Power Capping of GPU Servers for Machine Learning Inference Optimization

Yuan Ma
The Ohio State University
Columbus, Ohio, USA
ma.2800@osu.edu

Srinavasan Subramaniyan
The Ohio State University
Columbus, Ohio, USA
subramaniyan.4@osu.edu

Xiaorui Wang
The Ohio State University
Columbus, Ohio, USA
wang.3596@osu.edu

Abstract

Power capping, which is an essential component of power oversubscription, has been widely used in data centers to host more servers than allowed by the capacity of their power infrastructures, in order to avoid expensive power upgrade and reduce capital expenses. Traditionally, power capping is performed mainly with CPU frequency and voltage scaling, which cannot be directly applied to the GPU servers that are commonly deployed in today's data centers, because GPUs can have much higher power consumption than CPUs. Recently proposed GPU power capping solutions are designed for a single GPU and so cannot be used on GPU servers that have a host CPU and multiple GPUs to process machine learning (ML) workloads. Hence, a joint power capping solution must be designed to coordinate the host CPU and all the GPUs in a server for optimizing ML inference performance.

In this paper, we propose *CapGPU*, a power capping framework for today's GPU servers that run ML workloads on multiple GPUs and a host CPU in each server. In sharp contrast to existing solutions that try to control the server power consumption by throttling either the CPU or one GPU, in a separate manner, CapGPU features a multi-input multi-output (MIMO) power control methodology and a novel weight assignment algorithm that dynamically adjusts the weights assigned to each CPU/GPU based on their measured throughput. Consequently, the overall ML inference performance can be optimized. Our hardware testbed results demonstrate that CapGPU outperforms several state-of-the-art power capping solutions by having more precise power control, higher inference throughput, and better inference latency guarantees.

CCS Concepts

• **Hardware → Enterprise level and data centers power issues**; • **Computing methodologies → Computational control theory**; **Machine learning**.

Keywords

Power capping, power oversubscription, GPU server, data center.

ACM Reference Format:
Yuan Ma, Srinavasan Subramaniyan, and Xiaorui Wang. 2025. Power Capping of GPU Servers for Machine Learning Inference Optimization. In *54th International Conference on Parallel Processing (ICPP '25), September 08–11, 2025, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3754598.3754670

ICPP '25, September 08–11, 2025, San Diego, CA, USA

ACM ISBN 979-8-4007-2074-1/25/09.
https://doi.org/10.1145/3754598.3754670

1 Introduction

Recent years have witnessed a significant boom in the construction of new data centers [10], partially due to the fast AI development that has driven tech companies to build new AI infrastructures. It has been predicted by AFCOM (a leading association for data center professionals) that data center construction would increase sixfold in the next three years [12]. In addition to IT equipment, a significant capital expense (CapEx) of a data center is its power delivery infrastructure, which is a hierarchy of devices (e.g., power substation, power distribution unit) that supply power to different IT equipment in the data center. Of the $200B worldwide spending on data center systems annually, tens of billions of dollars have been spent on power infrastructures [24]. To reduce this CapEx, many data centers (e.g., Google [24], Meta [23], Microsoft [13]) have adopted *power oversubscription*, which aims to host more servers than allowed by the capacity of the power infrastructure, so that expensive upgrades of the power infrastructures can be avoided when more servers need to be added to the data center. Power oversubscription is based on the key observation that servers rarely have peak power draw at exactly the same time. However, power oversubscription indeed imposes a risk of power overload, which could trip the circuit breakers on the power devices and cause undesired server shutdowns and outages. Hence, power oversubscription must be used together with *power capping*, which monitors the server power consumption in real time and caps their peak power draw to enable safe power oversubscription by preventing power overload.

Power capping has been used in production data centers for years (e.g, Google [24], Meta [23], Microsoft [13], IBM [16]). For example, IBM servers have been equipped with the "power capping" feature as part of the IBM Active Energy Manager product, which has been deployed in many IBM System x and System p server models since 2007 [14]. Power capping can be enforced on a server, a server rack, or even on an entire data center. For example, Meta has deployed Dynamo, a data center-wide power capping system since 2013, across all of Meta's data centers [34]. Traditionally, power capping is performed by conducting DVFS (Dynamic Voltage and Frequency Scaling) on the CPUs in a server, because CPUs used to be the biggest power consumers. The objective of power capping is to achieve the best server performance by using as much power as allowed by the desired power cap. Unnecessarily throttling the CPU DVFS to keep server power consumption way below its cap (or budget) is highly undesired, because doing so can slow down the applications running on the servers.

Unfortunately, CPU-based power capping *cannot* be directly applied to the GPU servers that are commonly deployed in today's data centers to accelerate AI computation. GPUs can have much higher power consumption than CPUs, due to their higher densities of memory and computing units executing in parallel. For example, the power consumption of a high-end GPU used for AI (e.g., Nvidia H100) can reach 700 Watts. In such GPU servers, adjusting CPU DVFS has small or even negligible impacts on the high GPU power consumption. As a result, the traditional CPU-based power capping solutions have become much less effective for GPU servers, because they cannot adapt the GPU-side power consumption. To that end, some recent studies have proposed GPU power capping by scaling the GPU frequency [4] or adjusting the batch sizes of machine learning (ML) tasks [20]. However, these solutions are designed to cap the power consumption of *a single GPU*, while there are commonly multiple GPUs equipped in a GPU server for ML processing (referred to as Multi-GPU servers [5]). For example, a server is usually equipped with one host CPU and up to eight GPUs [22]. In addition, those solutions focus on one GPU only without coordinating with the host CPU for better ML performance.

In today's data centers, GPUs and the host CPU in a server are commonly configured to carry out the same ML tasks together. For example, the host CPU is often used to prepare ML data for the GPUs and receive the inference results from the GPUs afterwards. Thus, controlling CPU and GPU frequencies in isolation can degrade the ML inference performance, resulting in lower throughput or longer inference latency. For instance, throttling the host CPU may delay the preprocessing tasks, which might stall the subsequent ML inference processing on GPUs. On the other side, throttling a GPU can increase its ML inference processing time, causing CPU threads and other GPUs to idle while waiting for results. Therefore, while the existing power capping solutions focus on either CPU or GPU, a joint power capping solution must be designed to coordinate the host CPU and GPUs in a server for optimizing ML inference performance. Although some previous work tries to do power capping for both CPU and GPU [2], it is not designed for ML workloads that require CPU and GPUs to coordinate. So, it simply divides the server power cap between one CPU and one GPU and then controls them independently.

In this paper, we propose *CapGPU*, a power capping framework for today's GPU servers that run ML workloads on multiple GPUs and the host CPU in each server. In sharp contrast to existing solutions that try to control the server power consumption by throttling either the CPU or one GPU, in a separate manner, CapGPU features a multi-input multi-output (MIMO) power control methodology that coordinates the CPU DVFS and the frequency level scaling of each individual GPU for optimizing the ML inference performance. Specifically, we propose a novel weight assignment algorithm that monitors the inference throughput of each GPU and the CPU in real time and gives higher weights to CPU/GPU with higher throughput, so that they can run at higher frequencies. Consequently, the overall ML inference performance can be optimized.

Specifically, this paper makes several major contributions.

- While the existing power capping solutions focus on either the CPU or one GPU, in a separate manner, we observe that ML inference workloads demand for coordinated power capping. To that end, we propose the first power capping solution that is specifically designed for today's GPU servers that have one CPU and multiple GPUs to run ML workloads.
- We propose a MIMO power control methodology that coordinates the CPU DVFS and the frequency level scaling of each individual GPU for optimizing the ML inference performance. Our controller is designed based on MIMO control theory for guaranteed system stability and control accuracy. A novel weight assignment algorithm is designed for ML inference optimization.
- We evaluate CapGPU on a hardware testbed with three Nvidia V100 GPUs and various ML inference workloads. Our results demonstrate that CapGPU outperforms several state-of-the-art power capping solutions, including CPU-only, GPU-only, and separate GPU and CPU capping with simple power cap division, by having more precise power control, higher inference throughput, and better inference latency guarantees.

2 Related Work

We divide the related work into two main categories: (1) Power capping and (2) inference power optimization.

Power Capping. Power capping has been used in many data centers (e.g, Google [24], Meta [23], Microsoft [13], IBM [16]). Power capping can be enforced on a server [14], a server rack [28], or even on an entire data center [29, 34]. For example, Meta has deployed Dynamo, a data center-wide power capping system since 2013, across all of Meta's data centers [34]. In addition to server power capping, Thunderbolt [15] discusses power capping techniques for cloud CPUs, while [19] address power capping in chip multiprocessors. Ma et al. [18] propose a holistic solution for energy efficiency in GPU-CPU heterogeneous architectures. OptimML [4] employs DVFS and latency control to meet power budgets but focuses on a single GPU. These solutions are mainly designed for best-effort workloads and do not offer explicit SLO guarantees for multiple CPUs and GPUs.

Inference Power Optimization. Several works use algorithmic approaches to reduce power consumption in inference workloads. Dynamic batching [11, 20] leverages the batch size of a workload as a control knob to lower energy usage. EnergyNet [31] proposes a CNN-based energy-aware dynamic routing approach, providing adaptive-complexity inference based on input characteristics. Zhang et al. [35] propose a power control method for edge ML inference with hypernetwork meta-parameters. For CPUs, pruning [6] has been extensively studied to improve energy efficiency in DNN applications. Quantizing DNN parameters to lower numerical precision (e.g., 16-bit floating point or 8-bit integer) is also proposed to reduce energy consumption [6]. However, these studies focus on energy efficiency without capping server power consumption.

3 Overview of CapGPU

This section presents a high-level overview of *CapGPU*, which adaptively manages server power consumption within specified power limits. It achieves this by applying Dynamic Frequency Scaling to both CPUs and GPUs within the server. We later show a motivation example that illustrates performance optimization using *CapGPU*.

Table 1: End-to-end performance under different frequency controls.

Config	CPU Frequency	GPU Frequency	Preprocessing Latency (s/img)	GPU Latency (s/batch)	Queue (s/img)	Throughput (img/s)	Power (W)
CPU-only	1.1 GHz	810 MHz	0.1	1.3	3.2	5.3	406.4
GPU-only	2.1 GHz	495 MHz	0.2	2.0	2.7	5.9	421.3
CapGPU	1.6 GHz	660 MHz	0.1	1.6	2.5	**6.4**	415.1

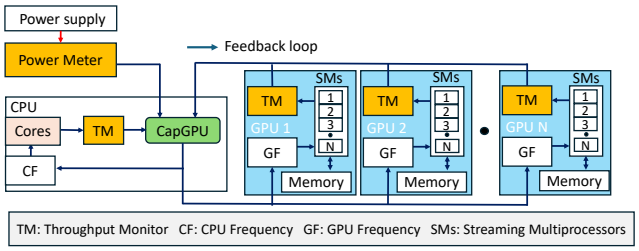


Figure 1: System design of CapGPU

3.1 System Overview

In this work, we adopt CPU and GPU frequency scaling as our actuation knobs for two main reasons. First, modern GPUs and CPUs typically provide well-defined interfaces for frequency adjustment, such as `nvidia-smi`, `AMD ROCm`, and `cpupower` [1, 21]. Second, CPUs and GPUs usually account for the majority of a server's total power consumption [22]. This is especially true in modern AI-driven data centers, where GPUs dominate energy use. Consequently, the power difference between their highest and lowest performance states is significant enough to compensate for fluctuations in other components, making them effective for enforcing power budgets during supply constraints or failures.

As illustrated in Figure 1, the power control loop consists of several key components: a centralized controller, a power monitor at the server level, CPU and GPU throughput monitors, and actuators capable of adjusting CPU and GPU frequencies. The feedback control loop executes at the end of each control period and operates as follows:

(1) The server's power monitor (e.g., a power meter) measures the average power consumed during the previous control period and transmits this value to the controller through a feedback path. This total power consumption serves as the controlled variable in the loop.
(2) Each GPU's throughput monitor reports its average inference throughput, measured by the number of inference tasks completed per second. The CPU throughput monitor reports the number of feature subsets evaluated per second. The normalized throughput of each device is computed by dividing its throughput by the maximum throughput of respective device. These normalized throughput metrics help the controller make informed decisions about weight allocation in the upcoming control period.
(3) The controller receives the power and throughput data, calculates the appropriate CPU and GPU frequency levels for each device, and sends them to the respective frequency modulators. The CPU and GPU frequency levels are the manipulated variables in the control loop.
(4) The frequency modulators on each CPU/GPU apply the new CPU and GPU frequency.

Note that though modern GPUs and CPUs commonly have interfaces, such as `nvidia-smi` and `RAPL`, which allow users to set desired power limits, these mechanisms cannot 1) cap the power consumption of an entire GPU server with a host CPU and multiple GPUs, 2) optimize the performance of ML inference processing. In the following section, we will focus first on system modeling along with the design and analysis of CapGPU. Details on the implementation of other components can be found in Section 5.

3.2 Motivation

In this section, we explain the importance of coordinated control for adjusting both CPU and GPU frequencies simultaneously. Throttling either the CPU frequency alone (referred to as *CPU-only*) or the GPU frequency alone (referred to as *GPU-only*) can negatively impact overall inference performance. To illustrate this, we conduct the following experiment.

We emulate a cloud server that processes ten parallel requests, each classifying twenty wildlife images using the GoogLeNet model trained on the Oregon Wildlife dataset [7]. Each request runs in a dedicated CPU process pinned to a distinct physical core. The CPU performs full preprocessing (resize, normalization, and tensor conversion via `torchvision` transforms) and pushes the resulting tensors into a shared queue. A single GPU-bound consumer then assembles batches and runs inference on an NVIDIA RTX 3090.

We evaluate three frequency-control modes: (1) *CPU-only*, where the CPU is set to its lowest frequency (1.1 GHz) and the GPU runs at a high frequency (810 MHz); (2) *GPU-only*, where the GPU is set to its lowest frequency (495 MHz) and the CPU runs at its highest frequency (2.1 GHz); and (3) *CapGPU*, where both the CPU (1.6 GHz) and the GPU (660 MHz) are set to approximately the midpoint of their respective high and low frequencies. We measure CPU preprocessing delay (seconds per image), queue delay (seconds per image), GPU latency (seconds per batch), throughput (images per second), and average power draw (W).

Table 1 summarizes the results across all performance metrics. *CapGPU* outperforms both *CPU-only* and *GPU-only* by using coordinated control such that neither resource remains idle for extended periods. In the *CPU-only* mode (CPU at 1.1 GHz, GPU at 810 MHz), the CPU preprocessing time is 0.14 seconds per image, but the GPU, running at a higher clock, often starves for data, driving queue delays up to 3.2 seconds per image and yielding a throughput of only 5.3 images per second. In the *GPU-only* mode (CPU at 2.1 GHz, GPU at 495 MHz), the CPU preprocessing time is 0.15 seconds per image, but the GPU becomes the bottleneck, with GPU latency increasing to 2.0 seconds per batch and queue delays reaching 2.7 seconds per image, resulting in a throughput of 5.9 images per second. With

CapGPU (CPU at 1.6 GHz, GPU at 660 MHz), the CPU preprocessing time is 0.15 seconds per image, the queue delay reduces to 2.5 seconds per image, GPU latency improves to 1.6 seconds per batch, and throughput increases to 6.4 images per second.

4 Design

In this section, we introduce the details of the proposed CapGPU design. Our control objective is the following. Whenever the GPU server's power consumption deviates from its respective set point, we dynamically adjust the frequencies of each GPU and the CPU (i.e., the manipulated variables) in each control period, so that total server power converges back to its set points within a finite settling time.

4.1 Problem Formulation

The control of system power can be formulated as a dynamic optimization problem. We first introduce the following notation:

- T: control period.
- $f_g(k)$: GPU frequency level at the k^{th} control period.
- $f_c(k)$: CPU frequency level at the k^{th} control period.
- N_c, N_g: Total number of CPUs and GPUs in the system.
- $p(k)$: total power consumption of the server at the k^{th} period.
- P_s: desired power set point of the system.
- $f_{c,\min}, f_{c,\max}$: minimum and maximum allowable CPU frequencies.
- $f_{g,\min}, f_{g,\max}$: minimum and maximum allowable GPU frequencies.
- $e_i(k)$: Inference latency of the i^{th} inference task at time k.
- $e_{\min,i}$: Minimum inference latency of task t_i when $f_g = f_{g,\max}$.
- γ: Empirical factor modeling non-linear frequency-latency scaling.
- $d(k)$: Increment in frequency command applied at time k.
- $\mathbf{A}$: Matrix of identified coefficients relating CPU/GPU frequencies to total power.
- $\mathbf{F}(k)$: Vector of CPU/GPU frequencies at time k.

The optimization problem is formulated to control system power by adjusting CPU and GPU frequencies. The objective is to minimize the squared difference between the desired power set point and the actual system power consumption. The formulation is:

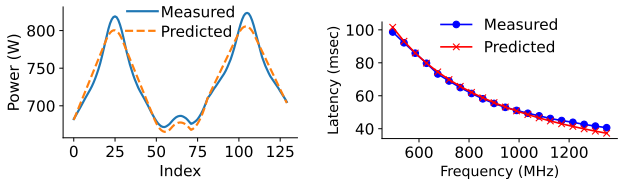
$$\min_{u(k)} f\big(u(k)\big), \quad \text{where } f\big(u(k)\big) = \big(P_s - p(k)\big)^2, \quad u(k) = \begin{bmatrix} f_c(k) \\ f_g(k) \end{bmatrix}. \tag{1}$$

subject to frequency constraints:

$$f_{c,\min} \le f_c(k) \le f_{c,\max}, \quad f_{g,\min} \le f_g(k) \le f_{g,\max}. \tag{2}$$

The goal is to minimize the difference between the power set point P_s and measured power $p(k)$ by adjusting CPU and GPU frequencies within their specified limits.

We adopt a control-theoretic methodology to design the centralized power controller, because control theory provides standardized methods for selecting control parameters, eliminating the need for exhaustive manual tuning and testing. This methodological advantage significantly reduces the time required for model calibration and enhances robustness.



(a) Power **(b) Frequency**

Figure 2: (a) System identification results for a system with one CPU and one GPU. (b) Variation between measured and predicted inference latency.

4.2 System Modeling

For effective controller design, it is essential to accurately model the dynamics of the system being controlled. This involves understanding the relationship between the manipulated variables, such as GPU frequencies and CPU frequency, and the controlled variable, which in this case is the server's power consumption. In order to have an effective controller design, it is crucial to model the dynamics of the controlled system, namely the relationship between the manipulated variables (GPU frequencies and CPU frequency) and the controlled variables (i.e., the server power consumption). Since there is no well-established physical equation directly relating these variables, we employ a standard method called system identification [9][28]. Because the power consumption of both a CPU and a GPU varies linearly with its frequency [14][4], we model the relationship as a linear system. For a system with N_c CPUs and N_g GPUs, we write the server power as in Equation (3):

$$p = \sum_{j=1}^{N_c} A_j \times f_{cj} \; + \; \sum_{i=1}^{N_g} B_i \times f_{gi} \; + \; C, \tag{3}$$

where A_j and B_i are coefficients (gains) for CPU and GPU frequencies, respectively, and C is a constant offset.

Equation (3) can be represented in matrix form as

$$p = \mathbf{A} \times \mathbf{F} + C, \tag{4}$$

where $\mathbf{F} = \begin{bmatrix} f_{c1} & f_{c2} & \dots & f_{cN_c} & f_{g1} & f_{g2} & \dots & f_{gN_g} \end{bmatrix}^T$ and $\mathbf{A} = \begin{bmatrix} A_1 & A_2 & \dots & A_{N_c} & B_1 & B_2 & \dots & B_{N_g} \end{bmatrix}$.

The model in Equation (4) can be represented in the time domain as:

$$p(k) = \mathbf{A} \times \mathbf{F}(\mathbf{k} - \mathbf{1}) + C, \tag{5}$$

In system identification, we systematically vary one frequency input (e.g., GPU frequency) while holding the other fixed (e.g., CPU frequency) and record the resulting power consumption; then we reverse the process. We collect these measurements into a set of linear equations and solve for $\mathbf{A}$ via least square regression. We keep C as a constant offset, and any noise or unmodeled dynamics are captured in the measurement error.

Difference Equation. The first step in controller design is to derive the difference equation from the linear power model in Equation (5). We have:

$$p(k-1) = \mathbf{A} \times \mathbf{F}(k-2) + C, \tag{6}$$

where $\mathbf{F}^\top(k-2) = \begin{bmatrix} f_{c1}(k-2) & f_{c2}(k-2) & \cdots & f_{cN_c}(k-2) \\ f_{g1}(k-2) & f_{g2}(k-2) & \cdots & f_{gN_g}(k-2) \end{bmatrix}$
$p(k-1)$ is the power consumption at control period $k-1$, $\mathbf{A}$ is the matrix of coefficients obtained via system identification, and C is a constant offset. To predict the power consumption at the next control period, we write $p(k) = \mathbf{A} \times \mathbf{F}(k-1) + C$. Taking the difference $p(k) - p(k-1)$ leads to the final model:

$$p(k) = p(k-1) + \mathbf{A} \times \Delta\mathbf{F}(k-1). \tag{7}$$

Example. The workload we use for system identification is described in detail in Section 6. To perform system identification, we vary the GPU frequency between 495 MHz and 1350 MHz, and the CPU frequency between 1 GHz and 2.1 GHz. We assume a single CPU and a single GPU. First, we increase the GPU frequency from 435 MHz to 1350 MHz while keeping the CPU frequency at 1.4 GHz. Next, we fix the GPU frequency at 495 MHz and vary the CPU frequency between 1 GHz and 2.1 GHz. We record the power consumption and use least squares to compute the predicted power. The variation in the predicted and measured power is shown in Figure 2(a). The resulting model achieves an R^2 of 0.96. This modeling process can be automatically performed for another GPU server with different GPU/CPU hardware, which only results in different model coefficients.

Impact of Frequency Scaling on Inference Latency: Inference tasks in modern data centers often have strict Service-level Objective (SLO), usually in terms of inference latency, which require that their latency remain below a specified threshold. However, reducing the frequency of the GPU f_g can significantly increase the latency of these tasks [4]. Let $f_{g,\max}$ be the maximum allowed GPU frequency, and suppose that a task t_i reaches its minimum inference latency $e_{\min,i}$ when operating at $f_{g,\max}$. As the frequency drops, the GPU processes tasks more slowly, leading to longer latencies and potential SLO violations if the frequency is not carefully managed. To capture the effect of frequency scaling, we model the inference latency e_i of task t_i at GPU frequency f_g as:

$$e_i = e_{\min,i} \cdot \left(\frac{f_{g,\max}}{f_g}\right) \cdot \gamma, \tag{8}$$

where γ is an empirically determined parameter accounting for non-linear frequency scaling. In our experiments, we set $\gamma = 0.91$, achieving a high modeling accuracy ($R^2 \approx 0.91$). Although our controller adjusts the CPU frequency f_c for data cleaning tasks, these tasks do not have SLO requirements. Consequently, the strict latency bound applies primarily to GPU-based workloads. Thus, the controller must carefully regulate f_g to balance power efficiency and ensure latency-sensitive GPU tasks stay within SLO thresholds.

4.3 CapGPU Controller Design

A model predictive controller (MPC) optimizes a cost function defined over a future time interval. The controller uses a system model to predict control behavior over P sampling periods, known as the *prediction horizon*. Its goal is to determine an input trajectory that minimizes the cost function while adhering to system constraints. This input trajectory consists of the control inputs for the next M sampling periods: $d(k), d(k+1 \mid k), \ldots, d(k+M-1 \mid k)$, where M is the *control horizon*. The notation $x(k+i \mid k)$ indicates that the value of x at time $(k+i)T$ depends on the state at time kT.

After computing the input trajectory, the controller applies only the first control input, $d(k)$. At the end of the next sampling period, the prediction horizon shifts forward by one step, and the input trajectory is recalculated using updated feedback $p(k)$ from the power monitor. This feedback is essential for dealing with uncertainties and inaccuracies in the model.

In this way, CapGPU integrates performance prediction, real-time optimization, constraint handling, and feedback control into a unified framework. The controller comprises a least squares solver, a cost function, a reference trajectory, and a system model. At the end of each sampling period, it computes the control input $d(k)$ that minimizes the following cost function:

$$V(k) = \sum_{i=1}^{P} \| p(k+i \mid k) - P_s \|^2_{Q(i)} + \sum_{i=0}^{M-1} \| d(k+i \mid k) + f(k+i \mid k) - f_{\min} \|^2_{R(i)}, \tag{9}$$

where P is the prediction horizon and M is the control horizon, $Q(i)$ is the weight of the tracking error, $R(i)$ is the vector of weights for the control penalty, and $p(k+i \mid k)$ is the predicted power from the system model. The second term in (9) represents the control penalty and balances system performance by comparing the newly adjusted frequency $d(k+i \mid k) + f(k+i \mid k)$ against a reference frequency such as $f_{\min}$; to handle varying workloads, the controller can assign larger weights to busier components by normalizing and inverting their throughput. This control problem is subject to three constraints, shown in (10): the first ensures each component's frequency remains within allowable limits, the second enforces the latency model for inference tasks based on GPU frequency scaling, and the third ensures that latency does not exceed the SLO of each inference job. Formally,

$$f_{\min,j} \;\le\; d_j(k) + f_j(k) \;\le\; f_{\max,j}, \quad (1 \le j \le N), \tag{10a}$$

$$e_i(k) = e_{\min,i}\left(\frac{f_{g,\max}}{f_g(k)}\right)^{\gamma}, \quad (1 \le i \le N), \tag{10b}$$

$$e_i(k) \;\le\; \mathrm{SLO}_i, \quad (1 \le i \le N). \tag{10c}$$

where $f_{g,\max}$ is the maximum GPU frequency, γ is an empirically determined parameter, and $e_{\min,i}$ is the minimum latency for task t_i at $f_{g,\max}$. Based on this formulation, server-level power management becomes a constrained MPC optimal control problem; we implement the solver with `SLSQP` in Python, which has polynomial complexity in both the number of controlled components and in the control/prediction horizons.

The MPC controller has small overhead and can complete its computation in just a few milliseconds when a server has about 4 to 8 GPUs [30]. In addition, the computational complexity and runtime overhead of the MPC controller can be further reduced by using a multi-parametric approach that 1) divides the MPC control problem into an offline part and an online part, and 2) solves the online part incrementally as a piecewise linear function [32].

4.4 Stability Analysis

A key advantage of our control-theoretic approach is that it provides theoretical assurance of system stability, even when the estimated model parameters (i.e., entries of $\mathbf{A}$) change due to different workloads. We say the system is *stable* if its total power $p(k)$ converges to the desired set point P_s; formally, $\lim_{k\to\infty} p(k) = P_s$.

Our CapGPU controller solves a finite-horizon optimal tracking problem, and, based on standard results in optimal control theory [9], its control decisions become linear functions of the current power $p(k)$, the power set point P_s, and the previous CPU/GPU frequency decisions.

We now outline the general process for analyzing the stability of a server when the *actual* system model differs from the estimated one (i.e., the true parameters $\mathbf{A}'$ deviate from $\mathbf{A}$):

(1) **Nominal Control Inputs:** Given the estimated model $p(k) = \mathbf{A}\,\mathbf{F}(k-1) + C$, we derive the sequence of control inputs $\{d(k), d(k+1 \mid k), \dots\}$ that minimize the cost function in (9). These inputs correspond to the controller's decision under the assumed parameters $\mathbf{A}$.
(2) **Actual System Model:** Suppose the *true* parameters are $\mathbf{A}'$, where each entry of $\mathbf{A}'$ is $g_i\, A_i$. Here, g_i is an unknown gain factor that differs from the nominal assumption. The actual system evolves according to $p(k) = \mathbf{A}'\,\mathbf{F}(k-1)+C$. We form a composite system by combining the actual system dynamics with the controller inputs computed from the nominal model.
(3) **Closed-Loop System:** We substitute the nominal controller inputs (obtained in Step 1) into the *actual* system model, yielding the closed-loop dynamics. These equations reflect how the real system responds to the controller's actions.
(4) **Pole Analysis:** We then analyze the poles (eigenvalues) of the resulting closed-loop system. According to linear control theory, if all poles lie strictly inside the unit circle in the complex plane, the power $p(k)$ converges to P_s. We also examine the DC gain from the control inputs to the system state; if it equals the identity matrix (or satisfies the necessary conditions in a multivariate setting), the closed-loop state approaches the desired set point. By tracking how the poles shift as g_i changes, we identify the range of parameter variations for which stability remains guaranteed.

Our analysis results show that the controlled GPU server remains stable as long as each A_i stays within a derived bound. The detailed proof is omitted due to space limitations. We assume that the constrained optimization problem is *feasible*, meaning there exists at least one combination of CPU and GPU frequency levels (within the ranges defined in Section 4.2) that achieves $p(k) = P_s$. If no such combination exists, then no single control algorithm can strictly enforce the set point through frequency adaptation alone. In such cases, additional system mechanisms (e.g., memory throttling) must be integrated. Exploring such multi-layer adaptations is part of our future work.

5 System Implementation

Hardware Testbed. We conduct our experiments on a server that hosts three Nvidia Tesla V100 16 GB GPUs. The server runs Ubuntu 20.04.6 with kernel 5.15.0. It is equipped with an Intel Xeon Gold 5215 @ 2.20 GHz processor (40 cores) and 128 GB of RAM. We use PyTorch 1.12.1 as our ML framework. Prior studies have highlighted that the fan is a significant contributor to system power consumption in modern ML servers [22]. To isolate the effect of workload-driven power variations, we fix the fan speed to a constant value throughout our experiments. Each inference workload running on a GPU is assigned one dedicated CPU core for its data preparation. An additional CPU core is allocated to the controller. The remaining CPU cores are used for the CPU-based workload.

Power Monitor. We measure server-level power consumption using the *power_meter-acpi-0* interface exposed by the Linux *lm-sensors* framework. This interface represents an ACPI-compliant power meter device, as defined in the ACPI Specification v6.4 [27]. The power meter reports real-time power consumption in watts, typically sampled at one-second intervals. It provides a system file that writes the power data to the user space. The power meter continuously samples power and appends new readings to the system file, which is updated periodically. The controller reads this data from the file and performs control computations based on the latest power measurements. We also used Nvidia-smi to monitor the power consumption of GPUs for baseline power capping solutions.

Controller. The controller runs as a multi-threaded process. The main thread uses a timer to periodically invoke the control algorithm described in Section 4, while a child thread uses the `select` function to collect CPU and GPU utilization data. Each time the periodic timer fires, the controller retrieves the average of power reading and the current CPU and GPU utilizations during the latest control period, and then invokes the control algorithm. Based on the algorithm's output, new CPU and GPU frequency levels are computed and sent to the frequency modulator on each server to be enforced during the next control period.

Frequency Modulators. We configure the CPU frequency governor by `cpupower` tool, with the command of `sudo cpupower frequency-set -f {CPU_freq}GHz`, where `{CPU_freq}` is the desired frequency value. The CPU supports iscrete frequency levels ranging from 1.1 GHz to 2.4 GHz. To configure GPU frequency, we use the Nvidia System Management Interface running `nvidia-smi -ac 877,435-1350`, where the memory clock is fixed at 877 MHz and the GPU core clock range is set from 435 MHz to 1350 MHz. Since the new CPU and GPU frequency levels received from the controller are floating-point (fractional) values, the modulator code locally resolves them into a sequence of discrete frequency levels to approximate the target value. To approximate a frequency of 2.4 GHz, we use a first-order delta-sigma modulator that switches between the two nearest discrete steps, which are 2 GHz and 3 GHz. For example, by toggling between the values 2, 2, 2, and 3, the time-averaged frequency converges to the desired value [14].

6 Results

In this section, we present the experimental results. We first introduce the baselines and the workloads in Section 6.1. We then compare CapGPU with the baselines in Section 6.2. In Section 6.3, we evaluate CapGPU's control accuracy for different set points. Finally, in Section 6.4, we examine the adaptability of CapGPU.

6.1 Baselines and Workloads

Baselines. We compare our proposed CapGPU controller against four baseline approaches: *1) Fixed-step*, *2) GPU-Only*, *3) CPU-Only*, and *4) CPU+GPU*. These methods represent heuristic-based and control-theoretic strategies for power capping in ML servers and are adapted from recent work in the field. Additionally, they target different CPU and GPU throttling.

1) Fixed-step: Fixed-step is a simple heuristic controller inspired by the power control scheme described in [20]. It represents a practical, industry-style solution that does not rely on any system model.

All CPUs and GPUs initially operate at their lowest frequency levels. In each control period, if the total system power consumption is below the target set point, the controller selects a CPU or GPU with the highest normalized utilization and increases its frequency level by one fixed step size. If the power exceeds the set point, it selects the component with the lowest utilization and decreases its frequency by one step size. When all components have identical utilization values, the controller chooses among them in a round-robin fashion to ensure fairness. To improve convergence speed, the controller uses coarse-grained frequency steps (e.g., 150 MHz for GPUs and 1000 MHz for CPUs). While Fixed-step is lightweight and simple to implement, it does not account for workload dynamics or CPU-GPU interaction, which can lead to oscillations in power and suboptimal application performance.

2) GPU-Only: The GPU-Only uses a proportional controller to control power consumption throughout the system by adjusting GPU frequency using a linear model [4]. During each control period, the GPU-Only controller measures the total power consumption and modifies the frequencies of all the GPUs in the server based on the difference between the measured power and the desired target set point. The gain for this controller is determined by pole placement and choosing the one that minimizes oscillations. In GPU-Only, a single frequency is applied to all GPUs.

3) CPU-Only: The CPU-Only controller represents the traditional power capping solutions that mainly adapt the CPU DVFS to control the server power. CPU-Only retains the proportional control logic of GPU-Only but actuates only the CPU DVFS knobs: At the beginning of each control period, it samples the total power draw of the servers, compares it with the target cap to compute the error, and then adjusts the CPU frequency by a gain selected through pole placement, following the steps introduced in a power capping solution from IBM [14]. The CPU-Only applies a single frequency to all the CPU cores of the server.

4) CPU+GPU: CPU+GPU utilizes two separate power control loops to independently control the CPU and GPU power by respectively adapting their frequencies, which is similar to the solution presented in [2]. Given a total power budget for the GPU server, CPU+GPU simply divides the budget using fixed values, so that the two control loops can individually control CPU/GPU power to their caps, making the total server power consumption converge to the desired cap. However, it is commonly difficult to find the best way of dividing the power cap between CPU and GPU, so CPU+GPU may not have the best ML inference performance due to the lack of coordination between the two control loops.

Workloads. We use three deep learning models, Swin Transformer [17], ResNet50 [3], and VGG16[26], denoted as $t_1 - t_3$, to perform image classification with a batch size of 20. All models are initialized with pretrained weights from `torchvision`, trained on the ImageNet dataset. Among them, only Swin is transformer-based, while ResNet and VGG are purely convolutional. For the CPU workload, we implement an *exhaustive feature selection* algorithm [8] on the *Alibaba PAI dataset* [33], a real-world ML trace frequently used in data center resource management research. We perform feature selection to fit and test a model using every possible feature subset, and choose the feature subset yielding the lowest cross-validation (CV) Mean Squared Error (MSE).

Controller configuration. The CapGPU controller in all experiments uses a prediction horizon of 8 and a control horizon of 2. The power meter has a minimum sampling interval of 1 second, so the control period T is set to 4 seconds. This helps avoid errors from momentary power reading fluctuations. While a 4-second control period might appear long for managing power budget violations, the control algorithm responds more quickly in high-performance servers that have precise power monitors with shorter sampling intervals [28].

6.2 Comparison with Baselines

For this set of experiments, we chose three workloads: t_1 (ResNet50), t_2 (Swin Transformer), and t_3 (VGG16). For the CPU, we select the feature selection workload as described in Section 6.1. We assign each inference task to a separate GPU, so that t_1 is scheduled to GPU 0, t_2 is scheduled to GPU 1, and t_3 is scheduled to GPU 2. For this experiment, we chose the set point as 900 W.

We first test the *CPU-Only* controller described in Section 6.1. Figure 3 shows that the control range of *CPU-Only* is very minimal since it only adjusts the CPU frequency. Moreover, since today's servers can host multiple GPUs [22], this solution becomes infeasible. Next, we test the *GPU-Only*, where we control the system power by adjusting the GPU's frequency. Figure 3 illustrates the variation in power control using the *GPU-Only* controller. The system power under the *GPU-Only* configuration can converge precisely to the set point with minimal oscillation, clearly demonstrating the advantages of control-theoretic designs as shown in Figure 3. The performance of the *GPU-Only* controller surpasses that of the *fixed-step* controller. However, there are two limitations to the *GPU-Only* controller. While it effectively manages the power to the set point, it negatively impacts application performance. The *GPU-Only* controller can increase inference latency for workloads and often violates Service Level Objectives (SLOs) for tasks, which will be discussed in detail in a later section. Additionally, since *GPU-Only* will not change the CPU frequency, the CPU frequency must be set to the maximum level throughout the process, which may result in a lower power budget for the GPU's ML inference workload.

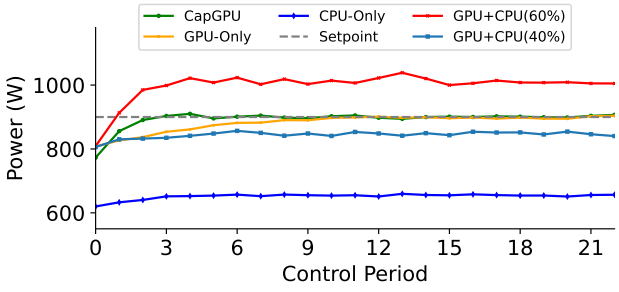


Figure 3: Power control using State-of-The-Art Baselines and CapGPU.

It is evident that both *CPU-Only* and *GPU-Only* approaches have their limitations: *CPU-Only* control is constrained by the small adjustable power range of CPUs, making it insufficient to manage power on today's GPU servers, while *GPU-Only* control, though effective in meeting power targets, cannot adjust the CPU frequency

and may affect GPU performance or cause SLO violations. To address these issues, we next explore three combined CPU and GPU control approaches, namely *GPU+CPU*, *Fixed-step*, and *CapGPU*.

We first test the *GPU+CPU* mentioned in Section 6.1. For this experiment, we select two ratios to test the power control performance. Here, we test two configurations: one where we assign 50% power to both the CPU and GPU, and another where we assign 60% power to the GPU and 40% power to the CPU. Figure 3 illustrates that different power assignment ratios may lead to different power control performance. However, since the GPU component and the CPU component have different power ranges, it is hard to find the perfect ratio for overall power control with this approach.

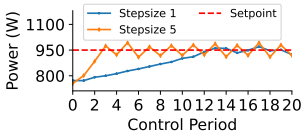


Figure 4: Fixed-Step controller for different step-sizes.

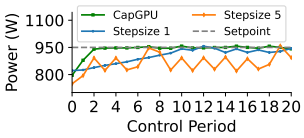


Figure 5: Safe Fixed-Step controller for different step sizes.

Next, we evaluate *Fixed-Step* where we use two separate CPU and GPU controllers as described in Section 6.1. The *Fixed-Step* controller increases or decreases the GPU or CPU frequency by one level, depending on whether the measured power is lower or higher than the set point. Figure 4 shows the power control behavior of the *Fixed-Step* controller. As mentioned before, for the *Fixed-Step* controller, if the total power consumption is lower than the set point, we increase the frequency level of the component (CPU or GPU) with the highest utilization by one step. If the power consumption exceeds the set point, we decrease the frequency level of the component with the lowest utilization by one step. If either the CPU or GPU frequency reaches its upper or lower bound, we alternate adjustments between the two components.

We define the CPU and GPU frequency step sizes as 100 MHz and 90 MHz, respectively. For this experiment, we chose two step sizes: stepsize 1 and stepsize 5 to illustrate their adaptation to different step size. For a step size of 1, the CPU and GPU frequency steps are 100 MHz and 90 MHz, respectively. For stepsize 5, they are 500 MHz and 450 MHz. We do not use the same step size for both CPU and GPU because the available frequency levels are hardware-dependent. For example, Nvidia GPUs offer frequencies in specific multiples, such as 135 MHz and 225 MHz. From Figure 5, we observe that with a small step size like stepsize 1, it takes a long time to reach the vicinity of the set point, and once it does, the system oscillates. For the larger step sizes, such as stepsize 5, the system still oscillates around the set point. One limitation of the *Fixed-Step* controller is that it requires precise and exhaustive tuning to select appropriate frequency levels that allow it to settle near its target. Any changes in workload can cause significant fluctuations.

To prevent power violations, we can introduce a safety margin to *Fixed-Step*, ensuring that the maximum power consumption remains just below the constraint for a specific workload. Figure 5 illustrates the power control using *Fixed-Step* for all three step sizes. Typically, Safe *Fixed-Step* operates at or below the set point. However, it does violate the power constraint once. This occurrence is attributed to the fact that the safety margin is calculated based on steady-state errors, which are averaged values. It is important to note that Safe *Fixed-Step* is generally infeasible in practice because obtaining accurate knowledge of the safety margin beforehand requires significant time spent measuring this margin. Nevertheless, we use Safe *Fixed-Step* as a baseline, as it achieves the best possible performance while still complying with the power constraint.

To overcome the limitations of the baselines mentioned above, we propose *CapGPU*, a MIMO power control methodology that coordinates CPU DVFS and the frequency level scaling of each GPU to optimize ML inference performance. For *CapGPU*, we utilize the delta-sigma modulation discussed in Section 5. We conduct the same experiments using the previously mentioned workloads. Figure 3 illustrates that our proposed solution can effectively converge to the set point without any violations. Furthermore, the proposed solution demonstrates better control capabilities and ensures that power can be maintained at its designated set point.

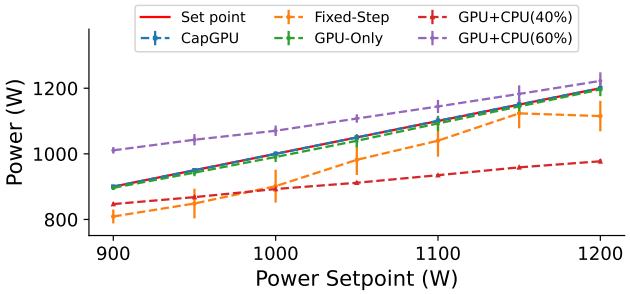


Figure 6: Power control comparison between GPU+CPU, *Fixed-Step*, and CapGPU across multiple power set points.

6.3 Comparison for Different set points

In this experiment, we evaluate our proposed solution alongside the baseline methods at various power set points to assess their control accuracy. To minimize the effects of any transient states that may occur at the beginning of each run, as discussed in the previous subsection, we calculate the average power and latency, after the system has reached a steady state. Specifically, we focus on the last 80 control periods out of a total of 100 periods for each run. We first test different power set points, ranging from 900 W to 1200 W, with a 50 W interval. *Fixed-Step* is excluded from comparison due to potential power violations, as previously discussed, so we opt for *Safe Fixed-Step* instead. Since the *CPU-Only* and *GPU+CPU* methods cannot reliably ensure system power convergence to the designated set point, these configurations are also excluded from further analysis.

Figure 6 illustrates power control between various set points for all solutions. The figure indicates that *Safe Fixed-Step* Power exhibits the worst control accuracy. This is primarily because it maintains a safety margin, ensuring that its highest power value remains below the set point by switching between two adjacent GPU frequency levels. Additionally, *Safe Fixed-Step* Power demonstrates the most significant oscillation and deviation.

Both *GPU+CPU* configurations (40% power on GPU and 60% power on GPU) failed to achieve convergence of system power

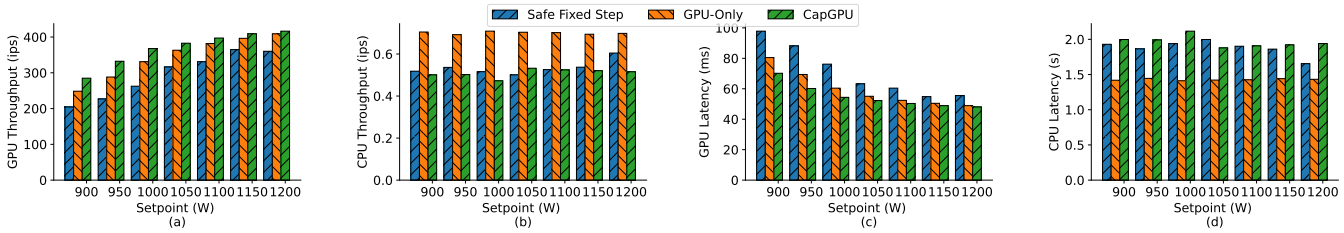
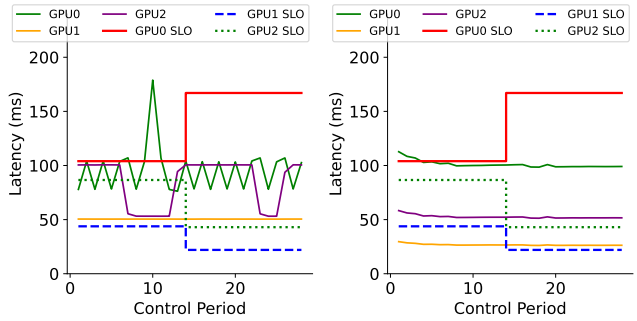


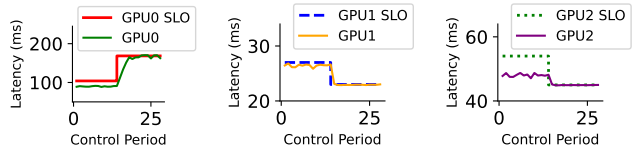
Figure 7: Comparison across control methods for: (a) GPU inference throughput, (b) CPU throughput, (c) GPU inference latency, and (d) CPU latency. In (a) and (c), CapGPU delivers the highest GPU inference throughput and the lowest latency.

to the set point, and the observed standard deviations indicate that this approach exhibits notable fluctuations. While *GPU-Only* shows relatively better performance than other baselines in power control, its control accuracy remains slightly lower than that of *CapGPU*. Additionally, the standard deviation indicates that *GPU-only* continues to exhibit fluctuations after converging to power set points. *CapGPU* outperforms all other baselines in both the stability and accuracy of power control.



(a) Safe Fixed-Step **(b) GPU-Only**

Figure 8: Illustration of inference latency and SLO requirements for each GPU: (a) Safe Fixed-Step, (b) GPU-Only. Neither method provides the capability to allocate computing resources according to SLO requirements.



(a) GPU0 **(b) GPU1** **(c) GPU2**

Figure 9: Illustration of inference latency under CapGPU alongside the SLO requirements for each GPU. CapGPU is capable of satisfying the SLOs for all tasks across the GPUs.

Next, we compare the performance of all the solutions. Figures 7(a) and 7(c) illustrate that the *CapGPU* has the lowest inference latency and highest throughput compared to *GPU-only* and *Safe Fixed-Step*. This advantage is due to the *CapGPU*'s ability to adjust the frequency based on the weight assignments discussed in Section 4. In addition, since the SLOs for the inference task are closely linked to the GPU's frequency, the controller effectively ensures that the SLOs are met. We also evaluate CPU latency for the preprocessing functions running in the cloud. The variation in CPU latency for all three techniques is shown in Figure 7(b) and Figure 7(d). The CPU latency is measured for each control period, with the inference time for every feature subset computed during the feature selection phase by recording the wall-clock time taken to perform cross-validation. Given that cloud platforms allow users to run additional CPU-bound tasks, as mentioned in Section 5, we specifically regulate the frequency of the cores dedicated to CPU tasks without adjusting the frequency for cores responsible for data copying for ML inference. It is also notable that in Figure 7(d), the latency for *CapGPU* is slightly higher than that for *GPU-Only*; however, this difference is acceptable since the preprocessing tasks do not operate under strict SLO agreements.

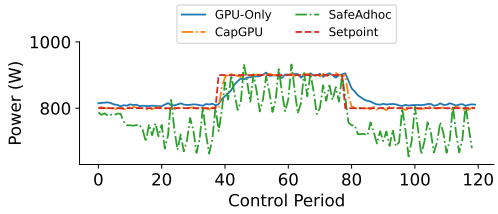


Figure 10: Online adaptation for changing power set points using 3 techniques.

6.4 Online Adaptability

In this section, we will test the online adaptability of *CapGPU* with changing power set points and changing SLOs. Firstly, we will test the adaptability for changing power set points.For this experiment, the system power is initially controlled to a fixed set point of 800W. During the 40th control period of the run, we simulate a scenario where there is a sudden surge in GPU inference requests. Since larger batches of images need to be processed, the power consumption of the servers slightly increases due to the greater computational demands placed on the GPUs. To ensure safe operation, the power budget of the data center is raised accordingly to 900W. At control period 80, when the burst is over, we change and reduce the power budget to 800W. Figure 10 shows that, while all the baselines can adapt to the changing set points, the *CapGPU* holds the least level of fluctuation while *GPU-only* has a long settling time.

Additionally, we show that *CapGPU* can adapt to changing SLOs through its SLO constraint mechanism. In many cloud-based ML inference services, SLO requirements often become more stringent during periods of high demand. For instance, during the recent Ghibli art event on ChatGPT [25], thousands of users submitted prompts simultaneously, expecting quick image generation. In such situations, cloud providers may impose stricter latency constraints to ensure responsiveness, especially during bursty real-time workloads such as image generation.

In this experiment, we demonstrate how the *CapGPU* controller meets SLOs and conduct a comparative study using *GPU-Only*, *Safe Fixed-step*, and *CapGPU* controllers. For this experiment, we chose three latency levels: 30% tail latency, 50% tail latency, and 80% tail latency. First, we calculate the 30%, 50%, and 80% tail latencies for three workloads and their corresponding GPU frequencies using Equation (8). We initially set the SLO for all workloads to the 50th percentile (50% tail) latency. At control period 14, we change the SLO requirements for workload on GPU 1 and GPU 2 to the 30th percentile (80% tail) latency, while changing the SLO of workload on GPU 0 to a more conservative level, 30% tail. We measure the inference latency and compute the deadline miss rate to evaluate how well each controller adapts to the changing SLO requirements. We set the power set point to 1000W to ensure that the system can meet all SLOs with appropriate frequency allocation. Figure 9 shows the variation in inference latency for the other two solutions.

Notably, *CapGPU* meets all SLO by explicitly incorporating SLO constraints. CapGPU also adapts effectively when the SLO for each workload on different devices varies. *GPU-only* uses a single shared frequency level across all GPUs, and Safe *Fixed-Step* controls only one frequency level per control period. In contrast, *CapGPU* adjusts the frequency for each device independently, enabling the system to support device-specific SLO requirements for all CPUs and GPUs.

7 Conclusions

Power capping is an effective way for data centers to avoid expensive power upgrade and reduce capital expenses. However, the existing power capping solutions focus on either the CPU or one GPU, in a separate manner, which cannot work well for ML inference workloads that demand for coordinated power capping. In this paper, we have presented *CapGPU*, a power capping framework for today's GPU servers that run ML workloads on multiple GPUs and a host CPU in each server. In sharp contrast to existing solutions that try to control the server power consumption by throttling either the CPU or one GPU, in a separate manner, CapGPU features a MIMO power control methodology and a novel weight assignment algorithm that dynamically adjusts the weights assigned to each CPU/GPU based on their measured throughput. Consequently, the overall ML inference performance can be optimized. Our hardware testbed results demonstrate that CapGPU outperforms several state-of-the-art power capping solutions by having more precise power control, higher inference throughput, and better inference latency guarantees.

Acknowledgments

This work was supported, in part, by the U.S. NSF under Grants CNS-2336886 and CNS-2344505. We would also like to thank the anonymous reviewers for their valuable comments.

References

[1] AMD. 2025. AMD ROCm SMI Power Control Documentation. https://rocm.docs.amd.com/projects/amdsmi/en/docs-5.6.1/doxygen/docBin/html/group__PowerCont.html. Accessed: 2025-04-04.
[2] Reza Azimi, Chao Jing, and Sherief Reda. 2018. PowerCoord: A coordinated power capping controller for multi-CPU/GPU servers. In *IGSC*. IEEE.
[3] Guoyu Chen, Srinivasan Subramaniyan, and Xiaorui Wang. 2024. Latency-Guaranteed Co-Location of Inference and Training for Reducing Data Center Expenses. In *ICDCS 2024*. IEEE.
[4] Guoyu Chen and Xiaorui Wang. 2022. Performance Optimization of Machine Learning Inference under Latency and Server Power Constraints. In *ICDCS*.
[5] Seungbeom Choi et al. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *USENIX ATC*.
[6] Amir Gholami, Sehoon Kim, et al. 2022. A survey of quantization methods for efficient neural network inference. In *Low-power computer vision*.
[7] Lixiang Han, Zimu Zhou, and Zhenjiang Li. 2024. Pantheon: Preemptible multi-dnn inference on mobile edge gpus. In *MOBISYS*.
[8] Trevor Hastie et al. 2001. *The elements of statistical learning data mining, Inference, and prediction*. Springer.
[9] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. 2004. *Feedback control of computing systems*. John Wiley & Sons.
[10] Bob Keaveney. 2024. A Data Center Building Boom Is About to Begin. https://biztechmagazine.com/article/2024/04/data-center-building-boom-about-begin.
[11] Osama Khan, Junyeol Yu, Yeonjae Kim, and Euiseong Seo. 2024. Efficient Adaptive Batching of DNN Inference Services for Improved Latency. In *ICOIN*.
[12] Bill Kleyman. 2024. AFCOM 2024 State of the Data Center Report. https://itchronicles.com/wp-content/uploads/2024/02/Data-Ctr-Report.pdf.
[13] Alok Gautam Kumbhare, Reza Azimi, Ioannis Manousakis, et al. 2021. Prediction-Based Power Oversubscription in Cloud Platforms. In *USENIX ATC*.
[14] Charles Lefurgy, Xiaorui Wang, et al. 2007. Server-level power control. In *ICAC*.
[15] Shaohong Li et al. 2020. Thunderbolt:Throughput-Optimized Quality-of-Service-Aware power capping at scale. In *OSDI*.
[16] Yang Li, Charles R. Lefurgy, Karthick Rajamani, et al. 2019. A Scalable Priority-Aware Approach to Managing Data Center Server Power. In *HPCA*.
[17] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, et al. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *IEEE CVF*.
[18] Kai Ma, Xue Li, Wei Chen, et al. 2012. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In *ICPP*. IEEE.
[19] Kai Ma and Xiaorui Wang. 2012. PGCapping: Exploiting power gating for power capping and core lifetime balancing in CMPs. In *PACT*.
[20] Seyed Morteza Nabavinejad, Sherief Reda, et al. 2022. Coordinated batching and DVFS for DNN inference on GPU accelerators. *TPDS* (2022).
[21] NVIDIA. 2025. NVIDIA System Management Interface (NVIDIA-SMI) Documentation. https://docs.nvidia.com/deploy/nvidia-smi/. Accessed: 2025-04-04.
[22] Pratyush Patel, Esha Choukse, Chaojie Zhang, et al. 2024. Characterizing power management opportunities for llms in the cloud. In *ASPLOS*.
[23] Leonardo Piga, Iyswarya Narayanan, et al. 2024. Expanding Datacenter Capacity with DVFS Boosting: A safe and scalable deployment experience. In *ASPLOS*.
[24] Varun Sakalkar et al. 2020. Data Center Power Oversubscription with a Medium Voltage Power Plane and Priority-Aware Capping. In *ASPLOS*.
[25] Rachel Shin. 2025. Sam Altman says GPTs are 'melting GPUs' as users flood ChatGPT with Studio Ghibli-style AI images. https://fortune.com/2025/03/28/sam-altman-chatgpt-gpus-melting-ai-images/ Accessed: 2025-04-16.
[26] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. https://arxiv.org/abs/1409.1556
[27] UEFI Forum. 2021. Advanced Configuration and Power Interface (ACPI) Specification Version 6.4. https://uefi.org/specs/acpi/6.4/. Accessed: 2025-04-06.
[28] Xiaorui Wang and Ming Chen. 2008. Cluster-level feedback power control for performance optimization. In *HPCA*. IEEE.
[29] Xiaorui Wang, Ming Chen, Charles Lefurgy, and Tom W. Keller. 2011. Ship: A scalable hierarchical power control architecture for large-scale data centers. *IEEE Transactions on Parallel and Distributed Systems* 23, 1 (2011), 168–176.
[30] Xiaorui Wang, Yingming Chen, Chenyang Lu, and Xenofon Koutsoukos. 2007. FC-ORB: A robust distributed real-time embedded middleware with end-to-end utilization control. *Journal of Systems and Software* 80, 7 (2007), 938–950.
[31] Yue Wang et al. 2018. Energynet: Energy-efficient dynamic inference. (2018).
[32] Yefu Wang, Kai Ma, and Xiaorui Wang. 2009. Temperature-Constrained Power Control for Chip Multiprocessors with Online Model Estimation. In *ISCA*. IEEE.
[33] Qizhen Weng, Lingyun Yang, et al. 2023. Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent. In *USENIX ATC*.
[34] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, et al. 2016. Dynamo: Facebook's Data Center-Wide Power Management System. In *ISCA*.
[35] Jiaying Zhang et al. 2025. Power Control for Edge ML Inference with Hypernetwork Meta-Parameters. *IEEE Communications Letters* (2025).