelligence (AI), DevOps, and cloud services, ReDX enables its clients to reach new levels of performance and digital maturity.

The company is also active in the fields of web development and automation, supporting a wide range of digital solutions. Through its growing professional network and collaborations with international partners, ReDX is steadily expanding its impact on a global scale, all while maintaining its role as a technological pioneer in the Tunisian and North African ecosystem

Figure 1.5. *ReDX Company Logo*

1.4 *System Analysis and Proposed Framework*

This section provides a description of the existing system, critiques its limitations, and presents the proposed solution.

1.4.1 Description of the Existing System

The Toubkal Supercomputer provides a high-performance computing environment with diverse hardware and software resources intended for scientific research. At present, despite its advanced hardware and software resources, the Toubkal Supercomputer lacks a comprehensive performance monitoring and evaluation framework. Administrators and users have limited visibility into real-time resource usage, job performance, or historical trends, making it challenging to identify performance bottlenecks, inefficient resource utilization, or regressions.

Software management on Toubkal is handled through automated frameworks, but these are outdated and do not fully support recent software applications or versions. Users must often rely on old modules or workarounds to run newer tools, creating inconsistencies and potential conflicts between different projects. This setup makes it difficult to ensure reproducibility and maintain a stable and efficient computing environment.

1.4.2 Critique of the Existing System

The absence of a performance monitoring system on Toubkal presents a significant limitation. Without proper logging, metrics collection, and visualization, issues such as hardware aging, misconfigurations, or inefficient resource usage may remain undetected until they noticeably impact scientific workflows. This reduces the overall reliability and usability of the supercomputer.

The software management framework, being outdated, exacerbates the problem. Users face a steep learning curve due to non-standardized installation processes and limited support for modern applications. Additionally, administrators struggle to maintain consistency across the cluster, increasing the likelihood of software conflicts, errors during execution, and reduced system stability.

In summary, while Toubkal provides substantial computational resources, its current system lacks proactive performance monitoring and modern software management capabilities, which hinders optimal utilization and can negatively affect research productivity.

1.4.3 Proposed Solution

In-service testing in HPC refers to evaluating and monitoring a system or component while it operates under real-world production workloads. The proposed solution introduces a comprehensive, automated, and integrated in-service testing framework for HPC clusters, designed to provide:

- Automated and scheduled benchmarking workflows.
- Continuous real-time performance monitoring.
- Robust management of software environments.
- Log-based analytics to improve reliability and provide administrators with deeper insights.

This framework enables early detection of performance anomalies, accelerates issue resolution, improves software reproducibility, and enhances the overall operational efficiency of HPC infrastructures across their lifecycle.

In this project, benchmarking, performance monitoring, software installation, and environment management were implemented on the GPU partition of the **Toubkal supercomputer**.

a) Rigorous Software Environment Management

Rigorous management of software environments ensures that users work with consistent, reliable, and reproducible configurations, while minimizing potential conflicts or execution errors.

This strategy involves:

- **Version Control and Compatibility:** Different releases of compilers, libraries, and applications are preserved to meet diverse project requirements. Stable, older versions

remain accessible alongside newer ones, ensuring continuity and preventing workflow disruptions.

- **Standardized Installation and Configuration:** Applications are installed and configured through controlled mechanisms such as environment modules or containerized setups. This minimizes misconfigurations and guarantees uniformity across all compute nodes.
- **Reproducibility:** Fixing dependencies and software versions allows computational experiments to be replicated with identical outcomes, which is critical for maintaining scientific credibility.
- **Ease of Use:** Researchers can quickly select or load the software stack suited for their tasks, lowering the barrier to entry and promoting efficient use of HPC resources.

Through these practices, the HPC cluster achieves stable and efficient operations, fosters reproducible research, and offers a dependable and user-friendly computing environment for all users.

b) HPC Benchmarking

Superior computational performance directly translates into greater scientific progress or business value, and high-performance computing (HPC) is the foundation for achieving this. Benchmarking provides quantitative measurements of system and application performance, establishes baselines, and evaluates the impact of modifications such as hardware upgrades, software optimizations, or system-level improvements.

Figure 1.6 illustrates a representative benchmarking workflow. As shown, benchmarking is far more involved than simply running code and collecting output. The process begins by selecting the target application or code to be assessed. Next, the hardware configuration to be tested is chosen. At this stage, it is crucial to define clear rules and consistent execution conditions, enabling fair comparisons. Although execution may appear simple, it often requires

tuning or adaptation to different architectures. After execution, performance metrics are recorded along with metadata that ensures meaningful analysis. Sometimes, the workflow includes iterative tuning and re-execution to refine results. Finally, extrapolation may generalize findings to untested scenarios.

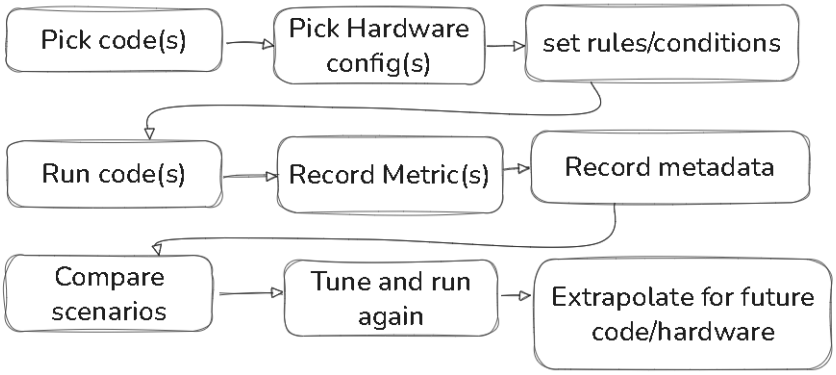


Figure 1.6. *Typical Benchmark Workflow*

A range of performance metrics can be collected during benchmarking:

- **Execution Time:** Wall-clock time required to complete a job or simulation.
- **Energy Consumption:** Expressed in joules, reflecting the energy cost of execution.
- **Rate Metrics:** Indicators such as FLOPS capturing computational throughput.
- **Ratio Metrics:** Derived measures like FLOPS per watt (performance per energy) or FLOPS per dollar (performance per cost).
- **Efficiency (% Peak):** Ratio of measured FLOPS to theoretical peak, assessing system utilization.

Within in-service testing, benchmarking is performed periodically to ensure HPC systems maintain expected performance. Automation is implemented via `ReFrame` and `Jenkins` for scheduling and result collection.

On the Toubkal supercomputer, two benchmarks were employed:

- **HPL** to evaluate floating-point computational throughput,

- **STREAM** to measure memory bandwidth.

Together, these automated workflows provide a continuous view of system performance, enabling early detection of anomalies and optimization of resource utilization.

c) Real-Time Performance Monitoring

Real-time monitoring is crucial in HPC clusters to optimize resource usage, detect bottlenecks, and maintain system reliability. It gives administrators and users continuous visibility into hardware and application states, supporting timely interventions.

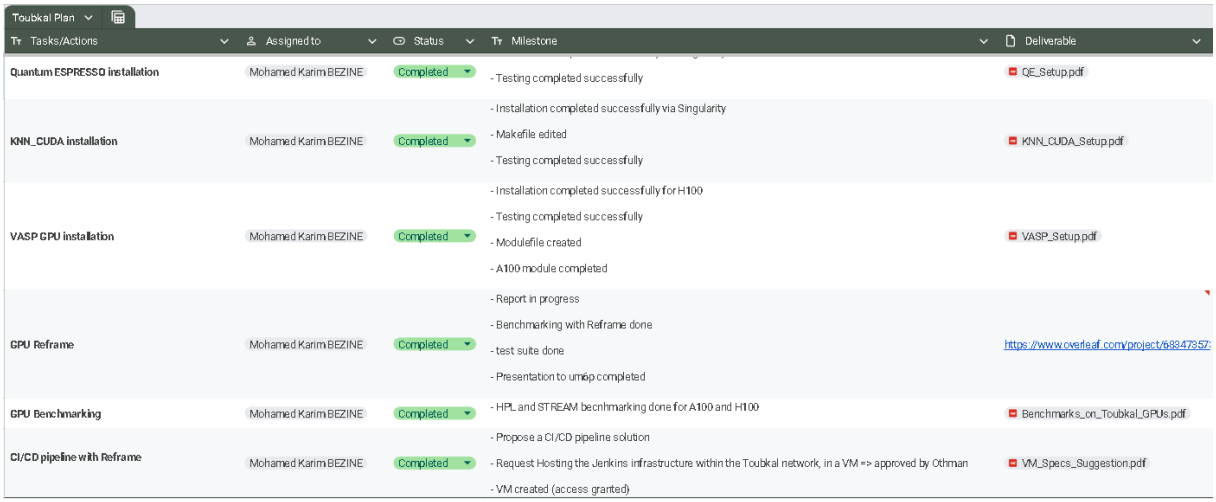
Agent-based monitoring deploys lightweight agents on cluster nodes. In this project, Prometheus and Grafana were used. Agents collect local metrics and forward them to a central server. Prometheus/Grafana is preferred in modern HPC due to scalability, flexible architecture, time-series database, PromQL, advanced visualizations, Docker deployment, and exporter support. Nagios and Ganglia are alternatives but have limitations in dynamic visualization or configuration complexity.

Log-based monitoring analyzes logs to understand system and application behavior. In this project, benchmark logs were collected and visualized with the ELK stack (Elasticsearch, Logstash, Kibana). Logs generated by Jenkins/ReFrame runs were parsed by Logstash, indexed in Elasticsearch, and visualized in Kibana. System-level logs (authentication, startup/shutdown events) were intended but inaccessible during the project.

1.5 *Project Management Method*

The project management methodology used in our end-of-studies project was `Kanban`. The project was divided into tasks, each with its own state: `Not Started`, `In Progress`, `Blocked`, or `Completed`. Upon completing each task, we delivered a report detailing the work carried out. Throughout the execution of tasks, continuous feedback was provided to improve performance

and ensure the quality of the deliverables. Figure 1.7 illustrates the executed tasks, their corresponding deliverables, and their states.



Toubkal Plan

Tasks/Actions	Assigned to	Status	Milestone	Deliverable
Quantum ESPRESSO installation	Mohamed Karim BEZINE	Completed	- Testing completed successfully	QE_Setup.pdf
KNN_CUDA installation	Mohamed Karim BEZINE	Completed	- Installation completed successfully via Singularity - Makefile edited - Testing completed successfully	KNN_CUDA_Setup.pdf
VASP GPU installation	Mohamed Karim BEZINE	Completed	- Installation completed successfully for H100 - Testing completed successfully - Modulefile created - A100 module completed	VASP_Setup.pdf
GPU Reframe	Mohamed Karim BEZINE	Completed	- Report in progress - Benchmarking with Reframe done - test suite done - Presentation to um6p completed	https://www.overleaf.com/project/68347357
GPU Benchmarking	Mohamed Karim BEZINE	Completed	- HPL and STREAM benhmarking done for A100 and H100	Benchmarks_on_Toubkal_GPUs.pdf
CI/CD pipeline with Reframe	Mohamed Karim BEZINE	Completed	- Propose a CI/CD pipeline solution - Request Hosting the Jenkins infrastructure within the Toubkal network, in a VM => approved by Othman - VM created (access granted)	VM_Specs_Suggestion.pdf

Figure 1.7. *Kanban Project Management Methodology*

1.6 *Conclusion*

In this chapter, we discussed the general context of our project, exploring the HPC environment and its current significance. We then presented the host company where the project was conducted, followed by an analysis of the Toubkal Supercomputer, highlighting its limitations and the proposed solution. Finally, we described the project management methodology applied throughout the work.

The next chapter introduces the theoretical background, providing an overview of the components of an HPC cluster, including both hardware and software elements, to build a solid foundation for understanding the rest of this work.

Chapter

2 Theoretical Background

2.1 *Introduction*

High-Performance Computing enables the execution of large-scale computations and complex simulations across scientific, engineering, and industrial domains. This chapter presents the theoretical foundations required to understand HPC systems, including GPU architectures, parallel computing models, cluster organization, job scheduling, and software management. These concepts provide the necessary background for analyzing and optimizing HPC performance.

2.2 *High Performance Computing Fundamentals*

This section focuses on the role of graphics processing units in High Performance Computing. It highlights their importance in accelerating scientific simulations and artificial intelligence workloads, and then provides an overview of GPU architecture, explaining the structural features that enable massive parallelism and high computational efficiency.

2.2.1 GPU Architecture

GPU architecture represents a key pillar of modern computing, enabling the execution of large-scale parallel tasks with remarkable efficiency. Unlike CPUs, which are primarily suited for sequential processing, GPUs are built to manage thousands of operations simultaneously. This capability makes them essential in fields such as high-performance computing, artificial intelligence, and virtualization. Today, industries ranging from retail and manufacturing to hospitality and maritime logistics leverage GPU acceleration to boost computational power, optimize workflows, and foster innovation.

As data processing demands continue to grow, GPUs are becoming an essential part of IT infrastructure, allowing businesses to accelerate AI-driven analytics, automate processes, and improve customer experiences. From facial recognition in retail security to predictive maintenance in manufacturing, GPUs enable real-time decision-making and operational agility. The ability to handle vast amounts of data at incredible speeds is what makes GPU technology so vital to modern computing.

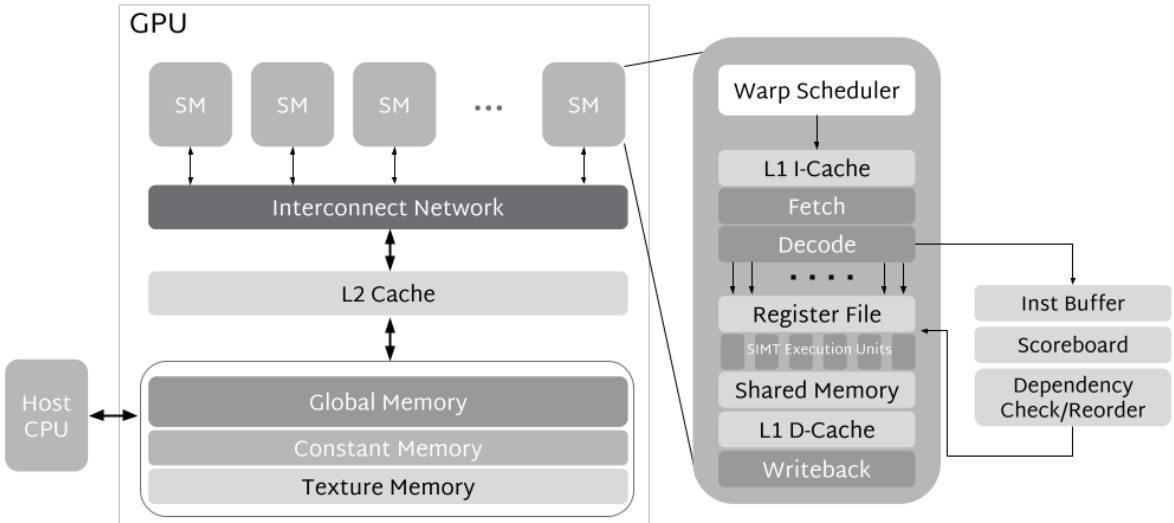


Figure 2.1. *GPU Architecture [6]*

2.2.2 The Structure of a GPU

The internal structure of a GPU is specifically designed to support highly parallel workloads. Unlike traditional CPUs, which rely on a few powerful cores for sequential execution, GPUs

integrate thousands of smaller cores organized into units optimized for concurrent processing. This architecture combines compute units, multiple memory levels, and high-bandwidth interconnects, all of which work together to maximize throughput. In what follows, the main structural components of a GPU are described:

- **Streaming Multiprocessors (SMs):** A Streaming Multiprocessor is the fundamental processing unit of a GPU. Each SM contains multiple GPU cores, a small memory pool (SRAM), and execution units. Each SM operates independently, handling multiple programs in parallel. The number of SMs in a GPU directly affects its computational power. When a program runs on a GPU, it gets split across multiple SMs, and each SM works on a chunk of data. More SMs mean better performance.

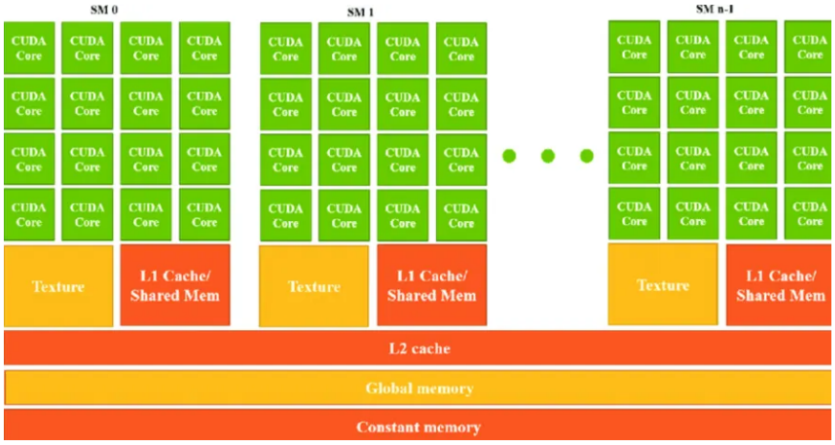


Figure 2.2. *Streaming Multiprocessors [4]*

- **GPU Memories and Cache Hierarchy:** A GPU has a complex memory hierarchy designed to maximize computational efficiency. The three main types of memory in a GPU include:
  - **SRAM Fastest Cache Memory:** Located inside the GPU core, it utilizes Registers, L1 Cache, and L2 Cache:
    - **Registers** : These are tiny, ultra-fast memory locations within each GPU core. Registers store immediate values that a core is actively processing, making them the fastest type of memory.

  - **L1 Cache** : This is the first-level cache inside a Streaming Multiprocessor (SM). It stores frequently accessed data to speed up calculations and reduce access to slower memory (like DRAM).
  - **L2 Cache** : This is a larger, second-level cache that is shared across multiple SMs. It helps store and reuse data that might not fit in L1 cache, reducing reliance on external memory (VRAM).

– **Texture Memory and Constant Memory:** Specialized types of GPU memory optimized for efficient access to particular data, such as image textures or constant values. Both types offer fast access speeds, comparable to shared memory.

– **Device Memory/VRAM or HBM/High-Performance VRAM:** VRAM is the main memory used in GPUs. Located on the GPU card itself, it stores large datasets such as textures, frame buffers, and model parameters. It is slower than SRAM but offers greater storage capacity. Modern GPUs implement VRAM using GDDR technology, which provides high bandwidth for intensive graphics and compute tasks. HBM is a high-performance variant designed for AI and deep learning, with stacked memory chips that reduce latency and increase bandwidth.

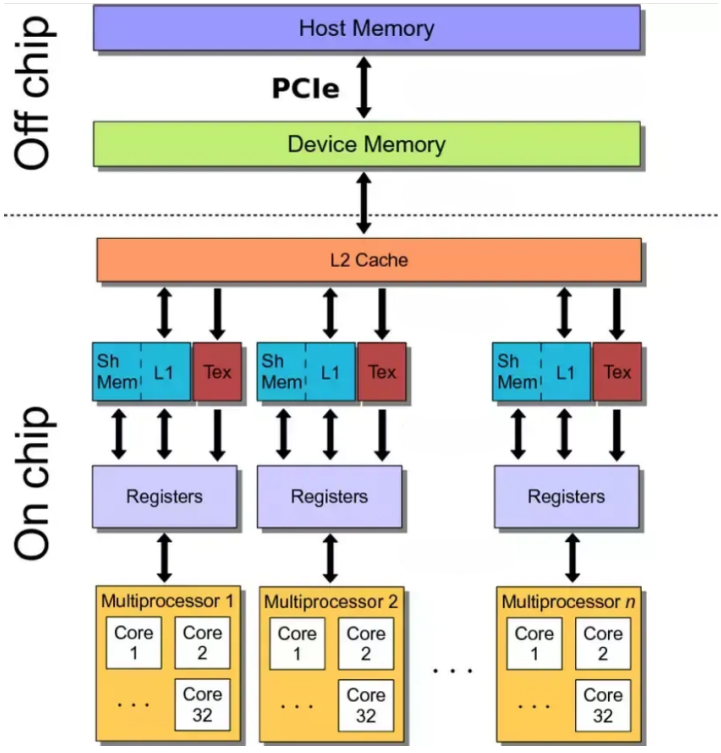


Figure 2.3. *Memory Types [7]*

To improve performance further, modern GPUs employ caching algorithms that predict and prefetch frequently used data, minimizing accesses to slower main memory. This is especially beneficial for AI-driven workloads processing vast datasets in real time.

- **Memory Bus and Bandwidth:** The memory bus connects the GPU's processing cores with VRAM, determining the rate at which data flows. A wider memory bus and higher bandwidth enable faster data transfer, which is crucial for applications that demand real-time data processing, such as machine learning and high-fidelity graphics rendering. As GPUs continue to evolve, memory bus widths have expanded, with some modern GPUs featuring 256-bit or even 512-bit memory buses. This expansion increases the volume of data that can be processed in a single clock cycle, enhancing GPU performance in high-compute environments such as real-time fraud detection, high-frequency trading, and large-scale simulations.

  As we mentioned before that CPU and GPU are distinct components, each with its own dedicated memory, and direct access between them is not possible. Data transfer between the two must occur through the PCIe bus, a standard interface for this purpose. The bandwidth of the PCIe bus is a critical factor when deciding whether to move data from the CPU to the GPU for processing, as it represents the slowest link in the data path.

- **Clock Speeds and Efficiency:** Clock speed (measured in MHz or GHz) dictates how quickly a GPU executes instructions. While higher clock speeds generally mean faster processing, efficiency improvements, such as dynamic frequency scaling and power-saving modes, help balance performance and energy consumption in enterprise environments. Recent advancements in GPU architecture include adaptive clock boosting, where the GPU dynamically adjusts its clock speed based on workload intensity and thermal conditions. This ensures peak performance when needed while maintaining energy efficiency, reducing operational costs for businesses utilizing GPU-powered infrastructure.

2.2.3 Layers

A GPU's architecture is organized into layers, each handling specific functions to ensure high performance and efficient operation.

- **Hardware Layer:** This layer consists of the physical components of the GPU, including CUDA cores, tensor cores (for AI acceleration), VRAM, and memory controllers. The hardware layer dictates the raw computing power and efficiency of a GPU.
- **Firmware & Driver Layer** GPU firmware and drivers act as intermediaries between hardware and software, optimizing performance, managing power consumption, and enabling compatibility with different workloads. Up-to-date drivers ensure maximum efficiency and security.
- **Software & API Layer** The software and API layer includes programming interfaces like CUDA, OpenCL, Vulkan, and DirectX, which allow developers to optimize applications for GPU acceleration. These APIs provide the tools needed to harness a GPU's computational power effectively.

2.2.4 GPU Performance Metrics

GPU performance is commonly evaluated using several key metrics:

- **TFLOPS and Processing Power:** Indicates the number of floating-point operations per second. Higher TFLOPS mean greater raw computational capacity, but real performance also depends on memory, architecture, and software optimization.
- **Memory Bandwidth & Latency:** Determines how fast data moves in and out of the GPU. High bandwidth and low latency, supported by technologies like HBM and GDDR6X, are essential for AI training, analytics, and high-resolution rendering.
- **Parallel Processing:** GPUs execute thousands of tasks simultaneously across many cores, enabling faster deep learning, real-time analytics, and large-scale simulations compared to CPUs.

- **Efficiency Metrics:** Measure performance relative to power consumption. Modern GPUs use techniques such as dynamic voltage scaling to balance speed with energy efficiency, which is vital for sustainable large-scale deployments.

2.3 *Parallel Computing Models and Architectures*

Parallel systems enable multiple computer resources to work simultaneously, which can involve a single machine with multiple processors, a network of interconnected computers forming a cluster, or a combination of both. Programming such systems is more challenging than for single-processor computers, as the architecture varies and multiple CPUs must be carefully coordinated and synchronized. Portability is one of the main difficulties in parallel processing.
An *Instruction Stream* refers to the sequence of instructions fetched from memory, while a *Data Stream* represents the operations performed on the data by the processor.
Flynns' taxonomy, introduced by Michael Flynn in 1966, classifies computer architectures based on the number of instruction and data streams that can be processed concurrently. Figure 2.4 illustrates the four categories in this taxonomy:

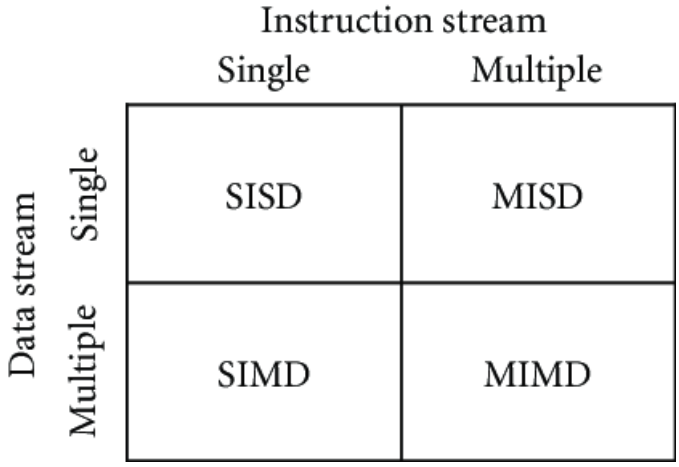


Figure 2.4. *Flynns' taxonomy [8]*

- **Single Instruction Single Data (SISD):** In a SISD architecture, there is a single processor that executes a single instruction stream and operates on a single data stream. This is the simplest type of computer architecture and is used in most traditional computers [9].

- **Single Instruction Multiple Data (SIMD):** In a SIMD architecture, there is a single processor that executes the same instruction on multiple data streams in parallel. This type of architecture is used in applications such as image and signal processing [9].
- **Multiple Instruction Single Data (MISD):** In a MISD architecture, multiple processors execute different instructions on the same data stream. This type of architecture is not commonly used in practice, as it is difficult to find applications that can be decomposed into independent instruction streams [9].
- **Multiple Instruction Multiple Data (MIMD):** In a MIMD architecture, multiple processors execute different instructions on different data streams. This type of architecture is used in distributed computing, parallel processing, and other high-performance computing applications.In practice, MIMD architectures also may include SIMD units within individual processors, allowing them to take advantage of both parallelization techniques. [9]

2.4 *HPC Cluster*

An HPC cluster, commonly known as a supercomputer, delivers high-performance computational power by linking together hundreds or even thousands of individual computers, called nodes, via a fast high-bandwidth communication network. While the majority of these nodes are optimized for intensive computational tasks, HPC clusters also incorporate specialized nodes that handle parallel file systems, databases, user login access, and job scheduling, as illustrated in Figure 2.5.

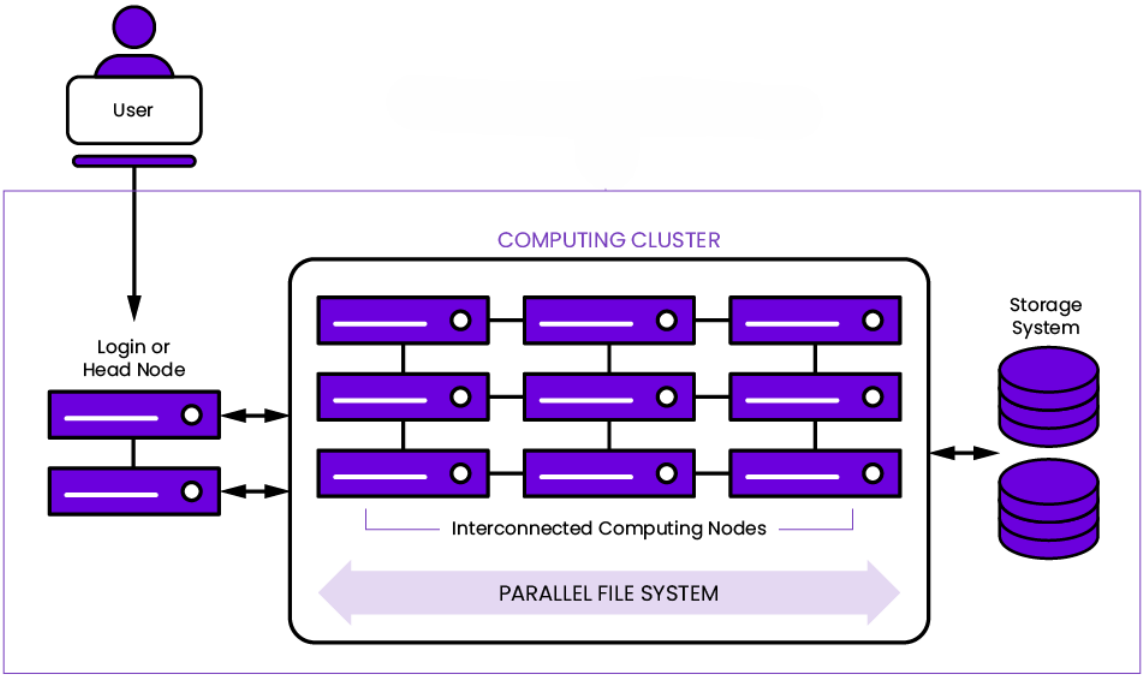


Figure 2.5. *HPC Cluster Architecture [10]*

2.4.1 Cluster Architecture

An HPC cluster is designed with a layered architecture to ensure high computational power, scalability, and reliability. Its structure brings together different types of nodes connected through a high-speed network, each serving a specialized role in the system. The architecture typically separates computation, data management, and user access to optimize efficiency.

2.4.2 Cluster Nodes

Cluster nodes are the fundamental building blocks of an HPC system. Each node type plays a distinct role in ensuring that the cluster operates smoothly and can handle diverse workloads. Below are the main categories of nodes:

- **Login Nodes:** These nodes act as the entry point to the cluster. Users connect to them via remote shell access, typically through SSH. From here, tasks such as preparing scripts, compiling programs, and submitting jobs to the scheduler are carried out.

- **Compute Nodes:** Compute nodes perform the actual heavy processing. They are equipped with multiple CPUs, large memory, or GPUs for intensive workloads. Clusters often provide different types of compute nodes, optimized either for high-memory or highly-parallel tasks. Users cannot directly access these nodes; instead, jobs are submitted through a scheduler.

- **Storage Nodes:** Storage nodes manage the cluster's data. They typically use RAID technology to ensure fault tolerance and high availability. All cluster nodes access shared storage through distributed file system technologies, allowing efficient data sharing across the system.

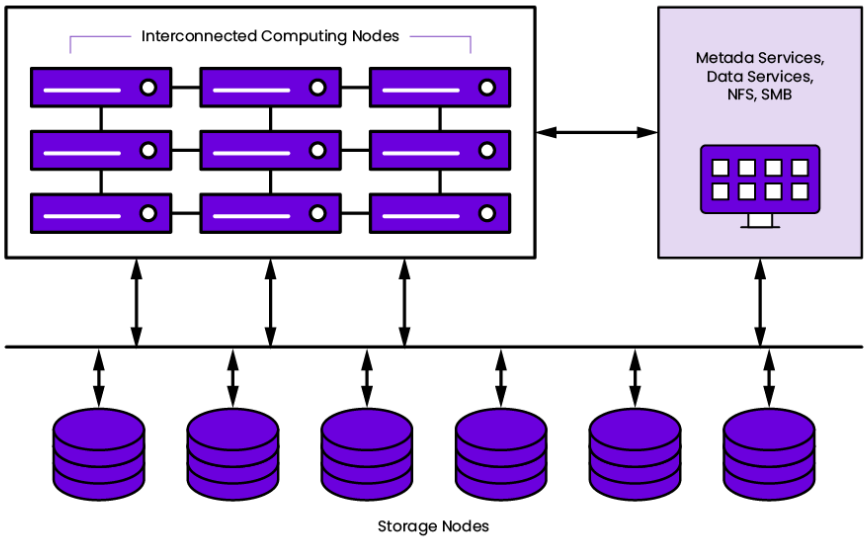


Figure 2.6. *HPC Storage [10]*

2.4.3 Job Scheduler

A Job Scheduler is a software system, often referred to as a Distributed Resource Management System (DRMS), that allows users to specify the computational resources required for their HPC workloads during runtime, the duration for which these resources are needed, and how they should be used when available. This scripted workflow of resource requests and execution commands is termed a *job*. Users typically write job scripts and submit them to the scheduler from the login node. The scheduler then queues and schedules these jobs to run at a specific

time on specific compute nodes, fulfilling the user's request. [11]

A job scheduler is generally responsible for:

- **Job Scheduling:** Allocating jobs to available compute nodes based on scheduling policies such as FIFO, fair-share, or priority.
- **Resource Management:** Tracking available, in-use, and reserved resources such as CPU cores, memory, and GPUs.
- **Job Queuing and Execution:** Accepting job submissions and launching them when the required resources become available.

This workflow is well illustrated in Figure 2.7.

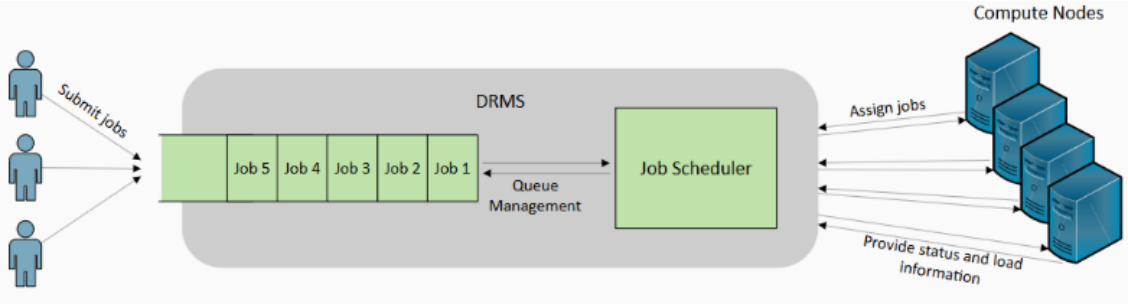


Figure 2.7. *Operating Principle of a traditional resource-manager based Computing Cluster [12]*

There are several implementations of Distributed Resource Management System , including **SLURM**, **Grid Engine**, and **PBS/Torque**.

2.4.4 Parallel File System

A parallel file system, sometimes referred to as a *clustered file system*, is a storage architecture that distributes data across several network-connected servers. It enables high-speed data access by allowing clients, such as compute or login nodes to perform coordinated input/output operations concurrently with the storage nodes.

In high-performance computing (HPC), parallel file systems enable distributed applications to read from and more importantly write to a single file concurrently from many clients without

locking, i.e., in parallel. This capability significantly improves I/O throughput and scalability, making parallel file systems essential for large-scale scientific and engineering workloads.

There are several implementations of parallel file systems, including **Lustre**, **GPFS** (also known as IBM Spectrum Scale), **BeeGFS**, and **OrangeFS**.

2.4.5 Interconnect / Network Fabric

Node-to-node communication in an HPC cluster is facilitated by a hardware component known as the interconnect network. The choice and efficiency of this communication system directly influence cluster performance, affecting both the speed at which nodes exchange data and the time required to access shared storage. The performance of an interconnect is typically evaluated based on two key metrics: bandwidth, which measures the volume of data transferable per second (usually in megabytes), and latency, which indicates the time needed to establish a connection with a remote node. Common interconnect technologies in HPC include Ethernet and InfiniBand. While Ethernet is widely adopted, its protocol introduces limitations that hinder low-latency performance. In contrast, InfiniBand provides very high throughput and minimal latency, enabling rapid communication between nodes. Modern solutions, such as NVIDIA Mellanox NDR, can achieve throughput rates of up to 400 GB/s, supporting demanding HPC workloads.

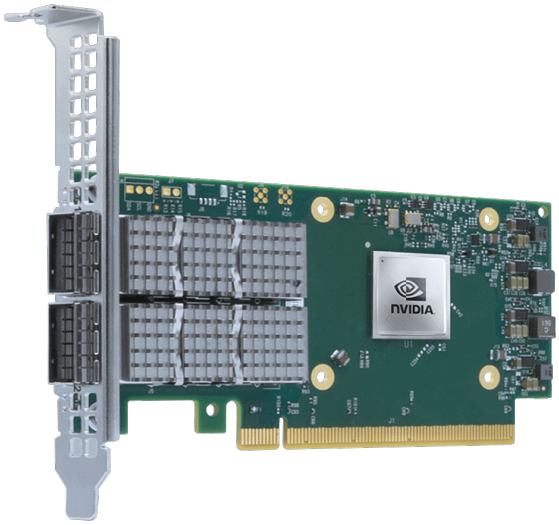


Figure 2.8. *NVIDIA Mellanox NDR InfiniBand Adapter [13]*

2.4.6 Software Management in HPC Clusters

Supercomputers are typically based on Unix-like operating systems, and managing scientific software in such environments requires careful planning to ensure performance, reproducibility, and user flexibility.

- **Files**

  Files serve as containers for different forms of information, such as text, programs, images, or videos.

  Modern operating systems provide utilities to create, read, edit, copy, and remove files.

  When working with HPC systems, it is important to be familiar with the most common file categories used in programming and execution.

  The main types of files include:

  - **Executables** – compiled binary files that contain machine instructions. When launched, the operating system runs the program encoded in them.
  - **Libraries** – collections of reusable functions or routines that can be linked or imported into applications.
  - **Configuration files** – structured data files that define parameters and settings required for a program's execution.
  - **Scripts and source code** – text files containing program instructions. Scripts (e.g., Bash, Python) can be executed directly, while source code (e.g., C, Java) must first be compiled.
  - **Header files** – special include files that declare functions, variables, and data structures without providing their implementation, ensuring modularity and code reuse.

- **Path environment variables**

  These variables list common locations of files and are used by the system to lookup requested executables and libraries, without the need to provide the full path, which

reduces the amount of configuration that would be required to run the same software on different systems.

- `PATH` - Defines the directories the shell searches when you type a command by name, for example: `/home/username/bin:/usr/local/bin:/usr/bin:/bin`.
- `LD_LIBRARY_PATH` - Instructs the dynamic linker on where to look for shared libraries, e.g., `/usr/local/lib:/usr/lib:/lib`.
- `CPATH` - Provides the directories to be checked for header files during compilation, particularly in C and C++ projects.
- `PKG_CONFIG_PATH` - Indicates the locations where `pkg-config` should search for `.pc` files that define compiler and linker flags.

When new software is installed or enabled, extra directories may be appended to `PATH` or `LD_LIBRARY_PATH`. Users can also set these variables locally in their environment, which is often necessary when compiling or running custom-built applications.

- **Software**

  Software consists of computer programs that can include executables, libraries, configuration files, datasets, or other resources. It may be launched directly by the user or executed automatically by the operating system. In many cases, several programs are bundled together in a single package, such as Anaconda, Microsoft Office, or Adobe Creative Cloud. For instance, Microsoft Office brings together Word, Excel, PowerPoint, and Outlook in one suite.

  Before use, most software must be installed, which can typically be done through:

  - a standalone executable installer
  - an installation package
  - a compressed archive containing setup and usage instructions
  - source code archives with build scripts and documentation

2.5 *Technologies Used*

In this section, we present the main technologies used in the project, including ReFrame, Jenkins, the ELK stack, Prometheus and Grafana, and Docker Compose.

2.5.1 ReFrame

ReFrame is a powerful framework designed for writing and automating system regression tests and benchmarks, specifically targeted at HPC systems. It provides a structured and efficient approach to executing tests while ensuring portability across different computing environments.

Writing a regression test in ReFrame involves two main components: the configuration script and the test code. The configuration script specifies the execution context of the tests, including the name of the HPC system, the module system to be used for software management (e.g., lmod, tmod31, nomod, spack, tmod4), the hostnames where the configuration is valid, and the system partitions.

In ReFrame, a system is an abstraction of an HPC platform managed by a workload manager. A system can contain multiple partitions, each representing a collection of nodes with similar characteristics (e.g., GPU architecture, GPU availability, memory size). For each partition, the configuration can define attributes such as its name, the scheduler type (e.g., Slurm, PBS), the parallel launcher used to start parallel programs (e.g., local for direct execution without a launcher, mpirun for launching MPI programs), and the environments that are valid for that partition.

An environment in ReFrame represents the software configuration in which a test will run. It consists of environment variables, loaded modules, and compiler definitions. By separating systems, partitions, and environments, ReFrame enables the same test code to be executed seamlessly across multiple HPC systems without modification. Figure 2.9 depicts this architecture.

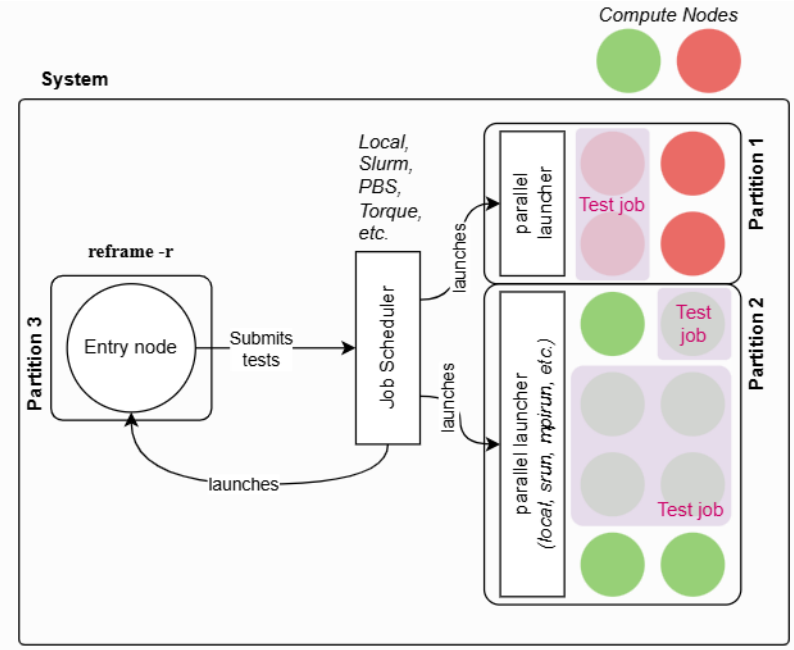


Figure 2.9. *ReFrame's System Architecture [14]*

2.5.2 Jenkins

Jenkins is an open-source automation server widely used for Continuous Integration and Continuous Deployment (CI/CD). It allows scheduling, automating, and monitoring tasks such as building software, running tests, and deploying applications. In the context of HPC, Jenkins can be used to automatically execute benchmark workflows and trigger regression tests. Its extensible plugin system and integration with tools like Git, ReFrame, and containerized environments make it a flexible solution for managing automated workflows in complex HPC infrastructures.

2.5.3 Elastic Stack

The Elastic Stack, commonly referred to as ELK (Elasticsearch, Logstash, and Kibana), is a suite of tools for collecting, storing, analyzing, and visualizing log data. In HPC environments, ELK can be used to process benchmark or application logs, enabling users and administrators to monitor system performance, detect anomalies, and gain insights into cluster behavior. Logstash

collects and parses logs, Elasticsearch stores and indexes the data for fast querying, and Kibana provides interactive dashboards for visualization and analysis.

2.5.4 Prometheus

Prometheus, an open-source monitoring and alerting toolkit, has gained prominence in agent-based monitoring setups. Developed by SoundCloud, Prometheus boasts reliability, scalability, and extensibility, making it suitable for monitoring dynamic cloud-native environments and HPC clusters alike [15]. Using a pull-based approach, Prometheus scrapes metrics from instrumented targets (nodes) at regular intervals. Node Exporter serves as a critical component in Prometheus-based monitoring setups, facilitating the collection of a diverse range of system-level metrics from Linux and Unix-like systems. Metrics encompass CPU utilization, memory allocation, disk I/O statistics, network traffic, and more. Node Exporter exposes these metrics in a format compatible with Prometheus, allowing for seamless integration into monitoring pipelines. Added to that, the DCGM Exporter extends Prometheus monitoring to the GPU level. It collects detailed metrics such as GPU utilization, memory usage, power consumption, and temperature from NVIDIA GPUs. These metrics are exposed in a Prometheus-compatible format, enabling operators to monitor GPU health and performance alongside standard system metrics, and to visualize them in dashboards such as Grafana.

2.5.5 Grafana

Grafana is a tool that visualizes metrics from a variety of data sources, such as Prometheus, Thanos , and InfluxDB. It can host a collection of dashboards and create graphs to visualize the metrics from the data sources. Some exporters even provide pre-designed Grafana graphs to import to an existing dashboard.

2.5.6 Docker Compose

Docker is a platform that enables containerization, allowing applications and their dependencies to run in isolated environments called containers. This ensures reproducibility, portability, and consistency across different systems. Docker Compose is a tool that simplifies the management of multi-container applications by allowing users to define and run multiple interconnected containers using a single configuration file. In the context of HPC and this project, Docker and Docker Compose were used to deploy monitoring and automation services, such as Prometheus, Grafana, and ELK components, in a controlled and reproducible environment.

2.6 *Conclusion*

This chapter provided an overview of the principles and components that make HPC systems effective. By understanding GPU design, parallel processing, and cluster organization, we gain the foundation needed to analyze, optimize, and efficiently manage computational workloads in high-performance environments.

Chapter

3 Toubkal Supercomputer & HPC Software Stack Build

3.1 *Introduction*

This chapter presents the architecture of the Toubkal supercomputer, detailing its hardware and software ecosystem. It describes the software management approach and outlines the process for installing scientific software. A case study demonstrates the installation steps in practice.

3.2 *Toubkal Supercomputer Architecture*

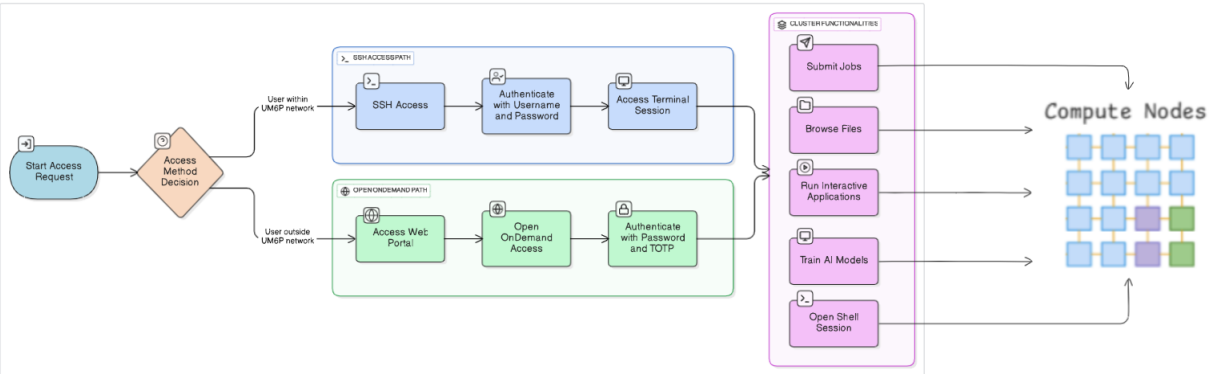


Figure 3.1. *Toubkal*

Toubkal provides two primary methods for accessing the cluster. The first is via SSH for users within the UM6P network, who can connect directly through the terminal using the command

`ssh <username>@toubkal.hpc.um6p.ma`. The second method is through Open OnDemand, a web-based HPC access portal available to users outside the UM6P network. This method requires two-factor authentication (password and TOTP). Open OnDemand allows users to browse files, open shell sessions, access Linux desktops, run interactive applications, and submit jobs to the cluster, all from within a web browser.

3.2.1 System Architecture

The Toubkal supercomputer is built on a heterogeneous architecture designed to support a wide range of computational workloads, from traditional CPU-based tasks to massively parallel GPU-accelerated applications. Its design combines general-purpose compute nodes with specialized GPU nodes, interconnected through a high-speed InfiniBand network and supported by large memory capacities. This balanced integration of CPUs, GPUs, memory, and interconnect ensures scalability, high throughput, and the ability to handle both compute- and data-intensive scientific applications efficiently. The following sections present detailed specifications of the different compute node types available in the system.

a) Compute Node CPU Spec

The Toubkal supercomputer's compute infrastructure consists of 1,219 Dell EMC PowerEdge C6420 nodes, each powered by Intel Xeon Platinum 8276L processors. Every node features two sockets, with each socket hosting a 28-core CPU, providing a total of 56 cores per node. Each node is equipped with 192 GB of memory and employs a dual NUMA (Non-Uniform Memory Access) architecture. In this architecture, memory is divided into two regions, each directly connected to one of the CPUs. Accessing local memory is faster than accessing the memory attached to the other CPU, which helps improve performance by reducing latency when processes are scheduled to use the memory closest to the CPU executing them. This design is particularly beneficial for multi-threaded and memory-intensive workloads.

Table 3.1. *CPU Specifications of the Node*

Attribute	Value
CPU Architecture	Intel Cascade Lake (2nd Gen Xeon Scalable)
Processor Model	Intel Xeon Platinum 8276L
CPU(s)	56
Thread(s) per Core	1
Core(s) per Socket	28
Socket(s)	2
NUMA Node(s)	2
Base Frequency	2.4 GHz
Maximum Boost Frequency	4.0 GHz
L1d Cache	32 KB
L1i Cache	32 KB
L2 Cache	1024 KB (1 MB)
L3 Cache	39424 KB (38.5 MB)

b) Compute Node A100 GPU Spec

The GPU nodes in the cluster are built on Dell XE8545 servers, each equipped with dual AMD EPYC 7713 processors, offering a total of 128 CPU cores per node across two sockets. These nodes are paired with 1 terabyte of system memory to support large-scale computations and data-intensive workloads. For GPU acceleration, each node features four NVIDIA A100 GPUs in the SXM4 form factor, each boasting 80 GB of high-bandwidth HBM2e memory. To ensure ultra-fast communication between nodes and GPUs, the cluster utilizes a dual-rail Mellanox HDR200 InfiniBand interconnect, providing high throughput and low latency necessary for demanding high-performance computing applications.

Table 3.2. *NVIDIA A100 80GB SXM GPU Specifications*

Attribute	Value
GPU Architecture	Ampere (GA100)
CUDA Cores	6912
Tensor Cores	432
Base Clock	1410 MHz
Memory Size	80 GB HBM2e
Memory Bus Width	5120 bits
Memory Bandwidth	2039 GB/s
TDP (Thermal Design Power)	400 W
Interconnect	NVIDIA NVLink 3.0
PCIe Support	PCIe 4.0
Compute Capability	8.0
Process Technology	TSMC 7nm FinFET

c) Compute Node H100 GPU Spec

The cluster includes three H100 GPU nodes, each composed of dual-socket CPUs with 48 cores per socket, totaling 96 CPU cores and around 1 TB of RAM per node. Each node is equipped with 4 NVIDIA H100 GPUs, providing high-performance acceleration for compute-intensive tasks. These nodes run a Linux kernel 4.18 environment and are interconnected through a high-speed InfiniBand network, ensuring efficient, low-latency communication for distributed workloads and multi-GPU scaling.

Table 3.3. *NVIDIA H100 80GB HBM3 GPU Specifications*

Attribute	Specification
GPU Architecture	Hopper
CUDA Cores	16,896
Tensor Cores	512
Base Clock	1460 MHz
Memory Size	80 GB HBM3
Memory Bus Width	5120 bits
Memory Bandwidth	3350 GB/s (3.35 TB/s)
TDP (Thermal Design Power)	Up to 700 W (configurable)
Interconnect	NVIDIA NVLink 4.0
PCIe Support	PCIe 5.0
Compute Capability	9.0
Process Technology	TSMC 4N (4nm-class)

d) Network System

The Toubkal internal network features an efficient HDR InfiniBand architecture from Mellanox, utilizing 200 Gbps interconnects to link compute nodes. It relies on PowerEdge C6420 servers powered by Intel Xeon Platinum processors for the CPU partition, ensuring robust computational performance. More precisely, the system employs Mellanox HDR100 InfiniBand switches, which provide both 100 Gbps and 200 Gbps ports to prevent data throughput bottlenecks. Each compute node is equipped with Mellanox ConnectX-6 HDR adapters, enabling high data transfer rates and facilitating efficient communication across the cluster.

In the GPU A100 partition, connectivity is provided through a dual-rail Mellanox HDR200 InfiniBand fabric, linking Dell PowerEdge XE8545 nodes that each house dual AMD EPYC 7713 processors and four NVIDIA A100 SXM4 80 GB GPUs. This dual-rail setup doubles available bandwidth per node, making it ideal for large-scale AI and HPC workloads.

The GPU H100 partition uses a similar high-performance networking approach, leveraging Mellanox HDR200 or NDR InfiniBand technology (depending on the specific deployment) to interconnect nodes equipped with NVIDIA H100 80 GB HBM3 GPUs. This ensures the latest generation of GPU nodes benefit from extremely low-latency communication and maximum throughput for data-intensive applications.

e) Prallel File System

Toubkal uses Lustre for the parallel file system

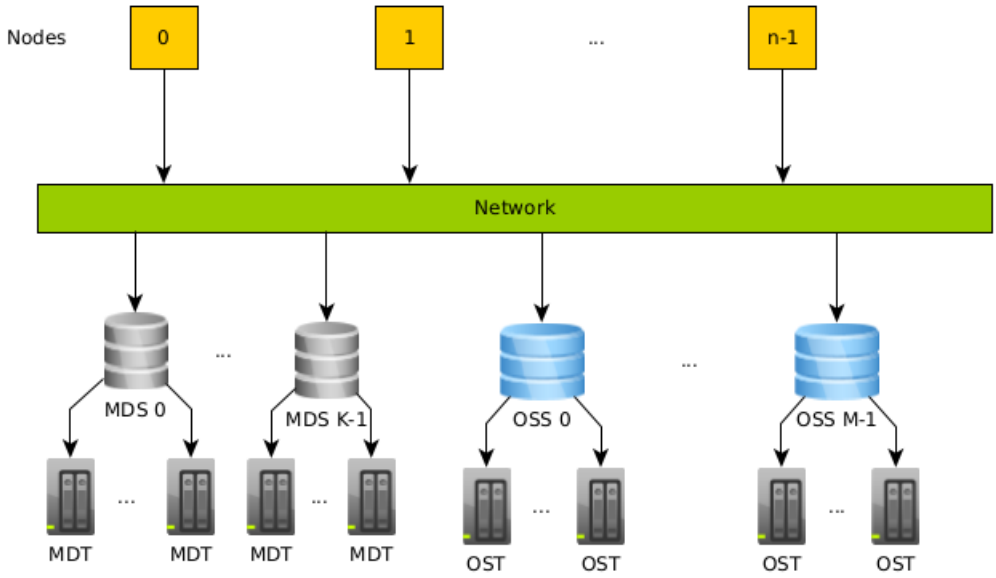


Figure 3.2. *Lustre File System View*

Figure 3.2 illustrates the architecture of the Lustre file system.
Lustre separates file data and metadata into separate services. Data is the actual contents of the file, while metadata includes information like file size, permissions, access date etc.

The Lustre file system consists of a set of I/O servers called Object Storage Servers (OSSs) and disks called Object Storage Targets (OSTs) which store the actual data. The metadata of a file are controlled by Metadata Servers (MDSs) and stored on Metadata Targets (MDTs). Basically, the servers handle the requests for accessing the file contents and metadata; the applications do not access disks directly. Lustre systems use typically multiple OSSs/MDSs together with multiple OSTs/MDTs to provide parallel I/O capabilities.

- **Object Storage Servers (OSSs)** : They handle requests from the clients in order to access the storage. Moreover, they manage a set of OSTs; each OSS can have more than one OST to improve the I/O parallelism.
- **Object Storage Targets (OSTs)** : Usually, an OST consists of a block of storage devices under RAID configuration. The data is stored in one or more objects, and each object is stored on a separate OST.
- **Metadata Server (MDS)** : A server that tracks the locations for all the data so it can decide which OSS and OST will be used. For example, once a file is opened, the MDS is not involved any more.
- **Metadata Target (MDT)**: The storage contains information about the files and directories such as file sizes, permissions, access dates. For each file MDT includes information about the layout of data in the OSTs such as the OST numbers and object identifiers.

The Lustre parallel filesystem in the Toubkal cluster is configured as follows:

- **Object Storage Servers and OSTs:** The cluster includes 8 OSS nodes, each connected to a PowerVault ME4084 storage array. Each array contains 80 disks, 4 SSDs used for caching and 76 HDDs for data storage yielding an approximate usable capacity of 1 petabyte per array.
- **Metadata Servers and MDTs:** The system features 2 MDS nodes, each attached to a PowerVault ME4024 storage array comprising 18 disks (2 SSDs and 16 HDDs), providing approximately 20 TB of metadata storage capacity per array.

3.2.2 Software Ecosystem

a) Operating System

Toubkal uses Rocky Linux 8.10 (Green Obsidian) as an operating system. Like Windows, a Unix-like operating system such as Linux organizes its files in a hierarchical directory structure.

This means files and directories are arranged in a tree-like pattern, where directories (sometimes called folders in other systems) may contain files and other directories. The first directory in the file system is called the root directory. The root directory contains files and subdirectories, which themselves contain more files and subdirectories, and so on.

Most users are probably familiar with a graphical file manager that represents the file system tree, as illustrated in Figure 3.3.

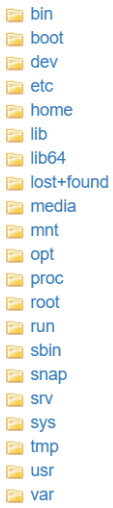


Figure 3.3. *File System Tree as shown by a Graphical File Manager*

Notice that the tree is usually shown upended, with the root at the top and the various branches descending below. However, the command line interface does not display images, so navigating the file system tree requires a conceptual approach. Imagine the file system as a maze shaped like an upside-down tree. At any given time, the user is inside a single directory, and can see the files contained within that directory, the pathway to the directory above (called the parent directory), and any subdirectories below.

Figure 3.4 shows the directory structure as a tree, and Table 3.4 provides descriptions of these directories.

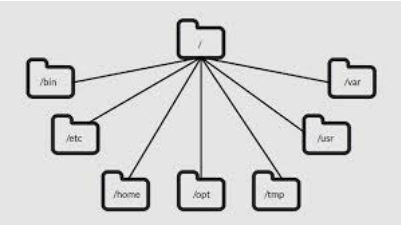


Figure 3.4. *Linux Directory Structure*

Table 3.4. *Common Linux Filesystem Directories and their Description*

Directory	Description
/	The directory called "root." It is the starting point for the file system hierarchy. Note that this is not related to the root, or superuser, account.
/bin	Binaries and other executable programs.
/etc	System configuration files.
/home	Home directories.
/opt	Optional or third party software.
/tmp	Temporary space, typically cleared on reboot.
/usr	User related programs.
/var	Variable data, most notably log files.

The directory currently being accessed in the terminal is referred to as the current working directory. To display it, the pwd (print working directory) command is used. As illustrated in Figure 3.5, the command is executed from within a shell session on the Toubkal cluster. The shell prompt indicates the username, the hostname (which identifies the login node the user is connected to), and finally, the output of the command shows the current working directory , in this case, the user's home directory.

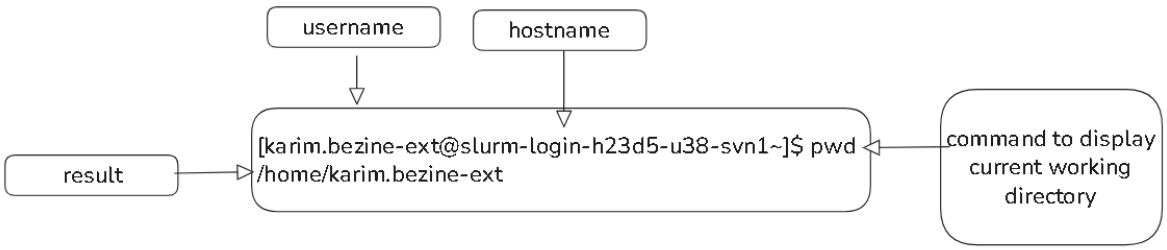


Figure 3.5. *Output of the pwd Command*

For example, when located in the root directory, the contents of that directory can be displayed using the ls (list) command, which shows all directories and files inside it.
Upon first logging in to the Toubkal Supercomputer, the current working directory is set to the user's home directory. Each user account is assigned a unique home directory , typically a path like /home/username , which is the only location where a regular user is permitted to write files.

b) Job Scheduler

The workload management on Toubkal is handled by Slurm, an open-source job scheduler that allocates computing resources, queues and schedules jobs, and monitors job execution to optimize cluster utilization. To run computations on the cluster, users must submit job scripts, which are typically written as shell scripts. Slurm provides a set of directives (prefixed with `#SBATCH`) that users include in their scripts to request specific resources such as the number of nodes, GPUs, memory, runtime, and output file paths. These directives guide Slurm in fulfilling the job's requirements efficiently.

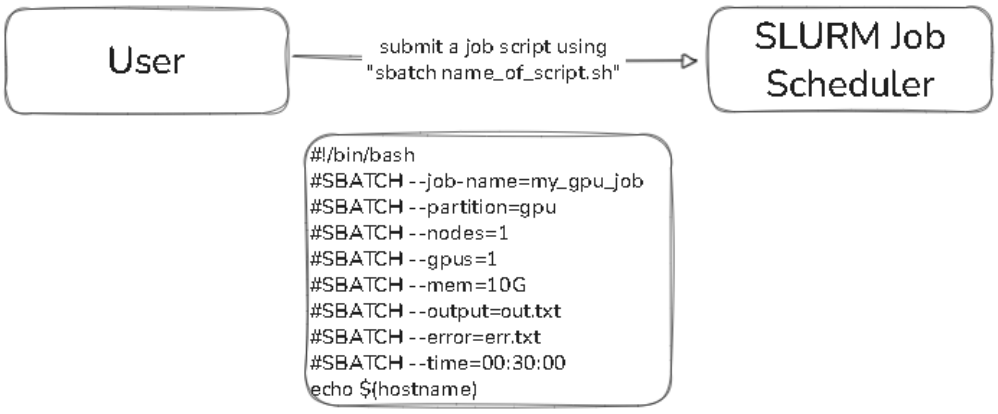


Figure 3.6. *Job Submission to the SLURM Scheduler using a Job Script*

As shown in Figure 3.6, the user submits the job to be executed on the compute nodes using the `sbatch` command. In the job script, a set of `#SBATCH` directives is specified. These directives include the number of nodes requested (in this case, 1 gpu node), the amount of memory required for the job (10 GB), and paths for output and error files. If the code produces any standard output or encounters an error during execution or exceeds its time execution which is set in our case to 30 minutes, these will be written to the respective output and error files. Finally, the script contains the actual command to be executed, which in this example is `hostname`, used to identify the node executing the job.

c) Compilers & Libraries

The Toubkal Supercomputer offers a comprehensive suite of HPC programming tools, including both licensed and open-source solutions. Its software environment is thoughtfully designed to support diverse research communities. Pre-installed software covers a broad range of domains such as chemistry, physics, deep learning, life and material sciences, and meteorology.

The system includes essential computational tools like compilers (GNU, Intel, NVIDIA) and communication libraries (OpenMPI, IntelMPI). It supports a wide variety of libraries and frameworks for numerical and scientific computing, including BLAS, LAPACK, OpenBLAS, MUMPS, PETSc (Portable, Extensible Toolkit for Scientific Computation), FFTW (Fastest Fourier Transform in the West), and ParaView for visualization.

For GPU-accelerated computing, Toubkal offers multiple versions of the NVIDIA CUDA Toolkit, enabling compatibility with a range of applications and frameworks. It also comes with popular deep learning and machine learning frameworks, including PyTorch, TensorFlow, and Keras, optimized for both CPU and GPU architectures. Additional AI-focused libraries such as cuDNN, NCCL, Horovod, and RAPIDS are provided to maximize performance for distributed training and large-scale data analytics.

Additionally, Toubkal provides access to high-level languages and software platforms such as Python, Anaconda, MATLAB, and Ansys Fluent, offering robust support for various

scientific workflows. For flexible and reproducible environment management, containerization tools like Singularity and Apptainer are also available, allowing researchers to package complex software stacks for consistent execution across the cluster.

d) Software Managment in Toubkal Supercomputer

The Toubkal supercomputer uses `Lmod` as its environment modules system. `Lmod` enables users to easily load, manage, and switch between software packages and their dependencies through `modulefiles`, supporting flexible and reproducible software environments.
A `module` provides a user-friendly interface for the dynamic modification of the user environment via `modulefiles`. These files adjust environment variables such as `LD_LIBRARY_PATH` , `PATH` and `CPATH`, allowing seamless transitions between different software configurations.[16]

When compiling a C, C++ or Fortran file with a specific version of the `GCC` compiler, it is necessary to load the corresponding module. For instance, to explore available versions of `GCC`, one can use the command `module avail GCC`, which lists several versions such as `GCC/11.3.0`, `GCC/12.3.0`, `GCC/11.2.0`, and `GCC/9.3.0`. To load a module, the user simply types `module load` followed by the module name. For example, loading the module `OpenMPI/4.0.3-GCC-9.3.0` allows the use of the `OpenMPI` communication library compiled with `GCC` version 9.3.0. The current list of loaded modules can be displayed using `module list`.
If a module is no longer needed, it can be unloaded with `module unload`, and to reset the environment completely, one can use `module purge` to unload all modules at once.
This modular system helps users avoid software conflicts and ensures a clean and controlled development environment on the cluster.

On the Toubkal supercomputer, software management can be performed using two main approaches: automated deployment with `EasyBuild` and manual installation from source.

`EasyBuild` provides a streamlined and automated framework for building and installing scientific software. It uses a large repository of predefined recipes that specify how to fetch,

configure, compile, and install software along with its dependencies. This approach ensures consistency across installations, facilitates reproducibility, and incorporates automated testing to verify installation correctness. Moreover, `EasyBuild` integrates seamlessly with the `Lmod` module system, enabling users to easily load software packages via environment modules.

Nevertheless, manual building remains necessary in certain cases, such as when new software versions are not yet supported by `EasyBuild`, when custom configurations are needed, or when the automated method fails. This aspect will be discussed in the future sections.

3.3 *Building Scientific Software on Toubkal Cluster*

3.3.1 Building From Source

Building software from source is a general term that refers to the process of creating executable binaries and other non-source files from the program's source files.[17]
In order to build software from source, we usually need some tools to help us automate and simplify the process. A build system is a tool that automates the process of compiling source code into executables, managing dependencies, and running custom build steps. It directly executes the instructions required to build software. Examples of build systems include Make and Ninja. On the other hand, a build system generator is a tool that creates the files needed by a build system based on higher-level project configurations. It does not build the software itself but generates build scripts tailored to the target platform or toolchain. Examples include CMake, which can generate Makefiles or Ninja files, and Meson, which typically generates Ninja build files. In short, a build system executes the build, while a build system generator prepares the instructions for it.

The NVIDIA HPC SDK (NVHPC) is NVIDIA's comprehensive development environment for building, profiling, and deploying GPU-accelerated applications in HPC systems. It integrates compilers, libraries, and tools tailored for NVIDIA GPUs, supporting multiple programming models including CUDA, OpenACC, and OpenMP GPU offload.

a) Compilers

NVHPC provides specialized compilers for heterogeneous programming:

- `nvc` for C11, `nvc++` for C++17, and `nvfortran` for modern Fortran standards—all supporting CPU and GPU code generation.
- `nvcc`, the CUDA C/C++ compiler, handles GPU kernel compilation and integrates with host compilers.
- The compilers support features like C++17 parallel algorithms and allow integration with custom GCC toolchains.

b) Programming Models

Supported programming paradigms include:

- **CUDA** and **CUDA Fortran** for explicit GPU kernel programming.
- **OpenACC** directives for portable, directive-based GPU acceleration.
- **OpenMP target offload** for standardized GPU parallelism.
- **C++17 Parallel Algorithms** enable GPU acceleration without CUDA-specific code.

c) Libraries

NVHPC bundles an extensive set of GPU-accelerated libraries:

- **Math libraries:** cuBLAS, cuSPARSE, cuFFT, cuSOLVER, cuTENSOR, cuRAND, with multi-GPU versions for distributed workloads.
- **Communication libraries:** NCCL for efficient multi-GPU collectives and NVSHMEM for GPU-optimized memory communication.

d) Tools

NVHPC includes tools for profiling, debugging, and optimizing GPU workloads:

- **Nsight Systems** for system-wide profiling of CPU-GPU interactions.
- **Nsight Compute** for detailed kernel-level performance analysis.
- **CUDA-GDB** and **Compute Sanitizer** for debugging and detecting memory errors or race conditions.
- **NVTX**, an instrumentation API to annotate and visualize code regions.
- **HPC Container Maker** for building HPC-focused containers.

The figure 3.7 illustrates the components and workflow of the NVIDIA HPC SDK which is the most popular commonly used toolchain in HPC environments, including its compilers, programming models, libraries, and tools used for GPU-accelerated HPC application development.

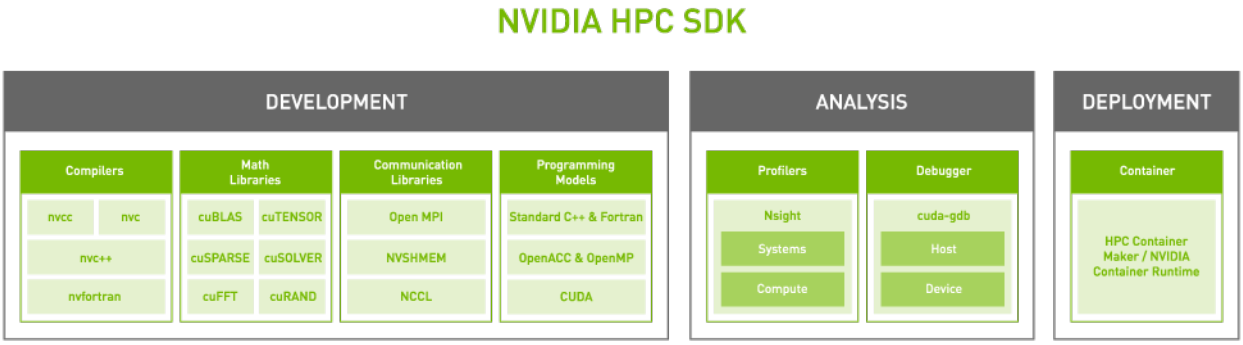


Figure 3.7. *NVHPC Toolchain*

Each toolchain is a self-contained and compatible collection of compilers, libraries, and MPI implementations, tailored for scientific computing and high-performance builds. These toolchains also play a key role in the naming conventions of installed software, indicating the specific toolchain version used during compilation , a topic that will be addressed in the following section.

3.3.2 Stages of Build Process

a) Software Installation Path and Naming Convention

On the Toubkal Supercomputer, a designated directory is provided for installing user-compiled software. This ensures that installed packages are accessible system-wide without interfering with system components. The target installation paths are `/srv/software/gpu_a100` and `/srv/software/gpu_h100`. Those paths are visible to all users and serves as the location where compiled software should be installed for shared use.
Once the software is installed, it typically follows a directory hierarchy similar to the standard Linux filesystem layout, particularly under the /usr directory.

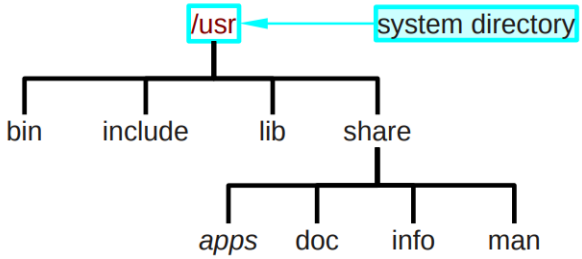


Figure 3.8. *usr System Directory Hierarchy*

The system hierarchy under /usr, as shown in Figure 3.8, contains a structured set of subdirectories designed to accommodate the diverse range of software typically used on a modern computer system. For example:

- `/usr/bin` contains user executables
- `/usr/include` contains the header files used to build C programs
- `/usr/lib` contains the libraries

The output generated from the installation of software typically follows a directory hierarchy similar to that of the /usr system directory. However, the exact structure can vary depending on the software. In some cases, the installation may only create a bin directory containing executables, while in others, it may include additional directories such as lib64, python or others specific to the software's requirements.

Building and using software for scientific research is comparable to constructing a laboratory instrument or conducting an experiment, and should be treated with the same level of care and documentation. It is important to record where the software was obtained from, along with its version. The system used to build the software should also be documented. Additionally, any options passed to the build process should be noted. Finally, it is essential to record whether the build succeeded or failed, and, in the case of failure, the reasons for it.

To ensure best practices, it is recommended to adopt a consistent naming convention for installed software. This is particularly important because software can be compiled using different compilers, and even the same compiler may have multiple versions. Each combination of software and compiler version may produce different binaries, and should therefore be installed in a separate directory.

For this reason, the name of each software installation directory should include the software name, its version, the compiler used, and the compiler version. The naming convention follows the pattern shown in Figure 3.9

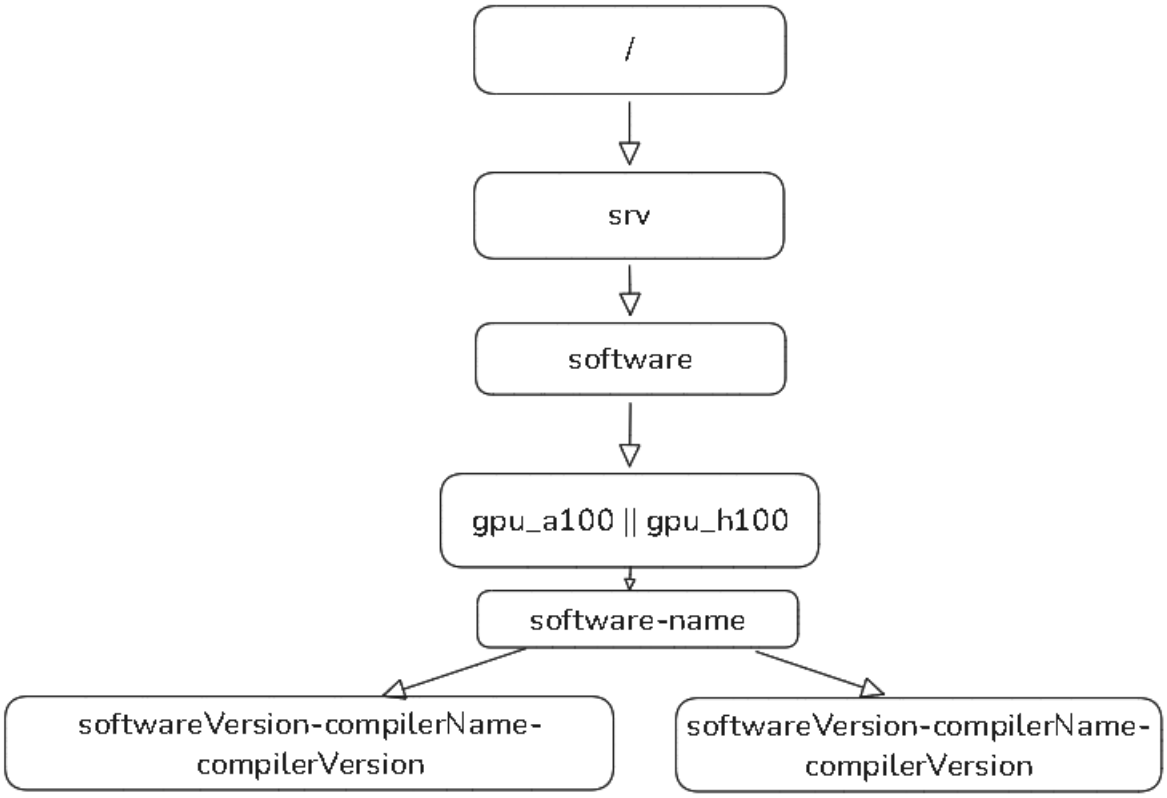


Figure 3.9. *Software Installation Path Structure*

This approach helps maintain clarity, reproducibility, and avoids conflicts between different builds of the same software.

b) Get Source Code

Software is typically distributed as an archive file containing the source code organized in a directory tree. This archive simplifies the transfer and distribution of the numerous files and subdirectories that make up a software package.

The most common format is a .tar archive, often compressed using utilities such as gzip (resulting in a .tar.gz or .tgz file). To extract the contents, the tar command is used with appropriate options. Modern versions of tar can automatically detect and handle compressed formats.

Alternatively, source code may be retrieved from online repositories using tools such as git clone for GitHub-hosted projects, or wget for direct downloads.

Once extracted, the source directory typically contains a README or INSTALL file. These files provide important instructions for building the software, including any prerequisites and the initial steps to begin the build process. It is important to consult these documents before proceeding.However, in some cases, the documentation may be incomplete or unclear, requiring additional investigation or consultation of external resources to successfully complete the build.

The next stage in the build process generally involves configuring the source code for the target system using a configuration script, typically named configure.

c) Configure

The configure script is a portable and widely used shell script designed to prepare source code for compilation on a variety of Unix-based systems. While it is complex and difficult to read, it provides a flexible and powerful mechanism for system-specific configuration.

Among its many command-line options, two are particularly important. The `--help` option displays a list of all available options, while the `--prefix` option specifies the installation

path where the software will eventually reside. Additional options may be used to define the compilers and flags to be used during the build process.

CC	C compiler
CFLAGS	C compiler options
CXX	C++ compiler
CXXFLAGS	C++ compiler options
FC	Fortran compiler
FFLAGS	Fortran compiler options
LDFLAGS	Library options

Figure 3.10. *Compiler Options*

During execution, the configure script processes template files , typically identified by the .in suffix , and generates corresponding output files. For example, if the source directory contains Makefile.in and config.h.in, the script will produce Makefile and config.h as output.

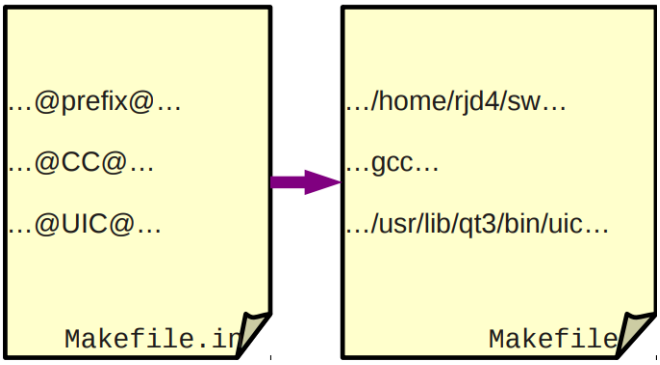


Figure 3.11. *Configure Flow*

These template files contain placeholders in the form of @word@, which are replaced with values determined during configuration. For example, @prefix@ is substituted with the value provided via the –prefix option, and @CC@ is replaced with the chosen C compiler. By default, this is typically cc, which often links to gcc on Linux systems, but it can be overridden by setting the CC environment variable.

Some placeholders depend on system-specific utilities, such as @UIC@, which is resolved only if the required program is available. If a mandatory component is missing, configure halts with an error. If the component is optional, the script issues a warning and disables related features in the build configuration.

d) Build

The build process of a program is typically managed by **make**, a tool that automates compilation and linking by reading instructions from a **Makefile**. The **Makefile** defines **targets**, which are usually the files to be generated (such as executables or object files), and **prerequisites**, which are the source files or other targets that must be up to date before building the target. Each target also specifies the commands needed to build it. This allows **make** to determine which parts of the program need rebuilding, avoiding unnecessary recompilation and speeding up the build process.

The program build itself involves three main stages: pre-processing, compilation, and linking. During pre-processing, directives such as `#include` (common in C and C++) and macros are expanded to produce a modified source code. In the compilation stage, each source file is translated into an object file (`.o`) containing machine code. Finally, linking combines these object files into a single executable, either statically embedding library code directly (`.a` files) or dynamically referencing shared libraries (`.so` files) to reduce memory usage and enable code sharing. Figure 3.12 illustrates this workflow.

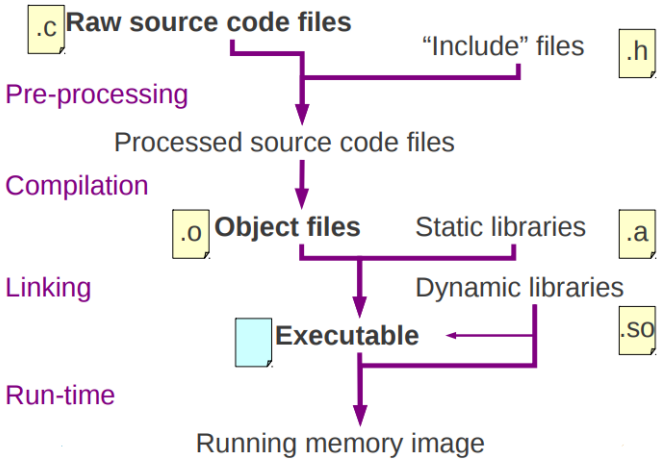


Figure 3.12. *Build Process Flow*

Typically, these phases are hidden behind a single command, usually referred to as **the compiler**, which handles pre-processing, compilation, and linking automatically, as shown in Figure 3.13.

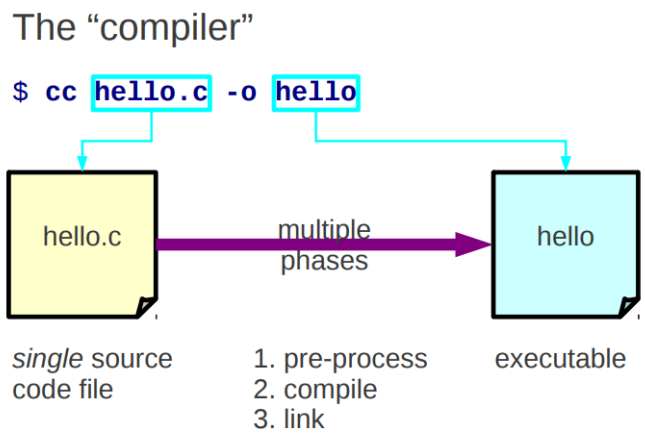


Figure 3.13. *Compiler Stages*

e) Install

Once the software is successfully built, it typically generates a set of executables, libraries, and auxiliary files such as manual pages or shared data. These files need to be organized into appropriate directories, which is handled during the installation phase.

The `make install` command performs this task by copying the compiled files into a structured directory tree. For example, executable binaries are placed under the bin directory, static libraries under lib, shared libraries under lib64 (when applicable), and documentation or manual pages under share/man. This hierarchy is created within the installation path specified earlier using the `--prefix` option during the configuration step.

The logic for this installation process is embedded in the Makefile, which was generated during the configuration phase. That file already contains instructions on where each component should be installed.

f) Iterative Development and Build Debugging

Building Software From Source are unlikely to succeed on the first attempt. Multiple iterations are typically required. The key is to approach the process systematically, treating it as a series of controlled experiments conducted in a calm and analytical manner. The procedure begins with a clean unpacking of the source code. A build is then attempted using the current

Makefile, with the output recorded in a log file for analysis. Once the build completes, its success is verified. If successful, the process proceeds to the installation phase. If the build fails, the log file and the contents of the build directory are examined to identify the cause. This analysis should answer two important questions:

- What modifications are required in the Makefile?
- Is it possible to continue using the current build tree, or is a fresh unpacking of the source code necessary?

Following this systematic approach increases the likelihood of achieving a successful and reproducible build.

This process is clearly illustrated in Figure 3.14.

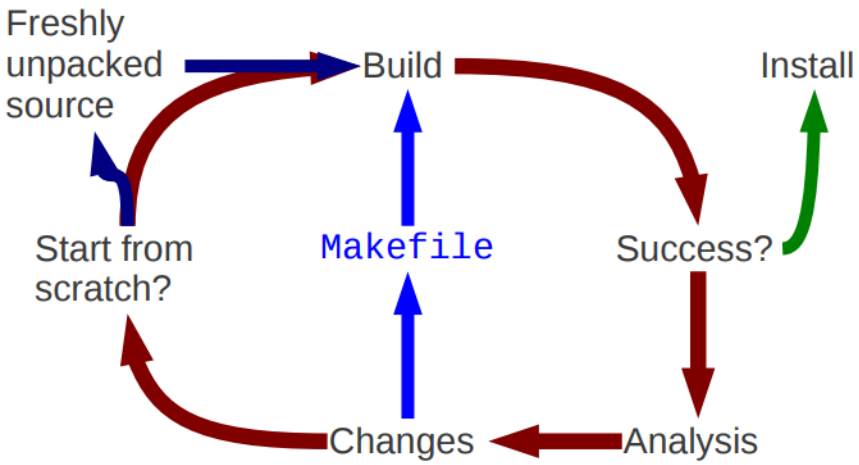


Figure 3.14. *Iterative Development*

3.4 *Case Study : VASP on Toubkal Cluster*

The Vienna Ab initio Simulation Package (VASP) is a licensed software used for atomic scale materials modelling, particularly electronic structure calculations and quantum-mechanical molecular dynamics. The Toubkal administrators obtained the necessary license and provided the source code for VASP.

Inside the VASP directory, as shown in Figure 3.15, there is no configure script typically used for setting up the build process. However, a Makefile is present, suggesting that the

software is built using make. Although a README file exists, it does not contain any build instructions.

```
$ ls
arch  bin  makefile  README_ELPHON.md  README.md  src  testsuite  tools
```

Figure 3.15. *Contents of VASP Directory*

The arch directory contains several makefile.include files tailored for different compilers, as illustrated in Figure 3.16.

```
$ ls arch/
makefile.include.aocc_ompi_aocl          makefile.include.gnu_omp
makefile.include.aocc_ompi_aocl_omp      makefile.include.gnu_ompi_aocl
makefile.include.fujitsu_a64fx           makefile.include.gnu_ompi_aocl_omp
makefile.include.fujitsu_a64fx_omp       makefile.include.gnu_ompi_mkl_omp
makefile.include.gnu                     makefile.include.intel
makefile.include.intel_omp               makefile.include.intel_ompi_mkl_omp
makefile.include.intel_serial            makefile.include.nec_aurora
makefile.include.nvhpc                   makefile.include.nvhpc_acc
makefile.include.nvhpc_omp               makefile.include.nvhpc_omp_acc
makefile.include.nvhpc_ompi_mkl_omp      makefile.include.nvhpc_ompi_mkl_omp_acc
makefile.include.oneapi                  makefile.include.oneapi_omp
makefile.include.oneapi_omp_off
```

Figure 3.16. *Contents of Arch Directory*

Those following steps describe setting up and running VASP on the NVIDIA H100 GPU, from installing the NVIDIA HPC SDK to building and running VASP.

3.4.1 Installing NVIDIA HPC SDK

To install VASP using the NVIDIA HPC SDK, first download the installation package from the NVIDIA developer website using the `wget` command. Next, extract the downloaded archive with `tar -xvf`. After extraction, navigate to the created directory and execute the `./install` script to begin the installation process, as illustrated in Figure 3.17.

```
$ wget https://developer.download.nvidia.com/hpcsdk/22.11/nvhpc_2022_2211_Linux_x86_64_cuda_multi.tar.gz
$ tar -xvf nvhpc_2022_2211_Linux_x86_64_cuda_multi.tar.gz
$ cd nvhpc_2022_2211_Linux_x86_64_cuda_multi
$ ./install
```

Figure 3.17. *Installing NVIDIA HPC SDK*

3.4.2 Configuring the Environment

Add your NVHPC module path to the `$MODULEPATH` environment variable in your `.bashrc`. Then, reload your environment and check the module availability, as illustrated in Figure 3.18.

```
$ source ~/.bashrc
$ module avail nvhpc
```

Figure 3.18. *Module Availibilty Check*

3.4.3 Allocating H100 GPU Node

Request a compute node equipped with an H100 GPU through your cluster's resource manager (e.g., `salloc`).

3.4.4 Loading Required Modules

Before building VASP, ensure the appropriate software environment is loaded. First, clear any previously loaded modules using `module purge`. Then, load the NVIDIA HPC SDK version 22.11, followed by the MPI-enabled FFTW library version 3.3.10, as shown in Figure 3.19.

```
$ module purge
$ module load gpu_h100/nvhpc/22.11
$ module load FFTW.MPI/3.3.10-gompi-2023b
```

Figure 3.19. *Required Modules for VASP*

3.4.5 Building VASP

Navigate to the VASP source directory and copy the appropriate makefile include file depending on your GPU architecture. In our case, for the `makefile.include` file we will set `-gpu=cc90` for `CC`, `FC`, and `FCL`, and the CUDA version to 11.8. Next, set the required environment variables

(replacing `/path/to/nvhpc` accordingly). Finally, compile VASP and run the testsuite, as shown in Figure 3.20.

```
$ export FFTW_ROOT=/srv/software/easybuild/software/FFTW.MPI/3.3.10-gompi-2023b
$ export PATH=/path/to/nvhpc/comm_libs/mpi/bin:$PATH
$ export LD_LIBRARY_PATH=/path/to/nvhpc/comm_libs/mpi/lib:$LD_LIBRARY_PATH
$ make DEPS=1 all
$ make test
```

Figure 3.20. *Environement Variables, Building, and Running Testsuite*

3.4.6 Public Deployment via Modulefile

The following modulefile is used to deploy VASP 6.5.0 built with NVIDIA HPC SDK 22.11 for H100 GPUs publicly on the cluster. This allows users to load the VASP environment easily with all necessary dependencies, as illustrated in Figure 3.21:

```
#%Module1.0#####################################################################
##
## VASP 6.5.0 built with NVIDIA HPC SDK for H100
##
proc ModulesHelp { } {
   puts stderr "This module loads VASP 6.5.0 (GPU build for H100 with NVHPC 22.11 and FFTW)."
}
module-whatis "VASP 6.5.0 (GPU build for H100 using NVHPC 22.11 and FFTW 3.3.10)"
# Purge any conflicting modules and load required dependencies
module load gpu_h100/nvhpc/22.11
module load FFTW.MPI/3.3.10-gompi-2023b
# Set VASP binary directory
set vasp_root /srv/software/gpu_h100/vasp.6.5.0
# NVHPC MPI paths
set nvhpc_mpi /srv/software/gpu_h100/NVHPC/Linux_x86_64/22.11/comm_libs/mpi
# Add VASP and NVHPC MPI to environment paths
prepend-path PATH            $vasp_root/bin
prepend-path PATH            $nvhpc_mpi/bin
prepend-path LD_LIBRARY_PATH   $nvhpc_mpi/lib
```

Figure 3.21. *Modulefile for VASP*

3.4.7 Notes on VASP Executables

After successfully building VASP, a bin directory is created inside the top-level directory of the VASP source tree. This directory contains three executables corresponding to the different software configurations: vasp_std, vasp_gam, and vasp_ncl, as shown in Figure 3.22.

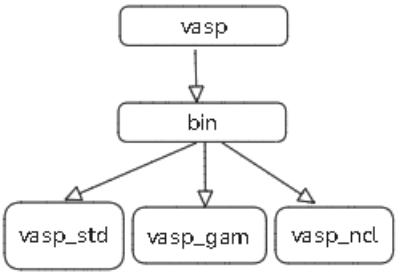


Figure 3.22. *Bin Directory Content*

- `vasp_std`: Standard VASP executable.
- `vasp_gam`: Gamma-point-only calculations.
- `vasp_ncl`: Non-collinear magnetic calculations.

3.4.8 Using VASP as an End User

To use VASP as an end user, first allocate an H100 GPU node using your cluster scheduler and load the appropriate VASP module. Prepare the required input files (`INCAR`, `POSCAR`, `POTCAR`, and `KPOINTS`), then create a test directory, navigate into it, and run VASP with MPI, selecting the executable based on the type of calculation. The sequence of commands is as follows in Figure 3.23:

```
$ module load vasp/6.5.0-gpu_h100
$ mkdir vasp_test
$ cd vasp_test/
$ mpirun -np 1 vasp_std
```

Figure 3.23. *Using VASP*

3.5 *Conclusion*

In this chapter, we presented the architecture of the Toubkal supercomputer, along with its hardware and software ecosystem. The installation and environment setup of VASP were described in detail, including directory structures and `modulefile` creation. Similar procedures were applied to other scientific software packages, such as KNN_CUDA and Quantum ESPRESSO.

While each software has specific configuration requirements, we adopted a workflow that is widely used in most HPC environments, consisting of organized versioning, source compilation, and module-based environment management. Following this established approach ensures reproducibility and simplifies the deployment and use of computational software across the HPC ecosystem.

Chapter

4 HPC Benchmarking

4.1 *Introduction*

In this chapter, we present the benchmarks executed on the Toubkal cluster. These benchmarks target various subsystems of the Toubkal supercomputer, including the network, storage, and compute components. The focus of this chapter is on the analysis and results of the benchmarks including the overall process of executing the benchmark.

4.2 *NVIDIA NGC Catalog*

The NGC catalog offers a wide range of GPU-optimized containers designed for AI, machine learning, and HPC workloads. These containers are pre-tested and ready to run on supported NVIDIA GPUs, whether deployed on-premises, in the cloud, or at the edge. In addition to containers, NGC provides pretrained models, model scripts, and complete industry solutions that can be easily incorporated into existing pipelines.

Each container comes with a pre-configured stack of GPU-accelerated software, including the selected application or framework, the NVIDIA CUDA® Toolkit, optimized libraries, and necessary drivers. All components are tested and tuned to work seamlessly together, requiring no further setup [18].

For our benchmarking study, we utilize the NVIDIA HPC Benchmarking container from NGC,

which contains optimized implementations of HPL and STREAM. Leveraging this container enables us to concentrate on performance evaluation and system tuning while maintaining consistency across various GPU platforms.

4.3 *Singularity*

Singularity is a container platform designed specifically for High-Performance Computing (HPC) environments. It enables the creation and execution of containers that package software in a portable and reproducible way. A key advantage is that a Singularity container is built as a single file, making it easy to move and run across heterogeneous environments, whether on a laptop, a workstation, a university cluster, a cloud platform, or even the world's largest supercomputers, without needing to reinstall or reconfigure software for each operating system. Unlike Docker, which requires root privileges and a daemon to run (posing security risks on multi-user HPC systems), Singularity integrates natively with shared HPC infrastructures by running containers without elevated privileges. This design guarantees that the same containerized application behaves consistently across different systems, aligning with the reproducibility and security requirements of HPC centers.

Originally developed at Lawrence Berkeley National Laboratory, Singularity rapidly spread across HPC and academic sites and is now widely adopted in both research and industry. Its open-source community continues to grow, ensuring ongoing improvements and support.

Figure 4.1. *Singularity [19]*

Now, we will explore the contents of the container by first pulling the appropriate image version 13.10. Then, after allocating either H100 or A100 GPU using `salloc`, we will run the container, navigate to the workspace directory, and list its contents, as shown in Figure 4.2. This directory includes several NVIDIA benchmark files and executables, among which we will focus on the HPL and STREAM benchmarks to evaluate computational performance and memory bandwidth.

```
$ singularity pull docker://nvcr.io/nvidia/hpc-benchmarks:23.10
$ singularity run --nv hpc-benchmarks_23.10.sif
$ cd /workspace/
$ ls
NVIDIA_Deep_Learning_Container_License.pdf  hpcg-linux-x86_64  hpcg.sh  hpl-linux-x86_64  hpl-mxp-linux-
x86_64  hpl-mxp.sh  hpl.sh  stream-gpu-linux-x86_64  stream-gpu-test.sh  third_party.txt
```

Figure 4.2. *Bechnmarks' Container Content*

`Important Note: All the benchmarks were run on a single GPU node.`

4.4 *HPL Benchmark*

HPL is a fundamental benchmark in HPC that evaluates a system's floating-point computational performance. It is widely used to assess the efficiency of both CPUs and GPUs when solving large dense linear systems. HPL not only measures raw performance in terms of gigaflops or teraflops, but it also helps to understand how well the system handles parallel computation, memory access patterns, and communication overhead. This makes it an essential tool for performance tuning and system comparison in HPC environments.

4.4.1 Characteristics of the HPL Benchmark

High-Performance Linpack (HPL) is a benchmark designed to measure the computational speed of systems when solving a dense system of linear equations. It is one of the most widely used benchmarks for evaluating the performance of high-performance computing (HPC) systems and serves as a key metric for ranking supercomputers in the TOP500 list.

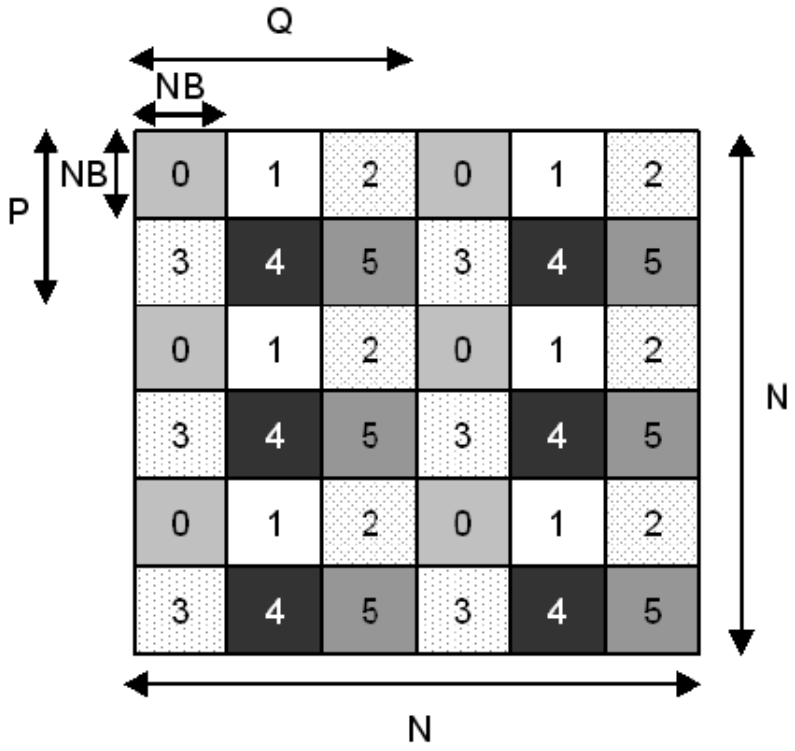


Figure 4.3. *HPL Parameters [20]*

Figure 4.3 displays the parameters of the HPL benchmark. `N` represents the global matrix size, and `NB` denotes the block size. These blocks form the basic unit of computation and are used to decompose the matrix into smaller tiles. The distribution is done in a 1D block-cyclic manner, which ensures a balanced workload among processes and minimizes communication overhead. The parameters `P` and `Q` define the dimensions of the process grid, where each MPI process is assigned multiple blocks based on its coordinates in the grid. For optimal performance, the product `P` $\times$ `Q` should match the total number of MPI processes, and `P` and `Q` should be chosen to be as balanced as possible.

4.4.2 HPL Benchmark Input File

The High-Performance Linpack (HPL) benchmark requires a configuration file, typically named `HPL.dat`, which specifies execution parameters such as matrix sizes, block sizes, process grids, and algorithmic options. This plain-text file has the extension `.dat` and controls the benchmark execution. An example is shown in Figure 4.4.

```
1  HPLinpack benchmark input file
2  Innovative Computing Laboratory, University of Tennessee and Frankfurt Institute for Advanced Studies
3  HPL.out      output file name (if any)
4  6            device out (6=stdout,7=stderr,file)
5  5        # of problems sizes (N)
6  20000 40000 60000 80000 100000       Ns
7  1            # of NBs
8  576           NBs
9  1            PMAP process mapping (0=Row-,1=Column-major)
10 1            # of process grids (P x Q)
11 2            Ps
12 1            Qs
13 0.1          threshold
14 1            # of panel fact
15 1            PFACTs (0=left, 1=Crout, 2=Right)
16 1            # of recursive stopping criterium
17 64           NBMINs (>= 1)
18 1            # of panels in recursion
19 2            NDIVs
20 1            # of recursive panel fact.
21 0            RFACTs (0=left, 1=Crout, 2=Right)
22 1            # of broadcast
23 6            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM,6=MPI)
24 1            # of lookahead options
25 3            LOOKAHEADs (enable = 1)
26 8            memory alignment in double (> 0)
27 100           Seed for the matrix generation
```

Figure 4.4. *HPL.dat [21]*

The main parameters of `HPL.dat` are:

- **Header (Lines 1 to 2):** Informational text, usually left as default.
- **Output (Lines 3 to 4):** Defines output file (e.g., `HPL.out`) and destination (stdout, stderr, or file).
- **Problem Sizes (Lines 5 to 6):** Number and values of matrix sizes to test (e.g., 3000, 6000, 10000).
- **Block Sizes (Lines 7 to 8):** Number and values of block sizes (e.g., 80, 100, 160).
- **Process Grid (Lines 9 to 11):** Defines row/column process mapping and grid dimensions ($P \times Q$).
- **Threshold (Line 13):** Residual check tolerance, typically set to 16.0 or negative to bypass.

- **Algorithmic Parameters (Lines 14+):** Options for panel factorizations (PFACT, RFACT), recursion depth (NDIV), stopping criteria (NBMIN), and broadcast methods.
- **Lookahead (Line 15):** Enables overlapping of computation and communication.
- **Memory Alignment (Line 16):** Boundary alignment in doubles (e.g., 8 = 64 bytes).
- **Random Seed (Line 17):** Ensures reproducibility of generated matrices.

In practice, HPL explores all combinations of these parameters to identify the optimal setup for a given system.

4.4.3 Execution Steps

To run the HPL (High Performance Linpack) benchmark, follow these steps: First, navigate to your desired working directory (replace `/path/to/working/directory` with the actual location). Create a new directory named `data` to store benchmark-related files, and move into this newly created directory. Then, create an empty `HPL.dat` file, which will store the configuration parameters for the benchmark, and open it using a text editor (e.g., Vim) to modify and define the necessary parameters.

```
$ cd /path/to/working/directory
$ mkdir data
$ cd data/
$ touch HPL.dat
$ vim HPL.dat
```

Figure 4.5. *HPL Step 1 Execution*

After setting up the `HPL.dat` file, we proceed with executing the benchmark using the appropriate configurations. The `HPL.dat` file contains key parameters that define the execution settings of the benchmark, including problem sizes, block sizes, process grid dimensions, and algorithmic parameters. These configurations must be carefully selected based on the available hardware resources to optimize performance.

Next, bind the local directory on the host machine to a directory inside the Singularity container. This ensures that files like `HPL.dat`, which reside on the host system, are accessible within the container. The `pwd` command prints the current working directory, which is assumed to contain the necessary data file (`HPL.dat`). Use the `singularity exec` command with the `-bind` flag to bind the local directory to the container's `/mnt` directory. You can then display the contents of `/mnt/HPL.dat` inside the container to make sure of the successfull binding.

```
$ singularity exec --bind $(pwd):/mnt ./hpc-benchmarks_23.10.sif cat /mnt/HPL.dat
```

Figure 4.6. *HPL Step 2 Execution*

If the session is closed, the bind must be re-executed each time; to avoid this, you can add the bind command to your `.bashrc` file, which runs automatically for new terminal sessions using the environment variable `$SINGULARITY_BINDPATH`.

```
$ echo export SINGULARITY_BINDPATH=/home/karim.bezine-ext/lustre/sw_stack-
373lcd9r8io/users/karim.bezine-ext/src/data:/mnt >> ~/.bashrc
$ source ~/.bashrc
```

Figure 4.7. *HPL Step 3 Execution*

Before running any GPU-accelerated benchmarks, allocate GPU resources. This ensures that the system reserves a node with a GPU available for your benchmark. You can verify the GPU allocation by checking the NVIDIA driver status.

Finally, navigate to the directory where the HPL benchmark files are located inside the container after the successful binding. Run the benchmark using MPI, specifying the number of processes (e.g., `-np 1` for a single process) and the location of the `HPL.dat` configuration file. Once executed, the benchmark begins its HPL test, performing matrix multiplications to measure the system's floating-point performance.

```
$ singularity run --nv hpc-benchmarks_23.10.sif
$ cd /workspace/hpl-linux-x86_64/
$ mpirun -np 1 hpl.sh --dat /mnt/HPL.dat --gpu-affinity 2 --no-multinode --cuda-compat
```

Figure 4.8. *HPL Step 4 Execution*

- **mpirun -np 1**: Runs an MPI process with a single instance.
- **hpl.sh**: The script that launches the HPL-NVIDIA benchmark.
- **–dat /mnt/HPL.dat**: Specifies the path to the HPL.dat configuration file.
- **–gpu-affinity 0**: Ensures execution runs on GPU 0.
- **–no-multinode**: Disables multi-node execution for a single-node setup.
- **–cuda-compat**: Enables CUDA forward compatibility for running binaries compiled with older CUDA versions.

4.4.4 A100 GPU Partition

a) Single GPU, Single MPI Process

To run the HPL benchmark on a single GPU, we allocate a GPU node with one task using the following `salloc` command:

```
salloc -t 4:00:00 -n 1 --ntasks=1 -A sw_stack-373lcd9r8io-default-gpu
-p gpu --gres=gpu:1
```

Each part of the command has a specific role:

- `salloc`: Allocates resources for an interactive Slurm job.
- `-t 4:00:00`: Sets the maximum job runtime to 4 hours.
- `-n 1`: Requests 1 node for the job.

- `-ntasks=1`: Requests 1 MPI task, ensuring a single process.
- `-A sw_stack-373lcd9r8io-default-gpu`: Specifies the account/project to charge the job to.
- `-p gpu`: Requests the job to run in the `gpu` partition (group of A100 GPU nodes).
- `-gres=gpu:1`: Requests 1 GPU on the allocated node.

This ensures that the benchmark runs on a single A100 GPU on one node with one MPI process. Below is a table summarizing the results from five different test cases with varying problem sizes ('N'). The results include the time taken for each test and the achieved computation speed in Gflops (Billions of Floating-Point Operations Per Second):

This table demonstrates how the time taken and the computation speed change as the problem size increases. It is important to note that larger problem sizes (higher 'N' values) typically result in higher computation speeds, though they also take longer to process.

The HPL benchmark is optimized to use FP64 Tensor Cores by default when running on A100-SXM4-80GB GPUs. These specialized cores significantly boost double-precision floating-point performance, enabling higher efficiency in large-scale numerical computations.

For the NVIDIA A100 GPU, the theoretical peak for FP64 operations is calculated as follows:

$$\text{Theoretical Peak (GFLOPS)} = \text{Number of CUDA Cores} \times 2 \times \text{Boost Clock (GHz)}$$

- Number of CUDA cores: 6912 - Boost clock: 1.41 GHz

$$\text{Theoretical Peak} = 6912 \times 2 \times 1.41 \approx 19,487 \text{ GFLOPS} \approx 19.5 \text{ TFLOPS}$$

The efficiency of a benchmark run can be calculated as:

$$\text{Efficiency (\%)} = \frac{\text{Measured Gflops}}{\text{Theoretical Peak Gflops}} \times 100$$

Using the result for the largest problem size (‘N=100000‘) with a measured computation speed of 17,860 Gflops:

$$\text{Efficiency} = \frac{17,860}{19,487} \times 100 \approx 91.7\%$$

This confirms that the HPL benchmark is achieving a high fraction of the theoretical peak performance of the A100 GPU, indicating that the system is well-optimized for double-precision operations.

Table 4.1. *HPL Results on one A100 GPU*

Problem Size, N	Time Taken (s)	Computation Speed (Gflops)
20000	0.51	9,731
40000	1.63	15,890
60000	8.36	17,150
80000	19.01	17,600
100000	36.94	17,860

b) Two GPUs, Two MPI Processes, One MPI Process Each

To utilize two A100 GPUs efficiently, we allocate a GPU node with two GPUs and launch one MPI process per GPU. This setup allows the problem to be distributed evenly across both devices, maximizing throughput.

The Slurm command for this configuration is:

```
salloc -t 4:00:00 -n 1 --ntasks-per-gpu=1 --gres=gpu:2 \
-A sw_stack-373lcd9r8io-default-gpu -p gpu
```

Each MPI process is bound to a separate GPU, ensuring that both devices operate independently on their respective data partitions in VRAM. The performance measurements from the HPL benchmark in this configuration are shown in Table 4.2.

Table 4.2. *HPL results on two A100 GPUs*

Problem Size, N	Time Taken (s)	Computation Speed (Gflops)
20000	0.55	9,106
40000	1.66	25,230
60000	4.56	31,430
80000	9.98	33,560
100000	19.00	34,720

The combined theoretical FP64 peak for two A100 GPUs is simply twice the single-GPU peak:

$$\text{Theoretical Peak}_{\text{2xA100}}(\text{GFLOPS}) = 2 \times \text{CUDA Cores} \times 2 \times \text{Boost Clock (GHz)}.$$

Using the same device characteristics as before (6912 CUDA cores, 1.41 GHz boost):

$$\text{Theoretical Peak}_{\text{1xA100}} \approx 6912 \times 2 \times 1.41 \approx 19{,}487 \text{ GFLOPS},$$

$$\text{Theoretical Peak}_{\text{2xA100}} \approx 2 \times 19{,}487 = 38{,}974 \text{ GFLOPS } (\approx 39.0 \text{ TFLOPS}).$$

The efficiency for the largest problem size ($N = 100{,}000$) is then:

$$\text{Efficiency } (\%) = \frac{\text{Measured GFLOPS}}{\text{Theoretical Peak GFLOPS}} \times 100 = \frac{34{,}720}{38{,}974} \times 100 \approx 89.1\%.$$

For completeness, the speedup relative to the single-GPU case at $N = 100{,}000$ is:

$$\text{Speedup} = \frac{34{,}720}{17{,}860} \approx 1.95\times,$$

which corresponds to a parallel efficiency of:

$$\text{Parallel Efficiency} = \frac{1.95}{2} \times 100 \approx 97.3\%,$$

indicating excellent scaling with minimal inter-GPU overhead at this problem size. Compared to the single GPU configuration (Table 4.1), the two-GPU setup shows a significant increase in computation speed, particularly for larger problem sizes. For instance, at $N = 100{,}000$, performance increases from 17,860 Gflops on a single GPU to 34,720 Gflops on two GPUs — achieving approximately $1.94\times$ the throughput. This near-linear scaling demonstrates excellent parallel efficiency, indicating that the workload is well-balanced and that inter-GPU communication overhead is minimal in this configuration, as illustrated in the graphical result in Figure 4.9.

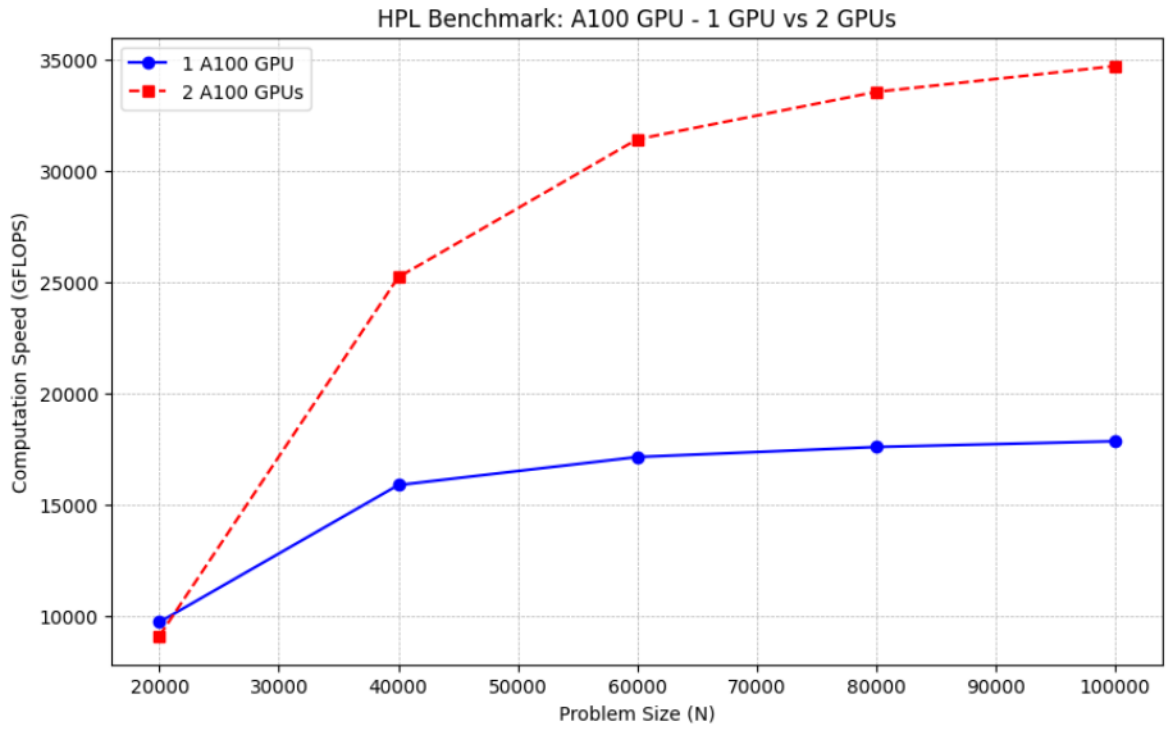


Figure 4.9. *HPL Results on A100*

It is observed that for the smallest problem size (N = 20,000), the single-GPU run performs slightly better than the two-GPU run. This occurs because the workload is too small to fully utilize multiple GPUs efficiently. When using two GPUs, additional time is required for data distribution and inter-GPU communication, which introduces overhead. For larger problem sizes, the computational workload dominates the communication cost, and the two-GPU configuration achieves significantly higher performance, demonstrating effective scaling.

4.4.5 H100 GPU Partition

a) Single GPU, Single MPI Process

To run the HPL benchmark on a single H100 GPU, we allocate a GPU node with one task using the following Slurm command:

```
salloc -t 4:00:00 -n 1 --ntasks=1 -A sw_stack-373lcd9r8io-default-gpu
-p gpu_h100 --gres=gpu:1
```

Each part of the command has a specific role:

- `salloc`: Allocates resources for an interactive Slurm job.
- `-t 4:00:00`: Sets the maximum job runtime to 4 hours.
- `-n 1`: Requests 1 total task for the job.
- `-ntasks=1`: Requests 1 MPI task, ensuring a single process.
- `-A sw_stack-373lcd9r8io-default-gpu`: Specifies the account/project to charge the job to.
- `-p gpu_h100`: Requests the job to run in the `gpu_h100` partition (group of H100 GPU nodes).
- `-gres=gpu:1`: Requests 1 GPU on the allocated node.

Below is a table summarizing the results from five different test cases with varying problem sizes (‘N‘). The results include the time taken for each test and the achieved computation speed in Gflops, as illustrated in Table 4.3:

Table 4.3. *HPL Results on one H100 GPU*

Problem Size, N	Time Taken (s)	Computation Speed (Gflops)
20000	0.31	16,130
40000	1.17	35,760
60000	3.43	41,760
80000	7.57	44,130
100000	11.43	45,110

The theoretical peak for FP64 operations on an NVIDIA H100 GPU is:

$$\text{Theoretical Peak (GFLOPS)} = \text{Number of CUDA Cores} \times 2 \times \text{Boost Clock (GHz)}$$

- Number of CUDA cores: 16,896 - Boost clock: 1.6 GHz

$$\text{Theoretical Peak} = 16,896 \times 2 \times 1.6 \approx 54,067 \text{ GFLOPS} \approx 54 \text{ TFLOPS}$$

Efficiency for the largest problem size (‘N=100000‘) is:

$$\text{Efficiency (\%)} = \frac{\text{Measured Gflops}}{\text{Theoretical Peak Gflops}} \times 100 = \frac{45,110}{54} \times 100 \approx 83.5\%$$

b) Two GPUs, Two MPI Processes, One MPI Process Each

To run the HPL benchmark on two H100 GPUs (one MPI process per GPU), we use the same command but request two GPUs and two tasks (value change of ntaks and gpu):

```
salloc -t 4:00:00 -n 1 --ntasks=2 -A sw_stack-373lcd9r8io-default-gpu
-p gpu_h100 --gres=gpu:2
```

All other options remain the same. The results for two GPUs are summarized below, as show in Table 4.4:

Table 4.4. *HPL Results on two H100 GPUs*

Problem Size, N	Time Taken (s)	Computation Speed (Gflops)
20000	0.40	11,510
40000	1.01	41,460
60000	1.11	64,700
80000	4.36	76,730
100000	7.95	81,970

The combined theoretical peak for two H100 GPUs is:

$$\text{Combined Peak} = 2 \times 54,067 \approx 108,134 \text{ GFLOPS}$$

Efficiency for the largest problem size (‘N=100000‘) is:

$$\text{Efficiency (\%)} = \frac{81,970}{108,134} \times 100 \approx 75.8\%$$

With two GPUs, performance reaches 81,970 GFLOPS, which is about 75.8% of the combined theoretical peak. Compared to the single-GPU case, this represents a speedup factor of 1.82×, resulting in a parallel efficiency of approximately 91%, which indicates good scaling with low inter-GPU communication overhead, as illustrated in the graphical result in Figure 4.10.

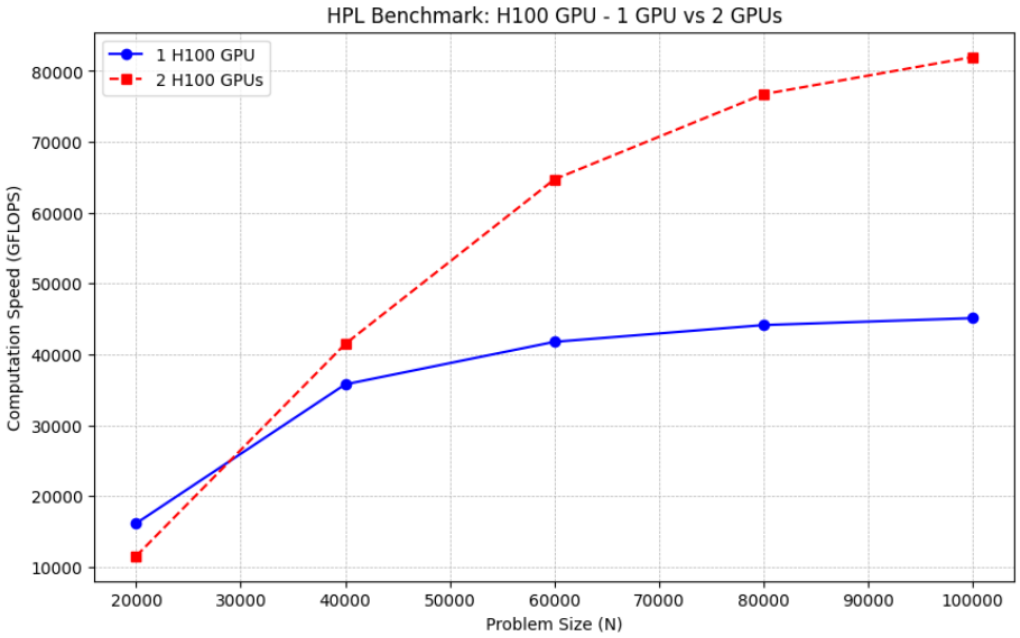


Figure 4.10. *HPL Results on H100*

4.4.6 Performance Analysis: A100 vs H100

Table 4.5. *HPL Results : A100 vs. H100*

Problem Size, N	A100 Computation Speed (Gflops)	H100 Computation Speed (Gflops)
20000	9,731	16,130
40000	15,890	35,760
60000	17,150	41,760
80000	17,600	44,130
100000	17,860	45,110

The results in Table 4.5 from the HPL benchmark clearly demonstrate the performance superiority of the NVIDIA H100 GPU compared to the A100 across all tested problem sizes.

- **Higher Gflops Throughput:** The H100 consistently delivers significantly higher Gflops than the A100, ranging from a 1.7× improvement at $N = 20{,}000$ to a 2.53× improvement at $N = 100{,}000$. This indicates that the H100 executes floating-point operations more efficiently with higher throughput.

- **Better Scalability:** The H100 shows better scalability with increasing problem size. While the A100's performance saturates around 17,800 Gflops, the H100 continues to scale linearly, achieving over 45,000 Gflops. This suggests that the H100 handles large-scale linear algebra computations with greater efficiency, making it ideal for HPC workloads.

- **Improved Architecture:** The performance improvements can be attributed to architectural advancements in the Hopper-based H100 over the Ampere-based A100:

  - *Tensor Core Enhancements:* Fourth-generation Tensor Cores in the H100 support a broader range of data types (including FP8) and offer higher throughput per SM compared to the A100's third-generation Tensor Cores.
  - *Increased SM Count and Clock Speeds:* More Streaming Multiprocessors (SMs) and higher clock frequencies increase parallelism and compute density.
  - *Transformer Engine:* Although not directly exploited in HPL, the Transformer Engine accelerates matrix-heavy operations, contributing to more efficient execution of large matrix solves.
  - *Memory Bandwidth and Latency Improvements:* Higher memory bandwidth reduces bottlenecks for large matrix data movement, critical in memory-bound operations like those in HPL.

- **Reduced Execution Time:** Execution times on the H100 remain low even for large problem sizes. For example, solving $N = 100{,}000$ takes only 11.43 seconds on a single GPU, significantly faster than the A100.

- **Energy Efficiency:** Although not explicitly measured, the H100 is expected to provide higher performance-per-watt compared to the A100, which is important for large-scale deployments where power efficiency is crucial.

- **Future-Proofing and Software Stack Optimization:** The H100 benefits from the latest NVIDIA software stack improvements, including CUDA 12+, optimized

cuBLAS libraries, and Hopper-specific tuning in frameworks like HPC-Bench. These enhancements further boost performance compared to legacy support for the A100.

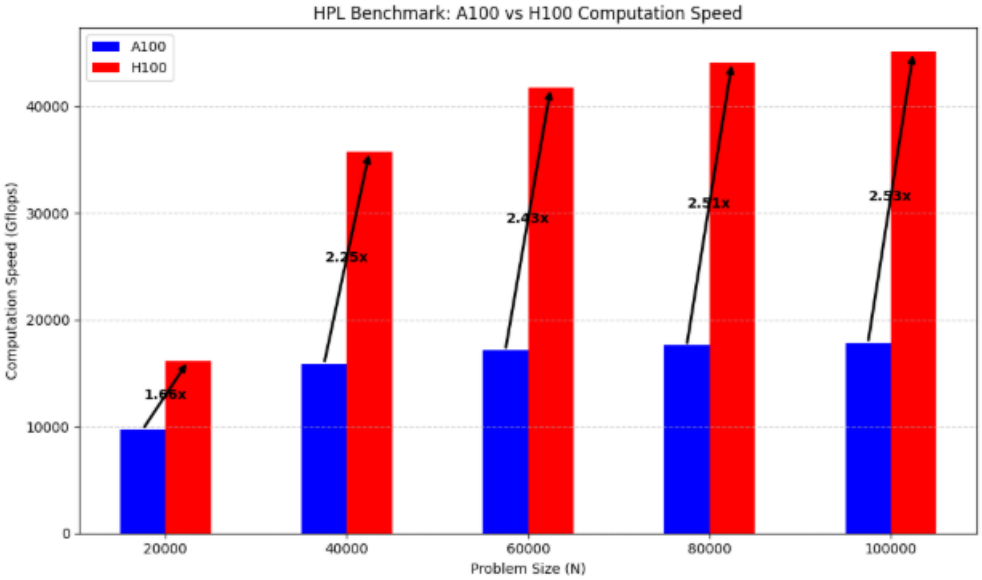


Figure 4.11. *HPL Results : A100 vs. H100*

4.5 *STREAM Benchmark*

The STREAM benchmark is widely used in HPC to evaluate the memory subsystem performance of a computing node. Unlike compute-intensive benchmarks such as HPL, STREAM focuses on the sustainable memory bandwidth and efficiency of basic vector operations. By measuring how quickly data can be read from and written to memory, STREAM provides insight into potential memory bottlenecks and helps in tuning applications that are memory-bound. This makes it a valuable tool for assessing the impact of memory hierarchy and optimizing system performance.

4.5.1 Characteristics of the STREAM Benchmark

The STREAM benchmark is a straightforward synthetic benchmarking tool designed to assess sustainable memory bandwidth in megabytes per second (MB/s) on a single node. It evaluates how effectively memory can be accessed and utilized under various workloads.

STREAM measures the computation rates associated with fundamental vector operations, such as vector addition, scalar multiplication, and dot products, as illustrated in Table 4.6.

Table 4.6. *Four Operations Used in the STREAM Benchmark*

Name	Kernel	Bytes/Iteration	FLOPs/Iteration
Copy	`a(i)=b(i)`	16	0
Scale	`a(i)=q` $\times$ `b(i)`	16	1
Sum	`a(i)=b(i)+c(i)`	16	1
Triad	`a(i)=b(i)+q` $\times$ `c(i)`	24	2

The benchmark thus provides valuable insights into both memory performance and processing efficiency, making it an essential tool for optimizing memory-related applications and understanding the computational capabilities of a system. Table 4.6 presents the four distinct operations `Copy, Scale, Sum, and Triad` along with the corresponding bytes required and FLOPs achieved per iteration. Notably, Triad stands out as the most complex operation, as it integrates the functionalities of copying, scaling, and summing.

The **Triad** operation involves two read operations (one for reading $b(i)$ and one for reading $c(i)$), and one write operation (to $a(i)$). Therefore, each iteration results in **24 bytes** of memory traffic: two reads (2×8 bytes) and one write (8 bytes), assuming double-precision floating-point numbers. The scalar value is stored in a GPU register and is not fetched from memory on each iteration. Only the array elements ($a[i]$, $b[i]$, and $c[i]$) contribute to memory traffic. Since the operation performs a multiplication and an addition, it results in **2 FLOPs (Floating Point Operations)** per iteration.

4.5.2 STREAM Benchmark Script Shell

The `stream-gpu-test.sh` script in Figure 4.12 is a wrapper for running the NVIDIA STREAM benchmark on GPUs, allowing the user to specify the problem size, target GPU device, and precision type from the command line.

```
#!/bin/bash
NUMBER_OF_ELEMENTS=""
DEVICE=""
EXEC_NAME="stream_test"
while ["$1" != ""]; do
 case $1 in
  --n )
     NUMBER_OF_ELEMENTS="-n$2"
     shift
     ;;
  --d )
     DEVICE="-d$2"
     shift
     ;;
  --dt )
     DATA_TYPE="$2"
     if ["$2" == "fp32"]; then
        EXEC_NAME="stream_test_fp32"
     fi
     shift
     ;;
 esac
 shift
done
echo "Command line: /workspace/stream-gpu-linux-x86_64/$EXEC_NAME $DEVICE $NUMBER_OF_ELEMENTS"
/workspace/stream-gpu-linux-x86_64/$EXEC_NAME $DEVICE $NUMBER_OF_ELEMENTS
```

Figure 4.12. *STREAM Benchmark Script Shell*

1. **Variable Initialization:**
   - `NUMBER_OF_ELEMENTS` – stores the number of elements for the benchmark array.
   - `DEVICE` – stores the target GPU device ID.
   - `EXEC_NAME` – stores the executable name; defaults to `stream_test` (double precision).
2. **Argument Parsing Loop:** The script uses a `while` loop to process all command-line arguments.
   - `-n <value>` sets `NUMBER_OF_ELEMENTS` in the format required by the executable (e.g., `-n1300000000`).
   - `-d <id>` sets the GPU device ID (e.g., `-d0` for device 0).
   - `-dt <type>` specifies the data type:
     - Default: double precision (`stream_test`)
     - If `fp32` is chosen: single precision (`stream_test_fp32`)
3. **Command Display:** Before execution, the script prints the full command to be run, e.g.:

```
Command line: /workspace/stream-gpu-linux-x86_64/stream_test -d0
-n1300000000
```

4. **Execution:** Finally, the script runs the STREAM benchmark with the selected options:

```
/workspace/stream-gpu-linux-x86_64/$EXEC_NAME $DEVICE $NUMBER_OF_ELEMENTS
```

Example Usage:

- Run on GPU 0, double precision:

```
./stream-gpu-test.sh --n 1300000000 --d 0
```

- Run on GPU 1, single precision:

```
./stream-gpu-test.sh --n 1300000000 --d 1 --dt fp32
```

This modular structure makes the script flexible for testing different GPUs and precision levels without modifying the benchmark executable directly.

4.5.3 Execution Steps

The execution procedure for the STREAM benchmark on A100 GPUs follows the same steps as described for the HPL benchmark, with the following differences:

- The `mpirun` command is executed from within the `/workspace/` directory inside the Singularity container.
- There is no need to export the `SINGULARITY_BINDPATH` environment variable for STREAM.

The command used to run the benchmark on one or two GPUs (one MPI process per GPU) is:

```
mpirun -np 2 bash -c './stream-gpu-test.sh --n 1300000000
--d $OMPI_COMM_WORLD_LOCAL_RANK'
```

In this command:

- `-n 1300000000` sets the number of elements in the benchmark arrays to $n = 1{,}300{,}000{,}000$.
- The `-np` argument sets the number of MPI processes. It can be set to 1 or 2:
  - If set to 1, the benchmark runs on a single GPU.
  - If set to 2, the benchmark runs on two GPUs, with one process per GPU. The measured performance results will be approximately twice those of a single process, since each GPU runs the same workload independently.
- These arrays are used for the Copy, Scale, Add, and Triad operations, with each element representing a data point on which the operation will be applied.
- For double-precision (FP64) operations, each element requires 8 bytes of memory.
- The size of each array is calculated as:

$$\text{Array Size (bytes)} = n \times 8 \times 8$$

  where:

  - The first 8 is the byte size of a single double-precision element.
  - The second 8 corresponds to the fact that each array element is stored as a vector of 8 double-precision numbers, making each vector 64 bytes.
- With $n = 1{,}300{,}000{,}000$, the size of each array is approximately 79,345 MB (roughly 80 GB), which fits within the available memory of an NVIDIA A100 80GB GPU.

`Important Note:` If the problem size n exceeds 1.3×10^9, the benchmark will encounter an out-of-memory error.

4.5.4 A100 GPU Partition

The STREAM benchmark was executed on a single NVIDIA A100-SXM4-80GB GPU using the procedure. Table 4.7 below summarizes the results for different STREAM operations: Copy, Scale, Add, and Triad.

Table 4.7. *STREAM Results on A100 GPU*

Function	Rate (MB/s)	Avg Time (s)	Min Time (s)	Max Time (s)
Copy	1,783,519.13	0.0117	0.0117	0.0117
Scale	1,790,751.93	0.0116	0.0116	0.0116
Add	1,805,341.67	0.0174	0.0173	0.0174
Triad	1,806,733.29	0.0174	0.0173	0.0174

These results indicate that the A100 GPU achieves memory bandwidths exceeding 1.7 TB/s for vector operations, demonstrating high efficiency for memory-bound computations.

4.5.5 H100 GPU Partition

The STREAM benchmark was also executed on a single NVIDIA H100 GPU using the same procedure. Table 4.8 presents the performance results.

Table 4.8. *STREAM Results on H100 GPU*

Function	Rate (MB/s)	Avg Time (s)	Min Time (s)	Max Time (s)
Copy	3,056,220.41	0.0068	0.0068	0.0068
Scale	3,073,925.53	0.0068	0.0068	0.0068
Add	3,132,077.26	0.0100	0.0100	0.0110
Triad	3,131,473.77	0.0100	0.0100	0.0100

The H100 GPU achieves bandwidths exceeding 3 TB/s, roughly doubling the performance of the A100. This demonstrates the architectural improvements in Hopper-based GPUs, including enhanced memory controllers and optimized data paths for memory-bound operations.

4.5.6 Performance Analysis: A100 vs H100

To provide a clear comparison of memory throughput between the A100 and H100 GPUs, Table 4.9 summarizes the STREAM benchmark results side by side, as shown in Table4.9.

Table 4.9. *STREAM Results: Comparison between A100 and H100 GPUs*

Function	A100 Rate (MB/s)	H100 Rate (MB/s)	Speedup (H100/A100)
Copy	1,783,519.13	3,056,220.41	1.71×
Scale	1,790,751.93	3,073,925.53	1.72×
Add	1,805,341.67	3,132,077.26	1.73×
Triad	1,806,733.29	3,131,473.77	1.73×

- **Higher Memory Bandwidth:** The H100 consistently achieves nearly double the memory throughput of the A100 for all STREAM operations.
- **Improved Latency:** The average execution times on the H100 are approximately 40–60% lower than the A100, demonstrating faster memory access.
- **Scalability:** Both GPUs show stable performance across Copy, Scale, Add, and Triad operations, confirming that memory-bound kernels scale predictably with problem size.
- **Architectural Advantage:** The performance gain is attributed to the H100's Hopper architecture, which features higher memory bandwidth, optimized caches, and enhanced data movement pipelines compared to the A100.

Overall, the STREAM benchmark results reinforce the H100's advantage in memory-intensive workloads, complementing the computational performance improvements observed in the HPL benchmark, as illustrated in Figure 4.13.

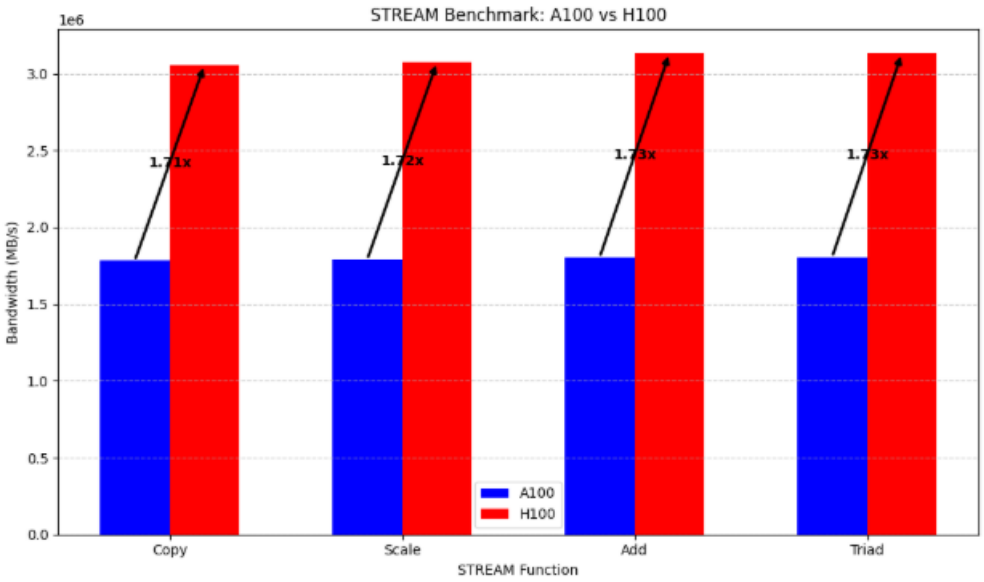


Figure 4.13. *STREAM Results : A100 vs. H100*

4.5.7 Conclusion

Benchmark results using the HPL and STREAM benchmarks confirm that the Toubkal Supercomputer infrastructure provides excellent performance and scalability for HPC workloads. Multi-GPU runs exhibit good scaling with minimal inter-GPU overhead, demonstrating that the system efficiently handles both computational and memory-intensive tasks. These results validate the effectiveness of the Toubkal cluster for demanding scientific and engineering applications.

Chapter 5

HPC Automation and Monitoring

5.1 *Introduction*

The previous chapter presented the manual execution of HPC benchmarks on the Toubkal cluster, providing insights into system performance across key applications and workloads. Building on this foundation, we will focus now on automating the benchmarking process using ReFrame and integrating it into a continuous pipeline.

In addition to automation, we will introduce a performance monitoring framework that continuously tracks system metrics, providing real-time insights into GPU utilization, memory usage, and execution efficiency. This chapter demonstrates how automated benchmarking and monitoring streamline performance evaluation, reduce human intervention, and enable reproducible, scalable, and systematic performance analysis across diverse workloads.

5.2 *Benchmark Integration to ReFrame*

Manual benchmarking typically involves multiple steps: compiling the source code, preparing the batch script for submission to the Slurm job scheduler, and validating results by reviewing output files. These steps are prone to human error and can be tedious.

ReFrame automates this process by providing Python-based classes and variables that define essential parameters such as Slurm directives, compiler flags, and regular expressions for metric

extraction. It can automatically check whether a metric's value falls within predefined upper and lower bounds passing the test if the condition is met, and rejecting it otherwise. This automation improves reproducibility, reduces human error, and accelerates benchmarking workflows.

5.2.1 Benchmark Integration

HPL, and STREAM benchmarks were integrated into the ReFrame framework. This section discusses the process of defining the different aspects of the ReFrame test code and configuration script for the HPL benchmark, while the other benchmarks follow the same steps.

a) ReFrame System Configuration

In the system configuration, a **system** named toubkal_cluster was defined. The module system was set up using Lmod, and two **partitions** were specified: `GPU A100`, and `GPU H100`. Each partition was configured with the Slurm job scheduler, and the launcher was selected according to the type of executable. In addition, three **programming environments** were defined: `CUDA`, `VASP A100`, and `VASP H100`. These environments, along with the partitions, are illustrated in Figure 5.1.

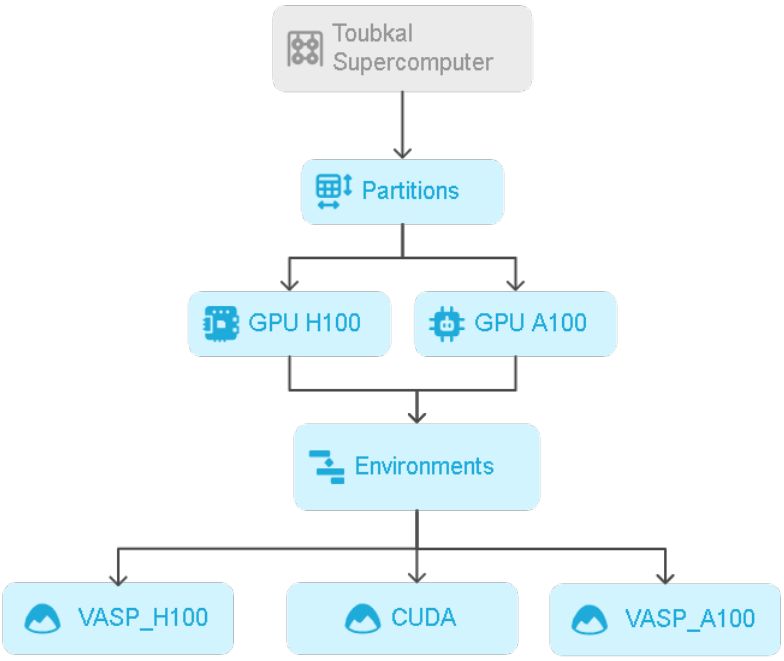


Figure 5.1. *Toubkal System: Partitions and Environments*

In each environment, the necessary modules required to execute the benchmarks are loaded, along with the appropriate compiler for building the source code. If a C source file is compiled, the `nvcc` compiler is used. When an environment is selected, ReFrame automatically loads the modules defined in the configuration. The defined environments are illustrated in Figure 5.2.

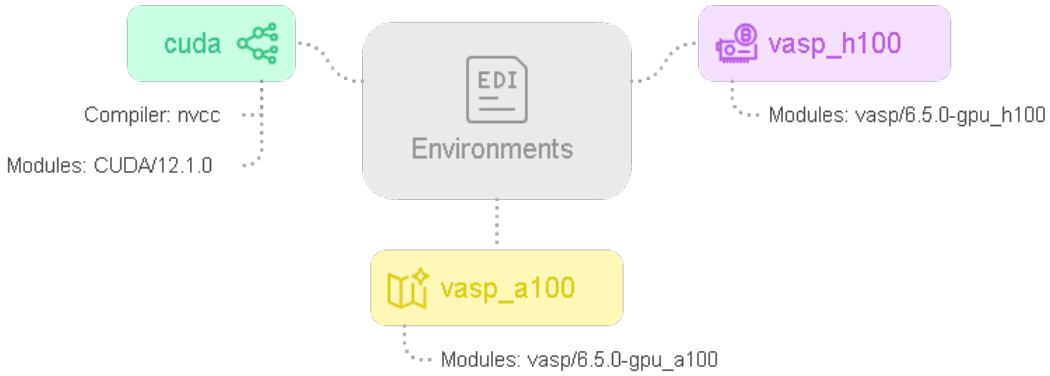


Figure 5.2. *Toubkal Environments*

b) ReFrame Test Code Walkthrough

The first step is to create a decorated class, where the decorator adds special behavior to the class within the ReFrame framework. Next, the valid systems and partitions for the test are specified, followed by the valid programming environments on which the test will run. The reference values for the benchmark are defined, the max pending time is defined and if it is exceeded the test will be classified as aborted and last but not least all paths that ReFrame will use them are mentioned. Finally, the executable type is defined as illustrated in Figure 5.3.

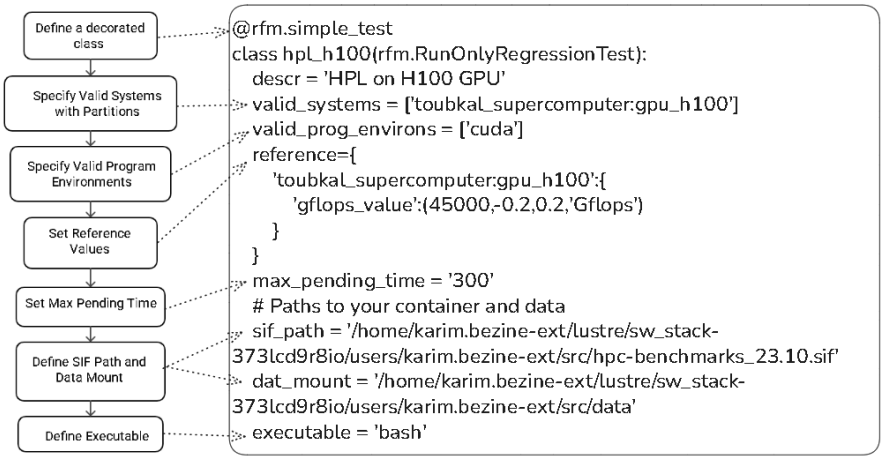


Figure 5.3. *ReFrame Test Configuration Flowchart*

Within the test definition, a method decorated with `@run_before('run')` is used to configure the executable options that will be executed when the benchmark runs. This block ensures that the environment and required resources are correctly set up before starting the HPL benchmark. Specifically, it first checks the allocation of the GPU using `nvidia-smi`, then verifies the proper binding of the data directory into the Singularity container by listing the benchmark configuration file `HPL.dat`. Afterward, the HPL benchmark is launched inside the container with `mpirun`, using one process and enabling GPU support while disabling multi-node execution. Finally, the output of the benchmark is captured in `HPL.out` and printed for inspection as shown in Figure 5.4.

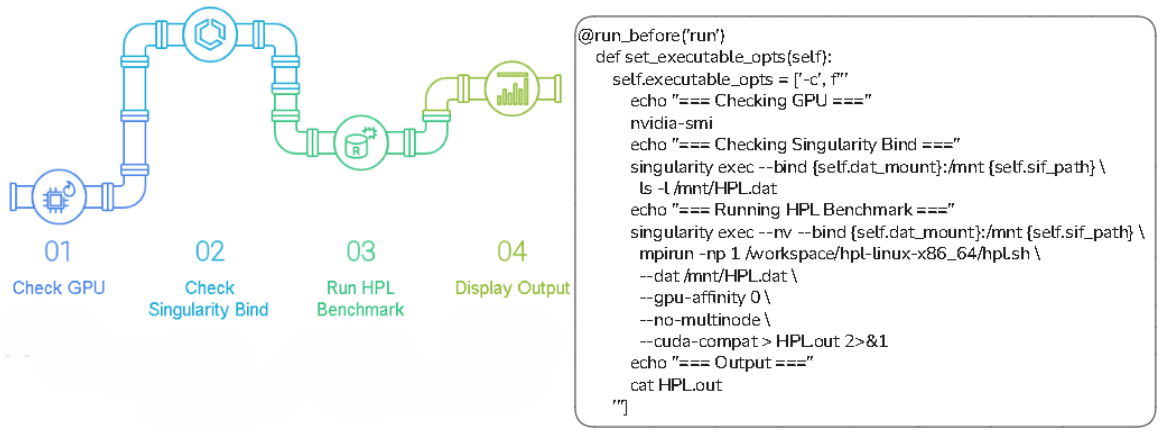


Figure 5.4. *Test Execution*

Within the test execution, this method sets the sanity patterns to verify the correctness of the HPL run. It checks the output file HPL.out for specific patterns, ensuring that the benchmark completed successfully, with no failed residual checks and no tests skipped due to invalid inputs as illustrated in Figure 5.5.

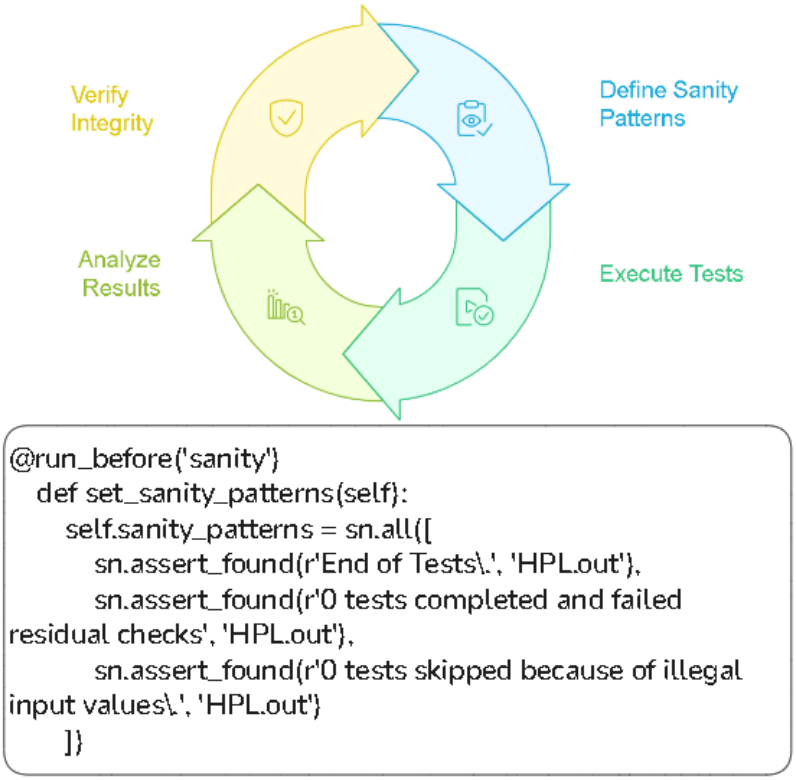


Figure 5.5. *Sanity Function*

The regular expression required to extract the GFLOPS value from the output file is defined, as shown in Figure 5.6.

Figure 5.6. *Extracting Performance Metrics*

5.2.2 Functional Validation Tests

Functional validation tests have been integrated into ReFrame, which involve:

- Validation that both the Lustre and Isilon file systems are properly mounted. This is verified using the `mount` command, with a regular expression defined to ensure that the output includes entries for both Lustre and Isilon.

  For example:

  - The Isilon file system is mounted from the network location `isilon:/HOME` is mounted on `/home`.
  - The Lustre file system is mounted from the servers `10.44.3.6@o2ib2:10.44.3.5@o2ib2:/lustre01` onto `/srv/lustre01`.

- Checking that the Slurm central daemon is up and running. The central management daemon of Slurm, `slurmctld`, monitors all other Slurm daemons and resources, accepts jobs, and allocates resources to them. Slurm is considered functional if this daemon is up and running, managing job submissions, scheduling, and coordinating compute nodes across the cluster.

  Additionally, the test verifies the correct behavior of essential Slurm commands:

  - `squeue -u <username>`: displays the list of jobs currently queued or running for a specific user.
  - `sacct`: provides detailed accounting information about completed jobs, such as job state, run time, and exit code.

- Ensuring that Lmod functions as expected by loading and unloading modules. The `module load` and `module unload` commands are used, and the `MODULEPATH` environment variable is checked to verify that Lmod can locate the installed software modules.

- Validating scientific application workloads on both A100 and H100 GPUs:

- **KNN_CUDA**: executed on A100 and H100 GPUs to confirm GPU compatibility and correctness.
- **Quantum ESPRESSO**: validated on both GPU architectures through the successful detection of the string `JOB DONE`.
- **VASP**: tested on A100 and H100 GPUs, where correctness is confirmed by the appearance of the string `reached required accuracy`.

Each of these tests leverages Singularity containers to ensure portability and reproducibility across environments, while ReFrame sanity functions check the expected output patterns to validate successful execution.

5.3 *Continuous Integration of HPC Benchmarks on the Toubkal Cluster*

Since Jenkins could not be installed directly on the login nodes due to lack of user privileges and for reasons of separation of concerns, the Toubkal administrators provided a dedicated virtual machine. On this virtual machine, Jenkins was deployed as a docker container to ensure portability and maintainability.

As illustrated in Figure 5.7, Jenkins runs on this virtual machine, while a Jenkins agent is configured on an entry node of the Toubkal supercomputer. This entry node, which can be either a login node or a compute node, will be discussed in more detail later regarding its selection.

The entry node is configured as a Jenkins node, responsible for executing build jobs delegated by the Jenkins controller. To enable this connection, the SSH agent method is used, allowing Jenkins to connect securely over SSH and run commands remotely on the entry node.

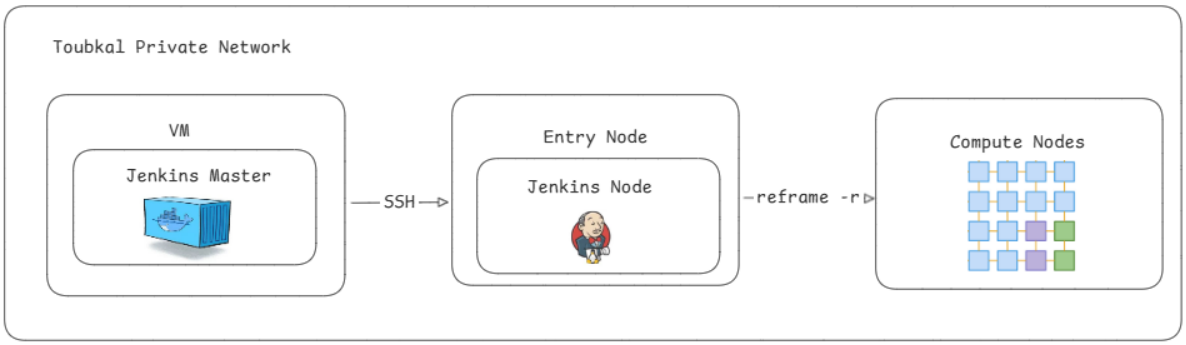


Figure 5.7. *Jenkins Workflow*

During testing, the following observations were made:

- When the entry node is a login node, SSH authentication requires both a password and a TOTP code, making automated access by Jenkins impractical, as Jenkins does not support two-factor SSH authentication.
- When the entry node is a compute node, Jenkins can only SSH into it if there is an active job already running on that node.

For the proof of concept, a compute node was used as the entry point, as Jenkins could not be configured for two-factor authentication.

For a production environment, it is recommended to use a login node as the Jenkins entry point. To support this setup, the login node should be configured to allow Jenkins to SSH into it with appropriate handling of TOTP-based authentication; for example, by bypassing TOTP for a specific Jenkins user. Figure 5.8 shows the configuration of a compute node set up as a Jenkins node. It includes the node description, number of executors, remote root directory, and labels. The labels ensure that jobs executed on the remote node are only those that match the assigned label.

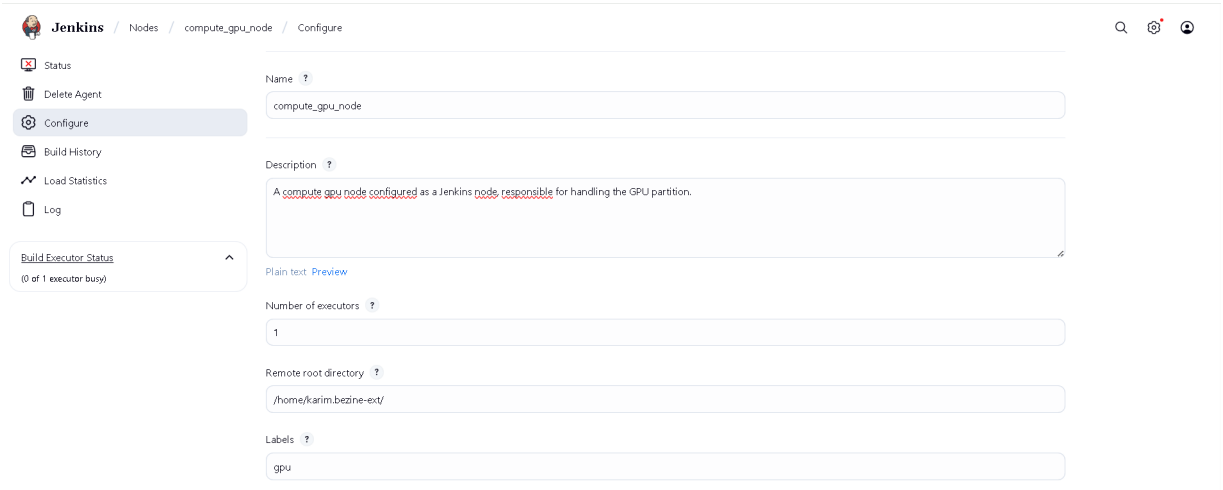


Figure 5.8. *Configuration of the Jenkins Compute Node*

Figure 5.9 illustrates how the Jenkins agent can access the Jenkins node. The launch method is configured to **Launch agents via SSH**, allowing secure communication between the master and the compute node.

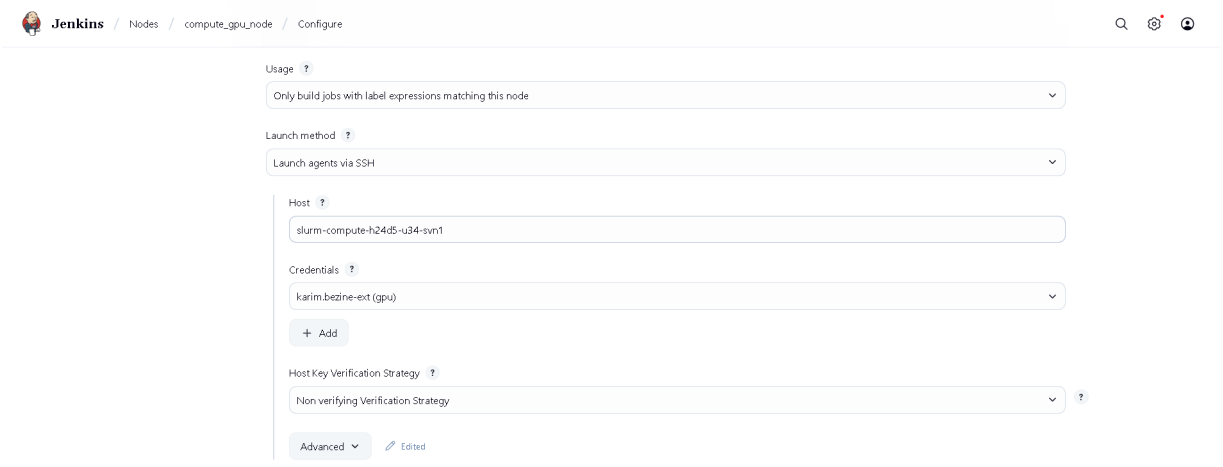


Figure 5.9. *Jenkins Agent Access Configuration for the Compute Node*

Figure 5.10 shows the Jenkins pipelines that were set up. We created a dedicated pipeline for each benchmark: STREAM, and HPL.

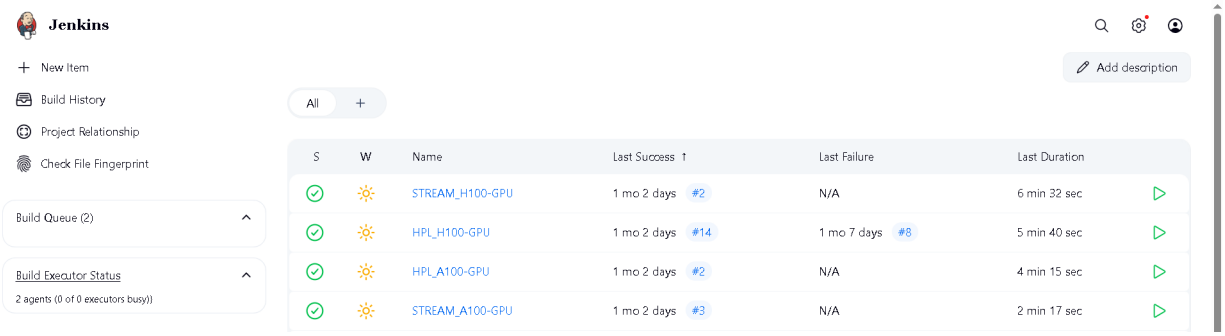


Figure 5.10. *Jenkins Pipelines*

As illustrated in Figure 5.11, we configured the trigger of the Jenkins agent using the expression `H 0 * * *`. This means that each pipeline is scheduled to run once per day at midnight. The letter `H` is specific to Jenkins and represents a hash value calculated from the job name, ensuring that jobs are evenly distributed to avoid running all at the exact same minute.

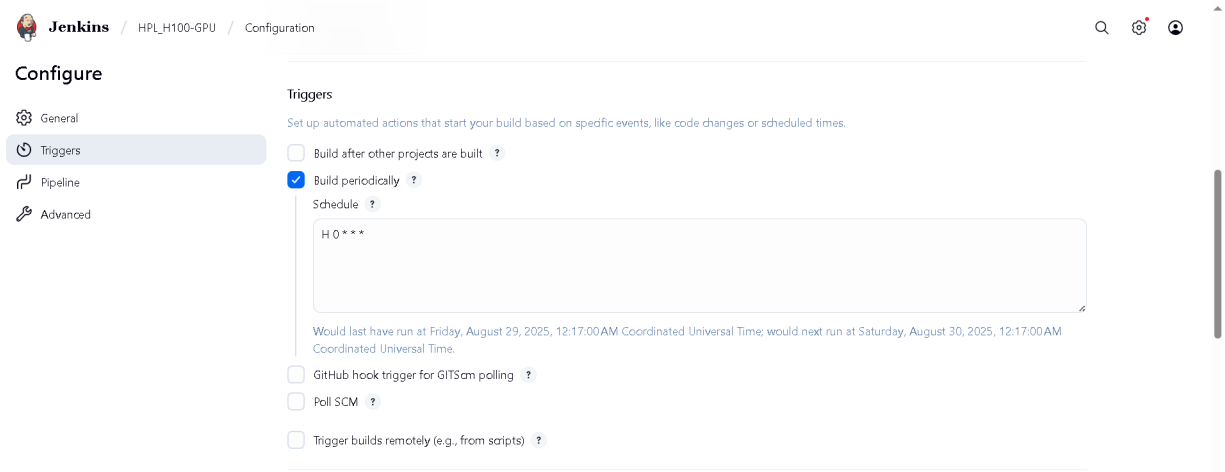


Figure 5.11. *Jenkins Pipeline Trigger*

Figure 5.12 shows the pipeline overview of the HPL benchmark pipeline. The pipeline consists of two main stages: one stage executes the benchmark, while the other stage copies the output generated by ReFrame to the virtual machine.

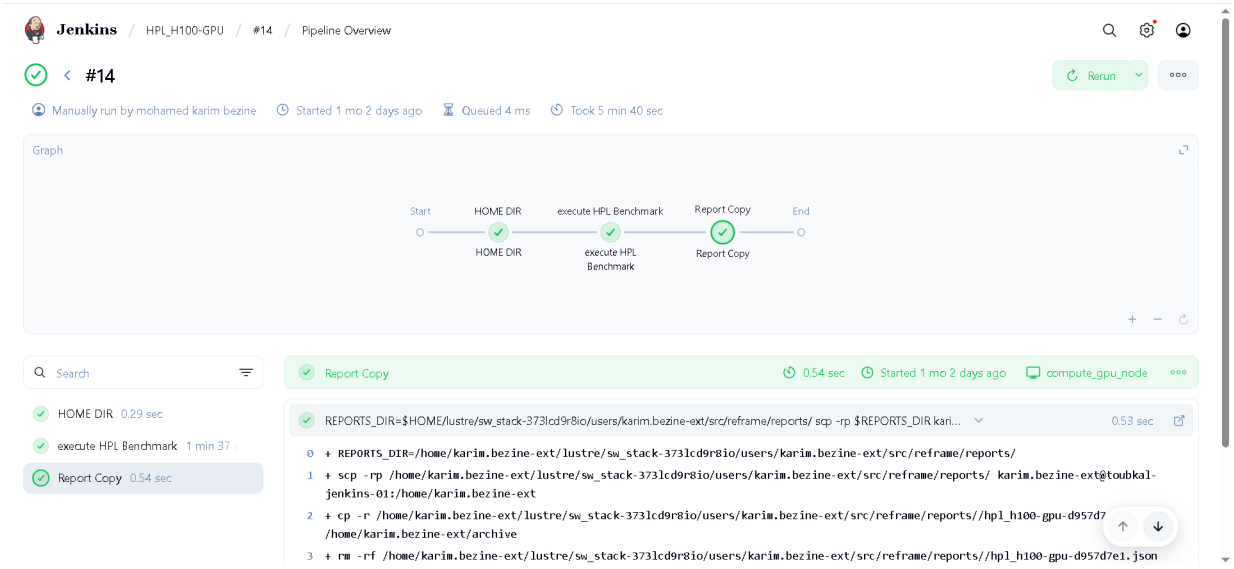


Figure 5.12. *Jenkins Pipeline Overview*

5.4 *Monitoring on Toubkal Supercomputer*

In this section, we present our work on log-based monitoring and agent-based monitoring on the Toubkal Supercomputer. The Elastic Stack, Prometheus, and Grafana were also deployed on the VM prepared by the Toubkal admins.

5.4.1 Log-Based Monitoring

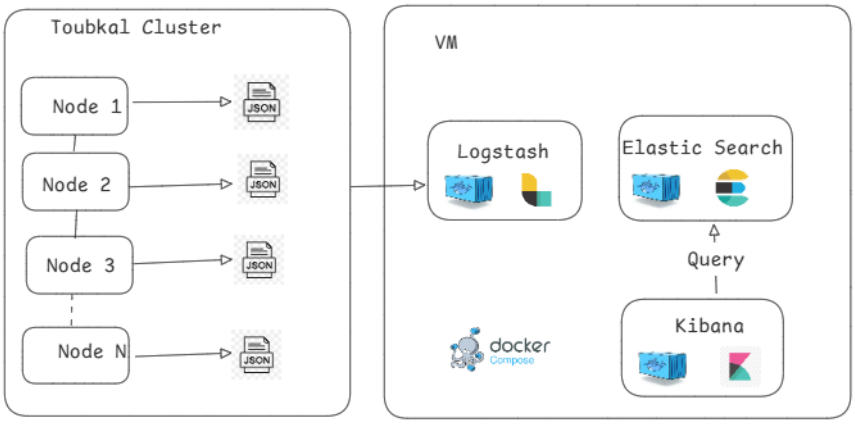


Figure 5.13. *Log Monitoring Architecture using ELK in the Toubkal Cluster*

As shown in Figure 5.13, the architecture of the log monitoring system is based on the ELK Stack, which includes Elasticsearch, Logstash, and Kibana. After ReFrame generates the output files, they are copied into a directory monitored by Logstash, which parses their contents and forwards the structured data to Elasticsearch, a NoSQL database optimized for search and analytics. The benchmark results are then visualized using Kibana dashboards, providing a clear and interactive way to monitor system performance over time.

Once Logstash sends the data to Elasticsearch, the indices are created with the naming convention: `benchmark-nametest-year.month.day`, as shown in Figure 5.14.

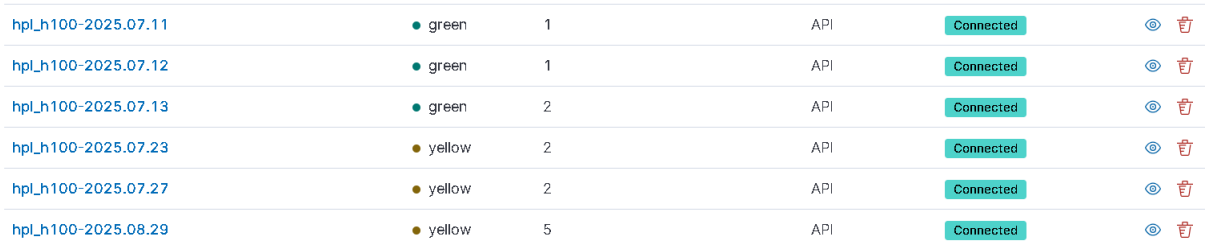


Figure 5.14. *Elasticsearch Indices Created for Benchmark Results*

In addition, a dedicated data view was created for each benchmark, configured to match all indices corresponding to that benchmark, as illustrated in Figure 5.15.

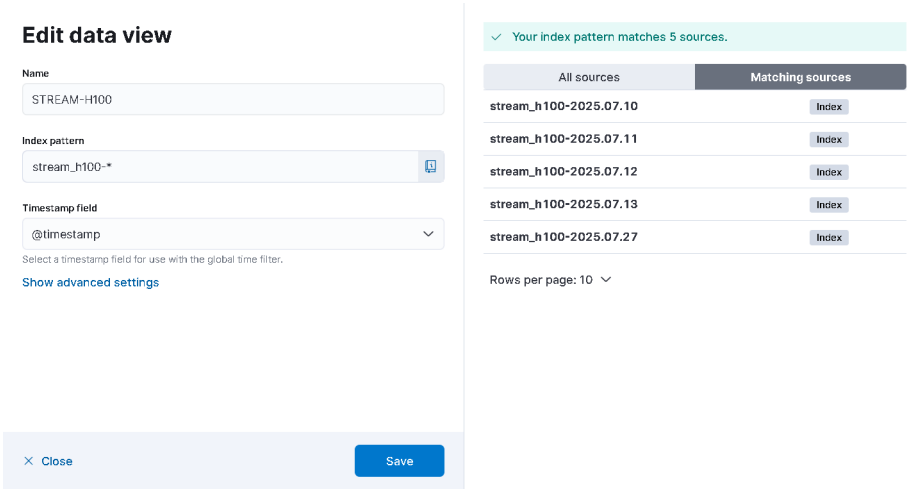
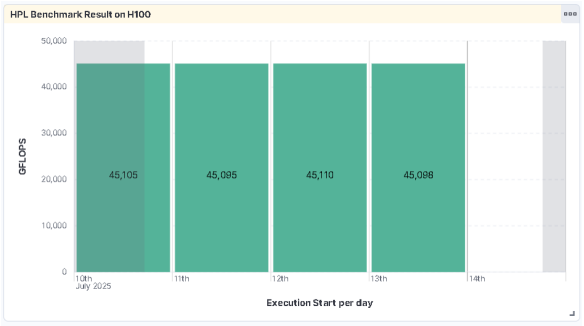
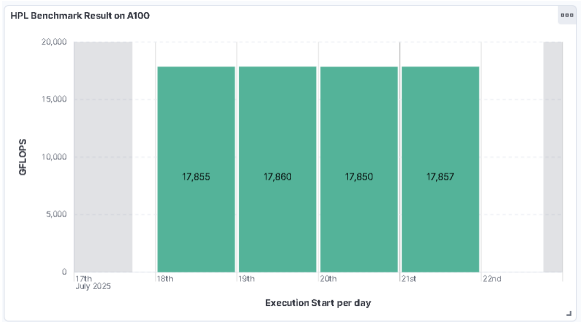


Figure 5.15. *Data Views in Kibana for Benchmark Indices*

Figure 5.16 presents the HPL benchmark results for both H100 and A100 GPU nodes. The x-axis represents the day of execution, and the y-axis shows the measured performance in GFLOPS.



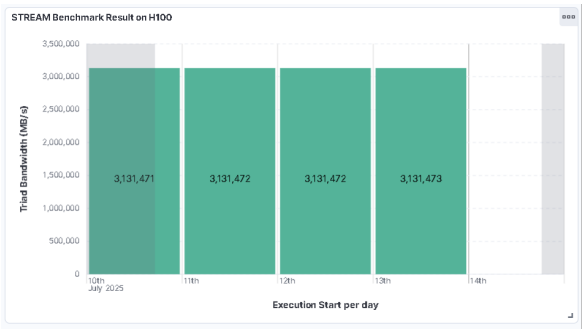
(a) *H100 GPU Node*



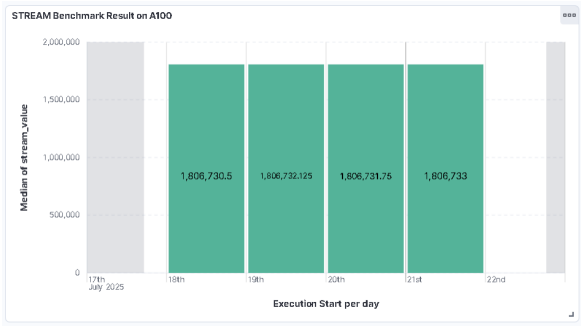
(b) *A100 GPU Node*

Figure 5.16. *HPL Benchmark Kibana Chart*

Similarly, Figure 5.17 illustrates the STREAM benchmark results for both H100 and A100 GPU nodes, focusing on the Triad Bandwidth metric. The x-axis represents the day of execution, while the y-axis indicates the Triad Bandwidth.



(a) *H100 GPU Node*



(b) *A100 GPU Node*

Figure 5.17. *STREAM Benchmark Kibana Chart*

Overall, the results across all benchmarks demonstrate solid performance, closely matching the expected values discussed in Chapter 4. These dashboards allow validation of any specific compute node's performance by selecting the node of interest and specifying the desired time range.

5.4.2 Agent-Based Monitoring

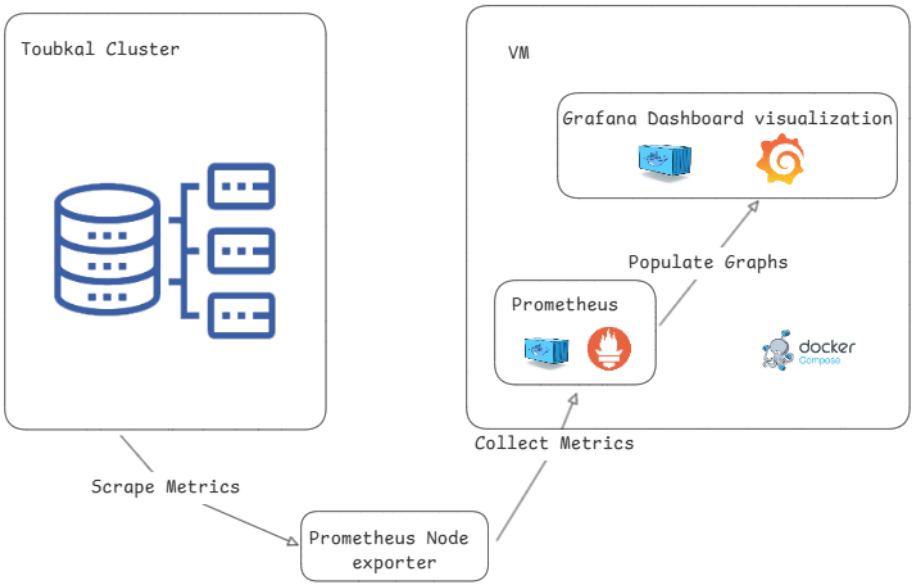


Figure 5.18. *Agent-Based Monitoring using Prometheus and Grafana in the Toubkal Cluster*

Figure 5.18 shows the architecture of the agent-based monitoring system. Prometheus and Grafana are deployed in Docker containers using Docker Compose. Prometheus collects metrics from the Toubkal supercomputer through Node Exporter, which exposes metrics at specific endpoint and port. The collected metrics are then visualized in real-time using Grafana dashboards.

a) Local Metrics

As shown in Figure 5.19, the dashboard displays several key metrics of a compute node with hostname `slurm-compute-h21a5-u7-svn2` and IP address `10.43.10.47`, which exposes metrics on port `9100`. The monitored information includes the average temperature of the node, CPU utilization, system load, RAM usage, swap usage, root filesystem usage, number of CPU cores, total swap memory, total RAM, and total root filesystem capacity.

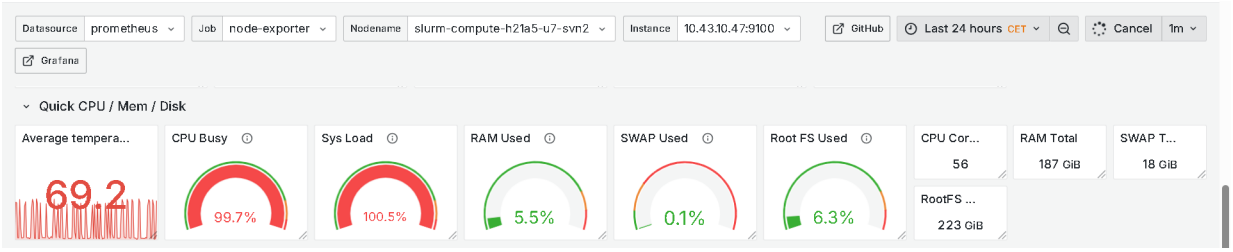


Figure 5.19. *CPU, Memory, and Disk metrics of a Node*

Figure 5.20 illustrates the percentage of time the CPU has spent in its different modes over the specified time range.

The modes in which the CPU can spend its time include:

- User mode: time spent executing user processes, such as applications and programs.
- System mode: time spent running kernel processes and handling system calls.
- I/O wait mode: time the CPU is idle while waiting for input/output operations, such as disk or network access, to complete.
- Idle mode: time when the CPU has no work to execute and remains idle.

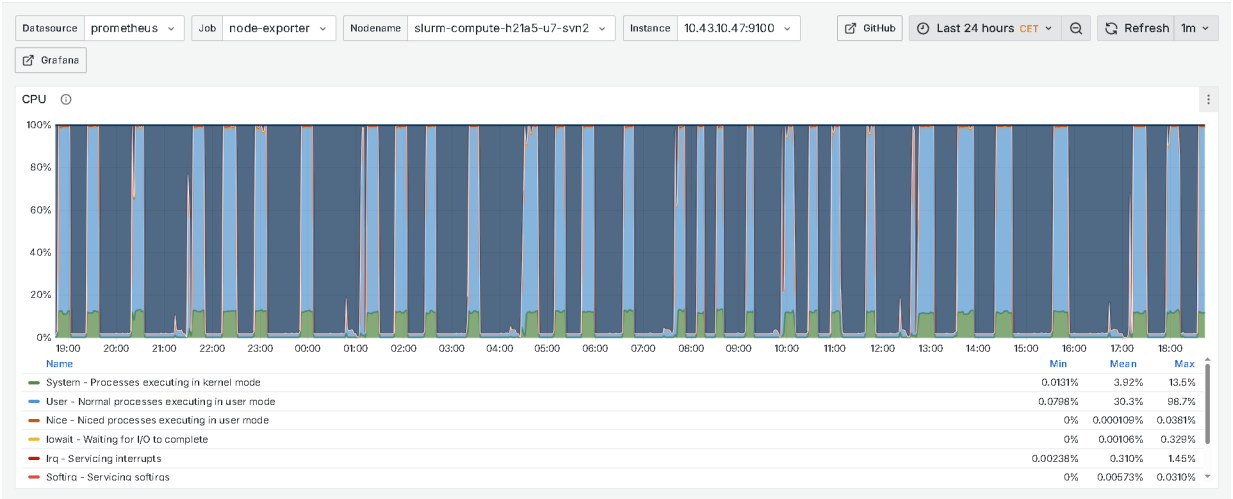


Figure 5.20. *CPU Utilization by Node*

Figure 5.21 presents the memory usage over time for a compute node, broken down into several categories.

- The apps memory corresponds to the space occupied directly by running processes.

- The cache corresponds to the memory used to store recently accessed files and data, allowing faster retrieval by avoiding repeated disk access.
- Page Tables represent the memory consumed by the kernel to manage virtual-to-physical address translations.

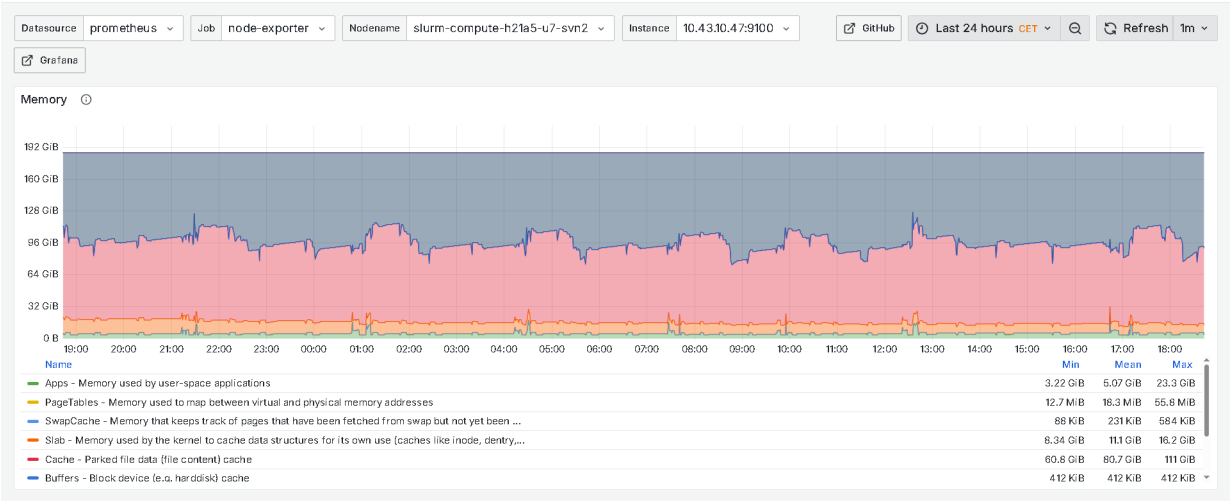


Figure 5.21. *Memory Usage over time for a Node*

b) Global Metrics

Since Node Exporters expose metrics only at the single-node level, it is also necessary to have a dashboard that provides an aggregated view of the entire cluster. This allows administrators to detect cluster-wide anomalies and performance issues more effectively.
Figure 5.22 illustrates the meta CPU utilization across the cluster.Monitoring the CPU `iowait` time at the cluster level makes it possible to detect issues with the shared Lustre file system. When `iowait` increases beyond a certain threshold, it indicates that a significant portion of the cluster's cores are stalled while waiting for disk I/O operations. This behavior points to serious performance bottlenecks affecting multiple nodes simultaneously.

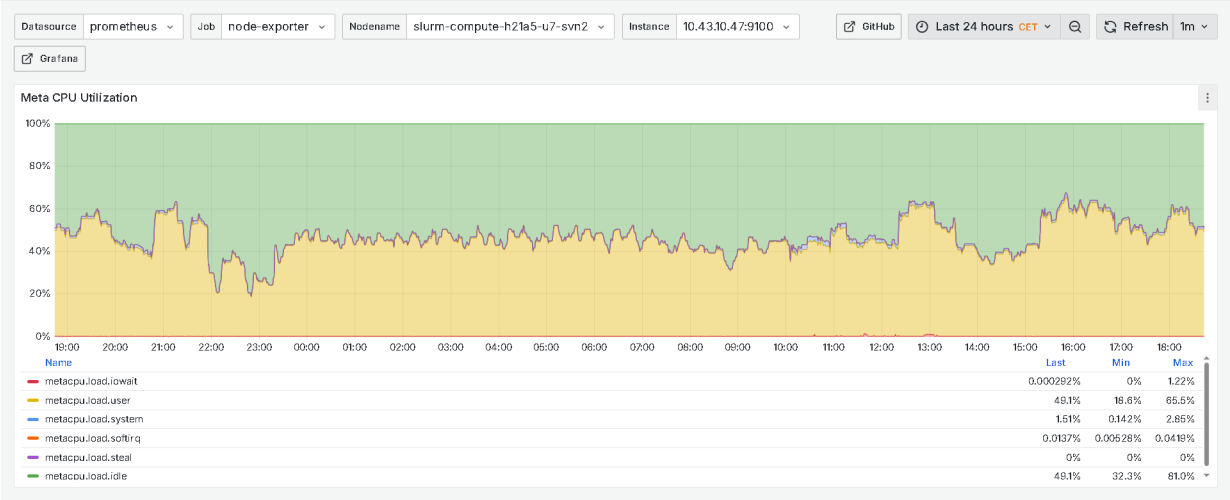


Figure 5.22. *Meta CPU Utilization*

Figure 5.23 presents the aggregated cluster status. This view helps in detecting overheated nodes, which can be caused by unbalanced resource allocations. For example, if a user allocates only a single node for a workload that requires several, the node may become overloaded and eventually overheat. Such conditions are visible in the dashboard as a reduction in the number of active nodes accompanied by an increase in overheated nodes.

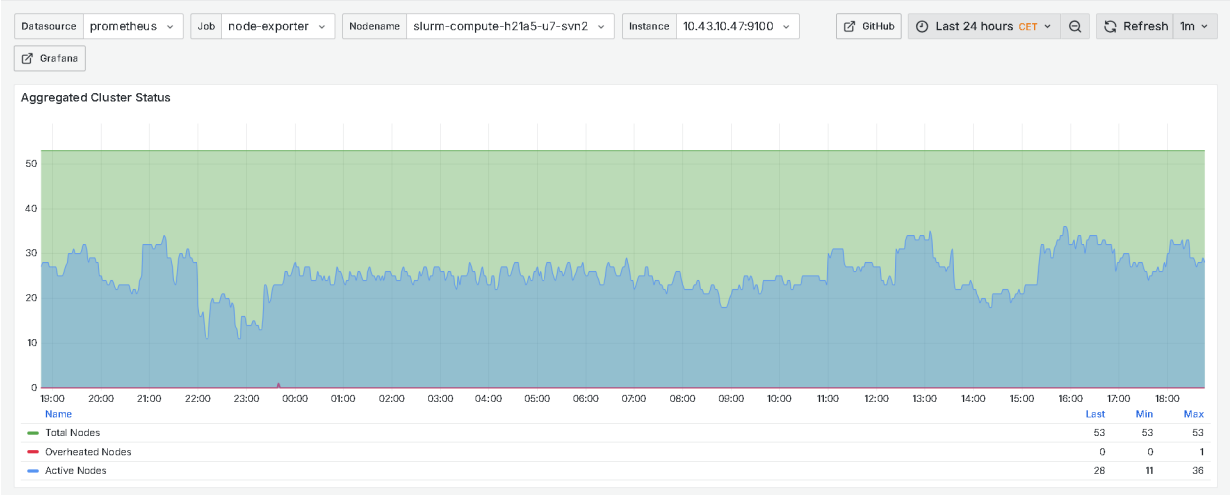


Figure 5.23. *Aggregated Cluster Status*

5.5 *Conclusion*

This chapter outlined the automation and monitoring workflow implemented on the Toubkal supercomputer. Benchmarks were integrated into ReFrame for reproducibility and validation,

while Jenkins enabled continuous integration of regression tests. Monitoring was achieved through the ELK stack for log-based analysis and Prometheus/Grafana for real-time metrics at both node and cluster levels. Together, these tools enhance reliability, reduce manual effort, and provide a solid foundation for scalable HPC operations.

General Conclusion

This work addressed the challenges of benchmarking and monitoring on modern HPC infrastructures by using the Toubkal supercomputer as a case study. We began by presenting the architecture of Toubkal, detailing its compute, memory, interconnect, and storage subsystems, as well as the supporting software stack. This architectural description established the foundation for understanding the performance characteristics of the system.

As a first step toward enabling scientific workloads, several widely used applications were built from source and optimized for the Toubkal environment, including VASP, KNN_CUDA, and QE. This ensured that the cluster was equipped with state-of-the-art simulation and machine learning tools, compiled and tuned to exploit the available hardware efficiently.

We then conducted a set of manual benchmarks, including HPL and STREAM, to evaluate the raw computational and memory bandwidth performance of the cluster. These experiments provided baseline reference values and allowed us to validate that the system delivers results close to its theoretical specifications.

Building upon these results, we automated the benchmarking workflow using the ReFrame regression testing framework. By integrating benchmarks into Python based test definitions and configuring system partitions, environments, and sanity checks, we enabled reproducible and scalable execution of performance tests. To further support continuous evaluation, a Jenkins-based pipeline was deployed, allowing automated scheduling and execution of benchmarks directly on the Toubkal cluster.

In parallel, monitoring capabilities were introduced to provide both log-based and agent-based insights into system health and performance. The ELK stack allowed systematic indexing and visualization of benchmark outputs, while Prometheus and Grafana enabled real-time monitoring of node-level and cluster-level metrics. Together, these tools provide administrators and users with a comprehensive view of system performance and resource utilization.

Overall, the contributions of this work can be summarized as follows:

- Building and optimizing key scientific applications (VASP, KNN_CUDA, QE) from source for efficient use on Toubkal.
- Establishing a benchmarking baseline for Toubkal's CPU, GPU, memory, and interconnect subsystems.
- Automating benchmarking workflows with ReFrame to ensure reproducibility, scalability, and reduced human intervention.
- Integrating automation into a continuous pipeline using Jenkins to streamline regression testing on HPC resources.
- Deploying complementary monitoring solutions (ELK, Prometheus, Grafana) for both benchmark results and real-time system metrics.

These contributions enhance the reliability, usability, and efficiency of benchmarking and monitoring on HPC systems, while also equipping them with optimized scientific tools ready for production use. Beyond Toubkal, this methodology can be applied to other supercomputing environments, enabling systematic performance evaluation and proactive detection of anomalies.

Future work may include extending the automation framework to cover a wider range of scientific applications, integrating anomaly detection and alerting mechanisms into the monitoring system, and exploring container orchestration tools such as Kubernetes for scaling monitoring and benchmarking services. Such developments would further strengthen the resilience and usability of HPC infrastructures for both administrators and end-users.

Bibliography

[1] Jack Dongarra et al. “High-Performance Computing: Status and Outlook”. In: *Communications of the ACM* (2020).

[2] *Introduction to Parallel Computing Tutorial | HPC @ LLNL — hpc.llnl.gov*. https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial. [Accessed: 26-07-2025].

[3] *How To Tell What CPU Your Motherboard Supports — softwareg.com.au*. https://softwareg.com.au/blogs/computer-hardware/how-to-tell-what-cpu-your-motherboard-supports?. [Accessed: 26-07-2025].

[4] https://medium.com/ai-insights-cobet/understanding-gpu-architecture-basics-and-key-concepts-40412432812b. [Accessed: 01-08-2025].

[5] *ReDX*. URL: https://redxt.com/.

[6] https://www.scalecomputing.com/resources/understanding-gpu-architecture. [Accessed: 04-08-2025].

[7] https://ai.gopubby.com/memory-types-in-gpu-6373b7a0ca47. [Accessed: 01-08-2025].

[8] Hübner Michael Juergen Becker Göhringer Diana Perschke Thomas. *A Taxonomy of Reconfigurable Single-/Multiprocessor Systems-on-Chip*. https://onlinelibrary.wiley.com/doi/10.1155/2009/395018. [Accessed: 02-08-2025]. 2009.

[9] *rcet.org.in*. https://www.rcet.org.in/uploads/academics/rohini_86590040491.pdf. [Accessed: 21-07-2025].

[10] WEKA. *Introduction to HPC: What are HPC & HPC Clusters?* https://www.weka.io/learn/guide/hpc/what-are-hpc-and-hpc-clusters/. July 25, 2020.

[11] *System Architecture 2014; KAUST Supercomputing Lab Support Documentation 0.1 documentation — docs.hpc.kaust.edu.sa*. https://docs.hpc.kaust.edu.sa/systems/index.html. [Accessed 25-07-2025].

[12] Thomas Kainrad. *Copy-paste ready commands to set up SGE, PBS/TORQUE, or SLURM clusters — tkainrad.dev*. https://tkainrad.dev/posts/copy-paste-ready-instructions-to-set-up-1-node-clusters/. [Accessed: 25-07-2025].

[13] PNY Networking Solutions. *NVIDIA Infiniband Adapters*. https://networking.pny.eu/infiniband-adapters/. [Accessed: 04-08-2025].

[14] Vasileios Karakasis et al. "Enabling Continuous Testing of HPC Systems Using ReFrame". In: *Tools and Techniques for High Performance Computing*. HUST - Annual Workshop on HPC User Support Tools (Denver, Colorado, USA, Nov. 17–18, 2019). Ed. by Guido Juckeland and Sunita Chandrasekaran. Vol. 1190. Communications in Computer and Information Science. Cham, Switzerland: Springer International Publishing, Mar. 2020, pp. 49–68. ISBN: 978-3-030-44728-1. DOI: 10.1007/978-3-030-44728-1_3.

[15] M Kamath. "Real-time Performance Monitoring of HPC Clusters: Techniques and Challenges". In: *INTERANTIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT* 08 (June 2024), pp. 1–5. DOI: 10.55041/IJSREM35707.

[16] https://modules.sourceforge.net/docs/modules_fosdem25.pdf. [Accessed: 27-07-2025].

[17] *Difference Between Building From Source and Installing From a Package File | Baeldung on Linux — baeldung.com*. https://www.baeldung.com/linux/build-source-vs-install-pkg. [Accessed 03-08-2025].

[18] NVIDIA. https://www.nvidia.com/en-us/gpu-cloud/. [Accessed: 02-08-2025].

[19] Sylabs. https://docs.sylabs.io/guides/3.5/user-guide/introduction.html. [Accessed: 03-08-2025]. 2017-2019.

[20] *High Perf. Linpack (HPL) - UL HPC Tutorials — ulhpc-tutorials.readthedocs.io*. https://ulhpc-tutorials.readthedocs.io/en/latest/parallel/mpi/HPL/. [Accessed 14-07-2025].

[21] https://www.netlib.org/benchmark/hpl/tuning.html. [Accessed: 02-08-2025].

HPC Software Stack Build, Benchmarking and Automation

Mohamed Karim BEZINE

الخلاصة: تم إعداد مشروع التخرج داخل شركة ReDX Technologies ، حيث ركز على قياس أداء الحاسوب الفائق Toubkalومراقبته. شملت المهام بناء التطبيقات العلمية مثل VASP و Quantum ESPRESSO و KNN_CUDA من المصدر لضمان أداء أمثل، بالإضافة إلى إعداد اختبارات القياس مثل HPL و STREAM لتقييم كفاءة النظام. تم تطوير منهج التكامل المستمر باستخدام Jenkins و ReFrame لآلية التحقق من الأداء وتنفيذ اختبارات الانحدار، مع معالجة السجلات وتخزينها في Elasticsearch وعرضها عبر Kibana. كما تم استخدام أدوات Prometheus و Grafana للمراقبة الآنية للنظام، مما عزز من القدرة على تتبع الأداء واكتشاف الانحدارات مبكرًا في بيئة تتميز بالحوسبة عالية الأداء.

Résumé: Ce projet de fin d'études, réalisé au sein de ReDX Technologies, a porté sur le benchmarking et l'évaluation des performances du supercalculateur Toubkal. Les applications scientifiques telles que VASP, Quantum ESPRESSO et KNN_CUDA ont été compilées à partir de leurs codes sources afin de garantir une performance optimale, tandis que les benchmarks HPL et STREAM ont été exécutés pour évaluer l'efficacité du système. Un pipeline d'intégration continue avec Jenkins et ReFrame a automatisé la validation et l'exécution des tests de régression, les journaux ayant été prétraités, stockés dans Elasticsearch et visualisés via Kibana. Prometheus et Grafana ont permis un suivi en temps réel, renforçant l'observabilité des performances et la détection précoce des régressions dans un environnement HPC.

Abstract: This graduation project, conducted at ReDX Technologies, focused on benchmarking and performance monitoring of the Toubkal Supercomputer. Scientific applications such as VASP, Quantum ESPRESSO, and KNN_CUDA were built from source to ensure optimal performance, while HPL and STREAM benchmarks were executed to evaluate system efficiency. A CI pipeline with Jenkins and ReFrame automated validation and regression testing, with logs parsed into Elasticsearch and visualized through Kibana. Prometheus and Grafana enabled real-time monitoring, enhancing performance observability and enabling early regression detection in an HPC environment.

المفاتيح: الحوسبة عالية الأداء، DevOps، CI/CD، ReFrame، Jenkins، Elasticsearch، Kibana، Prometheus، Grafana.

Mots clés: Calcul Haute Performance, DevOps, CI/CD, ReFrame, Jenkins, Elasticsearch, Kibana, Prometheus, Grafana.

Key-words: High Performance Computing, DevOps, CI/CD, ReFrame, Jenkins, Elasticsearch, Kibana, Prometheus, Grafana.