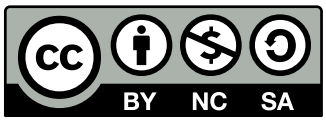
Aalto University
School of Science

Master's Programme in Security and Cloud Computing

Real-Time GPU Usage Alert Service on Pre-Exascale HPC Clusters

Songlin Jiang

Master's Thesis
2024



Aalto University
School of Science

Author Songlin Jiang

Title Real-Time GPU Usage Alert Service on Pre-Exascale HPC Clusters

Degree programme Security and Cloud Computing

Major Security and Cloud Computing

Supervisor Prof. Bo Zhao (Aalto University), Prof. Bernd Dammann (DTU)

Advisors Dr. Sebastian von Alfthan, Dr. Sami Ilvonen

Collaborative partner CSC – IT Center for Science

Date June 28, 2024 **Number of pages** 75+20 **Language** English

Abstract

Improving observability in large-scale distributed computing clusters has always been a complex problem, particularly in High-Performance Computing (HPC). Despite the growing popularity of GPU-accelerated jobs, traditional workload managers in HPC systems, such as Slurm, lack the feature for collecting GPU usage history at job levels. In addition, with increasing workloads that rely on extensive GPU resources, especially AI training jobs, GPUs have become the most power-consuming hardware in the HPC system, so it's essential to reduce resource waste on these devices.

To address these issues, in this master's thesis, we design and implement a real-time GPU usage alert service on top of the Slurm-based job monitoring system for supercomputer systems, i.e., Puhti, Mahti, and the pre-exascale supercomputer LUMI (the fastest supercomputer in Europe according to TOP500 by June 2024) at the CSC-IT Center for Science. We aim to have complete control over the data pipeline and tailor it to fit the characteristics of HPC systems so that it can be performant. As a result, we design our own GPU monitoring metrics collection infrastructure from the libraries provided by multiple GPU vendors and an in-memory real-time alert status checker service with the help of database triggers and LISTEN & NOTIFY. We also develop an alert algorithm to spot inefficient jobs with a bit of usage. In addition, we benchmarked the alert service with random data under extreme conditions designed for pre-exascale supercomputers, and the whole system was stable enough. Finally, we deployed the entire system in production for Puhti and Mahti, and it had been working well for months before we submitted the thesis.

The outcome of this master's thesis empowers supercomputer administrators at CSC – IT Center for Science to learn about sub-optimal GPU resource utilization for specific jobs in real-time, finding out the cause for them, thus improving energy efficiency and significantly reducing resource waste in HPC clusters.

Keywords HPC, GPU, Monitoring System, Alert Service, Observability

Preface

This M.Sc. thesis was prepared at the CSC – IT Center for Science Ltd., Finland, the Department of Computer Science at the School of Science of Aalto University, and the DTU Compute of the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a Master of Science (Technology) degree at the Aalto University and a Master of Science degree in Engineering at the Technical University of Denmark, for the SECCLO Master's Program in Security and Cloud Computing (Erasmus Mundus).

The **supervisors** of this thesis are Prof. Bo Zhao (main) from Aalto University and Prof. Bernd Dammann from the Technical University of Denmark.

The **advisors** of this thesis are Dr. Sebastian von Alfthan and Dr. Sami Ilvonen from CSC – IT Center for Science Ltd., Finland.

Without the assistance of others, this thesis would not have come to fruition:

First and foremost, I express my heartfelt gratitude to my supervisor, Prof. Bernd Dammann, and my advisor, Dr. Sami Ilvonen, for generously allocating time amidst their busy schedules to organize bi-weekly meetings throughout the thesis writing phase. Their invaluable advice has been instrumental in shaping this work. I would also like to thank Prof. Bo Zhao and my advisor, Dr. Sebastian von Alfthan, for their unwavering guidance and support. Special thanks to my colleague, Mr. Robin Karlsson, who helped me deploy the job graph and alert dashboard to CSC's Open OnDemand. Also, Mr. Simon Westersund and Dr. Mats Sjöberg gave me invaluable feedback on the dashboard design and usability of the system in production.

I sincerely thank CSC – IT Center for Science Ltd. for sponsoring this thesis project. I also want to thank the SECCLO Erasmus Mundus Program for granting me a full-ride scholarship, which covered my living expenses during my two-year journey at Aalto University and the Technical University of Denmark (DTU).

Last but certainly not least, I extend my heartfelt appreciation to my family and friends whose unwavering love and support sustained me throughout my journey over the past two years.

Otaniemi, June 28, 2024

Songlin Jiang

With the support of the
Erasmus+ Programme
of the European Union

Contents

List of Tables

List of Figures

Abbreviations

API	Application Programming Interface
Cgroups	Control group
CLI	Command Line Interface
CPU	Central Processing Unit
DB	Database
GCD	Graphics Compute Die
GPT	Generative Pre-trained Transformer
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HPC	High-Performance Computing
JSON	JavaScript Object Notation
LLM	Large Language Model
LSF	International Business Machines (IBM) Spectrum Load Sharing Facility
NVMe	Non-Volatile Memory express
NVML	NVIDIA Management Library
NUMA	Non-Uniform Memory Access
PCI	Peripheral Component Interconnect
QoS	Quality of Service
REST	Representational State Transfer
RDBMS	Relational Database Management System
ROCm	Advanced Micro Devices (AMD) Radeon Open Computing platform
SIMD	Single Instruction, Multiple Data
Slurm	Simple Linux Utility for Resource Management
SMI	System Management Interface
SQL	Structured Query Language
TDP	Thermal Design Power
TBP	Typical Board Power
TMU	Texture Mapping Units
TORQUE	Terascale Open-source Resource and Queue Manager
UGE	Univa Grid Engine

Chapter 1

Introduction

This Chapter provides an overview of the thesis, including its foundational background, the purpose and objectives to be pursued, and the identified problems. Additionally, we point out the thesis's contributions, and structure.

1.1 Background

In contemporary computational science and AI, using HPC systems has become advantageous and indispensable. As the boundaries of human knowledge expand, so do the complexities of the problems we seek to solve. From simulating climate models to analyzing vast datasets in genomics and training deep neural networks to powering advanced simulations in fluid dynamics, HPC emerges as the bedrock for modern scientific research and technological advancement.

Traditional HPC clusters emphasized CPU-centric computing. However, the rise of heterogeneous computing has introduced supercomputers with accelerators such as the GPU [21]. At the core of the current computational revolution lies the GPU, which has transcended its original purpose in graphics rendering to become the cornerstone of parallel computing. In artificial intelligence, the demand for GPU resources for training has surged dramatically, driven by the increasing complexity of neural network models and the rise of LLMs in recent years. The significance of GPUs in HPC cannot be overstated, as their massively parallel architecture and ability to handle thousands of computational tasks simultaneously have pushed scientific research and AI development to new heights. Indeed, GPUs have redefined the limits of what was once computationally feasible, enabling researchers and engineers to tackle problems of immense scale and complexity with remarkable efficiency and speed.

As a result, effective GPU monitoring has become an indispensable component for ensuring optimal performance, reliability, and resource utilization. Comprehensive monitoring solutions are paramount as computational workloads become increasingly complicated and resource-intensive. GPU monitoring provides invaluable insights into the utilization, temperature, power consumption, and memory usage of GPUs in real-time, enabling researchers and system administrators to identify bottlenecks, preemptively address hardware failures and code bugs, and optimize resource

allocation. This proactive approach enhances the overall efficiency and stability of HPC environments, minimizes downtime, and maximizes the return on investment in GPU-accelerated infrastructure.

Furthermore, with the growing popularity of AI applications, from natural language processing to computer vision, the demand for GPU resources continues to grow. Effective monitoring ensures the smooth operation of AI training jobs. It facilitates capacity planning and scalability to ensure that all the resources are used efficiently, allowing HPC users to seamlessly adapt to evolving computational demands. Through the lens of this thesis, we aim to underscore how we conduct GPU monitoring to harness the full potential of HPC in real-world applications, showing its pivotal role in driving innovation, enhancing system performance, and advancing the frontiers of both scientific research and artificial intelligence.

1.2 Purpose

Most workload managers in the HPC world lack a GPU monitoring feature. Implementing a real-time GPU usage alert service at the job level on HPC clusters can yield significant benefits across various domains. Notably, it can enhance the overall sustainability of utilizing the computing cluster while facilitating the advancement of generative AI development.

1.2.1 Sustainability

In HPC environments, GPUs are typically allocated by the device, while the CPU allocates cores. For instance, the LUMI supercomputer, ranked as the fastest in Europe according to TOP500 by June 2024 [45], uses AMD Radeon Instinct MI250X GPU Accelerator, which is equipped with 220 compute units (cores), in contrast to the 64-core 3rd Gen EPYC 7A53s Trento CPU [13]. While this abundance of cores enables GPUs to efficiently handle mathematical and spatial computations simultaneously, it makes GPUs the most power-intensive component in a computing system.

Compared with CPUs, inefficient utilization of GPUs can lead to substantial resource wastage, particularly during peak hours when numerous jobs await allocation. Referring to Subsection 2.2.3, as is shown in Table 1.1, in LUMI, the maximum TDP for the CPU is 280W [3], whereas the TBP for the GPU is 500W, peaking at 560W [2]. Thus, each allocation on systems like LUMI has a mere 4W energy consumption per core for CPUs, contrasting wildly with the over 250W consumption for GPUs (1 GPU has 2 GCDs in this case here) [13].

Device	Cores	Maximum Power	Allocable	Allocation Power
GPU	220	560W	2	280W
CPU	64	280W	64	4.375W

Table 1.1: Resources comparison of GPU and CPU for LUMI

However, for GPU, knowing the performance and getting alerted for sub-optimal jobs on time poses a challenge for supercomputer administrators. Existing workload managers and job schedulers in HPC systems, such as Slurm and LSF, provide descriptive accounts of CPU and memory usage but lack real-time hardware monitoring capabilities at the job level, and they have no recording of history monitoring data. They also have no built-in alert systems. As a result, a GPU alerting system is urgently needed to help administrators ensure sustainability and reduce carbon emissions. Users also need this information to debug or improve the code and make their programs more environmentally friendly.

1.2.2 Generative AI

In January 2024, the European Commission introduced comprehensive measures to bolster European startups and SMEs (Small and Medium-sized Enterprises) in developing AI that upholds EU values and regulations [8]. This initiative follows the political consensus reached in December 2023 regarding the EU AI Act, which is tailored to support AI startups and foster innovation. It includes a proposal aimed at amending the Regulation of the EuroHPC JU (European High-Performance Computing Joint Undertaking) [9], thereby granting startups and the broader innovation community access to AI-optimized supercomputers — AI Factories.

AI Factories leverage the supercomputing capacity of the EuroHPC Joint Undertaking to develop trustworthy, cutting-edge generative AI models. Generative AI, now a trendy research area, focuses on creating new content or data that is original and often indistinguishable from human-created content. These models are trained on vast amounts of text data to learn the intricacies of language. They can then generate coherent and contextually relevant text, pictures, sound, and videos in various styles and topics. Through fine-tuning and conditioning on specific prompts or input contexts (prompt engineering), LLMs can generate text that mimics human writing remarkably, making them valuable tools for various creative and practical applications, including content generation, story writing, and even code generation. Most of the generative AI models, such as LLaMA [49], Gemma [46] as well as OpenAI's GPT series [27], use transformers.

Transformers employ attention mechanisms to capture dependencies between words in a sequence, allowing them to understand and generate text with greater contextual awareness [50]. The architecture of transformers, characterized by their extensive layers and attention mechanisms, necessitates vast amounts of parallel computation, making them an ideal fit for GPU acceleration.

GPUs excel at handling the matrix multiplications and element-wise operations inherent in the forward and backward passes of transformer models, leveraging their massively parallel architecture to accelerate training and inference processes. Moreover, GPUs' large memory bandwidth helps efficiently process the immense datasets typically associated with LLMs, enabling scalability. As a result, integrating GPUs in LLM frameworks enhances computational performance. It allows researchers and practitioners to explore more complex models and larger datasets, pushing the boundaries of natural language understanding and generation.

However, due to the highly intensive model structure and the complicated nature of distributed parallel training and inference logic, those generative AI training programs and inference applications will likely not utilize all the GPUs fully. GPU monitoring and alerting can provide insights into the performance and help identify code bugs and optimization opportunities for AI workloads, thus advancing state-of-the-art AI research and applications.

1.3 Research questions

By the time we started the thesis, a monitoring system for GPU had already been ready as part of the work during my 2023 summer internship at CSC – IT Center for Science. As a result, this thesis aims to impact the above areas by implementing an alert service based on our existing monitoring system for supercomputers at CSC. It seeks to address the following research question:

How do we systematically design a service that efficiently and reliably analyzes jobs using GPUs in HPC systems in real-time?

This can be divided into the following sub-research questions:

- **RQ1**: *How to minimize the alert delay with a fix-size window in real-time?* This involves exploring efficient methods to streamline detecting and responding to anomalies or critical events promptly.
- **RQ2**: *How to minimize the performance impact on the database systems?* This includes optimizing database queries and resource allocation to ensure minimal disruption to overall system performance.
- **RQ3**: *How to reliably maintain a data structure for job alert status check?* This involves data organization, scalability, and fault tolerance to ensure reliable operation under varying workload conditions.
- **RQ4**: *How to find the most reliable algorithm for generating alerts?* This includes assessing different algorithms' accuracy, efficiency, and scalability to determine their suitability for real-time alert generation of HPC.

1.4 Contributions

The contributions of this thesis are as follows:

- We have successfully created and deployed the monitoring system and the alert service for Nvidia and AMD GPUs at job level with Slurm on HPC systems.
- Through an in-depth investigation of the collected monitoring data, an algorithm has been developed to identify and alert jobs with inefficient GPU resource usage. This achievement not only enhances the effectiveness of the alert service but also contributes to optimizing resource utilization and improving overall system efficiency.
- By analyzing the collected monitoring data and identifying patterns of inefficient GPU usage, this thesis contributes to a deeper understanding of resource

utilization changes and patterns within HPC environments. This knowledge can inform future research and development efforts to optimize GPU-accelerated computing and improve the performance of AI workloads on HPC.

1.5 Thesis structure

This thesis explores techniques and methodologies for designing and implementing a monitoring system and alert service for GPU resource utilization on HPC clusters. Following the introduction, which provides background information, outlines the application domains, and presents research questions and contributions, the thesis is organized as follows:

Chapter 2 focuses on the background, which discusses various techniques for building the monitoring system and the alert service. This includes an examination of job schedulers, an overview of HPC systems at CSC (including Puhti, Mahti, and LUMI), an overview of the time-series database TimescaleDB, and an exploration of different monitoring systems techniques, such as /proc, Cgroups, NVML, and ROCm-SMI, as well as database techniques, including triggers, LISTEN & NOTIFY, and continuous aggregates. Additionally, this Chapter delves into alert algorithms, covering descriptive statistics, decision trees, random forests, K-means clustering, and silhouette analysis.

Chapter 3 details the methods we use for implementing the system, which includes developing a monitoring daemon, timescale ingest, timescale reader, alert service, and alert dashboard, configuring TimescaleDB, and designing alert algorithms.

In Chapter 4, benchmark results from both experimental and production setups are presented and analyzed. This includes an overview of the experimental setup, benchmarking results, and insights from production environments by case studies.

Chapter 5 discusses the findings, analyzing related work and their limitations.

Finally, Chapter 6 provides concluding remarks, including a summary of critical points and suggestions for future work.

Chapter 2

Background

2.1 Job scheduler

This Chapter briefly overviews the well-known open-source workload manager we use at CSC – IT Center for Science, Slurm [54].

Let's first start with some definitions:

1. **Resource Manager** is a software component for overseeing the resources within a cluster, typically under the control of a scheduler. Its responsibilities include:
   - Management of various resources such as nodes, CPUs, GPUs, memory, and network.
   - Coordination of job execution across compute nodes to ensure efficient resource utilization.
   - Prevention of resource overlap among concurrent jobs.
2. **Scheduler** is a software module that manages user jobs within a cluster based on predefined policies. It interacts with users to receive and handle new jobs while controlling the Resource Manager. Key features of a Scheduler include:
   - Provision of partitions, queues, and Quality of Service (QoS) settings to enforce policies and limits on job execution.
   - Implementation of scheduling mechanisms such as backfilling, first-in-first-out (FIFO), etc.
   - Provision of interfaces for defining job workflows (e.g., job scripts), specifying job dependencies, and issuing commands for job management (e.g., submission, cancellation, etc.).
3. **Batch-System** or **Workload Manager** is the combination of a scheduler and a resource manager, combining the features of both components to manage workload within the cluster efficiently.

Slurm, initially known as the Simple Linux Utility for Resource Management, emerged in 2002 at Lawrence Livermore National Laboratory as a batch system tailored for Linux clusters. As HPC environments require sophisticated workload managers to efficiently manage and allocate computing resources among multiple users

and tasks, it evolved into a sophisticated scheduling framework incorporating advanced plugins.

The architecture of Slurm is modular and extensible, with many plugins available to cater to diverse requirements. At the core of Slurm, we have mainly the following three components:

- **Slurmctld** serves as the central management daemon within the Slurm framework, orchestrating the activities of all other Slurm daemons and resources. Its primary functions include the monitoring of system resources and the allocation of these resources to incoming workloads (jobs). Additionally, Slurmctld is a connecting point for accepting and processing job submissions, ensuring efficient utilization of available computing resources.
- **Slurmdbd** offers a secure and centralized interface to interact with the database system, specifically tailored to cater to the needs of Slurm. This daemon enables features such as archiving accounting records and storing essential metadata related to job executions. It ensures the integrity and confidentiality of data while providing seamless access to critical information for administrative and analytical purposes.
- **Slurmd** functions as the compute node daemon, operating at the node level to oversee the execution of computational tasks. Among its key responsibilities, slurmd actively monitors the status of tasks running on the compute nodes, facilitating efficient task management and resource allocation. It also acts as a connecting point for receiving task assignments, launching tasks onto the compute nodes, and terminating tasks as per system requirements or user requests.

Slurm plays a crucial role in maximizing the utilization of computational resources and ensuring fair access for users to run their parallel and distributed applications. It offers a powerful yet flexible solution for managing and scheduling computational workloads. Slurm is a preferred choice for orchestrating complex computing infrastructures worldwide, as used by over 65% of TOP500 systems [36].

2.2 HPC systems at CSC

Table 2.1 shows the GPU statistics for the three HPC systems at the CSC-IT Center for Science: Puhti, Mahti, and LUMI. Understanding the architecture and capabilities of these systems is crucial for tailoring GPU monitoring and alerting strategies and ensuring seamless integration with existing infrastructure. This Section provides an overview of the GPU computing nodes for these HPC systems.

Item	Puhti	Mahti	LUMI
Number of GPU nodes	80	24	2928
GPU Model	Nvidia V100	Nvidia A100	AMD MI250X
GPU number per node	4	4	4 / 8 (GCDs)
Total number of GPUs	320	96	11712 / 23424 (GCDs)

Table 2.1: GPU statistics for HPC systems at CSC – IT Center for Science

2.2.1 Puhti

Puhti [15] is an Atos BullSequana X400 cluster. Its AI partition comprises 80 GPU nodes, tailored specifically for artificial intelligence tasks, and collectively achieve a peak performance of 2.7 petaflops. Each node integrates the Intel Xeon Gold 6230 processors of the Cascade Lake family, which has 20 cores running at 2.1 GHz and supports AVX-512 vector instructions & VNNI instructions for AI inference workloads. We have a total thread count of 80 threads per node (40 per socket). Additionally, we have 4 Nvidia Volta V100 GPUs on each node, each with 32 GB of memory and connected via NVLink. Each node also features 384 GB of main memory and 3.6 TB of high-speed local storage.

2.2.2 Mahti

Mahti [14] is an Atos BullSequana XH2000 cluster. It comprises 24 GPU nodes, accumulating a theoretical peak performance of 2.0 petaflops. Each CPU and GPU node has 2 AMD Rome 7H12 CPUs featuring 64 cores each. The CPUs are based on the AMD Zen 2 architecture and operate at a base frequency of 2.6 GHz and a maximum peak frequency of up to 3.3 GHz. They also support the AVX2 vector instruction set. Each core supports simultaneous multi-threading, enabling 256 threads per node.

The GPU nodes have 512 GB of memory and a local 3.8 TB NVMe drive. They also have 4 Nvidia Ampere A100 GPUs, with a subset of nodes featuring split A100 GPUs, which enable lightweight workloads such as interactive tasks, courses, and code development and save energy consumption.

2.2.3 LUMI

By Jun 2024, LUMI [13], one of the three European pre-exascale supercomputers and an HPE Cray EX supercomputer, was the fastest supercomputer in Europe. The GPU partition, LUMI-G, features the primary compute power, which consists of 2978 nodes, as described in Figure 2.1.

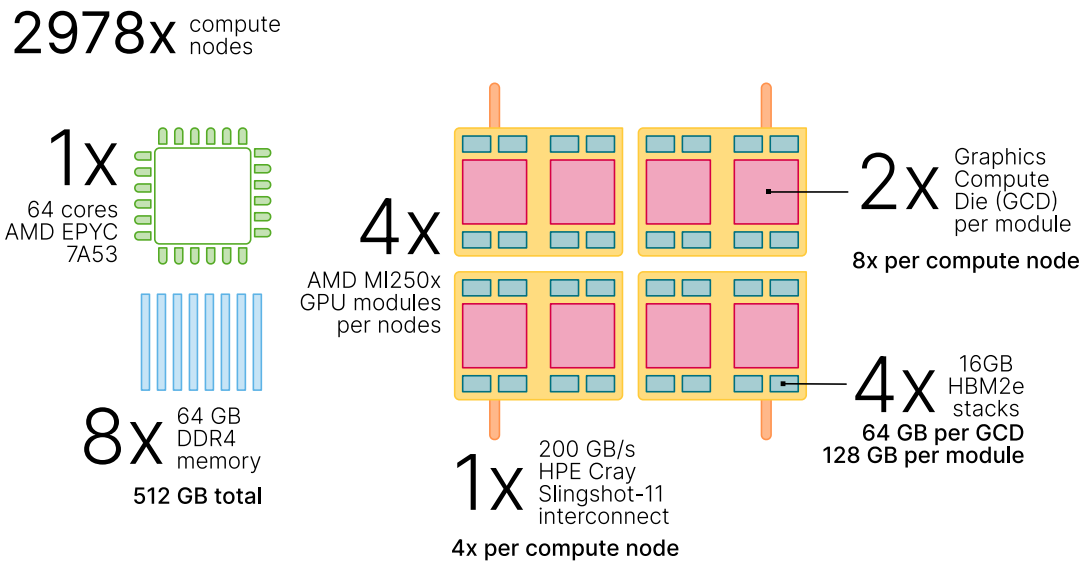


Figure 2.1: LUMI GPU partition overview [13]

GPU

The LUMI-G compute nodes have four AMD MI250X GPUs, each based on the 2nd Gen AMD CDNA architecture. The MI250X GPU is structured as a multi-chip module, which includes two GPU dies referred to as AMD Graphics Compute Dies (GCD). In Slurm and HIP runtime, a single MI250X module is recognized as two GPUs. Thus, in the actual system, the LUMI-G are 8-GPU nodes.

Each GCD has 110 compute units and accesses a 64 GB slice of high-bandwidth memory, totaling 220 compute units and 128 GB of memory per MI250X module. All compute units share an L2 cache with an 8 MB capacity to enhance memory throughput. The cache is divided into 32 slices, capable of delivering 128 B/clock/slice, amounting to a peak theoretical bandwidth of 6.96 TB/s. Including an L2 cache in the MI250X GPU modules enhances synchronization capabilities for algorithms reliant on atomic operations to coordinate communication across the entire GPU. These atomic operations are executed close to the memory within the L2 cache.

CPU

The CPU of LUMI-G nodes is a 64-core AMD EPYC 7A53 Trento CPU. These CPU cores, built on the Zen 3 architecture, support AVX2 256-bit vector instructions, enabling a maximum throughput of 16 double-precision FLOP/clock (AVX2 FMA operations). Each core has 32 KiB of private L1 cache, a 32 KiB instruction cache, and 512 KiB of L2 cache. The L3 cache has 32 MiB shared among groups of 8 cores, accumulating 256 MiB of L3 cache per CPU. Configured as 4 NUMA nodes (NPS4), the system allocates 128 GiB of DDR4 memory per NUMA node, resulting in 512 GiB of CPU memory.

2.3 Monitoring tools

A robust selection of monitoring techniques is crucial for capturing accurate metrics, diagnosing performance issues, and ensuring efficient resource utilization. This Section explores various components for building a monitoring system, each offering unique capabilities for monitoring within large-scale distributed computing clusters. Thus, it provides the cornerstone for building an accurate alert algorithm.

2.3.1 Cgroups

Cgroups [56] is a Linux kernel feature that enables the isolation and accounting of resources, including CPU, memory, and devices, among a collection of processes. It provides a means to track and manage CPU and memory utilization per process.

Slurm can be configured to use Cgroups to regulate resources allocated to jobs, steps, and tasks and for resource accounting. Cgroups have different controllers (subsystems), each responsible for managing specific resources. Slurm's plugins can utilize multiple controllers, including memory, CPU, devices, freezer, cpuset, and cpuacct. Each enabled controller empowers Slurm to enforce resource constraints on a defined set of processes. Slurm cannot enforce resource constraints through Cgroups for the associated resources if a controller is unavailable. Slurm supports two Cgroup modes through plugins: Legacy mode (Cgroups v1) and Unified Mode (Cgroups v2, introduced in Linux Kernel version 5.8).

At CSC, Puhti and Mahti still use Red Hat Enterprise Linux (RHEL) 8, which means Cgroup v1. In contrast, LUMI uses a more recent system, SUSE Linux Enterprise Server (SLES) 15 SP4, which means Cgroup v2. So, we need to support both Cgroup versions to cover the CPU and memory monitoring for all the systems.

Cgroups can form the foundation for understanding CPU and memory consumption. Take Cgroups v1, for example:

- **/sys/fs/cgroup/memory/slurm/uid_*/job_*/*/memory.usage_in_bytes** tells the memory usage for the job.
- **/sys/fs/cgroup/cpuset/slurm/uid_*/job_*/cpuset.cpus** tells the list of CPU cores that are used by the job.

2.3.2 /proc

The /proc [55] directory is unique within the Linux file system, as it functions as a virtual filesystem. Often described as a process information pseudo-file system, it diverges from traditional directories by not containing actual files of the same size since these files serve as pointers, directing users to the underlying location of process information within the kernel. Thus, it provides runtime system data such as system memory utilization, mounted devices, and hardware configurations.

When browsing the directory, we see those numbered subdirectories corresponding to the unique process ID (PID). By cross-referencing these PIDs with the process table, we can identify and examine specific processes. For instance, if the process

table denotes a process with PID 1234, accessing the directory /proc/1234 reveals detailed information about this process.

The /proc can also form the basis for understanding CPU and memory consumption. Monitoring processes often involve parsing information from /proc to extract consumption-related statistics according to process IDs, including their memory usage and CPU information:

- **/proc/cpuinfo** helps us know the frequency of CPU cores in the current node.
- **/proc/<PID>/status** helps us know how many CPU cores we got allocated, as well as the current Resident Set Size (RSS) and max RSS, which is the maximum amount of memory used at any time by the process in that job context.
- **/proc/<PID>/maps** provides information about memory mappings for the specified process. It includes a list of memory regions allocated to the process and details such as the starting and ending addresses, permissions, and file mappings, if applicable.

2.3.3 GPU management library

NVML

NVML [26] is a GPU-specific library provided by NVIDIA for managing and monitoring NVIDIA GPU devices. Its C-based API is designed to monitor and control the diverse states of NVIDIA GPU devices. This API offers direct access to the queries and commands accessible through Nvidia-SMI. It can help efficiently manage and monitor NVIDIA GPU devices. NVML exposes a comprehensive set of metrics, including GPU temperature, memory usage, and GPU utilization, allowing for detailed insights into GPU performance.

NVML is a cornerstone for GPU monitoring on systems equipped with NVIDIA GPUs, such as Puhti and Mahti.

ROCm-SMI

ROCm-SMI [4] is an analogous tool to NVML but tailored for AMD GPUs within the Radeon Open Compute ecosystem. Like NVML, ROCm-SMI provides GPU-specific metrics, allowing for the monitoring and managing of AMD GPU devices.

For HPC clusters that utilize AMD GPUs, such as LUMI, integrating ROCm-SMI into the GPU monitoring framework is essential. In contrast, Puhti and Mahti utilize NVIDIA GPUs, so NVML is used in these two systems.

2.4 Time-series databases

Time-series data management is a critical aspect of data lifecycle management, mainly due to its complexities. Time-series data, comprising a sequence of data points recorded over discrete time intervals, provides invaluable insights into evolving phenomena

spanning milliseconds to years. Widely applicable across various domains, time-series data uses time as the principal axis for data organization.

Effectively managing time-series data involves addressing numerous challenges, ranging from efficient data ingestion to optimized query performance and cost-effective resource utilization. Time-series databases play a pivotal role in storing and analyzing time-stamped data efficiently. Choosing the right time-series database is crucial for managing and querying the vast amount of data generated by monitoring metrics, thus contributing to resolving **RQ1** in Section 1.3.

One of the primary challenges in time-series monitoring for HPC jobs is handling high cardinality, which refers to the number of unique sets of data combinations. High cardinality can lead to performance bottlenecks as the database needs to manage and index many unique series, each potentially requiring separate storage and processing resources. This can strain the database system's capacity to ingest data rapidly and execute queries efficiently, especially as the data volume scales. Many non-time-series databases, and other time-series databases, including InfluxDB, will crash as the job to record (cardinality) increases.

TimescaleDB

TimescaleDB [5] emerges as a robust solution, excelling in both rapid data ingestion and streamlined query processing, ensuring comprehensive time-series data management capabilities. TimescaleDB is an open-source time-series database optimized for fast ingest and complex queries. It is engineered on top of the mature RDBMS system PostgreSQL and packaged as an extension. TimescaleDB extends PostgreSQL with time-series-specific optimizations and functions, allowing it to manage time series data efficiently. TimescaleDB supports the full range of SQL functionality, including aggregates, joins, subqueries, and window functions.

Central to TimescaleDB's architecture are **hypertables**, which serve as fundamental structures for efficient time-series data management as shown in Figure 2.2. Hypertables abstract multiple individual tables that store the data, referred to as chunks, providing users with a unified interface for data interaction. With hypertables, users can execute diverse operations, including data insertion, querying, and schema modifications, seamlessly integrating standard SQL functionalities. Hypertables are particularly advantageous for time-series data management, because they efficiently partition incoming data into smaller, manageable subsets. These data subsets, represented as time-based chunks, enable TimescaleDB to efficiently handle data retention and optimization. Consequently, TimescaleDB can do fast data ingestion and is scalable to ions of rows per second, which is crucial for time-series applications.

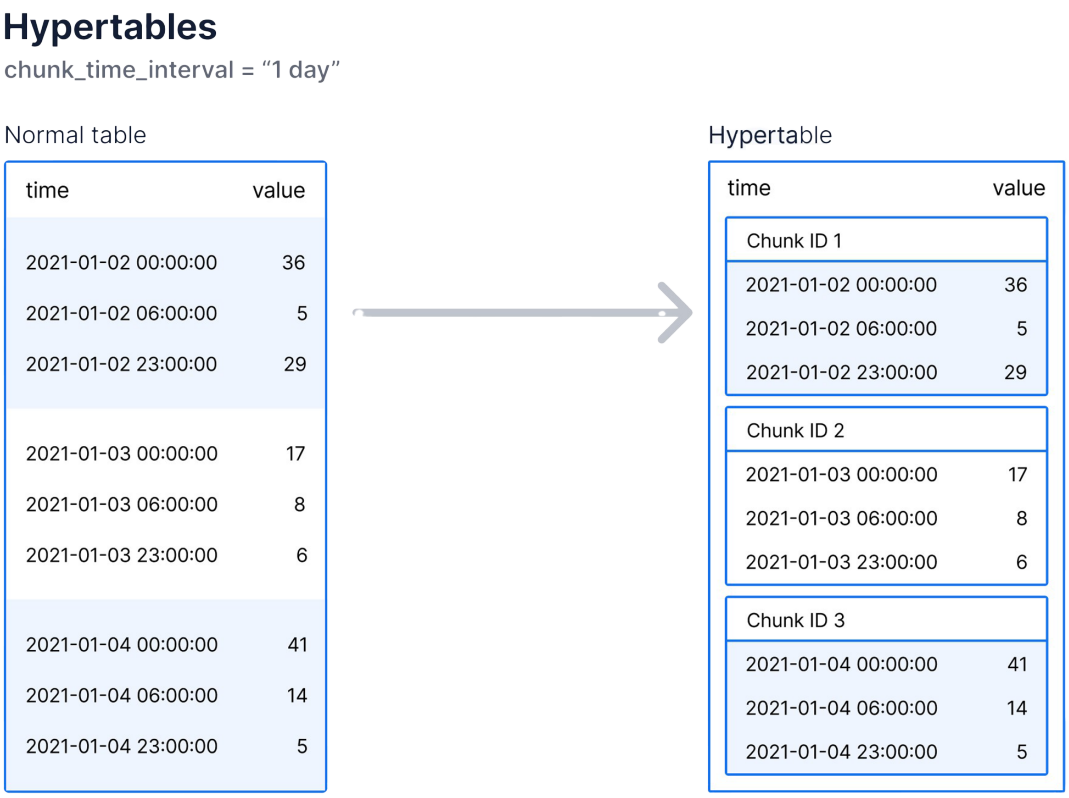


Figure 2.2: Hypertable compared with normal table [48]

Moreover, TimescaleDB effectively addresses cardinality issues. Leveraging hypertables and their automatic partitioning mechanism minimizes the overhead of managing high-cardinality data sets. This partitioning ensures the data is distributed across smaller chunks, reducing the performance impact when querying large datasets and maintaining efficient indexing.

Compression functionalities in TimescaleDB further enhance data storage efficiency. By employing compression algorithms such as delta encoding, delta-of-delta, simple-8b, run-length encoding, etc. [24], TimescaleDB significantly reduces disk space utilization, achieving very high compression ratios.

TimescaleDB also facilitates the implementation of **data retention** policies as background jobs, automating the removal of obsolete data chunks and ensuring efficient data lifecycle management.

These features collectively make TimescaleDB a preferred choice over other time-series databases, especially for applications requiring robust performance under high cardinality conditions.

2.5 Database techniques

Databases are crucial in effectively managing and analyzing time-series data generated in real-time GPU resource monitoring. Our motivation is to create a system

that can handle long-term reliable storage and provide near real-time status overviews without heavy SQL searches, thus contributing to resolving both **RQ1** and **RQ2** in Section 1.3. To achieve this, we explore three significant database techniques that can be employed: Triggers, LISTEN & NOTIFY, and Continuous Aggregates.

2.5.1 Triggers

Triggers [39, 32] are database elements that automatically respond to specific events or changes in the database. In GPU resource monitoring, a trigger can be activated upon inserting new GPU monitoring data, instantly processing this data and notifying relevant services. This ensures that the data is immediately available for long-term storage and real-time monitoring, facilitating a seamless integration between the two tiers of data management.

A trigger in database management serves as a specification, that executes a designated function automatically, whenever a particular type of operation is conducted. The function must be declared as accepting no arguments and returning type triggers. It is important to note, that the trigger function receives input via a specially passed TriggerData structure, rather than conventional function arguments. Triggers can be configured to execute before or after an INSERT, UPDATE, or DELETE operation on a per-row or per-statement basis. Upon occurrence of a trigger event, the designated trigger function is invoked at the appropriate time to handle the event.

PostgreSQL offers two distinct types of triggers: **per-row (row-level)** triggers and **per-statement (statement-level)** triggers. Per-row triggers invoke the trigger function once for each row affected by the triggering statement. Conversely, per-statement triggers are invoked only once, when the corresponding statement is executed, irrespective of the number of rows impacted by the statement. Statement-level triggers also lack a mechanism to examine the rows modified by the statement.

Triggers can further be categorized as **before** triggers or **after** triggers. Statement-level before triggers are inherently activated before the start of the relevant statement, while statement-level after triggers are triggered upon the end of the statement. On the other hand, row-level before triggers are activated immediately before an operation on a specific row, while row-level after triggers are triggered after the statement but before any statement-level after triggers.

Typically, row-level before triggers are employed for data validation or modification before insertion or updating. Conversely, row-level after triggers are generally utilized to update other tables or perform consistency checks against other tables. This distinction is generated from the fact, that an after trigger can make sure that it is observing the final value of the row, whereas a before trigger cannot. Consequently, if there is no specific rationale for selecting between a before or after the trigger, the before the trigger is preferred for its efficiency, as information about the operation need not be retained until the end of the statement.

The input data within the trigger function includes the trigger event type (e.g., INSERT or UPDATE) and any specified arguments in the CREATE TRIGGER command. For row-level triggers, the input data consists of the NEW row for all types of triggers, and the OLD row for UPDATE and DELETE triggers precisely.

2.5.2 LISTEN & NOTIFY

LISTEN & NOTIFY mechanism [38, 32] is a powerful feature in PostgreSQL that provides asynchronous event notification. This technique enables efficient communication between the monitoring infrastructure and other system components, such as the alert service. It helps avoid the overhead of continuous polling and heavy SQL searches, enhancing system efficiency and responsiveness. When a relevant event occurs, such as the arrival of new GPU metrics or job state changes, the NOTIFY command can trigger notifications to any components that have registered interest with the LISTEN command, providing near real-time status updates.

LISTEN command registers the current session as a listener on the notification channel specified by the parameter. All sessions currently listening on the notification channel receive notifications. As a result, each listening session can be notified of its associated client application. Listen registrations of a session are automatically cleared upon the termination of that session.

It is up to the underlying PostgreSQL application programming interface for the client application to detect notification events. For the PostgreSQL driver and toolkit (**pgx**) package in Golang, listening for notification is a blocking read operation on the underlying socket. It will allocate a connection exclusively for listening purposes, allowing it to be blocked indefinitely.

NOTIFY enables processes accessing a shared PostgreSQL database to exchange information. The data transmitted to the client during a notification event includes the name of the notification channel, the process ID (PID) of the notifying session's server, and the payload string, which defaults to an empty string if unspecified, to client applications previously registered in listening for events on a specified channel within the current database. These notifications are broadcast to all listeners. More complex data structures can be established with database tables, to convey additional information from the notifier to the listeners.

A practical programming approach to signaling changes to a particular table using NOTIFY involves embedding the NOTIFY command within a trigger, triggered by table insertion.

2.5.3 Continuous aggregates

Time-series data tends to experience rapid expansion over time. Consequently, aggregating such data into meaningful summaries often encounters considerable latency. Continuous aggregates [42] offer a solution by fastening the data aggregation process.

In scenarios where data is collected at high frequencies, it becomes advantageous to aggregate the data into more considerable time intervals, such as minutes or hours. For instance, when GPU usage readings are recorded every second, computing the average GPU usage for each hour means we must scan the entire dataset and recalculate the average with each query execution. There are generally three ways to do the aggregation within TimescaleDB [47]:

- **Materialized views** is a conventional PostgreSQL feature used for caching the

results of complex queries for subsequent reuse. While materialized views do not update regularly, they can be manually refreshed.

- **Continuous aggregates**, exclusive to TimescaleDB, operate similarly to materialized views, but undergo automatic background updates, as new data is appended to the database. These aggregates are continuously and incrementally updated, resulting in lower resource requirements than materialized views. Furthermore, continuous aggregates are compatible with hypertables, and can be queried like standard tables.
- **Real-time aggregates**, another feature unique to TimescaleDB, share similarities with continuous aggregates. However, incorporate the latest raw data with previously aggregated data, to deliver accurate and up-to-date results without necessitating real-time data aggregation.

Continuous aggregates represent a materialized view that undergoes automatic refresh in the background, as new data is introduced or existing data is modified. These aggregates effectively track alterations to the dataset, ensuring the underlying hypertable is consistently updated. Furthermore, the maintenance overhead associated with continuous aggregates is substantially lower than conventional PostgreSQL materialized views. This efficiency allows users to focus on data analysis, rather than database maintenance.

Continuous aggregates comprise the following components:

- **Materialization hypertable**: The intermediary repository for aggregated data, which is retrieved as required. It consists of columns representing various group-by clauses in the query, a chunk column identifying the corresponding data chunk, and partial aggregate columns for each aggregate function specified in the query. The partial columns are crucial in aggregating data across chunks, particularly when groups span multiple chunks.
- **Materialization engine**: Responsible for orchestrating two transactions— the first determines the time range for materialization and updates the invalidation threshold. In contrast, the second executes the actual materialization process. Notably, most work occurs during the second transaction to prevent interference with other operations.
- **Invalidation engine**: Monitors changes to data in the hypertable and ensures timely re-materialization of affected rows. The invalidation prioritizes recent changes to minimize performance overhead.
- **Query engine**: Allows access to aggregated data in materialization hypertable.

For real-time continuous aggregates, it provides data by combining pre-aggregated data from materialized views with recent unaggregated data, ensuring that query results remain up-to-date.

2.6 Alert algorithms

This Section explores the alert algorithms we can use, including descriptive statistics, decision trees, random forest, and K-Means clustering, thus trying to resolve **RQ4** in

Section 1.3. Although deep learning models are powerful tools with great flexibility and capacity to learn feature hierarchies from raw data, they come with challenges. They can hardly be used in our case, since they require a large amount of data and substantial computational resources, which makes them less efficient compared to simpler models like decision trees and random forests when dealing with smaller datasets or when computational resources are limited. Furthermore, deep learning models can be challenging to interpret and require careful tuning to avoid over-fitting.

2.6.1 Descriptive statistics

Descriptive statistics is a branch of statistics that summarizes the data through numerical calculations. Its main purpose is to simplify and present data in an easy-to-understand format. Key measures include:

- **Central Tendency:** Mean, median, and mode. These measures of central tendency describe a distribution's center position.
- **Dispersion:** Standard deviation, range, variance, and interquartile range. These are measures of dispersion that describe the spread of data.
- **Skewness and Kurtosis:** These are measures of shape that describe the asymmetry and peakedness of a distribution, respectively.

Here is a list of definitions of these possible descriptive statistics:

- **Percentiles (25%, 50%, 75%)**: Statistical measures used to describe the distribution of a dataset. The 25th, 50th, and 75th percentiles are commonly known as the first quartile (Q1), median, and third quartile (Q3) respectively. They represent the values below which a given percentage of observations fall.

$$\text{Percentile}(X, p) = \text{value below which } p\% \text{ of the data fall}$$

- **Kurtosis**: Measure of the *tailedness* or *sharpness* of the peak of a distribution. Positive kurtosis indicates a sharper peak (leptokurtic), while negative kurtosis indicates a flatter peak (platykurtic). The formula for kurtosis is given by:

$$\text{Kurtosis}(X) = \frac{n(n+1)}{(n-1)(n-2)(n-3)} \sum_{i=1}^{n} \left(\frac{X_i - \bar{X}}{s} \right)^4 - \frac{3(n-1)^2}{(n-2)(n-3)}$$

- **Maximum**: The highest value in a dataset, which is the extreme upper end of the distribution.
- **Mean**: The average of all values in the dataset and is calculated as:

$$\text{Mean}(X) = \frac{1}{n} \sum_{i=1}^{n} X_i$$

- **Minimum**: The lowest value in a dataset represents the distribution's extreme lower end.

- **Skewness**: The asymmetry of a distribution. Positive skewness indicates a longer right tail, while negative skewness indicates a longer left tail. The formula for sample skewness is given by:

$$\text{Skewness}(X) = \frac{n}{(n-1)(n-2)} \sum_{i=1}^{n} \left(\frac{X_i - \bar{X}}{s} \right)^3$$

- **Variance**: Measure the average squared deviation of each data point from the mean. The formula for sample variance is given by:

$$\text{Variance}(X) = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{X})^2$$

- **Standard Deviation**: Measure the variation or dispersion in a dataset. It is calculated as the square root of the variance:

$$\text{Standard Deviation}(X) = \sqrt{\text{Variance}(X)}$$

2.6.2 Decision tree

A decision tree [43] represents possible solutions to a decision based on branches of certain conditions and reaches several conclusions based on the conditions. As shown in Figure 2.3, the main components of a decision tree are:

- **Root Node:** Collection of all the data that can be divided into two or more homogeneous sets.
- **Splitting:** The process of dividing a parent node into two or more sub-nodes.
- **Decision Node:** The split sub-nodes.
- **Leaf Node:** Terminal nodes that do not further split.

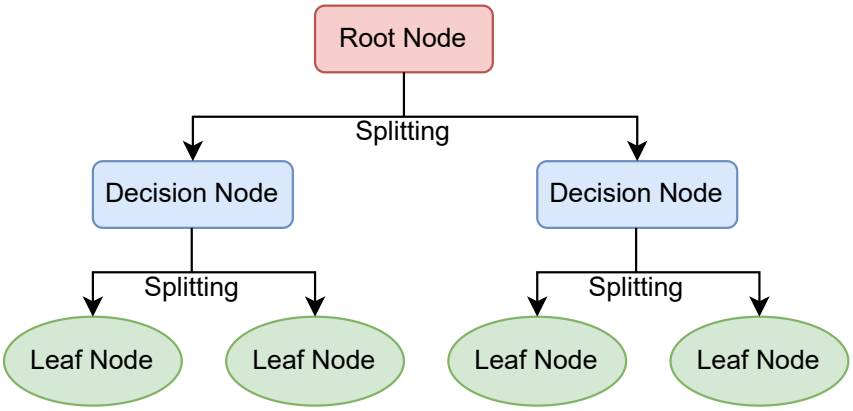


Figure 2.3: Decision tree structure

2.6.3 Random forest

Random forest [18] is an ensemble machine learning algorithm with many individual decision trees. Each tree in the random forest gives a prediction, of which the model's final prediction is the most voted class.

Extremely Randomized Trees (or Extra Trees) [16] is a type of ensemble learning technique that aggregates the results of multiple de-correlated decision trees to reach its final result. The fundamental difference between random forests and extra trees is selecting the cut points to create the decision trees. Extra Trees selects these cut-points comprehensively at random, while traditional decision tree-based algorithms (like Random Forest) select the optimal ones.

2.6.4 K-Means clustering and silhouette analysis

K-means clustering [53] is a linear clustering method widely used in data mining and machine learning that aims to partition data into several linearly separable homogeneous groups. It is an unsupervised learning method, meaning a training set is not required, making it more sensible for unlabelled data.

Silhouette analysis [51] is an unsupervised method used for performance evaluation in machine learning for clustering algorithms such as k-means clustering, which examines the separation distance among resulting clusters and offers a visual means to evaluate parameters such as the number of clusters. The silhouette index in the plot effectively illustrates the proximity of each point within one cluster to points in neighboring clusters that fall within the range of [-1, 1].

Silhouette coefficients close to +1 imply that a sample is far from neighboring clusters, which means good separation. A value of 0 suggests the sample data is on or near the decision boundary between two adjacent clusters. In contrast, negative values imply the possibly wrong classification of a cluster. Furthermore, the thickness of the silhouette plot allows for the visualization of cluster sizes.

2.7 Summary

In this Section, we have delved into the intricacies of designing and implementing a real-time GPU monitoring and alerting system for large-scale distributed computing environments, focusing mainly on HPC clusters. By leveraging monitoring techniques, database technologies, and event stream management platforms, we aimed to address the need for efficient resource utilization and performance monitoring in these complex computing infrastructures.

The exploration began with an overview of Slurm, emphasizing its role in resource allocation, job scheduling, and management.

Subsequently, we provided insights into the architecture and specifications of three HPC systems at the CSC-IT Center for Science: Puhti, Mahti, and LUMI. These systems represent state-of-the-art computing clusters with diverse GPU configurations, highlighting the need for a flexible and scalable monitoring solution to

accommodate varying hardware architectures.

We then delved into monitoring systems, exploring essential techniques and tools such as /proc, Cgroups, NVML, and ROCm-SMI for capturing resource utilization metrics at both the CPU and GPU levels. Additionally, we discussed the significance of time-series databases like TimescaleDB and related techniques such as triggers, LISTEN & NOTIFY, and continuous aggregates in efficiently storing and querying monitoring data, enabling real-time analysis and visualization.

Finally, we explored various alert algorithms, including descriptive statistics, K-means clustering, silhouette analysis, decision trees, and random forests, as potential methods for identifying patterns or anomalies necessary to catch attention.

Chapter 3

Methods

Implementing real-time GPU resource monitoring and alerting involves systematically integrating various components. This Chapter provides detailed insight into how we construct such a system, including the architecture, design, and implementation, covering key elements such as the monitoring system, alert service, and algorithms.

3.1 Research process

This Section outlines the approach to achieving the thesis objectives, including designing, implementing, and evaluating the monitoring system and alert service for GPU utilization on HPC clusters.

1. **Requirement Analysis**: We begin with a thorough analysis of the monitoring system and alert service requirements. This involves understanding the needs of HPC administrators and users, and identifying key performance indicators for assessing GPU utilization and job efficiency.
2. **System Design and Architecture**: The system design and architecture are developed based on the identified requirements. This phase involves defining the components of the monitoring system, including data collection mechanisms, storage infrastructure, and alert generation algorithms. Special attention is given to ensuring scalability, reliability, and compatibility with existing HPC cluster environments.
3. **Implementation and Deployment**: With the system architecture finalized, the implementation phase commences. This involves developing the necessary software components for data collection, processing, and alert generation and integrating the monitoring system with the HPC cluster infrastructure. The deployment process includes configuration, testing, and validation to ensure the system operates effectively in a production environment.
4. **Data Collection and Analysis**: Once deployed, the monitoring system collects real-time data on GPU usage and job performance. This data is then analyzed to identify patterns of inefficient resource utilization and inform the development of alert generation algorithms. Statistical analysis and machine learning tech-

niques can be employed to extract insights from the collected data and optimize the performance of the alert service.

5. **Evaluation and Validation**: The effectiveness and reliability of the monitoring system and alert service are evaluated through comprehensive testing and validation. This includes assessing the accuracy of alerts, evaluating the system's responsiveness to dynamic workload conditions, and validating the impact of the alert service on improving GPU utilization and job efficiency.
6. **Feedback and Iterative Improvement**: Feedback from HPC administrators and users is solicited to identify areas for improvement and refinement. This feedback is incorporated into iterative cycles of system enhancement, enabling continuous improvement of the monitoring system and alert service over time.
7. **Documentation and Knowledge Transfer**: Finally, comprehensive documentation is prepared to facilitate knowledge transfer and ensure the sustainability of the implemented solution. This includes user manuals, technical specifications, and best practice guides to support HPC administrators in effectively utilizing the monitoring system and alert service.

Through this systematic methodology, the thesis aims to deliver a robust and effective solution for monitoring and alerting GPU resource utilization on HPC clusters, ultimately contributing to optimizing job scheduling, resource allocation, and overall system efficiency in GPU computing environments.

3.2 Monitoring system

One of the issues faced by HPC resource users is that it is hard for them to see how well they are using the resources. This is an even bigger issue with GPU nodes, where users often request GPUs without actually using them, or they may run jobs on the GPUs without putting any significant load on them.

Our monitoring system at the job level aims to improve observability and enable HPC system admins to find out these situations. It consists of several components with various roles that collect, process, store, and preset different metrics about jobs that are run in HPC systems:

- **Monitoring Server**: Runs on all compute nodes and polls the performance metrics of jobs.
- **Monitoring Client**: Used by the Slurm prolog and epilog script to initiate the job collection.
- **TimescaleDB**: PostgreSQL database with an optimized extension for storing the time-series data and used to store job metrics and metadata.
- **Timescale Ingest**: Receives the metrics data from all monitoring servers and stores them into the TimescaleDB.
- **Timescale Reader**: Backend API loads data from TimescaleDB, enabling frontend UI or command line interface to render job statistics.
- **Timescale Chart**: Web interface to render user usage graphs.

- **Seff**: Command line interface to render user usage tables.
- **Lmod**: Environment module system used at CSC, and we configured the module load hook to send a notification to the monitoring server every time a module is loaded in the hope that we can better use this information to help debug the job as well as do classification on those jobs in the future.

Communication between different components during the lifetime of a job is presented in Figure 3.1. This Section will then give a detailed description of the design and how they work together.

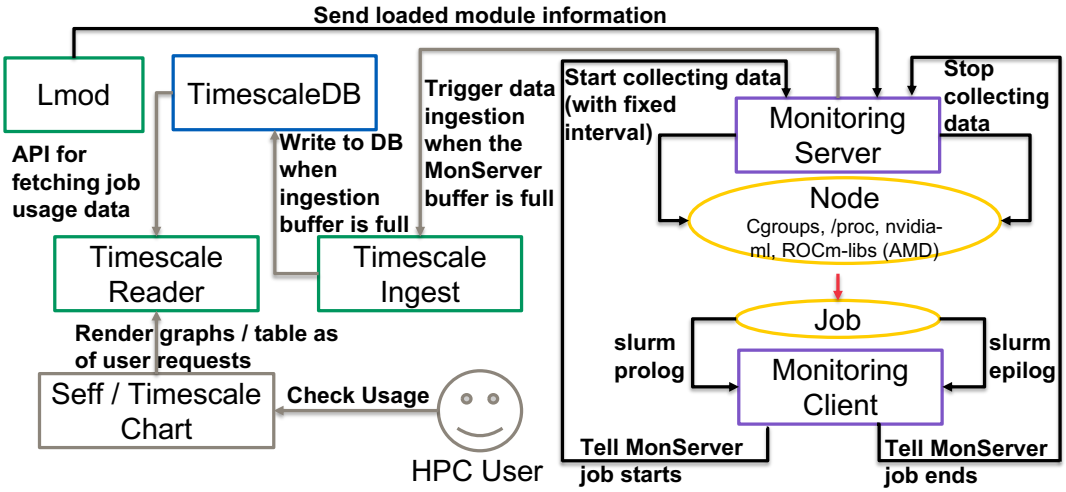


Figure 3.1: Monitoring system structure

3.2.1 Monitoring Daemon

Monitoring Daemon is written in C++ and consists of MonClient and MonServer. MonClient and MonServer are utilities that run on each compute node. They collect the job metadata and metrics and send them out from the node.

MonClient is a CLI utility that passes information about the job start and end to the MonServer process via UDP. MonClient commands are run in prolog and epilog scripts of jobs and tasks in Slurm.

MonServer runs continuously in the background of the compute nodes. It gets information about the job, such as hardware specifications, job ID, and step ID, from the client, collects the metrics of each job accordingly, and sends them to the Timescale Ingest server once the local buffer is complete so we can monitor how the actual hardware resources are used. The metrics that MonServer collects are as follows:

- **CPU usage**: Reads the CPU metrics from `/proc` for the cores assigned to the job. Since the number of CPU cores can be too large to analyze individual jobs, we can also use the aggregated result of the assigned cores per node.
- **Memory usage per job**: Gets the total memory usage from Cgroups for the job per node. This requires that Slurm is set up to use the Cgroups plugin.

- **Memory usage per process**: Uses the `ps` command line utility to get the memory usage of processes for a job. `ps` needs the job step information from prolog and epilog scripts passed by MonClient to get the process ID information.
- **GPU usage**: Reads GPU load, memory, power, temperature, and energy information from Nvidia Management Library / ROCm-SMI for the GPUs assigned to the job.

We have intentional delays in collecting the monitoring data to filter out the IO loading period caused by dataset loading and environment initialization so that monitoring can genuinely reflect the use of GPU. Also, those jobs only last for a very short time, so we only collect from those jobs with a meaningful length of data to analyze and reduce the burden of the database and maintain the stability of the whole monitoring infrastructure. Data sent to the Timescale Ingest will be tagged with the job ID, the username, and the step ID if it is for memory usage per process.

We also support dropping those jobs from the collection loop that have exceeded the maximum possible end time configured by the partition or after two weeks in case the job is killed accidentally and the Slurm epilog does not run.

3.2.2 Timescale Ingest

Timescale Ingest offers API for data ingestion through HTTP or UDP. We do symmetric encryption on the monitoring data. We derive a cryptographic key from an API key using the PBKDF2-HMAC-SHA256 (Combining Password-Based Key Derivation Function 2, Hash-based Message Authentication Code, and Secure Hash Algorithm 256-bit), and then use the derived key as the key for AES-GCM (Combining Advanced Encryption Standard and Galois/Counter Mode) encryption. The server is configured to handle incoming data according to the specified protocol. A UDP server is spun up in a separate thread (goroutine in Golang) if UDP is chosen. If HTTP is selected, the server creates a route to handle incoming POST requests to the **/write** API endpoint. The transmitted data format follows the InfluxDB line protocol [19].

We have an ingestion buffer to store the data temporarily in memory. Whenever we receive the data, we parse it into an instance of Go's structs. When the buffer is full, or after a timeout, we sort the buffer by timestamp, convert them into SQL insert statements, and execute them together. This allows us to reduce the database's write load in batches.

Timescale Ingest implements the graceful shutdown, where a Goroutine uses a channel to wait for an interrupt signal (SIGINT or SIGTERM). Once the interrupt signal is received, the server initiates a graceful shutdown process, which involves creating a context with a timeout of 5 seconds, during which the server attempts to finish handling any ongoing requests and stops receiving new updates. The server is forcibly shut down after the timeout or when all requests are completed. Once the server is shut down, any remaining tasks should be completed. This includes closing the database connection, sending all the data in the buffer, and logging.

In addition to the /write API endpoint as demonstrated above, the API design for the Timescale Ingest is as follows:

- **/version**: Displaying the version information and build time. If the current commit is tagged, `git describe` starts from the tagged commit and counts how many commits are on top of that tag. It then generates a string in the format of <*tag*>-<*number_of_commits*>-<*short_commit_hash*>, where the tag is the name of the closest annotated one reachable from the commit, the number of commits is the count between the tagged commit and the current commit. The short commit hash is the current commit, abbreviated commit hash (typically seven characters).
- **/status**: A JSON-encoded representation of the server's current status, including information about the metrics being ingested. The JSON output contains fields such as *ok* to indicate whether the operation was successful, *msg* to provide any additional messages if the service has any error, and *status* to contain the JSON-encoded heartbeat information with the key as follows:
  - **BufferSize**: The number of metric items currently stored in the buffer.
  - **BufferLimit**: The maximum number of metric items allowed in the buffer.
  - **CommitCount**: The count of commits made to the database.
  - **ReceiveCount**: The count of metric items received by the server.
  - **NextCommitTime**: The next time scheduled for committing to the database if the buffer is not full.
  - **LastCommitDuration**: The duration to commit the last buffer batch to the database.
  - **CommitMetric**: The time taken per metric item to commit the last batch to the database.
- **/healthStatus**: A liveliness check for connection status between the ingest and the database.

3.2.3 TimescaleDB

TimescaleDB, as the time-series database, introduces a relational aspect to monitoring. This component stores metrics in a structured manner, allowing for complex SQL queries and analysis. TimescaleDB accommodates the evolving nature of HPC workloads by enabling the retention of historical data. It also accelerates the query speed, essential for identifying trends and patterns over time.

Table 3.1 shows how we define the database table for storing the GPU usage data at the job level collected by MonServer. These give us a basic idea of how well the job performs using GPUs.

Column	Type	Explanation
timestamp	TIMESTAMP	Time when the data is collected
host	TEXT	Name of the host where data is from
GPU	INT	ID of the GPU in the host machine
username	TEXT	Name of the user that starts the job
job	INT	ID of the job that uses the GPU
load_gpu	INT	Instantaneous GPU core load (%, 0-100)
load_memory	INT	Instantaneous GPU memory load (%, 0-100)
used_mem	BIGINT	Instantaneous GPU memory in use (Byte)
total_mem	BIGINT	Instantaneous GPU memory in total (Byte)
power	INT	Instantaneous GPU power (mW for Nvidia, uW for AMD)
temperature	INT	Instantaneous GPU temperature (°C for Nvidia, m°C for AMD)

Table 3.1: Table for storing GPU usage

Table 3.2 shows how we define the database table for storing the unstructured job information collected by MonClient. The JSON string in the metadata column represents the data structure containing information related to jobs, tasks, and allocated hardware information. Each JSON object corresponds to a specific event or action within Slurm indicated by the type, such as starting or stopping a job, task, or event. Every type follows a similar structure, containing fields such as *method* (indicating the type of event), *slurmInternalID* (job ID assigned by the Slurm workload manager), *hostname* (name of the computing node), and *gpu_energy* (energy consumption data for GPUs), which contains the PCI Bus ID of each GPU as well as the energy counter when the event is triggered (in Millijoule, mJ). Here is a breakdown of each unique field of the types that happen in order during a job lifetime:

Column	Type	Explanation
timestamp	TIMESTAMP	Time when the data is collected
host	TEXT	Name of the host where data is from
job	INT	ID of the job that uses the GPU
type	TEXT	Type of the metadata (start/stop job/task, event)
metadata	TEXT	Actual key-value data in JSON format

Table 3.2: Table for storing Slurm job metadata

1. **start-job** indicates initiating a new job. It includes details such as the user name (*user*), user's ID (*uid*), the group ID that the user belongs to (*gid*), the partition that the job belongs to (*partition*), and resource allocations, e.g., GPU and CPU id lists that get allocated to the job). Additionally, it includes the GPU energy counter of all the GPUs that belong to the node.

```
{
  "method":"start-job",
  "slurmInternalID":8934,
  "hostname":"g1101",
  "user":"dowjohn",
  "uid":100567,
  "gid":100567,
  "partition":"gputest",
  "gpu":"1",
  "cpu":"0-255",
  "gpu_energy":[
    {
      "pciBusId":"00000000:03:00.0",
      "energy":1260156826
    },
    {
      "pciBusId":"00000000:44:00.0",
      "energy":999985444
    },
    {
      "pciBusId":"00000000:84:00.0",
      "energy":793570107
    },
    {
      "pciBusId":"00000000:C4:00.0",
      "energy":708110133
    }
  ]
}
```


2. **start-task**: indicates the start of a specific task within a job. It contains metadata such as the task's process ID, step ID (*-1* indicates the batch step), node-local task ID for the process within a job (*locaID*), number of processes in the job step or whole heterogeneous job step (*stepTasks*) and loaded modules, which records the modules name as well as their versions separated by a colon. It also includes the GPU energy counter assigned to the job.

```
{
  "method":"start-task",
  "slurmInternalID":8934,
  "hostname":"g1101",
  "taskPID":44689,
  "stepID":-1,
  "locaID":0,
  "loadedModules":"gcc/11.2.0:openmpi/4.1.2:openblas/0.3.18-
     omp:csc-tools:StdEnv",
  "gpu_energy":[
    {
      "pciBusId":"00000000:44:00.0",
      "energy":999996492
    }
  ]
}
```



```
{
  "method":"start-task",
  "slurmInternalID":8934,
  "hostname":"g1101",
  "taskPID":45279,
  "stepID":0,
  "locaID":0,
  "stepTasks":1,
  "loadedModules":"csc-tools:StdEnv:gcc/9.4.0:tensorflow/2.12:
     openblas/0.3.18-omp:openmpi/4.1.2:cuda/11.5.0",
  "gpu_energy":[
    {
      "pciBusId":"00000000:44:00.0",
      "energy":1000040684
    }
  ]
}
```


3. **event**: denotes the event that happens when the task is running. This can be that a new module is loaded in the job context.


```
{
  "method":"event",
  "slurmInternalID":8934,
  "stepID":0,
  "eventKind":"module-load",
  "eventField":"tensorflow/2.15"
}
```


4. **stop-task**: denotes completing or terminating a task within a job. Like the start task, it includes metadata for the task and the energy consumption of GPUs allocated to the task-related job during its execution.

```
{
  "method":"stop-task",
  "slurmInternalID":8934,
  "hostname":"g1101",
  "taskPID":44689,
  "stepID":0,
  "locaID":0,
  "stepTasks":1,
  "gpu_energy":[
    {
      "pciBusId":"00000000:44:00.0",
      "energy":1001166893
    }
  ]
}
```



```
{
  "method":"stop-task",
  "slurmInternalID":8934,
  "hostname":"g1101",
  "stepID":-1,
  "locaID":0,
  "gpu_energy":[
    {
      "pciBusId":"00000000:44:00.0",
      "energy":1001173148
    }
  ]
}
```


5. **stop-job**: indicates the completion or termination of a job. It includes metadata about the job and the total energy consumption of all the GPUs inside the job running node during its execution.

```
{
  "method":"stop-job",
  "slurmInternalID":8934,
  "hostname":"g1101",
  "gpu":"1",
  "gpu_energy":[
    {
      "pciBusId":"00000000:03:00.0",
      "energy":1261452017
    },
    {
      "pciBusId":"00000000:44:00.0",
      "energy":1001180831
    },
    {
      "pciBusId":"00000000:84:00.0",
      "energy":794875873
    },
    {
      "pciBusId":"00000000:C4:00.0",
      "energy":709266506
    }
  ]
}
```


We use TimescaleDB-specific features to improve query performance. We turn the metrics table into a hypertable with 6-hour chunking. We also create indexes on job IDs, hostnames, and GPU IDs, as those are the keys most commonly used for our SQL group queries. We compress data older than one day and drop data older than six months.

We define two triggers and corresponding notification functions in PL/pgSQL – SQL Procedural language. These triggers are designed to automatically send notifications whenever new data is inserted into table *slurm_job_metadata* and *gpu_usage*.

```
-- Notify of new job metadata
CREATE OR REPLACE FUNCTION notify_new_job_metadata_insertion()
  RETURNS trigger AS $notify_new_job_metadata_insertion$
BEGIN
  PERFORM pg_notify('job_updates', (NEW.host || '|' || NEW.job || '
      |' || NEW.type || '|' || NEW.metadata)::text);
  RETURN NEW;
END;
$notify_new_job_metadata_insertion$ LANGUAGE plpgsql;

CREATE TRIGGER job_metadata_insertion_notify_trigger
AFTER INSERT ON slurm_job_metadata
FOR EACH ROW EXECUTE FUNCTION notify_new_job_metadata_insertion();

-- Notify on new gpu_usage_aggregate data
CREATE OR REPLACE FUNCTION notify_new_gpu_usage_insertion()
  RETURNS trigger AS $notify_new_gpu_usage_insertion$
BEGIN
  PERFORM pg_notify('gpu_usage_insertion', (NEW.host || ',' || NEW.
      gpu || ',' || NEW.job || ',' || NEW.username || ',' || NEW.
      load_gpu || ',' || NEW.load_memory || ',' || NEW.used_mem || '
      ,' || NEW.power || ',' || NEW.temperature)::text);
  RETURN NEW;
END;
$notify_new_gpu_usage_insertion$ LANGUAGE plpgsql;

CREATE TRIGGER gpu_usage_insertion_notify_trigger
AFTER INSERT ON gpu_usage
FOR EACH ROW EXECUTE FUNCTION notify_new_gpu_usage_insertion();
```

- **notify_new_job_metadata_insertion()**: This function retrieves the newly inserted row and constructs a notification message using concatenation (||) separated with | to avoid any collision with the JSON format text (metadata). The message includes all the fields except the timestamp from the inserted row. Finally, the *pg_notify()* function is called to send a notification to a specific channel named *job_updates*.
- **job_metadata_insertion_notify_trigger**: This trigger fires after each insertion into the *slurm_job_metadata* table. It is associated with the *notify_new_job_metadata_insertion()* function, causing the function to execute automatically whenever new data is inserted into the table.
- **notify_new_gpu_usage_insertion()**: Similar to the first function, it constructs a notification message using all the fields except the timestamp, username, and total_mem from the inserted row separated by commas (,). It then notifies through the *gpu_usage_insertion* channel.
- **gpu_usage_insertion_notify_trigger**: This trigger fires after each insertion into the *gpu_usage* table and is associated with the *notify_new_gpu_usage_insertion()* function, triggering it automatically upon insertion of new data.

These triggers and notification functions facilitate real-time communication within the database system. They enable other parts of the system to be notified instantly whenever new job metadata or GPU usage data is inserted, allowing for timely updates and actions based on the newly inserted data.

3.2.4 Timescale Reader

The Timescale Reader component facilitates the retrieval of metrics from TimescaleDB for analysis and reporting. It provides a RESTful API for building other elements that support administrators in gaining insights into historical resource utilization, aiding in capacity planning, performance optimization, and trend analysis. It also enables the integration of other components to show statistics to users via GUI or CLI. The Timescale Reader complements the real-time monitoring capabilities, providing a comprehensive view of metrics across different time intervals.

We have a centralized web page that allows users to use the APIs provided by Timescale Reader to check job history data with interactive graphs about the GPU load, memory, power, and temperature, as shown in Figure 3.2. Figure 3.3 shows one example of a graph rendered via Timescale Chart through Timescale Reader API data. It also has accessibility support to help viewers with vision deficiencies (e.g., color blindness or partial sight) more easily understand the data with patterns and gradients, as shown in Figure 3.4.

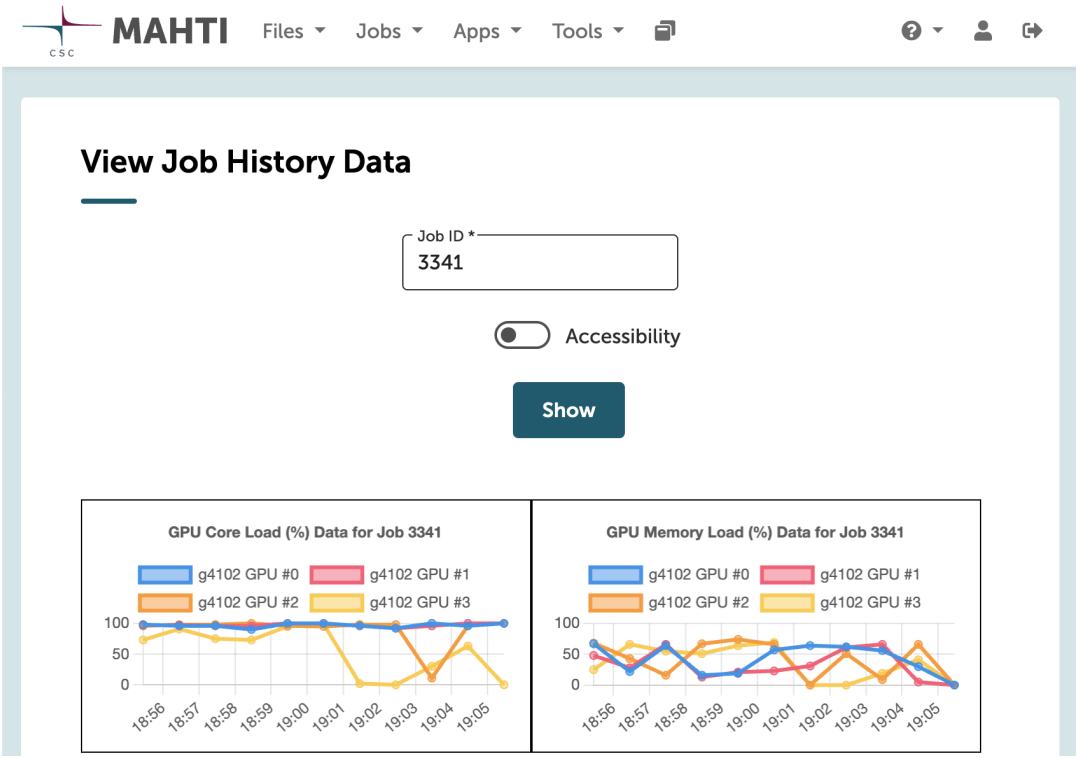


Figure 3.2: GPU usage history checking dashboard

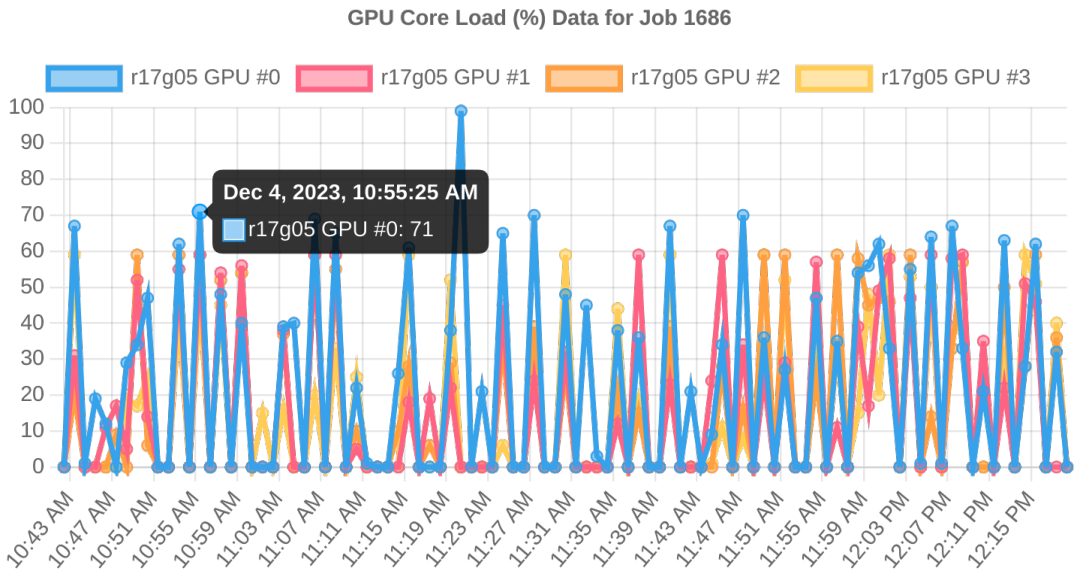


Figure 3.3: GPU usage history graph from Timescale Chart

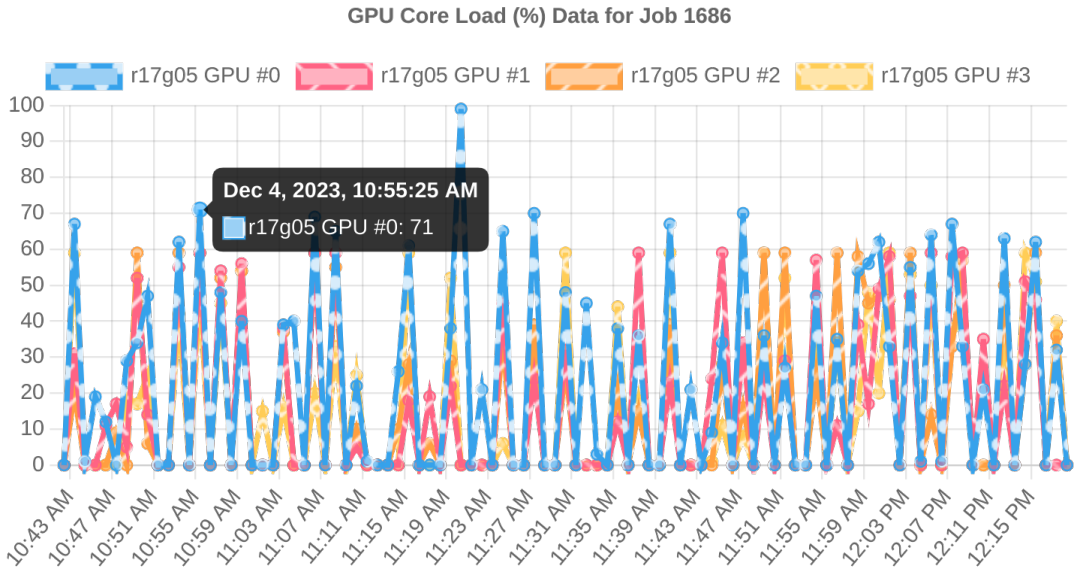


Figure 3.4: GPU usage history graph from Timescale Chart with accessibility

Below is an example showing the output of our modified *seff* command. The GPU job efficiency section is printed via the Timescale Reader API. Here, we show the metrics related to the GPU, including the load, memory, and energy. Each entry shows the hostname, the GPU ID about the metrics, and the aggregated mean, standard deviation, and maximum value for the whole job history.

```
$ seff 5465
Job ID: 5465
Cluster: mahti
User/Group: johndoe/pepr_johndoe
State: COMPLETED (exit code 0)
Nodes: 1
Cores per node: 16
CPU Utilized: 19-19:45:45
CPU Efficiency: 82.58% of 24-00:05:52 core-walltime
Job Wall-clock time: 1-12:00:22
Memory Utilized: 14.41 GB
Memory Efficiency: 22.51% of 64.00 GB
Job consumed 3600.61 CSC billing units based on the following used
   resources.
Billed project: project_1008888
Non-Interactive BUs: 3600.61
GPU BU: 7201.22
NVME BU: 38.89
GPU job efficiency:
----------------------------------------------------------------------
GPU load
    Hostname      GPU Id      Mean (%)    stdDev (%)        Max (%)
       g5101           0         86.21         18.69          99.00
       g5101           3         78.53         19.65          97.00
----------------------------------------------------------------------
GPU memory
    Hostname      GPU Id    Mean (GiB)  stdDev (GiB)     Max (GiB)
       g5101           0         22.64          0.00          22.64
       g5101           3         22.32          0.00          22.33
----------------------------------------------------------------------
GPU energy
    Hostname      GPU Id   Energy (Wh)
       g5101           0       8532.60
       g5101           3       8664.34
----------------------------------------------------------------------
```

The API design for the Timescale Reader is as follows:

- **/version**: Same as Timescale Ingest, it displays the version information and build time.
- **/status**: Liveness checking endpoint.
- **/chart**: Serving the static files for Timescale Chart JS web interface (an encapsulated web component using Chart.js), available parameters can be referred from Table 3.3.
- **/getJobs**: Endpoint for listing all the available job IDs in the database with valid GPU monitoring data, meaning those jobs are long enough to view data.
- **/getData/:table/:jobid**: Fetching the raw (unaggregated) history data in JSON format to be rendered by Timescale Chart.
- **/getLoad/:metric/:resource/:jobid/:type**: Displaying aggregated result of the history monitoring data.

- **/getGPUEnergy/:jobid**: Displaying the GPU energy counter.

Table 3.3: Timescale Chart parameter description

Parameter	Description
api	API endpoint to read data from, such as `http://localhost:8001/getData`. Default to be `/getData` at the same site.
domain	Database table name to read from, such as `gpu_usage`.
job	Job ID to be displayed.
title	Title of the chart.
group	Grouping of the data for different domain datasets, such as `core` (for `CPU usage`), `gpu` (for `GPU usage`), `pid` (for `Memory usage by PID`).
metric	Column name of the data to be displayed in the specific table.
name	Name of the group that will be displayed.
divide	Value to divide the data with.
mode	Chart zoom and pan mode, available values are `xy`, `x`, `y`. The default is `x`, which means zoom only at the x-axis.
type	Chart graph type, default is `line`

The parameters we support for displaying the aggregated result of the history monitoring data, as well as the GPU energy counter, are as follows:

- **display**: 1 for printing in human-readable format, 2 for printing a table (similar to human-readable format but separated by tabs), and any other values will be in JSON format organized by a list of values using the hostname as the top level and hardware ID as the second level.
- **unit**: Specify the unit of the value to be printed.
- **type**: Specify the name of the value.
- **metric**: Specify the metric of the value (average, minimum, maximum, standard deviation, etc.), support multiple metrics separated by , (comma).
- **divide**: Divide the data value stored in the database by this specified number.
- **precision**: Specify the precision of the value to be printed. The default is 2. -1 for no rounding.
- **index**: Specify the device's index to get printed. The default is all devices.
- **hide_device**: 1 for hiding the `[device]`, other values for printing the `[device]`.

The human-readable printing format is as follows. Note that #[index] will only get printed when there is more than one.

[device] #[index] ([type] [metric]) : [value] [unit]

3.3 Alert algorithms

The Alert algorithms component comprises predefined algorithms, that determine the conditions under which alerts are triggered. These algorithms consider various factors, including GPU utilization thresholds and temperature limits. The flexibility of the alert algorithms allows for customization — based on the specific requirements of HPC clusters — ensuring that alerts are triggered for conditions deemed critical by administrators.

Regarding the design of the alert algorithms for GPU usage: many high-performance computing applications have load-balancing issues regarding pipeline parallelism, and the data has highly fluctuated characteristics for multiple GPU jobs. Thus, alerting only according to average is neither reliable nor practical, as it may not be fixable easily by the user, so those can be non-critical. Machine learning models could be one way to address the data fluctuation issue by recognizing the data pattern. Algorithm 1 defines how we can use them.

Algorithm 1: Checking GPU load alert based on machine learning

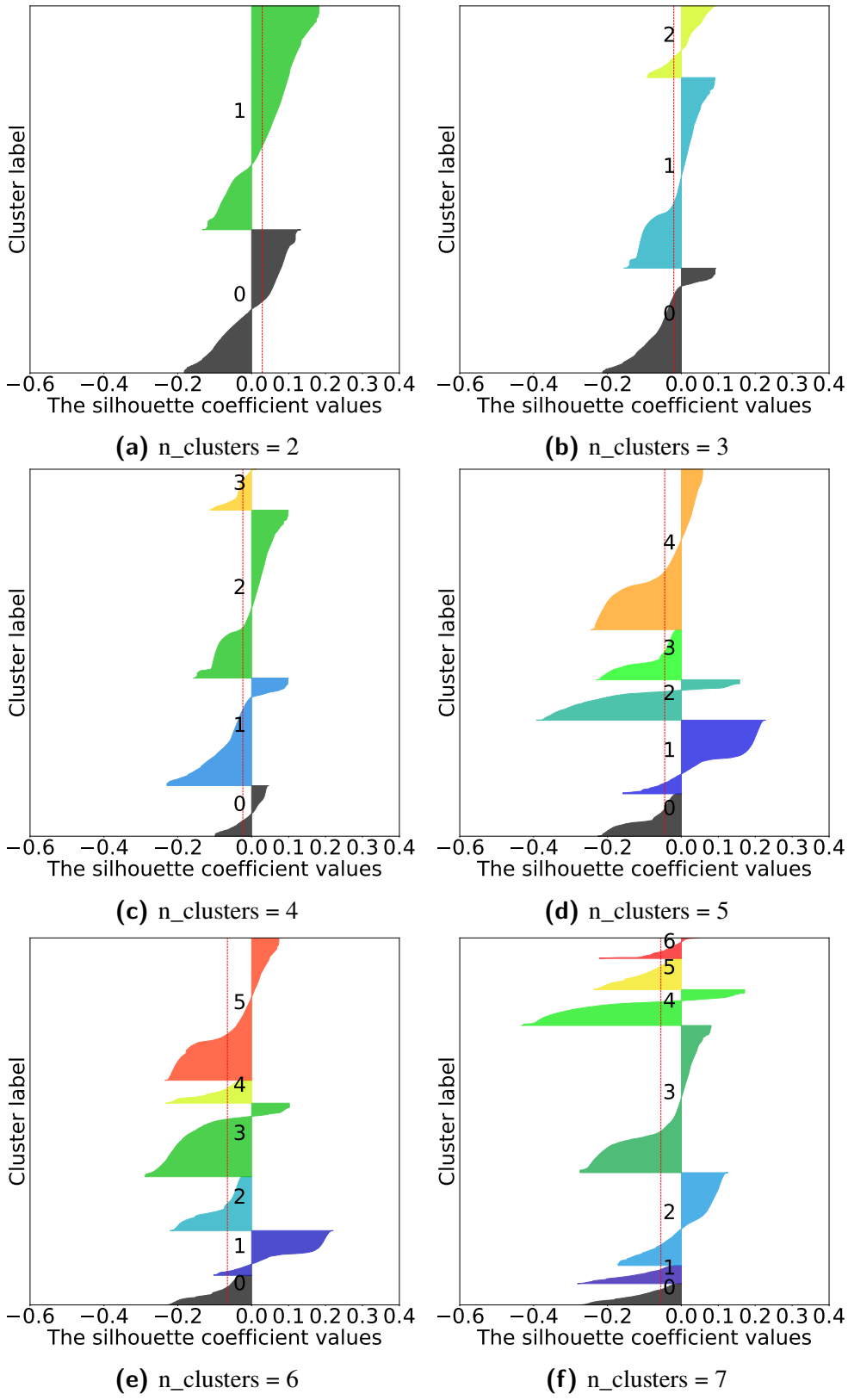
Data: Array, Fixed-size sliding window of latest GPU usage history: A
Result: Boolean, indicating if an alert should be raised

```
Function mlAlgo(array):
    Run a machine learning algorithm with an input of array;
    return classification result (0 or 1) for good or bad jobs;
Function checkAlert(A):
    R ← mlAlgo(A);
    if R == 1 then
        return true;
    end
    else
        return false;
    end
```

Unsupervised machine learning, such as reinforcement learning, is hard to train and interpret. Deep learning algorithms are also very slow to run, and do not fit our need for real-time job analysis. Most importantly, we cannot find a good reward function. The only way is from human feedback.

For supervised machine learning, such as random forests that involve decision trees, although it can be much faster to run, all the data collected from the monitoring system is unlabeled. It is unrealistic for humans to label all those data manually.

In the hope of doing labeling automatically, we also did the silhouette analysis of K-means clustering on the collected GPU monitoring data starting from December 2023 directly, as shown in Figure 3.5, as well as features generated with statistical aggregation, as shown in Figure 3.6. Each color in the figure represents a cluster. The red vertical line denotes the average silhouette coefficient value across all clusters. More explanation of silhouette analysis can be found in Subsection 2.6.4. The result shows that neither of the methods works well since most clusters have the majority proportion of negative coefficient value, and they cannot be reasonable classifications.



(a) n_clusters = 2 **(b)** n_clusters = 3

(c) n_clusters = 4 **(d)** n_clusters = 5

(e) n_clusters = 6 **(f)** n_clusters = 7

Figure 3.5: Silhouette analysis of KMeans on raw windowed GPU load data

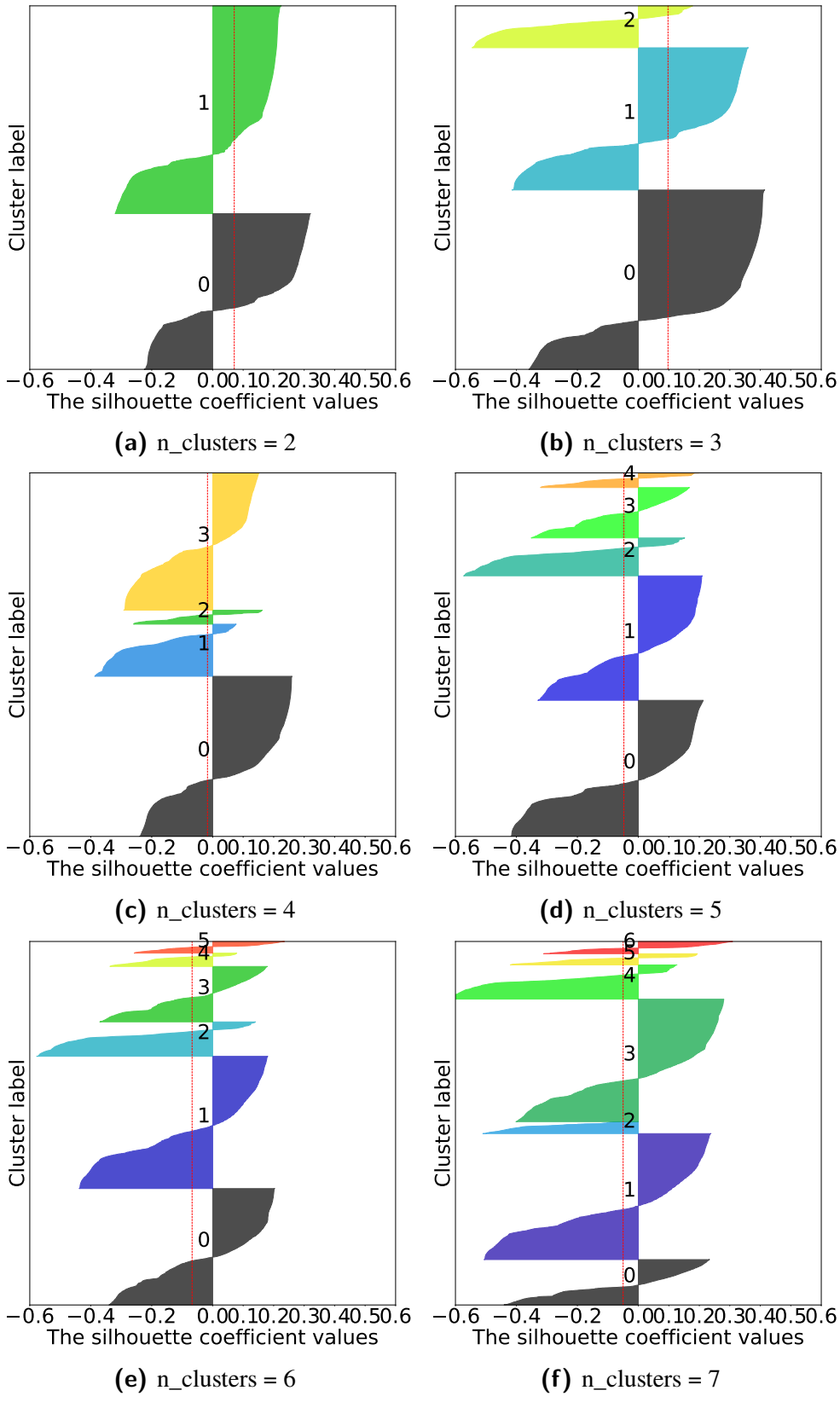


Figure 3.6: Silhouette analysis of KMeans on windowed GPU load statistics

Ultimately, we used more descriptive statistics to help understand its central tendency, spread, and shape to aid alerting. We can use nine descriptive statistics metrics to provide valuable insights into the characteristics of the data: percentiles (25%, 50%, 75%), kurtosis, maximum, minimum, skewness, standard deviation, and variance, as defined in Subsection 2.6.1.

Figure 3.7 and Figure 3.8 show the histogram distribution graph. The graph shows that kurtosis, mean, skewness, and standard deviation have a central tendency. We cannot find a good separation for these data, so these metrics cannot separate good and bad jobs. Combined with the analysis of the monitoring data in production and the actual need for job alerting, which is to find out the bad jobs that can be significantly improved, we eventually fine-tuned and discovered that 20 in the 75th percentile and 32 in the maximum are good separations, as defined in Algorithm 2.

Algorithm 2: Checking GPU load alert based on percentile and maximum

Data: Array, Fixed-size sliding window of latest GPU usage history: A
Result: Boolean, indicating if an alert should be raised

Function `getPercentile`$(array, divide)$**:**
    return $array[\lfloor(\text{len}(array) - 1)/divide\rfloor]$;
Function `checkAlert`(A)**:**
    $array \leftarrow$ Sort A in ascending order;
    $P_{75} \leftarrow$ `getPercentile`$(array, \frac{4}{3})$;
    $M \leftarrow$ `getPercentile`$(array, 1)$;
    if $P_{75} < 20$ ***or*** $M < 32$ **then**
        return true;
    end
    else
        return false;
    end

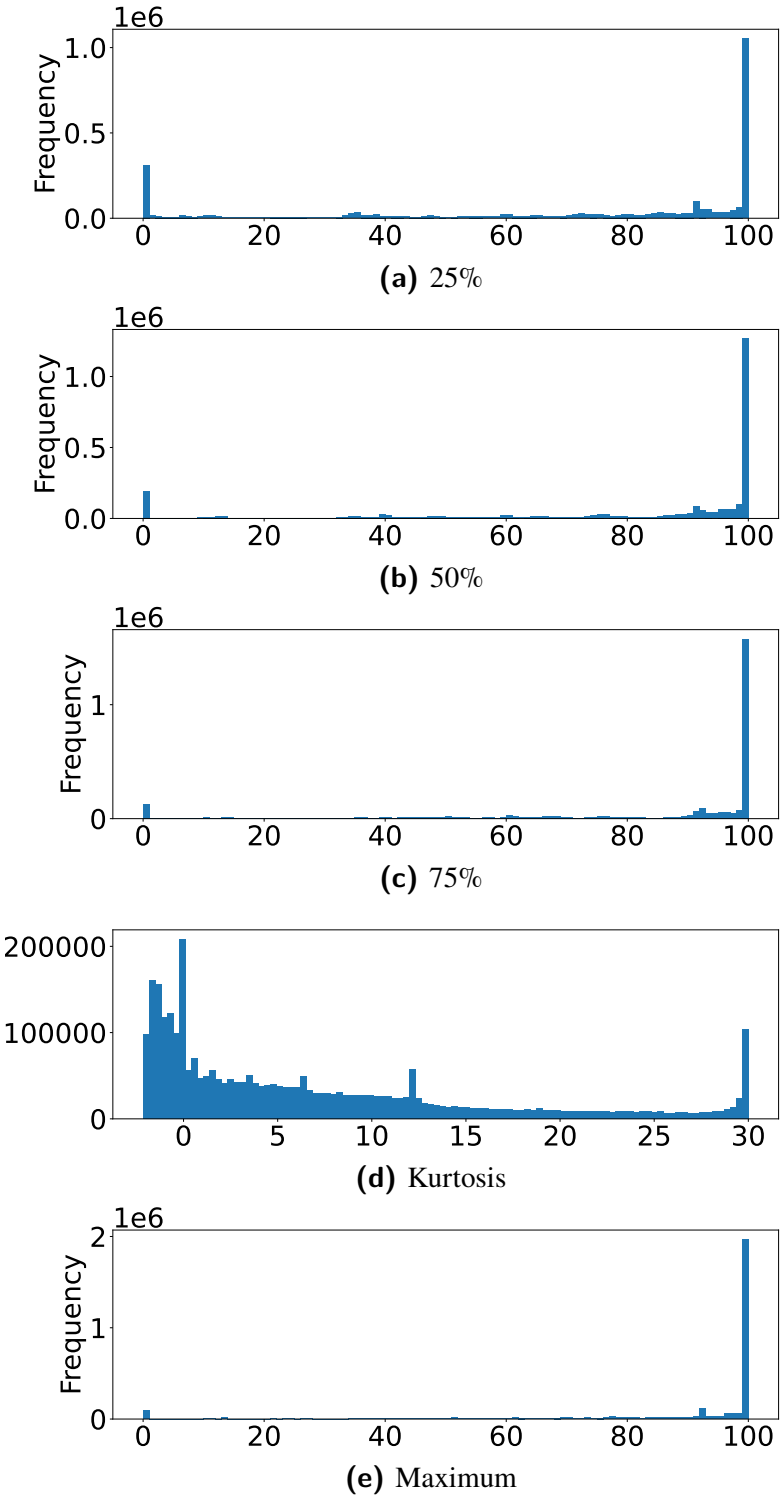
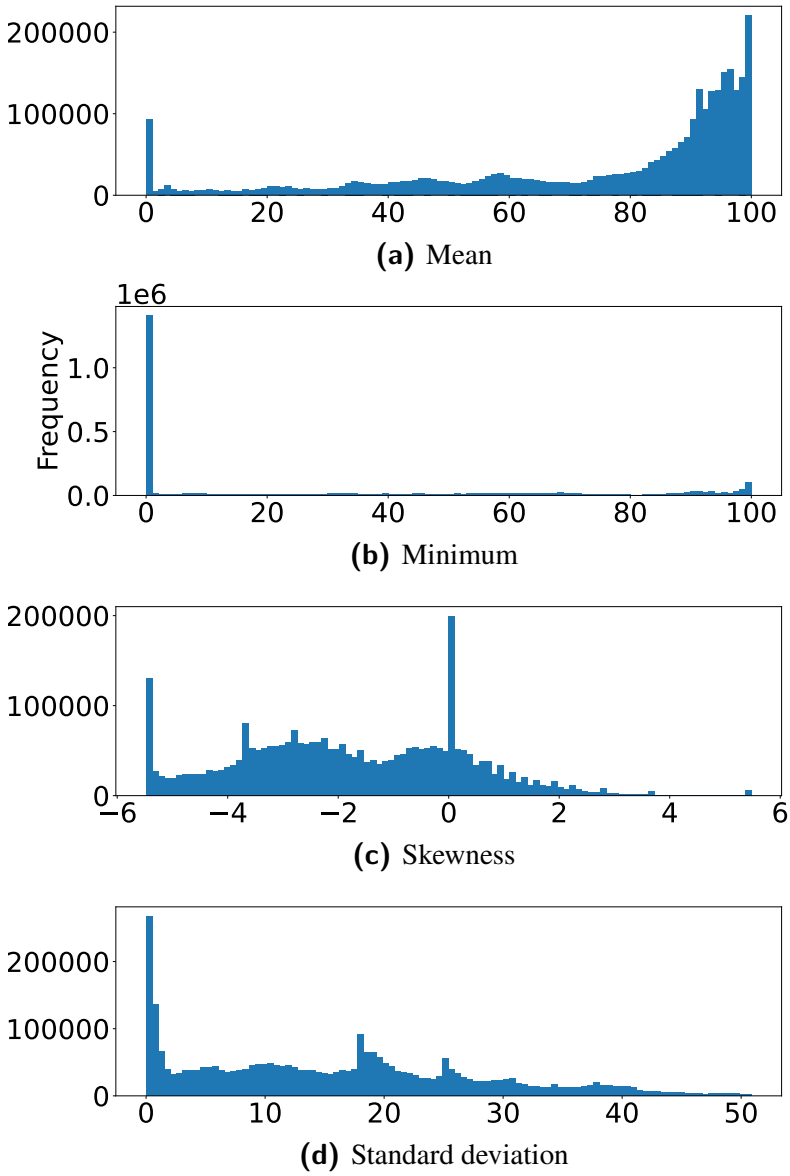


Figure 3.7: Histogram of statistics analysis on windowed GPU load data, Part 1



(a) Mean

(b) Minimum

(c) Skewness

(d) Standard deviation

Figure 3.8: Histogram of statistics analysis on windowed GPU load data, Part 2

3.4 Alert service

The alert service we implemented is built on the monitoring system. A good design is crucial for addressing all the research questions in Section 1.3.

For **maintaining the internal state** for tracking different jobs, The alert service can create or destroy the internal state for specific jobs by following job metadata updates. Whenever we have a new alert, or the alert is dismissed, we write the event into logs. We garbage collect stale jobs to avoid memory leaks. We get the aggregation result as soon as data are inserted. For the dashboard RESTful API integration, handling the one-writer, multiple-reader problem can be very complex if we directly let

the API server access our internal state, as starvation can quickly happen, thus causing significantly reduced performance. In the end, we chose to convert the issue into the one-writer, one-reader problem by dumping the alert state with a thread separately at regular intervals into the JSON string, so that we can easily tackle the issue by simply using the thread-safe map and mutual exclusion lock, to solve **RQ3** in Section 1.3.

For **following the job data updates**, since message queues introduce a single point of failure, and can be challenging to debug, we decided not to use them. This increases the difficulty of our design, but aside from polling, we do have an alternative: TimescaleDB, which is based on PostgreSQL. PostgreSQL has LISTEN and NOTIFY, so we can use triggers to execute NOTIFY after the insertion of each row, and Timescale Alert can LISTEN to the updates. We also have continuous aggregates in TimescaleDB [47], which can also be our choice.

As a result, we devised four solutions combining polling/triggers, and with or without continuous aggregates, to follow the job data updates, so that we can balance between **RQ1** and **RQ2** as mentioned in Section 1.3.

- **Polling with continuous aggregates**: Start/Stop polling as soon as job metadata updates. Use the continuous aggregates feature offered by TimescaleDB.
- **Polling with SQL aggregates**: Start/Stop polling as soon as job metadata updates, directly use SQL for aggregates (without using continuous aggregates for caching).
- **Triggers with continuous aggregates**: use the continuous aggregates feature offered by TimescaleDB and query the aggregated result according to the information sent by the **NOTIFY** as soon as the message is received.
- **Triggers with in-memory aggregates**: Since our monitoring system can ensure that data arrives in order and has evenly distributed intervals, we can use a fixed-size container/ring (Circular Linked List) as the sliding window to store the history data, and do the aggregation by Golang. Here, we listen to the metrics data sent by the **NOTIFY** from the database and update the internal state in memory.

Here is an empirical analysis of the four design choices mentioned above: We can choose either polling or triggers. With polling, the alert delay increases and will likely burden the database heavily with read operations. The alert delay is minimized with triggers, but we might slow down the database for writing operations because of the transactional overhead.

For continuous aggregates in TimescaleDB, Spark [40], and Flink [6], after investigation, we found they do not have sliding-window aggregation support with a high-level API:

Since we want real-time alerts, we want to aggregate with a fixed-size sliding window that considers the latest data and drops the old data at any time, as shown in Figure 3.9. However, TimescaleDB continuous aggregates, Spark, and Flink high-level API can only aggregate with a fixed start time, as shown in Figure 3.10, which will work badly if we want to check the latest aggregation result at o'clock in this case since we will only have one data to aggregate in that window (only data at 11

o'clock as shown in the figure). Then, it is not very sensible to have an aggregation. Although the sliding window is not supported, we can overcome this issue by using a weighted average on the last two windows. However, it is still not a good candidate since triggers for continuous aggregates are not supported currently, as confirmed by the issue the author opened[1]. So we will have to burden the database by polling, which means many read operations plus many background jobs.

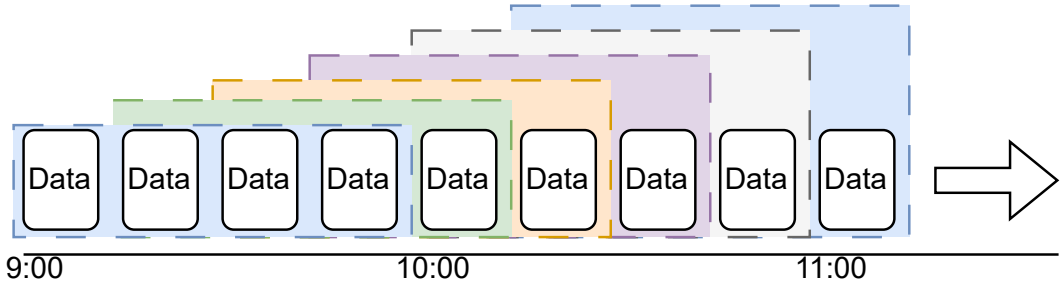


Figure 3.9: Sliding-window aggregation on the last 1 hour's data

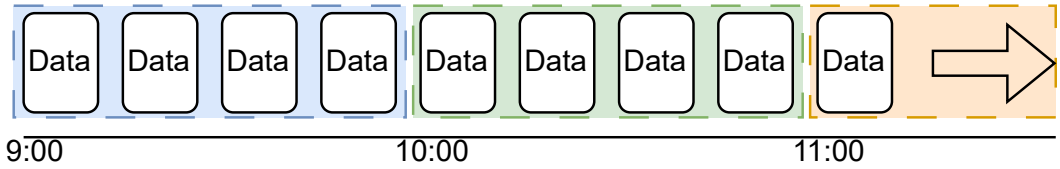


Figure 3.10: Aggregation with a fixed start time

We need to benchmark all four design choices to reach a final decision, which will be further discussed in Subsection 4.1.2.

The **API design** of the Timescale Alert is as follows, which provides the data for other services and allows easier integration:

- **/version**: Same as Timescale Ingest, it displays the version information and build time.
- **/healthStatus**: Checking connection status between the timescale ingest and the database.
- **/dismiss/:host/:job/:gpu**: Disable or enable the alert for the specified GPU on the host from a specific job ID.
- **/history**: Displaying a list of JSON objects or a table that shows the alert history. It can be filtered by the host, username, job ID, and type.
- **/**: Displaying the current alert status. Host, username, and job ID can also be the filter.

Below is an example showing the GPU alert history output section of the modified `seff` command we added for Timescale Alert. The alert information is printed via the

[1]https://github.com/timescale/timescaledb/issues/6500

above-mentioned `/history` endpoint. Here are a few alert histories related to GPU usage. Each entry shows the time when the alert happened, the hostname, and the GPU ID that generated the alert event. It also shows the 75th percentile aggregation result for the sliding window time frame that contributes to the alert status change, and the `Normal` column shows whether this entry generates a new alert (with `x`) or clears the old alert (with `v`).

```
$ seff 5465
GPU Alert History
------------------------------------------------------------------------
GPU Usage
                        Time    Hostname    GPU Id    75% (%)    Normal
2024-04-25T17:05:55+03:00      r14g04         2          5       x
2024-04-25T17:06:41+03:00      r14g04         2         21       v
2024-04-25T17:10:49+03:00      r14g05         1          8       x
2024-04-25T17:11:17+03:00      r14g05         1         23       v
------------------------------------------------------------------------
```

3.5 Alert dashboard

The alert dashboard reads data to display and learn about the current job status. It renders tables through the web page to internal users, such as CSC user support experts. As shown in Figure 3.11, the dashboard has a counter for the viewers to know the total number of GPUs in use, and how many are currently on alert.

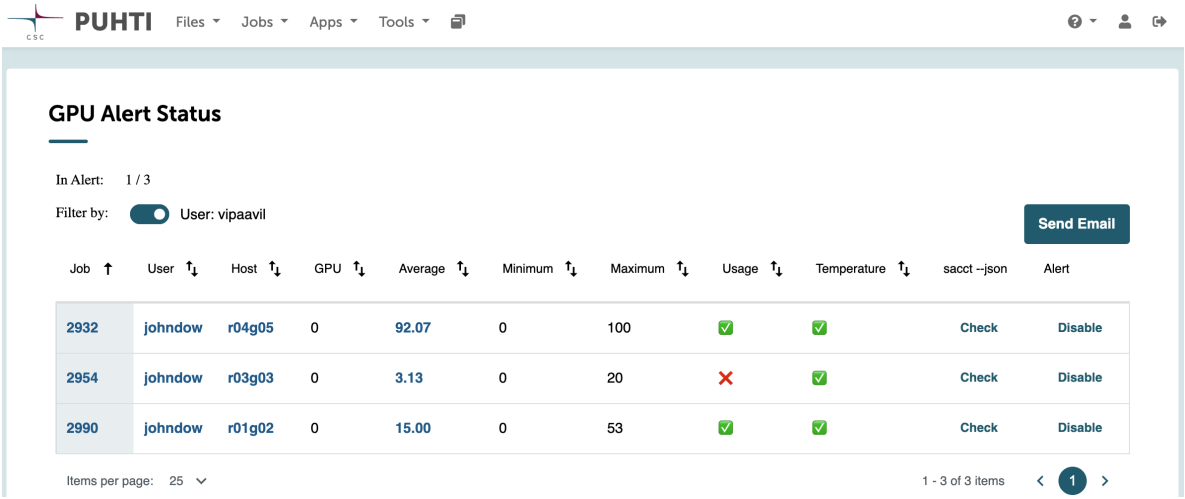


Figure 3.11: GPU alert status dashboard

In addition, the dashboard allows users to disable or re-enable job alerts, so that admins can improve the alert manually and focus more on those jobs, that have not been checked or can be improved. The users can also click on the button to check more detailed JSON information about jobs from the output of `sacct --json`. (We

don't render that information directly into the table, since the SQL database behind Slurm runs full steam for several seconds when calling that, even for a single job)

Besides showing the alert status, the dashboard also displays the watermark information for the job lifetime metrics, such as the average, minimum, and maximum GPU load, to give admins a whole picture of what might happen to the job.

Aside from sorting the data, the dashboard allows the viewers to filter the data by job ID, user name, hostname, and alert status, so viewers can quickly target different entities in question. When we filter by user name or job ID, the dashboard also allows admins to send emails to the user in question, with one click, from the preset template, for the current jobs in the alert. One example email generated by the dashboard, when we click the button in Figure 3.11, is as follows:

```
To: johndow@users.csc.fi
Subject: Low GPU utilization rate with the job you are running on Puhti

Dear johndow,

We have noticed that you have a low utilization rate at least during the
last 30 min, according to our monitoring system, for the jobs you are
running on Puhti:

- GPU 0 on host r03g03 for job 21850954, with lifetime average 3.13%,
maximum 20%, miminum 0%.

Please review them and make improvements at your earliest convenience. We
recommend checking your job scripts and programs to ensure that the jobs are
running as intended.

You might also want to consider the following:
1. Running "seff <job_id>" on the login node to check the job's resource
usage.
2. Find out the history of the job's GPU data by visiting:
https://puhti-ood-testing.csc.fi/pun/sys/dashboard/custom/job_monitor
3. Check the job's output and error logs for any error messages.
4. Reduce the number of GPUs requested in your job script.

if the situation continues and we receive no response from you, we might
take actions to ensure fair usage for all our users. Thank you for your
cooperation! If you have any questions or need help, please feel free to
contact us!

Best regards,
CSC computing services
```

We also have a dashboard showing the alert history, as presented in Figure 3.12, which shows the alert status change time, job ID, user name, hostname, GPU ID, alert type, alert value (for usage alert, it will be the 75% percentile), and whether the status change generates a new alert or clear out an old one. The history dashboard also shares the same features as the status dashboard.

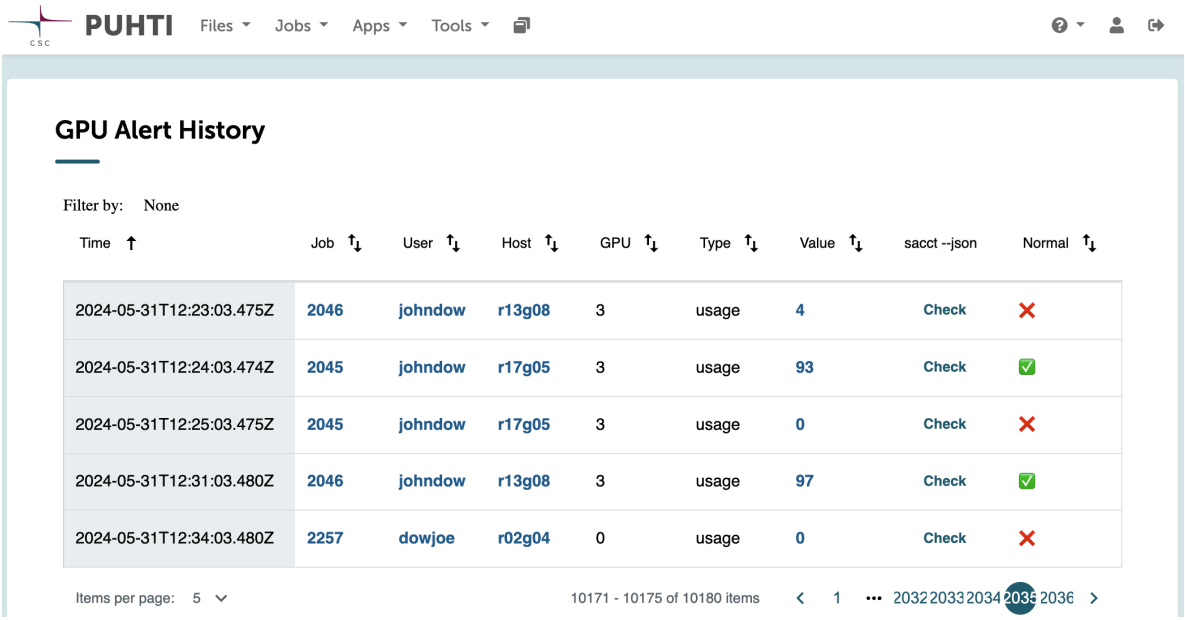


Figure 3.12: GPU alert history dashboard

Chapter 4

Results and analysis

This Chapter presents the results of benchmarking the real-time GPU resource monitoring and alerting system in experimental and production setups. The evaluation includes a comprehensive analysis of the system's performance, scalability, and impact on energy efficiency and resource utilization for production within High-Performance Computing (HPC) clusters.

4.1 Benchmark

4.1.1 Experimental setup

The experimental setup involves deploying the proposed GPU monitoring system in a controlled environment, that emulates the characteristics of a typical HPC cluster. The monitoring infrastructure continuously collects and analyzes the simulated GPU metrics, and an alert service is configured to notify any sub-optimal GPU utilization.

To integrate all the components for testing, we deployed Slurm in a containerized environment with an isolated network to simulate the actual HPC job context. Everything, including the software versions, replicates what we have in production. We use Podman as the container runtime and Rocky Linux 8 as the base container image to keep up with the operating system we use in Puhti and Mahti (RHEL 8).

We have two compute nodes in our test environment. All the containers are listed as follows:

- **mysql**: Database for Slurm to store job data.
- **slurmctld**: Central controller daemon of Slurm monitors all other Slurm daemons and resources, accepts jobs, and allocates resources.
- **slurmdbd**: Interface to the MySQL database for Slurm can be used to archive accounting records.
- **cpn01**: Compute node 1 for executing actual jobs received by Slurm.
- **cpn02**: Compute node 2 for executing actual jobs received by Slurm.
- **frontend**: Similar to the login node in HPC for submitting jobs.
- **timescaledb**: TimescaleDB instance for storing the monitoring data.

- **timescaleingest**: Timescale Ingest instance for ingesting the monitoring data into TimescaleDB.
- **timescalereader**: Timescale Reader instance for fetching the monitoring data and serving as the API for displaying to end users.
- **timescalealert**: Timescale Alert instance for checking alerts and displaying related information.
- **ondemand**: Open OnDemand serves as a web-based UI to computer clusters in HPC, in addition to SSH access, to help new users who are not familiar with Linux get started quickly. It offers an intuitive entry point for straightforward tasks and shell access for more intricate operations.
- **ldap**: Lightweight Directory Access Protocol (LDAP), for access control in Open OnDemand.

To make testing easier with an arbitrary number of GPUs in a container without needing them, we created a fake Nvidia library that implements the Nvidia-ML interface for generating random data.

We also have separate testing setups, either by randomly generated data or by replaying the data we collected in production from the database dump.

- For the random data benchmark, we spawn maximum 23824 (2978*8) Go routines that write to the database simultaneously, which is, i.e., the maximum GPU number we have in LUMI, to simulate the heaviest situation we can have in the pre-exascale supercomputer.
- For real data replaying, we read all the records from the database dump ordered by time stamp, and send it to timescale ingest with accelerated speed to reproduce the production situation.

4.1.2 Performance

The benchmarking process focused on evaluating the system's responsiveness in detecting anomalies and the overall impact on system performance during monitoring, to assess the system's adaptability and robustness in different usage patterns.

We use the local environment to run the container setup mentioned in Subsection 4.1.1 on the 12th Gen Intel Core i7-12700H CPU with SSD and 16 GB of RAM by writing random GPU metrics data to the database continuously in an infinite loop, without any interval for writing and dumping data. We run each solution for 10 minutes and take the average of the last 2-minute delay. The benchmark result is shown in Figure 4.1 (We apply the logarithmic scale on both the x and y axes):

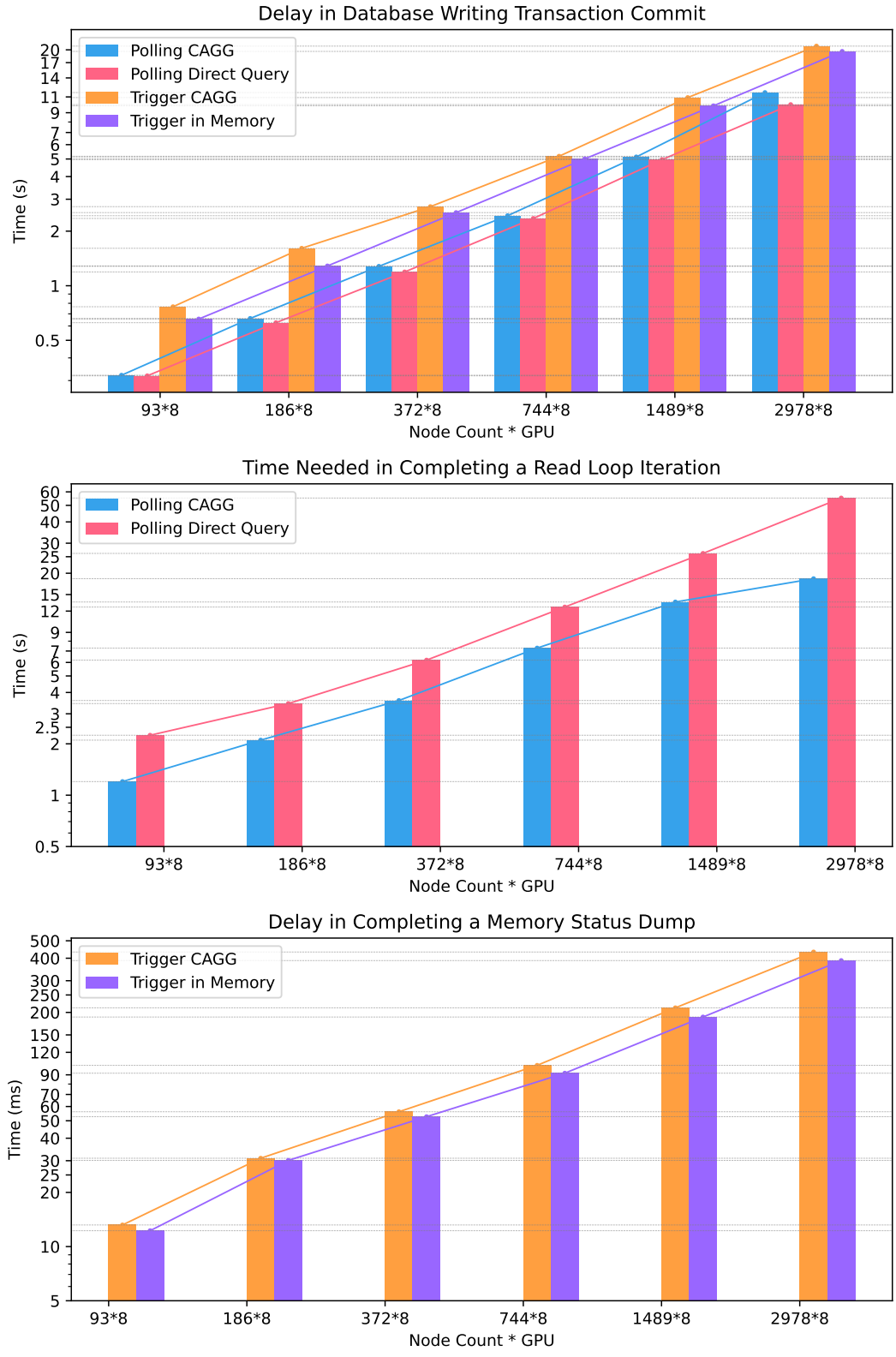


Figure 4.1: Timescale Alert benchmark result

From Figure 4.1, we can learn that:

- Generally, all the delays scale linearly with the number of GPUs.
- Triggers generally double the delay in database transaction commit time for writing, which the overhead of NOTIFY should be the cause.
- Continuous aggregates (CAgg) cause a slight increase in writing delay and the memory status dump delay for the triggers solution, which is likely to be caused by the overhead of CAgg background jobs. However, CAgg accelerates the query speed dramatically compared to traditional SQL queries aggregate, as we can see a significant time reduction in completing a read loop among all the jobs available.
- Polling is very inefficient for accessing data in real-time, since it creates a delay that is almost 100x more than the triggers solution.

Hence, the results confirmed our empirical analysis in Section 3.4 and led to our final design after comparison: **triggers without continuous aggregates**. The reason is that triggers have the best real-time access, while only doubling the delay in writing compared to polling, which is acceptable, and the in-memory solution has a better performance in both read and write delay. Most importantly, the in-memory solution allows us to access raw data for more complicated statistics calculations and use machine learning models. We used the Golang data race detector to ensure the correctness during benchmarking and testing. We also checked the memory usage during the benchmark: The maximum Resident Set Size (RSS) is around 84.6 MB, which is also acceptable.

4.2 Production

The production evaluation involved deploying the real-time GPU monitoring system in a live HPC environment, including Puhti and Mahti. This evaluation allows us to test the practicability of alerting for real-world workloads from diverse scientific domains, including molecular dynamics simulations, computational chemistry, and machine learning. Unfortunately, due to the time limit for the thesis writing process, we cannot finish deploying the alert services in production for the pre-exascale HPC system, LUMI. Still, through a benchmark, we have verified the possibility of deploying it in a pre-exascale system in Section 4.1.1.

The production evaluation focused on the practical implications of integrating the alerting and monitoring system into the daily operations of HPC clusters. Key performance indicators, such as the system's ability to facilitate timely alerts, were measured. Additionally, the impact on user experience was evaluated considering the introduction of real-time alerts to administrators.

4.2.1 Setup

As is presented in Figure 4.2, we have two copies of the monitoring infrastructure separately for Puhti and Mahti deployed as microservices inside containers:

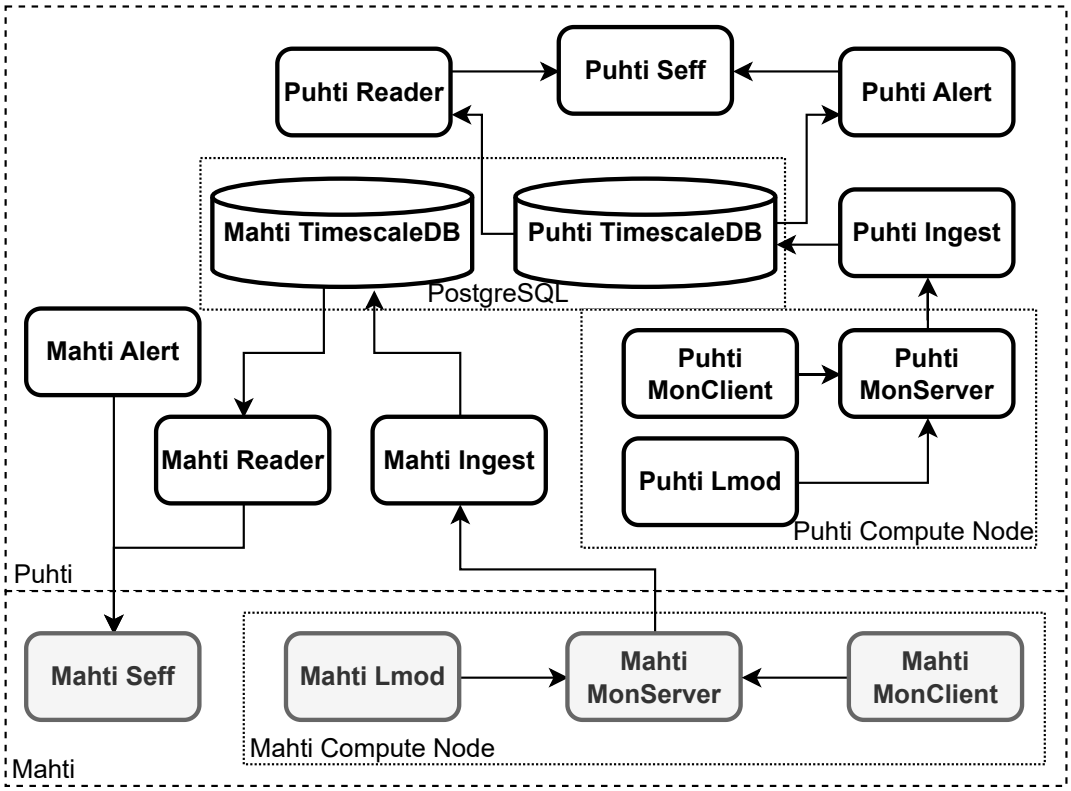


Figure 4.2: GPU monitoring infrastructure production setup for Puhti and Mahti

- **Monitoring Daemon** is written in C++ and deployed on each compute node for both Puhti and Mahti. The **Monitoring Server** is deployed as a systemd service. In contrast, the **Monitoring Client** is deployed as a CLI utility to be called by the Slurm prolog and epilog scripts.
- **Lmod** is the environment module system in Lua. It is deployed on each compute node as well.
- **Timescale Reader**, **Timescale Ingest**, and **Timescale Alert** are all written in Go and deployed as systemd services. Both copies for Mahti and Puhti are deployed on the Puhti MonDB Utility node.
- **TimescaleDB** is deployed as a plugin on the PostgreSQL node. We use different database names to isolate the monitoring data between Puhti and Mahti.
- **Seff** script is the CLI utility written in Perl. It is installed as an RPM package by admins in all the nodes and can be called by the user.
- Other **visualization** services, such as the job history and alert status dashboard, are deployed as Flask Apps for Open OnDemand.

We set the alert sliding window size at 30 minutes. Initially, we collected the monitoring data at 1-minute intervals, then increased that to 20 seconds, and the whole monitoring and alert system was still stable enough. We set the compression policy to compress the data after one day and do the retention every half-year. We also put an index on the hostname, job ID, and GPU ID to accelerate the querying speed.

4.2.2 Case studies

Here are some real-world scenarios in which we use our monitoring infrastructure and Timescale Alert to help users solve issues when they submit jobs in HPC. These case studies illustrate how the system proposed in this thesis enables the support team to proactively identify, diagnose, and resolve issues, ensuring efficient utilization of HPC, reducing user queuing time, and improving user satisfaction.

Configuration Error

From the GPU alert status dashboard, the user support team was alerted by a job that reserved two full nodes, each with 4 A100 GPUs. It has nearly 100% usage almost all the time for 1 GPU but zero usage for the remaining 7, as illustrated in Figure 4.3. By checking the GPU alert history dashboard, the support team found that the alert began long ago. The disk I/O currently used by that job is low, so it shouldn't be the case that the job is still loading data. The situation leads them to suspect a configuration error or code bug. By examining the module load information stored in the job metadata, they discovered that Pytorch was loaded into the job context, a mature machine-learning library that shouldn't have such a situation if everything is configured right. The user support team then contacted the user to investigate possible configuration errors. It was eventually discovered that `CUDA_VISIBLE_DEVICES` was mistakenly set to `0` (utilizing one GPU only), and they didn't use `srun` to distribute the job steps to all nodes properly with `torchrun`. The job was terminated to fix this, and a new job with the correct configuration was subsequently submitted.

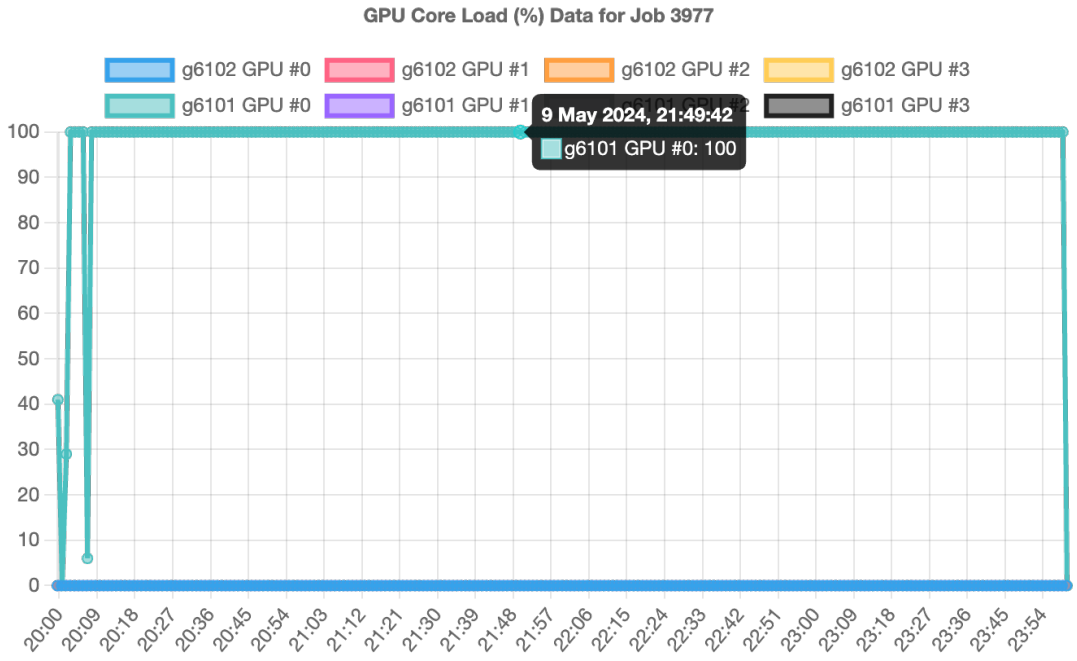


Figure 4.3: Job usage graph for a job with configuration error

Code Bugs

In another scenario, the support team received similar alerts about a job allocated multiple GPUs, described in the *Configuration Error*. Still, it showed low utilization on all GPUs except a few. Those with high usage constantly change and take turns, as Figure 4.4 demonstrates. By analyzing the job's execution patterns and checking with the user, they found that the workload was not adequately distributed across the GPUs because of code bugs and how the user's code was written. The user was advised to improve their code to parallelize tasks and fully utilize the allocated GPUs. This resulted in significantly improved performance and more efficient use of the cluster's GPU resources.

Figure 4.4: Job usage graph for a job with code bugs

GPU Over-Provisioning

For most cases, the support team was alerted by jobs with low overall GPU utilization across all allocated GPUs, as shown in Figure 3.3 and Figure 4.5 for examples. After contacting the users, most of the time, it was found that their applications did not scale well beyond a certain number of GPUs, which means there were more allocated GPUs than needed, resulting in the under-utilization of the resources. Ultimately, the support team recommended shrinking the GPU requests to match the application's needs better, leading to more efficient GPU usage and freeing for other jobs.

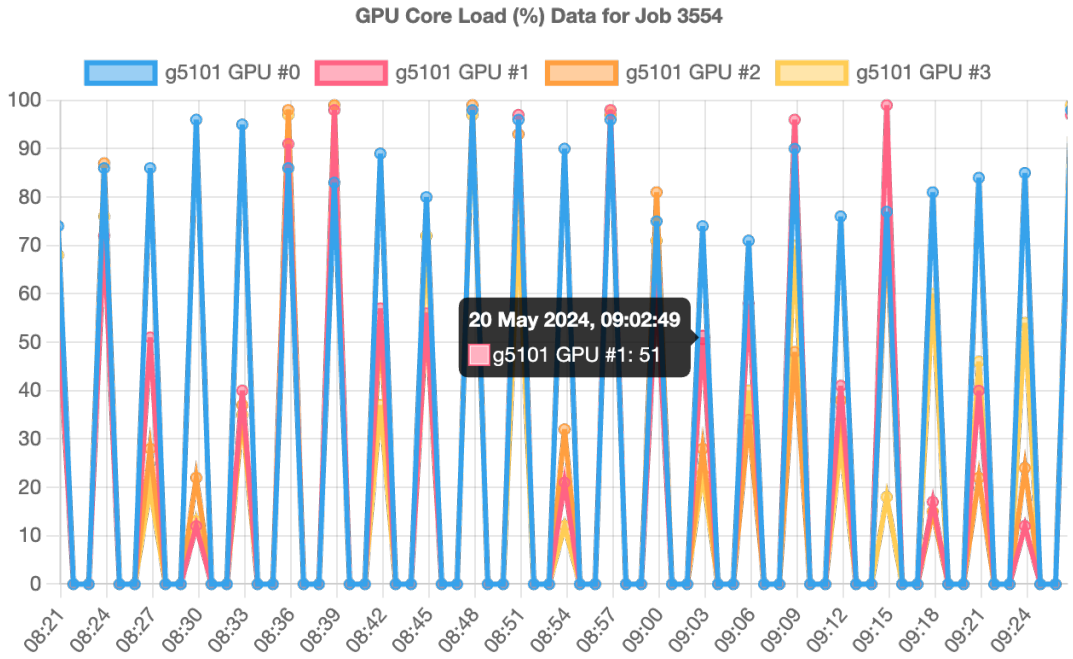


Figure 4.5: Job usage graph for a job with over-provisioning

Interactive use on GPU compute node

For some cases, the support team also gets alerted by jobs that only requested 1 GPU but still have low GPU utilization, on partitions meant for heavy GPU computing jobs, as demonstrated in Figure 4.6 for one example. After checking with the users, most of the time, it was found that they develop their code interactively. The user was then advised to switch their job to the partition meant for lightweight GPU computing.

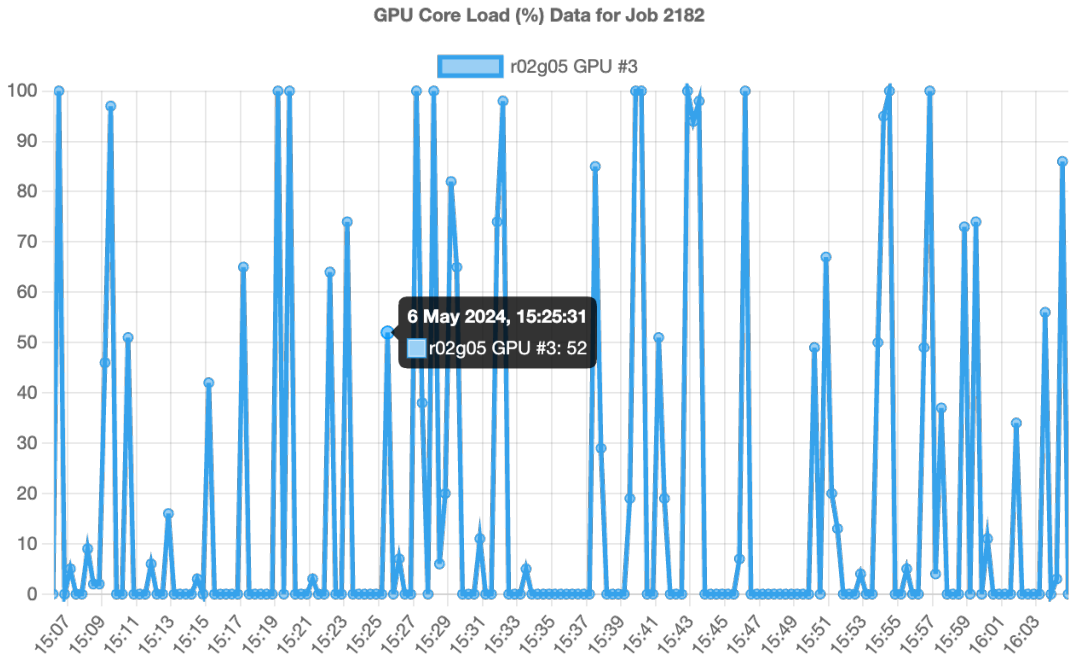


Figure 4.6: Job usage graph for an interactive job on heavy GPU compute partition

GPU Memory Leaks

A user raised a ticket, asking for help from the user support team about why his long-running AI inference job eventually crashed after some time. After checking the monitoring history, as shown in Figure 4.7, the support team found progressively increasing GPU memory usage, ultimately leading to the job being killed due to exceeding memory limits. The support team worked with the user to review the code and identified a memory leak, caused by improper handling of GPU memory allocations in a loop. The memory usage was stabilized by fixing the code, allowing the job to be finished successfully, without exceeding memory limits.

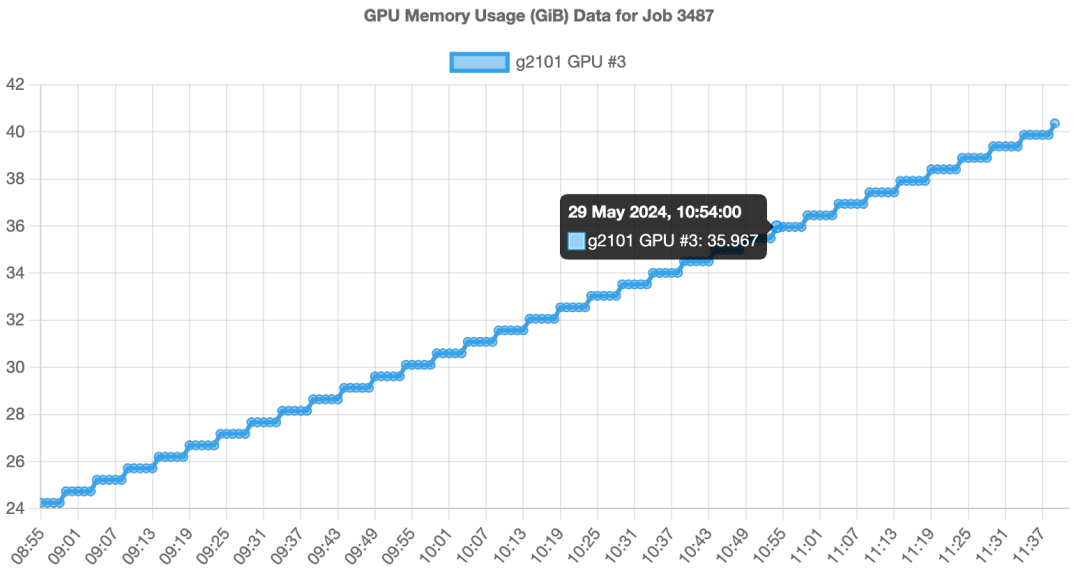


Figure 4.7: Job usage graph for a job with GPU memory leaks

Chapter 5

Discussion

5.1 Findings

Developing and implementing the monitoring system and alert service for GPU resource utilization in HPC clusters has garnered several vital insights, paving the way for further exploration and advancement in this domain.

One of the primary findings of this research is the effectiveness of the implemented algorithms in detecting inefficient GPU resource usage. By analyzing real-time monitoring data, the system can identify instances, where jobs are not effectively utilizing allocated GPU resources, potentially leading to performance bottlenecks or resource wastage. This capability is crucial for optimizing job scheduling and resource allocation in HPC environments, enhancing overall system efficiency and throughput.

Moreover, integrating GPU energy consumption data into the monitoring system offers valuable insights into the sustainability aspects of HPC operations. With increasing focus on energy efficiency and environmental sustainability, understanding and managing power consumption in HPC clusters is paramount. By monitoring GPU energy usage and identifying energy-intensive tasks or jobs, HPC users can implement strategies to reduce power consumption, lower operational costs, and minimize their environmental footprint.

Furthermore, the research's practical implications extend beyond HPC cluster management to various application domains, including AI development and scientific computing. The ability to monitor and analyze GPU resource utilization in real time provides researchers and practitioners with valuable insights into the performance of AI algorithms, computational simulations, and data analysis workflows. By optimizing GPU resource allocation and usage, HPC users can accelerate AI model training, improve scientific simulations, and drive innovation in various fields.

Additionally, we highlight the potential challenges and limitations we have tackled in the monitoring system and alert service for this master's thesis, such as scalability issues with large-scale HPC clusters and compatibility with different hardware configurations (Nvidia/AMD).

5.2 Related work

Several job monitoring platforms for HPC have emerged in recent years, including Ganglia [25], TACC Stats [12], XDMoD [30], LIKWID [35], LDMS [20], PIKA [11], and MAP [28, 29]. However, these platforms lack GPU monitoring support, and few have real-time alerting and history visualization features.

Other notable works in this area use the Prometheus monitoring framework and the Grafana visualization toolkit, including Jobstats [31] and the work down by Jaelyn et al. [23]. These platforms are designed for both CPU and GPU clusters, and they leverage the Prometheus monitoring framework [33] and the Grafana visualization toolkit [7] to provide job-level information on CPU/GPU efficiencies and CPU/GPU memory usage. However, these out-of-the-box solutions operate with high-level APIs, and it is hard to access the streaming raw data for alert customization, such as machine learning algorithms to identify jobs in real-time. Performance can also be an issue, and it is tough to debug if something goes wrong when we implement these solutions to pre-exascale supercomputers such as LUMI.

Chapter 6

Conclusions and future work

In this Chapter, we summarize the thesis and conclude with future work that we could not cover during the thesis period.

6.1 Summary

The thesis comprehensively explores designing, implementing, and evaluating a monitoring system and alert service for GPU resource utilization in High-Performance Computing (HPC) clusters. Through a systematic methodology, the research addresses critical challenges in efficiently analyzing jobs in HPC systems in real-time, focusing on minimizing alert delays and performance impacts on database systems, maintaining reliable data structures for job alert status checks, identifying optimal algorithms for generating alerts, and addressing all research questions in Section 1.3.

The thesis's contributions are multifaceted. First, a monitoring system and alert service with visualization are successfully created and deployed for Nvidia and AMD GPUs within Slurm-managed supercomputer systems. Then, an algorithm is developed to detect and alert jobs with inefficient GPU resource usage by investigating collected monitoring data. Additionally, the thesis provides insights into GPU resource utilization dynamics with case studies in production. By addressing critical challenges and proposing innovative solutions, the research enhances the effectiveness and sustainability of GPU-accelerated computing environments, paving the way for future advancements in HPC infrastructure management and AI research.

6.2 Future work

There are several avenues for future research and improvements due to the limited time during this master's thesis project:

6.2.1 Additional job schedulers

Extend support for real-time GPU monitoring to additional job schedulers commonly used in HPC environments, such as LSF [22], TORQUE [44], or UGE [1].

6.2.2 More monitoring metrics

Collect monitoring data from other hardware (e.g., CPU, disk I/O) and collectively contribute to the alert for the whole system. The monitoring infrastructure already has the capabilities for monitoring additional hardware, but due to performance considerations, they have not been tested in production.

In addition, we can also explore the possibility of capturing and analyzing fine-grained GPU usage metrics, including memory bandwidth, cache utilization, and instruction throughput. This granular level of monitoring can provide deeper insights into application performance and identify optimization opportunities at the code level.

6.2.3 Flexible alerting

Investigate adaptive alerting strategies that dynamically adjust the thresholds in our alert algorithms based on workload characteristics, ensuring effective alerting across varying HPC workloads.

It's also worth exploring the integration of possible other advanced machine learning algorithms to predict GPU resource usage patterns, enabling proactive alerting based on historical data analysis.

We can also try to implement mechanisms to directly gather feedback from administrators or support teams regarding the effectiveness of the jobs, and improve our alert strategies automatically, so that we can continuously refine them based on practical usage experiences.

6.2.4 Resource optimization

Investigate optimization techniques for dynamically allocating GPU resources based on real-time workload demands and system utilization. This could involve developing algorithms for intelligent resource provisioning and load balancing to maximize overall cluster efficiency and performance.

We can also investigate the predictive maintenance techniques for GPUs based on real-time monitoring data. Predictive maintenance models can anticipate failures or performance degradation by analyzing hardware health metrics and performance degradation, enabling proactive maintenance actions to minimize downtime and maximize system reliability.

It's also worth investigating integrating the real-time GPU monitoring system with energy management systems to optimize power consumption. By correlating GPU usage metrics with power consumption data, administrators can implement energy-efficient computing strategies, such as dynamic voltage and frequency scaling, to minimize power consumption without sacrificing performance.

Bibliography

[1] Misha Ahmadian, Eric Rees, Yu Zhuang, and Yong Chen. Reducing Faulty Jobs by Job Submission Verifier in Grid Engine. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, New York, NY, USA, 2019. ACM.

[2] AMD. AMD Instinct MI250X Accelerators. https://www.amd.com/en/products/accelerators/instinct/mi200/mi250x.html, 11 2021. Accessed: Jun 2024.

[3] AMD. AMD EPYC 7003 Processors Data Sheet. https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf, 10 2023. Accessed: Jun 2024.

[4] AMD. ROCm System Management Interface (ROCm SMI) Library - Documentation. https://rocm.docs.amd.com/projects/rocm_smi_lib/en/latest/, 1 2024. Accessed: Jun 2024.

[5] Ryan Booz. Guide to Postgres Data Management. https://www.timescale.com/blog/guide-to-postgres-data-management/, 10 2023. Accessed: Jun 2024.

[6] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink®: consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, 8 2017.

[7] Mainak Chakraborty and Ajit Pratap Kundan. *Grafana*, pages 187–240. Apress, Berkeley, CA, 2021.

[8] European Commission. Commission launches AI innovation package to support Artificial Intelligence startups and SMEs. https://ec.europa.eu/commission/presscorner/detail/en/ip_24_383, 1 2024. Accessed: Jun 2024.

[9] Council of the European Union. Council Regulation (EU) 2021/1173 of 13 July 2021 on establishing the European High Performance Computing Joint Undertaking and repealing Regulation (EU) 2018/1488, 2021. Official Journal of the European Union, L 256, 3-34.

[10] Christophe Cérin, Nicolas Greneche, and Tarek Menouer. Towards Pervasive Containerization of HPC Job Schedulers. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 281–288, 2020.

[11] Robert Dietrich, Frank Winkler, Andreas Knüpfer, and Wolfgang Nagel. PIKA: Center-Wide and Job-Aware Cluster Monitoring. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 424–432, 2020.

[12] Todd Evans, William L. Barth, James C. Browne, Robert L. DeLeon, Thomas R. Furlani, Steven M. Gallo, Matthew D. Jones, and Abani K. Patra. Comprehensive Resource Use Monitoring for HPC Systems with TACC Stats. In *2014 First International Workshop on HPC User Support Tools*, pages 13–21, 2014.

[13] CSC IT Center for Science. GPU nodes - LUMI-G - Documentation. https://docs.lumi-supercomputer.eu/hardware/lumig/, 11 2023. Accessed: Jun 2024.

[14] CSC IT Center for Science. Mahti - Docs CSC. https://docs.csc.fi/computing/systems-mahti/, 3 2023. Accessed: Jun 2024.

[15] CSC IT Center for Science. Puhti - Docs CSC. https://docs.csc.fi/computing/systems-puhti/, 3 2023. Accessed: Jun 2024.

[16] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63:3–42, 2006.

[17] Ibrahim Abaker Targio Hashem, Nor Badrul Anuar, Abdullah Gani, Ibrar Yaqoob, Feng Xia, and Samee Ullah Khan. Mapreduce: Review and open challenges. *Scientometrics*, 109:389–422, 2016.

[18] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1, 1995.

[19] InfluxData. Line protocol | InfluxDB Cloud (TSM) Documentation. https://docs.influxdata.com/influxdb/cloud/reference/syntax/line-protocol/, 3 2024. Accessed: Jun 2024.

[20] Ramin Izadpanah, Nichamon Naksinehaboon, Jim Brandt, Ann Gentile, and Damian Dechev. Integrating Low-latency Analysis into HPC System Monitoring. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP '18, New York, NY, USA, 2018. ACM.

[21] Volodymyr V. Kindratenko, Jeremy J. Enos, Guochun Shi, Michael T. Showerman, Galen W. Arnold, John E. Stone, James C. Phillips, and Wen-mei Hwu. GPU clusters for high-performance computing. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, 2009.

[22] Shin F. Leong and Craig S. Pohl. Traditional High-Performance Computing with Container Technology (THPC): HPC using Container Technology, EasyBuild, Spack, and IBM LSF scheduler. In *Practice and Experience in Advanced Research Computing*, PEARC '23, page 306–311, New York, NY, USA, 2023. ACM.

[23] Jaelyn Litzinger, Roy Hallquist, and James Tessmer. HPC Monitoring & Visualization: Understanding usage of HPC systems. In *Practice and Experience in Advanced Research Computing*, PEARC '23, page 453–456, New York, NY, USA, 2023. ACM.

[24] Joshua Lockerman and Ajay Kulkarni. Time-Series Compression Algorithms, Explained. https://www.timescale.com/blog/time-series-compression-algorithms-explained/, 10 2023. Accessed: Jun 2024.

[25] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[26] Nvidia. NVIDIA Management Library (NVML) | NVIDIA Developer. https://developer.nvidia.com/nvidia-management-library-nvml, 1 2024. Accessed: Jun 2024.

[27] OpenAI, Josh Achiam, Steven Adler, et al. GPT-4 Technical Report, 2024.

[28] Ashish Pal and Preeti Malakar. MAP: A Visual Analytics System for Job Monitoring and Analysis. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 442–448, 2020.

[29] Ashish Pal and Preeti Malakar. An Integrated Job Monitor, Analyzer and Predictor. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 609–617, 2021.

[30] Jeffrey T. Palmer, Steven M. Gallo, Thomas R. Furlani, Matthew D. Jones, Robert L. DeLeon, Joseph P. White, Nikolay Simakov, Abani K. Patra, Jeanette Sperhac, Thomas Yearke, Ryan Rathsam, Martins Innus, Cynthia D. Cornelius, James C. Browne, William L. Barth, and Richard T. Evans. Open XDMoD: A Tool for the Comprehensive Management of High-Performance Computing Resources. *Computing in Science & Engineering*, 17(4):52–62, 2015.

[31] Josko Plazonic, Jonathan Halverson, and Troy Comi. Jobstats: A Slurm-Compatible Job Monitoring Platform for CPU and GPU Clusters. In *Practice and Experience in Advanced Research Computing*, PEARC '23, page 102–108, New York, NY, USA, 2023. ACM.

[32] PostgreSQL. PostgreSQL 15.6 Documentation. https://www.postgresql.org/docs/15/index.html, 4 2024. Accessed: Jun 2024.

[33] Björn Rabenstein and Julius Volz. Prometheus: A Next-Generation Monitoring System. In *Open Access Media*, Dublin, May 2015. USENIX Association.

[34] Theofanis P. Raptis and Andrea Passarella. A Survey on Networked Data Streaming With Apache Kafka. *IEEE Access*, 11:85333–85350, 2023.

[35] Thomas Röhl, Jan Eitzinger, Georg Hager, and Gerhard Wellein. LIKWID Monitoring Stack: A Flexible Framework Enabling Job Specific Performance monitoring for the masses. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 781–784, 2017.

[36] SchedMD. Slurm History – SchedMD. https://www.schedmd.com/about-schedmd/slurm-history/, 5 2024. Accessed: Jun 2024.

[37] scikit-learn Docs Authors. Selecting the number of clusters with silhouette analysis on KMeans clustering. https://scikit-learn.org/1.5/auto_examples/cluster/plot_kmeans_silhouette_analysis.html, 10 2023. Accessed: Jun 2024.

[38] Baji Shaik and Dinesh Kumar Chemuduru. *Listen and Notify*, pages 283–292. Apress, Berkeley, CA, 2023.

[39] Baji Shaik and Dinesh Kumar Chemuduru. *Triggers*, pages 241–262. Apress, Berkeley, CA, 2023.

[40] James G. Shanahan and Laing Dai. Large Scale Distributed Data Science using Apache Spark. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 2323–2324, New York, NY, USA, 2015. ACM.

[41] Rahul Sharma and Mohammad Atyab. *Introduction to Apache Pulsar*, pages 1–22. Apress, Berkeley, CA, 2022.

[42] Chunhui Shen, Qianyu Ouyang, Feibo Li, et al. Lindorm TSDB: A Cloud-Native Time-Series Database for Large-Scale Monitoring Systems. *Proc. VLDB Endow.*, 16(12):3715–3727, 8 2023.

[43] Yan-Yan Song and LU Ying. Decision tree methods: applications for classification and prediction. *Shanghai archives of psychiatry*, 27(2):130, 2015.

[44] Garrick Staples. TORQUE resource manager. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, page 8–es, New York, NY, USA, 2006. ACM.

[45] Erich Strohmaier, Jack Dongarra, Horst D. Simon, and Hans Meuer. TOP500. https://www.top500.org/lists/top500/2024/06/, 6 2024. Accessed: Jun 2024.

[46] Gemma Team, Thomas Mesnard, Cassidy Hardin, et al. Gemma: Open Models Based on Gemini Research and Technology, 2024.

[47] Timescale. About continuous aggregates | Timescale Documentation. https://docs.timescale.com/use-timescale/latest/continuous-aggregates/about-continuous-aggregates/, 3 2024. Accessed: Jun 2024.

[48] Timescale. About hypertables | Timescale Documentation. https://docs.timescale.com/use-timescale/latest/hypertables/about-hypertables/, 3 2024. Accessed: Jun 2024.

[49] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, et al. LLaMA: Open and Efficient Foundation Language Models, 2023.

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[51] Fei Wang, Hector-Hugo Franco-Penya, John D. Kelleher, John Pugh, and Robert Ross. An Analysis of the Application of Simplified Silhouette to the Evaluation of k-means Clustering Validity. In Petra Perner, editor, *Machine Learning and Data Mining in Pattern Recognition*, pages 291–305, Cham, 2017. Springer International Publishing.

[52] Tim Wickberg. Slurm and/or/vs Kubernetes. https://slurm.schedmd.com/SC23/Slurm-and-or-vs-Kubernetes.pdf, 11 2023. Accessed: Jun 2024.

[53] Junjie Wu. *Cluster Analysis and K-means Clustering: An Introduction*, pages 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[54] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[55] Li Yuanyuan, Xiao Peng, and Deng Wu. The method to test Linux software performance. In *2010 International Conference on Computer and Communication Technologies in Agriculture Engineering*, volume 1, pages 420–423, 2010.

[56] Zhenyun Zhuang, Cuong Tran, Jerry Weng, Haricharan Ramachandra, and Badri Sridharan. Taming memory related performance pitfalls in linux Cgroups. In *2017 International Conference on Computing, Networking and Communications (ICNC)*, pages 531–535, 2017.

Appendix A

More about Slurm

In addition to the three main components (Slurmctld, Slurmdbd, Slurmd) mentioned in Section 2.1, Figure A.01 shows the general Slurm architecture.

Slurm also adopts the plugin-based architecture. These plugins, from network topologies to authentication mechanisms, offer a customizable framework adaptable to different computing environments. Moreover, Slurm's robust plugin design allows for seamless integration of custom functionalities, empowering users to tailor the system to their specific needs.

This Chapter introduces Slurm's login-node commands and Slurm's key features. We also compare Slurm with Kubernetes.

A.1 Login-node commands

Here are the available Slurm commands for users to interact in the login node:

- **sbatch**: For submitting batch scripts, which can be written in bash, Perl, or Python, enabling users to automate the execution of tasks in a batch mode.
- **scancel**: Enabling the cancellation of pending or running jobs or job steps, providing users with control over job management and resource allocation.
- **squeue**: For querying pending and running jobs, providing users with visibility into the status of their submitted tasks and overall system workload.
- **salloc**: Users can interactively access computing resources for their tasks to request interactive job allocations.
- **sinfo**: Retrieving comprehensive information about partitions, reservations, and the state of nodes within the system, helping users understand the availability and status of computing resources.
- **scontrol**: Enabling users to manage jobs, query system configurations, and retrieve resource utilization and allocation information.
- **sprio**: Enabling users to query job priorities, assisting in resource allocation decisions and prioritization of tasks based on predefined criteria.
- **seff**: Providing a concise overview of resource utilization for active and completed batch jobs, detailing the requested and actual usage of resources.

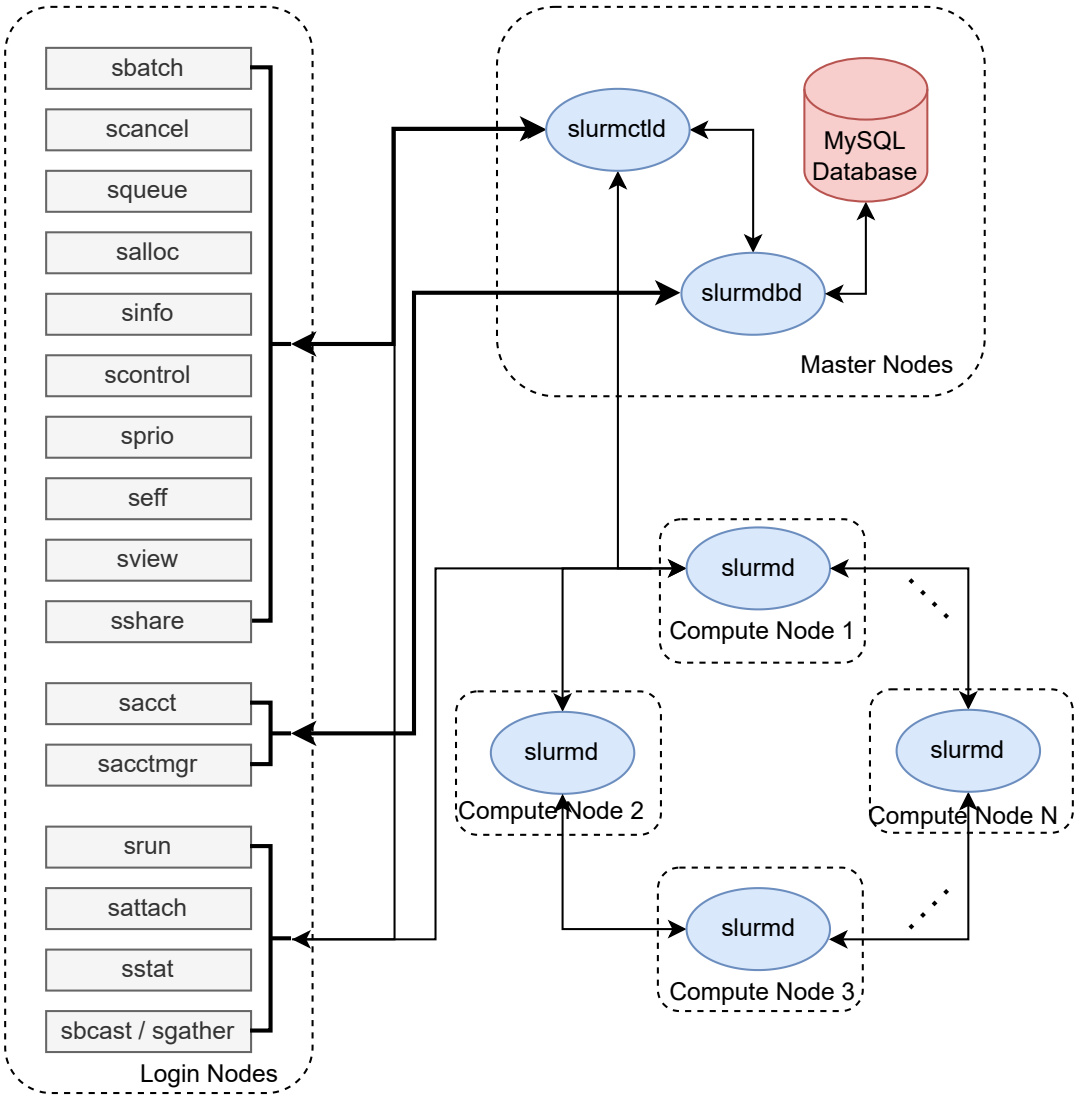


Figure A.01: Slurm architecture overview

- **sview**: GUI that offers state information for jobs, partitions, and nodes, enhancing user experience in monitoring and managing computing resources.
- **sshare**: Providing users with fair-share information, offering insights into resource allocation fairness among different users based on usage history and system policies.
- **sacct**: Retrieving accounting information about jobs and job steps stored in Slurm's database, assisting in resource usage analysis, billing, and reporting.
- **sacctmgr**: Enabling users to query accounting-related information and other accounting data stored in Slurm's database, facilitating user accounting management and administration tasks.
- **srun**: Initiating job steps, primarily within a job or starting interactive jobs, allowing for executing multiple steps sequentially or in parallel on allocated nodes within the job's resource allocation.

- **sattach**: Attaching standard input, output, and error streams, along with signal capabilities, to a currently running job or job step, facilitating real-time monitoring and interaction.
- **sstat**: Querying status information about running jobs, providing real-time updates on job progress, resource utilization, and other relevant metrics.
- **sbcast**: Facilitating the transfer of files to all nodes allocated for a specific job, so that we can ensure necessary data or resources to be available across the computing environment.
- **sgather**: Allowing the retrieval of files from all allocated nodes to the currently active job, serving as a mechanism for aggregating results or data produced during job execution.

A.2 Key features

Key features for Slurm include:

- **High-availability** for the primary daemons, namely Slurmctld and Slurmdbd, ensuring uninterrupted operation.
- Compute nodes are grouped into **partitions** by Slurm, allowing for configuring various limits and policies for each partition, such as permitted users, maximum nodes, and maximum wall-time limit per job.
- **Quality-of-Services (QoS)** enforce additional limits based on the status of the user's group.
- Utilization of the **backfilling scheduling algorithm** to optimize job scheduling and enhance resource utilization.
- Job scheduling based on **priorities** allows efficient allocation according to user-defined criteria.
- **Accounting** mechanism utilizing Slurmdbd and the MySQL/MariaDB database to track resource usage and job statistics for Trackable RESources (TRES). Default TRES include nodes, CPUs, memory, and billing.
- **No preemption** policies support, meaning that administrators can configure Slurm so that running jobs are not subject to interruption.
- **Prologue and Epilogue** scripts to execute tasks before and after job execution.
- **Generic RESources (GRES)** such as GPUs and NVMe allow flexibility but lack built-in accounting support.

A.3 With Kubernetes

With the rise of cloud computing, container orchestration systems such as Kubernetes become increasingly popular. However, these tools are unsuitable for deployment in the HPC world, as HPC job schedulers like Slurm revolve around job completion. At the same time, Kubernetes is tailored for hosting and sustaining services over time [10]. To be more precise, the primary distinction between an HPC workload and the typical application suited for Kubernetes lies in their operational characteristics.

HPC workloads focus on efficiently executing a complex task within the shortest time frame, even if the duration is considerable. Conversely, Kubernetes excels in managing continuously running applications, particularly those designed as services.

However, there is a growing interest in integrating batch job systems with Kubernetes to provide the best of both worlds. Such integration aims to leverage Kubernetes's capabilities in managing scalable, long-running services while benefiting from the efficient scheduling and resource management of HPC systems like Slurm. Several models have been proposed for converging these environments, including *Adjacent* and *Under* models. In the *Adjacent* model, both control planes are overlapped, with Slurm managing both traditional HPC and Kubernetes workloads, utilizing Kubernetes' capabilities like sidecars and operators. The *Under* model involves running Slurm clusters within a Kubernetes environment, allowing for traditional user experiences and higher throughput for MPI workloads. At the same time, Kubernetes manages scaling and dynamic resource allocation. One such example project is SUNK (Slurm on Kubernetes) [52], which is working towards seamless integration to support large-scale AI training, as well as inference and other complex workflows on hybrid environments that combine cloud-native and HPC.

Appendix B

More about HPC systems at CSC

This Chapter gives more information about the HPC systems at CSC, on top of Section 2.2, including the overview of CPU for Mahti, more details about the AMD MI250X GPU, and how the hardware is connected for each HPC system.

B.1 Mahti CPU

The NUMA configuration of Mahti [14] involves a hierarchical structure within each node. Each node contains two sockets, each accommodating a single CPU alongside memory DIMMs. Although the memory within the node is shared, the performance of memory access varies based on the proximity of the core to the memory. Mahti operates each CPU in NPS4 (NUMA per socket 4) mode to optimize memory performance, dividing each CPU into four NUMA domains. Each NUMA domain includes 16 cores and two memory controllers, providing 32 GiB of memory. Thread allocation within each core follows a specific pattern: core 0 runs threads 0 and 128, core 1 runs threads 1 and 129, and so forth. Figure B.11 shows how the threads are distributed over each core and NUMA node.

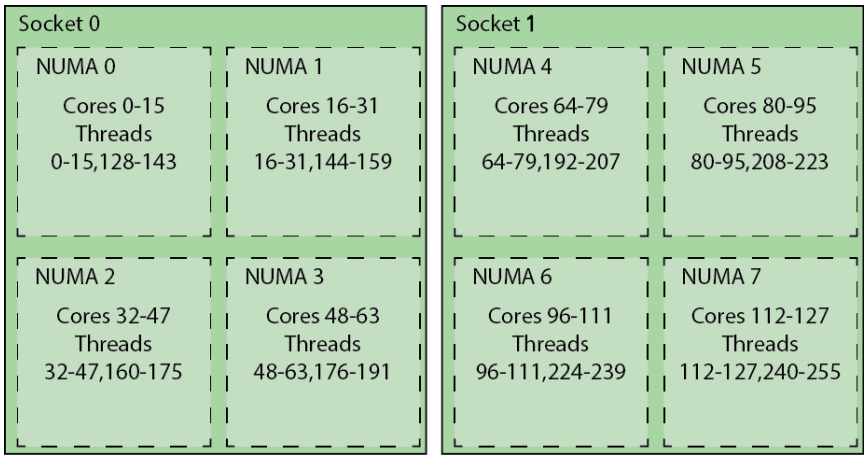


Figure B.11: Mahti NUMA structure overview [14]

Each core possesses 32 KiB of L1 data cache, 32 KiB of L1 instruction cache, and a private 512 KiB L2 cache. Additionally, each core has two FMA (fused multiply-add) units capable of processing operations on full 256-bit vectors. Consequently, each unit can execute operations on 8 single-precision floats or 4 double-precision floats per clock cycle, resulting in 16 double-precision floating point operations per clock cycle.

As shown in Figure B.12, cores in the CPU are grouped into core complexes (CCXs) and further combined into compute dies (CCDs). At the CCX level, four cores share a 16 MiB L3 cache within the CCX, and two CCX parts combine to form a compute die (CCD).

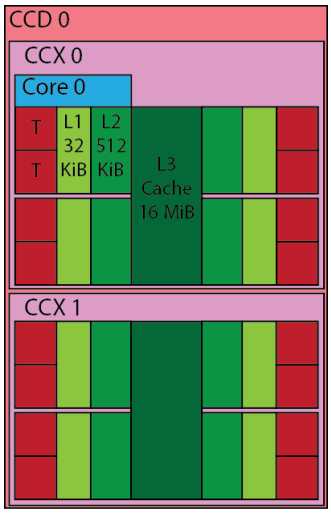


Figure B.12: Mahti CCD structure overview [14]

Each processor comprises eight compute dies and an additional I/O die, including memory and PCI-e controllers. Furthermore, each node consists of two processors and one 200 Gbps HDR network adapter, as shown in Figure B.13.

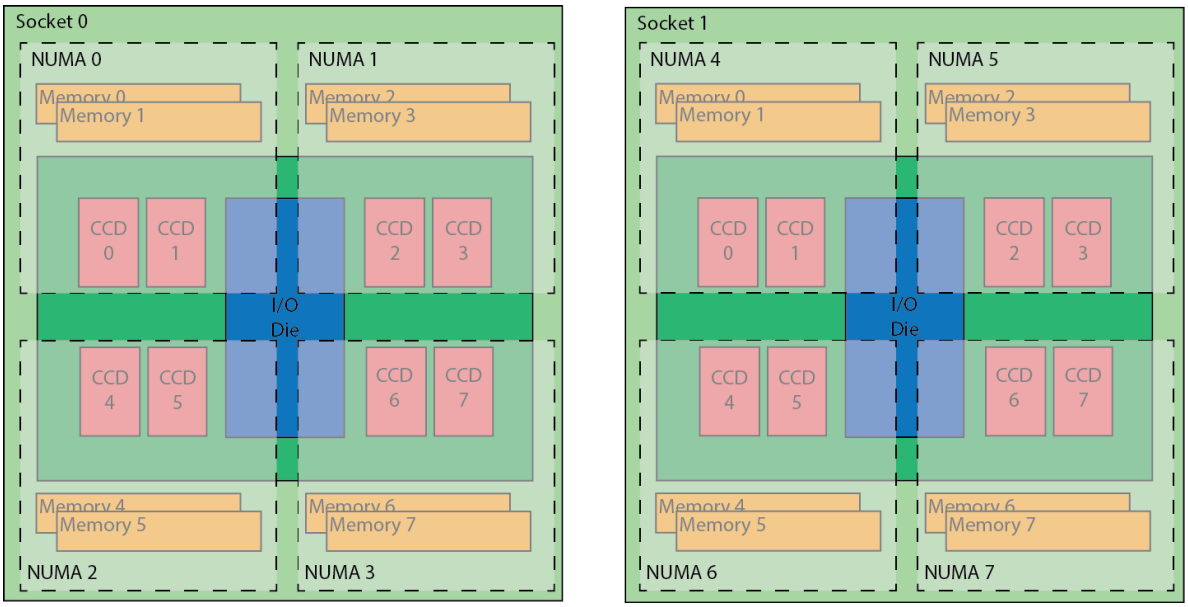


Figure B.13: Mahti node structure overview [14]

B.2 AMD MI250X GPU

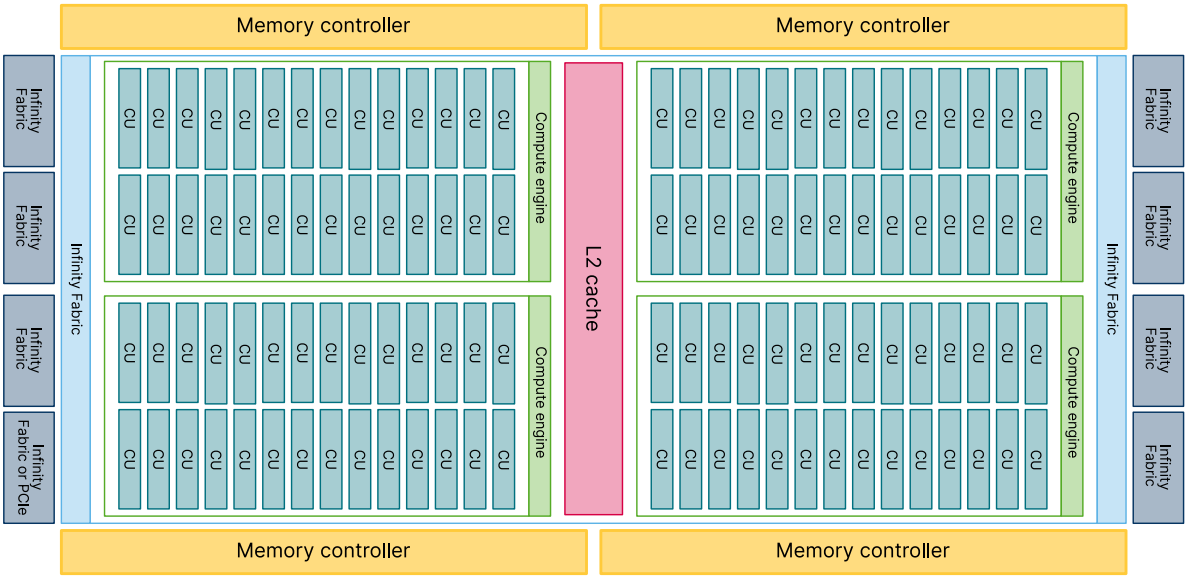


Figure B.24: AMD MI250X GCD structure [13]

The GCD structure of AMD MI250X GPU is demonstrated in Figure B.24. When a kernel is dispatched for execution on the GPU, it is organized as a grid of thread blocks (workgroups), with the grid and thread blocks being one, two, or three-dimensional. The grid can have a maximum number of blocks specified along each dimension of (2147483647, 2147483647, 2147483647), while the maximum number of threads

(work-items) for each dimension of a block is (1024, 1024, 1024), with a thread block size limit of 1024, which means size.x * size.y * size.z must be less or equal to 1024.

The thread blocks are assigned to one of the 110 compute units and are scheduled in groups of 64 threads, known as wavefronts. This is analogous to a warp on NVIDIA hardware, except that a warp consists of 32 threads, while for AMD hardware, a wavefront comprises 64 threads.

The execution process of wavefronts by a compute unit can be outlined as follows:

1. Each wavefront has 64 work items (threads) allocated to one of the 16-wide SIMD units.
2. Most instructions are executed within a single cycle, although one instruction requires four cycles per wavefront.
3. With 4 SIMD units available per compute unit, up to 4 wavefronts can be processed simultaneously, ensuring a consistent throughput of one instruction per wavefront per compute unit.

Figure B.25 shows that each compute unit has 512 64-wide 4-byte vector general-purpose registers. Additionally, the unit provides access to low-latency storage through a 64 kB local data share (LDS, shared memory), accessible to all threads within a block. The programmer manages the LDS allocation. Furthermore, each compute unit has access to 16 kB of L1 cache.

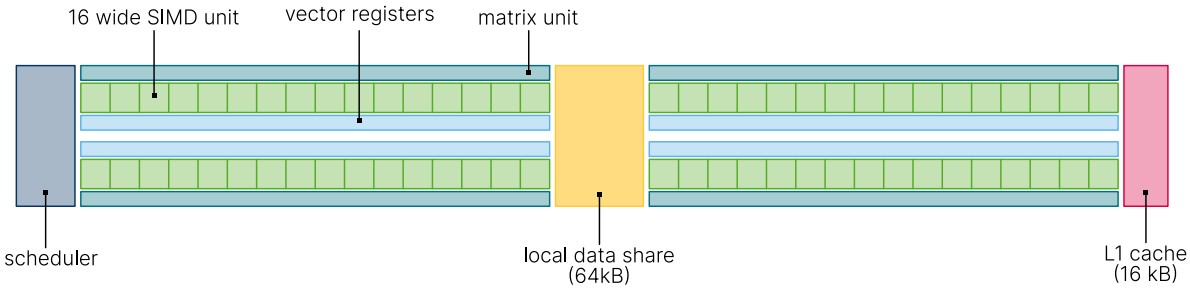


Figure B.25: AMD MI250X compute unit structure [13]

The vector ALUs are complemented by matrix cores optimized to execute matrix-fused multiply-add instructions. These cores offer significant acceleration for generalized matrix multiplication computations, which is crucial for linear algebra in High-Performance Computing applications and AI workloads. Each compute unit (CU) has four matrix cores capable of achieving a throughput of 256 double-precision floating-point format (FP64) Flops/cycle/CU.

B.3 Connections

This Section introduces how the hardware or node is inter-connected, for the three HPC systems at the CSC: Puhti, Mahti, and LUMI.

B.3.1 Puhti

The Puhti[15] interconnect architecture is based on a dual-rail Mellanox HDR100 InfiniBand setup. It offers a non-blocking fat-tree topology with a blocking factor of approximately 2:1 and delivers an impressive aggregate bandwidth of 200 Gbps, ensuring efficient connectivity across the network.

B.3.2 Mahti

The network interconnect in Mahti is based on Mellanox HDR InfiniBand, with each node connected via a single 200 Gbps HDR link. The network topology adopts a dragonfly+ configuration, in which multiple groups of nodes are internally connected using a fat tree topology. These fat trees are interconnected using all-to-all links to ensure fully non-blocking connectivity between groups, as shown in Figure B.36.

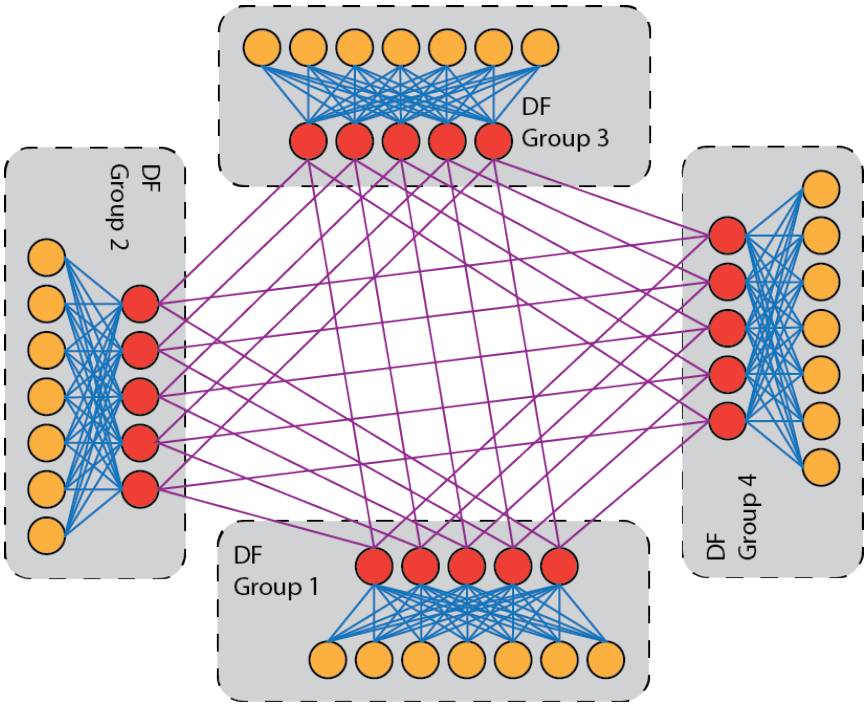


Figure B.36: Mahti dragonfly+ configuration overview [14]

In Mahti, each dragonfly group comprises 234 nodes, with an internal fat tree featuring a blocking factor of 1.7:1 and 20 or 18 nodes connected per leaf switch. Each leaf switch connects to the spine switch in the group via 12 200 Gbps links. Figure B.37 shows the topology of such a group. There are six groups, with five 200 Gbps links connecting each spine switch to a spine switch in every other group, facilitating comprehensive inter-group communication.

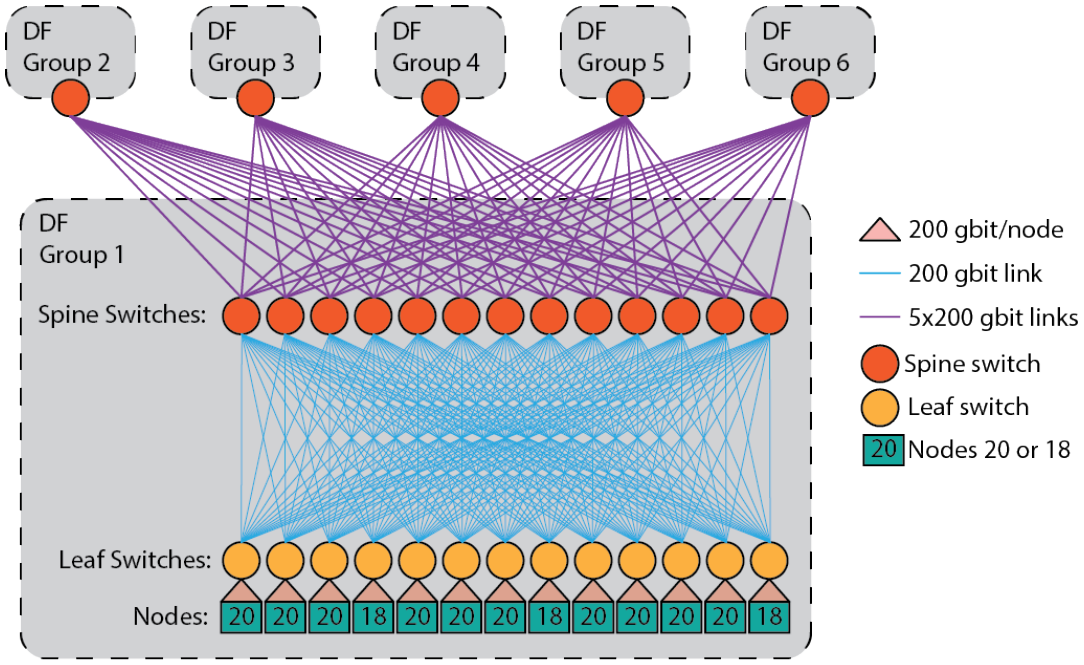


Figure B.37: Mahti dragonfly topology [14]

B.3.3 LUMI

Figure B.38 shows the node LUMI topology. Each MI250X module includes 5 GPU-GPU links, 2 CPU-GPU links, and 1 PCIe link to the slingshot-11 interconnect. The MI250X modules are connected via an in-package Infinity Fabric interface, capable of delivering a theoretical peak bidirectional bandwidth of up to 400 GB/s. Furthermore, GCDs across different MI250X modules are linked through single or double Infinity Fabric links, offering peak bidirectional bandwidths of 100 GB/s and 200 GB/s, respectively. Each MI250X module directly connects to the slingshot-11 network, affording peak bandwidths of up to 25+25 GB/s.

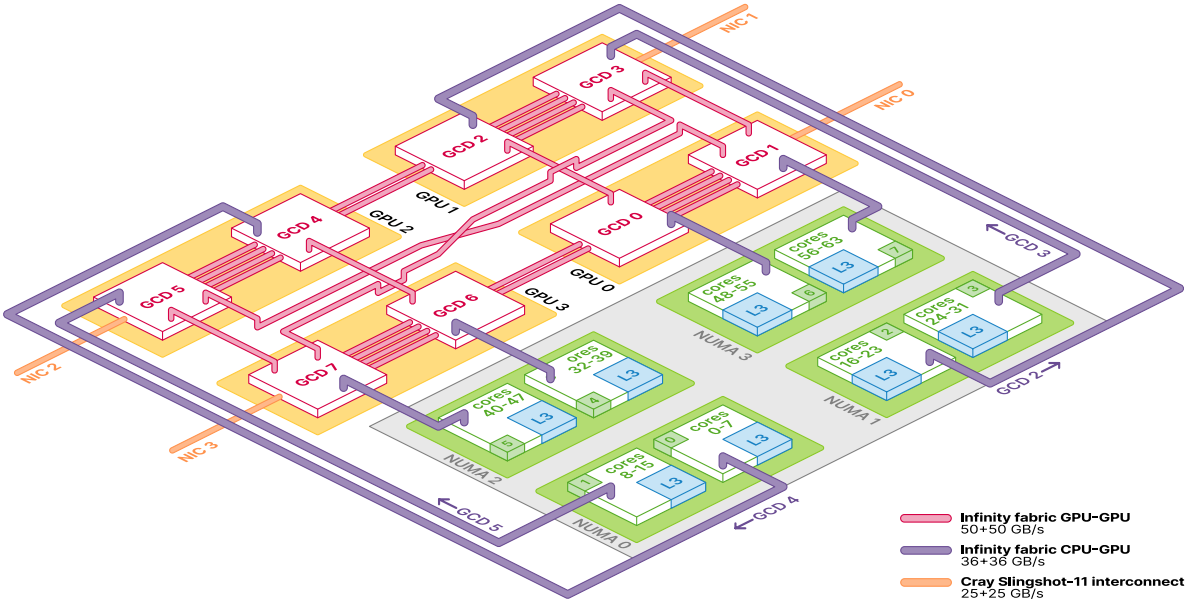


Figure B.38: LUMI GPU node topology overview [13]

Figure B.39 shows the CPU-GPU links from a CPU-centric or GPU-centric point of view of the LUMI GPU node. Proper binding of the NUMA node to the GPU can be essential to ensure optimal application performance.

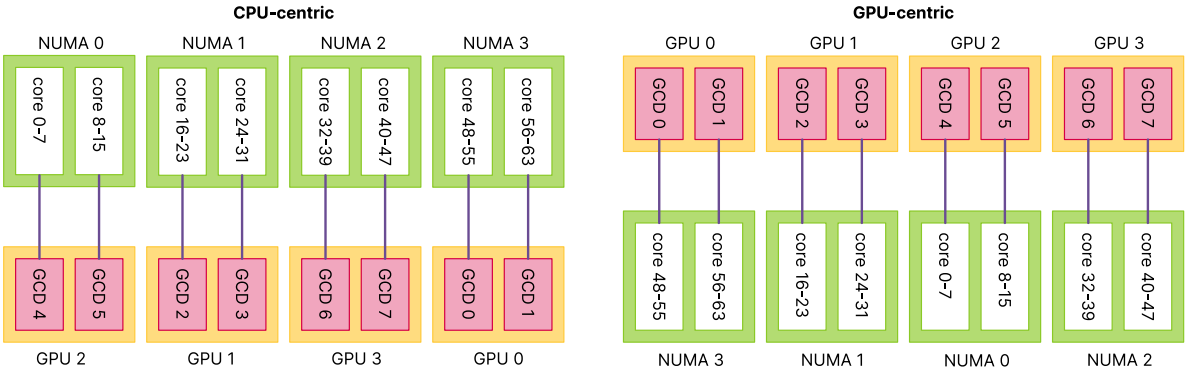


Figure B.39: CPU-GPU links on LUMI GPU node [13]

Appendix C

Event stream management

Event Stream Management involves ingesting, analyzing, and storing streams of events, which are discrete data points denoting state changes. This allows for real-time analytics and decision-making.

- **MapReduce** [17] is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. A MapReduce program comprises a map procedure, which performs filtering and sorting, and a reduce method, which performs a summary operation. The MapReduce algorithm contains two important tasks, namely Map and Reduce. The map script takes some input data and maps it to `<key, value>` pairs according to the specifications. The reduce script takes a collection of `<key, value>` pairs and *reduces* them according to the specifications. MapReduce is primarily used for batch processing of large datasets and is not designed for real-time processing or low-latency queries.
- **Apache Spark** [40] is an open-source cluster-computing framework. It provides elegant development APIs for Scala, Java, Python, and R that allow developers to execute a variety of data-intensive workloads across diverse data sources, including HDFS, Cassandra, HBase, S3, etc. Spark provides a faster and more general data processing platform. Spark lets users run programs up to 100x faster in memory or 10x faster on disk than Hadoop. Spark's versatility makes it suitable for various applications and industries.
- **Apache Flink** [6] is a Big Data processing framework that allows programmers to process vast data efficiently and in a scalable. Flink primarily focuses on real-time stream processing, efficiently processing large volumes of data with low latency. Flink's processing engine is built on top of its streaming runtime and can handle batch processing. Flink provides robust Java, Scala, and Python APIs for developing data processing applications.
- **Apache Kafka** [34] **/ Apache Pulsar** [41] is a real-time event-streaming platform that collects, stores, and processes messages. It provides excellent performance, too, at scale. On top of that, it provides capabilities such as stream processing, distributed logging, and pub-sub messaging. An event (or message) in Kafka consists of Key and Value.

All four technologies can handle big data, but each has strengths and use cases:

- MapReduce is primarily used for batch processing of large datasets. It is not designed for real-time processing or low-latency queries.
- Apache Spark, on the other hand, while initially designed for batch processing, has evolved to handle real-time data processing through micro-batching.
- However, Apache Flink was designed as a stream-first framework, excelling in real-time stream processing. It efficiently processes large volumes of data with low latency.
- Apache Kafka, similar to Flink, is designed for real-time data streaming. However, Kafka is more focused on the messaging system, providing a robust platform for storing, reading, and analyzing streaming data.
- Apache Pulsar combines the strengths of both a message queue system and a streaming platform, making it a versatile choice for many use cases.

Appendix D

Code for figures

In this Chapter, we present the code for some of the figures drawn using Python script. These figures are drawn using the Matplotlib plotting library, together with pandas and NumPy.

D.1 Figure 3.5 & Figure 3.6

Section 3.3 shows the silhouette analysis on windowed GPU load data. Here is the code for the direct one [37], as in Figure 3.5. Note that for `load_gpu.csv`, we have the format as follows, where the data for each entry is the fixed-size (30) sliding window:

$$hostname, jobid, GPUid, data0, data1, ..., data28, data29$$

```
import pandas as pd

data = pd.read_csv('load_gpu.csv', usecols=[i for i in range(3, 33)
    ])
X = data.to_numpy()

import matplotlib.cm as cm
import matplotlib.pyplot as plt
import numpy as np

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score

range_n_clusters = [2, 3, 4, 5, 6, 7]

plt.rcParams.update({'font.size': 56})

for n_clusters in range_n_clusters:
    print("Start creating plot for n_clusters =", n_clusters)
    # Create a subplot with 1 row and 1 column
    fig, ax1 = plt.subplots(1, 1)
    fig.set_size_inches(18, 18)

    # The 1st subplot is the silhouette plot
```

```
    # The silhouette coefficient can range from -1, 1
    ax1.set_xlim([-0.6, 0.4])
    # The (n_clusters+1)*10 is for inserting blank space between
        silhouette
    # plots of individual clusters, to demarcate them clearly.
    ax1.set_ylim([0, len(X) + (n_clusters + 1) * 10])

    # Initialize the clusterer with n_clusters value and a random
        generator
    # seed of 10 for reproducibility.
    clusterer = KMeans(n_clusters=n_clusters)
    cluster_labels = clusterer.fit_predict(X)
    print(cluster_labels, cluster_labels.shape)

    # The silhouette_score gives the average value for all the
        samples.
    # This gives a perspective into the density and separation of
        the formed
    # clusters
    silhouette_avg = silhouette_score(X, cluster_labels)
    print(
        "For n_clusters =",
        n_clusters,
        "The average silhouette_score is :",
        silhouette_avg,
    )

    # Compute the silhouette scores for each sample
    sample_silhouette_values = silhouette_samples(X, cluster_labels
        )

    y_lower = 10
    for i in range(n_clusters):
        # Aggregate the silhouette scores for samples belonging to
        # cluster i, and sort them
        ith_cluster_silhouette_values = sample_silhouette_values[
            cluster_labels == i]

        ith_cluster_silhouette_values.sort()

        size_cluster_i = ith_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_cluster_i

        color = cm.nipy_spectral(float(i) / n_clusters)
        ax1.fill_betweenx(
            np.arange(y_lower, y_upper),
            0,
            ith_cluster_silhouette_values,
            facecolor=color,
            edgecolor=color,
            alpha=0.7,
        )

        # Label the silhouette plots with their cluster numbers at
```

```
            the middle
72          ax1.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))
73
74          # Compute the new y_lower for next plot
75          y_lower = y_upper + 10  # 10 for the 0 samples
76
77      # ax1.set_title("The silhouette plot for the various clusters
            .")
78      ax1.set_xlabel("The silhouette coefficient values")
79      ax1.set_ylabel("Cluster label")
80
81      # The vertical line for average silhouette score of all the
            values
82      ax1.axvline(x=silhouette_avg, color="red", linestyle="--")
83
84      ax1.set_yticks([])  # Clear the yaxis labels / ticks
85      ax1.set_xticks([-0.6, -0.4, -0.2, 0, 0.1, 0.2, 0.3, 0.4])
86
87      plt.savefig("plot/directly/silhouette_directly_" + str(
            n_clusters) + ".pdf", format="pdf", bbox_inches="tight")
88
89  plt.show()
```

For the one with statistics analysis, as in Figure 3.6, the code for drawing it is similar. The only difference is that, we do an aggregation for each entry (fixed-size (30) sliding window), and append those results to the end of each entry, using the nine descriptive statistics metrics, as defined in Subsection 2.6.1: percentiles (25%, 50%, 75%), kurtosis, maximum, minimum, skewness, standard deviation, as well as the variance.

D.2 Figure 3.7 & Figure 3.8

Section 3.3 also has the histogram of statistics analysis on windowed GPU load data. Here is the code for drawing those figures:

```
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 32})
plt.figure(figsize=(14,4))

data = pd.read_csv('load_gpu.csv', usecols=[i for i in range(3, 33)])
new_data = data.transpose()
stat = new_data.describe()
stat.loc['skew'] = new_data.skew()
stat.loc['kurt'] = new_data.kurt()
stat = stat.drop(['count'], axis=0)
stat = stat.transpose()
for kind in ["mean", "std", "min", "25%", "50%", "75%", "max", "skew", "kurt"]:
    pd.cut(stat[kind], bins=100).value_counts().sort_index().to_csv("load_gpu_" + kind + "_histogram.csv")
    figure = stat[kind].plot.hist(bins=100).get_figure()
    figure.savefig("load_gpu_" + kind + "_histogram.pdf")
    figure.clear()

# Verify the threshold selection
def getJobs(name, threshold):
    jobs = []
    for index in stat.index[stat[name] < threshold].tolist():
        jobs.append(data.iat[index, 1])
    jobs.sort()

    return set(jobs)

for job in (getJobs("75%", 20).union(getJobs("max", 32))):
    print(job)
```

D.3 Figure 4.1

In Subsection 4.1.2, we have a performance benchmark. The code for drawing the related Figure 4.1 is here. You can also find the raw data for the benchmark result in the code below:

```
import matplotlib.pyplot as plt
import numpy as np

# Data
methods = ["Polling CAGG", "Polling Direct Query", "Trigger CAGG", "Trigger in Memory"]
```

```
colors = ["#36A2EB", "#FF6384", "#FF9F40", "#9966FF"]
gpu_write_delay = [
    [0.321007643, 0.660196607, 1.279614867, 2.433749516,
       5.131623884, 11.624568741],
    [0.31859915, 0.625456453, 1.191235464, 2.350517039,
       4.958478354, 9.977873813],
    [0.765987412, 1.609155407, 2.73052539, 5.176909349,
       10.908855175, 20.96940738],
    [0.65528452, 1.285481557, 2.53125466, 5.007170087, 9.824896244,
        19.604722893]
]
read_iteration = [
    [1.201504459, 2.102009134, 3.58900885, 7.282948943,
       13.567867101, 18.581255946],
    [2.245655323, 3.443534234, 6.191009134, 12.676023435,
       26.139007435, 55.204802484],
    [13.184558, 30.957367, 56.111442, 101.592729, 212.071778,
       433.089509],
    [12.250112, 30.117734, 52.671703, 91.90153, 188.53631,
       388.158221]
]
index = range(len(gpu_write_delay[0]))

# Plot for GPU Write Delay
fig, axes = plt.subplots(3, 1, figsize=(8, 12))
for i, method in enumerate(methods):
    axes[0].bar([x + i * 0.2 for x in index], gpu_write_delay[i],
       0.2, label=method, color=colors[i])
    axes[0].plot([x + i * 0.2 for x in index], gpu_write_delay[i],
       color=colors[i], marker='o', linewidth=1, markersize=2)
    # Add horizontal lines to the first plot
    for tick in gpu_write_delay[i]:
        axes[0].axhline(y=tick, color='grey', linestyle='dotted',
           linewidth=0.3)
axes[0].set_xlabel('Node Count * GPU')
axes[0].set_ylabel('Time (s)')
axes[0].set_title('Delay in Database Writing Transaction Commit')
axes[0].set_yscale('log')
axes[0].set_xticks([x + 0.2 * 3 / 2 for x in index])
axes[0].set_xticklabels(['93*8', '186*8', '372*8', '744*8', '1489*8
    ', '2978*8'])
axes[0].legend()
y_ticks_0 = [0.5, 1, 2, 3, 4, 5, 6, 7, 9, 11, 14, 17, 20]
axes[0].set_yticks(y_ticks_0)
axes[0].set_yticklabels([str(tick) for tick in y_ticks_0])

# Plot for Read Iteration
for i in range(2):
    method = methods[i]
    axes[1].bar([x + i * 0.2 for x in index], read_iteration[i],
       0.2, label=method, color=colors[i])
    axes[1].plot([x + i * 0.2 for x in index], read_iteration[i],
       color=colors[i], marker='o', linewidth=1, markersize=2)
    # Add horizontal lines to the second plot
```

```
    for tick in read_iteration[i]:
        axes[1].axhline(y=tick, color='grey', linestyle='dotted',
            linewidth=0.3)
axes[1].set_xlabel('Node Count * GPU')
axes[1].set_ylabel('Time (s)')
axes[1].set_title('Time Needed in Completing a Read Loop Iteration'
    )
axes[1].set_yscale('log')
axes[1].set_xticks([x + 0.2 * 3 / 2 for x in index])
axes[1].set_xticklabels(['93*8', '186*8', '372*8', '744*8', '1489*8
    ', '2978*8'])
axes[1].legend()
y_ticks_1 = [0.5, 1, 2, 2.5, 3, 4, 5, 6, 7, 9, 12, 15, 20, 25, 30,
    40, 50, 60]
axes[1].set_yticks(y_ticks_1)
axes[1].set_yticklabels([str(tick) for tick in y_ticks_1])

# Plot for Memory Status Dump
for i in range(2, 4):
    method = methods[i]
    axes[2].bar([x + i * 0.2 for x in index], read_iteration[i],
        0.2, label=method, color=colors[i])
    axes[2].plot([x + i * 0.2 for x in index], read_iteration[i],
        color=colors[i], marker='o', linewidth=1, markersize=2)
    # Add horizontal lines to the third plot
    for tick in read_iteration[i]:
        axes[2].axhline(y=tick, color='grey', linestyle='dotted',
            linewidth=0.3)
axes[2].set_xlabel('Node Count * GPU')
axes[2].set_ylabel('Time (ms)')
axes[2].set_title('Delay in Completing a Memory Status Dump')
axes[2].set_yscale('log')
axes[2].set_xticks([x + 0.2 * 3 / 2 for x in index])
axes[2].set_xticklabels(['93*8', '186*8', '372*8', '744*8', '1489*8
    ', '2978*8'])
axes[2].legend()
y_ticks_2 = [5, 10, 20, 25, 30, 40, 50, 60, 70, 90, 120, 150, 200,
    250, 300, 400, 500]
axes[2].set_yticks(y_ticks_2)
axes[2].set_yticklabels([str(tick) for tick in y_ticks_2])
plt.tight_layout()
plt.savefig("benchmark-data.pdf")
```