Minimizing Power Waste in Heterogenous Computing via Adaptive Uncore Scaling

Zhong Zheng
University of Illinois Chicago
Chicago, USA
zzheng33@uic.edu

Seyfal Sultanov
University of Illinois Chicago
Chicago, USA
ssulta24@uic.edu

Michael Papka
Argonne National Laboratory (ANL)
Lemont, USA
University of Illinois Chicago
Lemont, USA
papka@anl.gov

Zhiling Lan
University of Illinois Chicago
Chicago, USA
Argonne National Laboratory (ANL)
Chicago, USA
zlan@uic.edu

Abstract

High-performance computing (HPC) systems are essential for scientific discovery and engineering innovation. However, their growing power demands pose significant challenges, particularly as systems scale to the exascale level. Prior uncore frequency tuning studies have primarily focused on conventional HPC workloads running on CPU-only systems. As HPC advances toward heterogeneous computing, integrating diverse GPU workloads on heterogeneous CPU-GPU systems, it becomes imperative to revisit and enhance uncore scaling. Our investigation reveals that uncore frequency scales down only when CPU power approaches its thermal design power (TDP), which is rare in GPU-dominant applications. As a result, modern computing systems experience unnecessary power waste. In this study, we present MAGUS, a user-transparent uncore frequency scaling runtime for heterogeneous computing. MAGUS dynamically adjusts uncore frequencies according to distinct application execution phases, effectively minimizing power waste caused by consistently using maximum uncore frequencies. Our design incorporates several key techniques, including real-time monitoring and prediction of memory accesses, intelligent handling of frequent phase transitions, and leveraging vendor-provided power management features. We evaluate MAGUS with various GPU benchmarks and applications on multiple heterogeneous systems with different CPU and GPU architectures. Experimental results demonstrate that MAGUS achieves up to 27% energy savings compared to the default settings, while maintaining a performance loss of less than 5% and an overhead of under 1%.

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759879

Keywords

GPU workloads, heterogeneous CPU-GPU systems, uncore frequency scaling, energy efficiency, performance-power trade-offs

ACM Reference Format:

Zhong Zheng, Seyfal Sultanov, Michael Papka, and Zhiling Lan. 2025. Minimizing Power Waste in Heterogenous Computing via Adaptive Uncore Scaling. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3712285.3759879

1 Introduction

High-performance computing (HPC) plays a vital role in advancing numerous research fields by utilizing extensive high-end computational, memory, storage, and network resources to solve complex problems. However, these powerful capabilities often lead to significant energy consumption. As HPC systems have scaled toward the exascale level in recent years, their energy consumption has become a critical concern. For example, the recently deployed Aurora supercomputer at Argonne National Laboratory achieves a sustained performance of 1.01 exaFLOPS while consuming approximately 38.7 MW of power [42]. The growing power demands highlight the urgent need for energy-efficient computing to conserve energy while maintaining performance. Our work aims to address this need. Here, *energy efficiency* is defined as minimizing the total energy-to-solution, that is, the total energy required to complete an application.

Various efforts have investigated energy efficiency using techniques such as dynamic voltage and frequency scaling (DVFS), frequency scaling, and power capping [10, 22, 23, 57, 62, 63]. A typical CPU consists of *core* and *uncore* components. The core encompasses the CPU cores, while the uncore includes the Last Level Cache (LLC), Memory Controller (MC), and Quick Path Interconnect (QPI) [3, 7, 30]. Several studies show that the uncore contributes significantly to overall power consumption, averaging 5-15% of total CPU power, depending on processor architecture and workload characteristics [14, 29]. Recognizing the potential for

energy savings through uncore frequency scaling (UFS), several studies have explored various methods to achieve this goal. Broadly, these methods can be classified into model-based and model-free approaches. Model-based methods typically use multiple hardware counters to construct analytical or machine-learning models that guide uncore tuning decisions [60, 68]. In contrast, model-free methods rely on real-time feedback data for decision-making [5, 24, 26]. UPScavenger [24] is a pioneering model-free method that dynamically adjusts uncore frequency by detecting phase transitions between compute-intensive and memory-intensive regions. It monitors DRAM power and Instructions Per Cycle (IPC) without requiring complex modeling.

We identify several *limitations* in existing studies. First, some methods detect phase transitions between compute-intensive and memory-intensive phases by tracking multiple hardware counters, which can potentially introduce considerable overhead. Others rely on extensive offline profiling, complex model construction, or manual modifications to user code. Additionally, few studies have investigated the impact of uncore frequency scaling specifically on GPU workloads (e.g., emerging AI-enabled applications), as prior efforts have focused mainly on traditional HPC workloads in homogeneous CPU systems. As HPC environments increasingly adopt heterogeneous CPU-GPU architectures and execute hybrid workloads, particularly emerging applications that primarily rely on GPUs, it becomes essential to address these gaps.

In this work, we investigate whether the uncore frequency is dynamically adjusted when executing GPU workloads or AI-enabled scientific applications. While vendor-supplied dynamic uncore scaling solutions may exist [6], our study reveals that uncore frequency is only adjusted when CPU power usage approaches the thermal design power (TDP). In *GPU-dominant applications*, applications relying primarily on GPUs rather than CPU cores, CPU utilization rarely reaches TDP. Consequently, the uncore frequency remains at its maximum, leading to significant power waste. The primary reason is that GPU workloads typically do not require extensive computational tasks to be performed on the CPU. We further conduct a case study to examine the impact of uncore scaling on application performance and energy consumption. Our findings highlight the need for uncore frequency scaling to reduce power waste, hence optimizing energy efficiency in heterogeneous computing.

We present *MAGUS*, a model-free, lightweight, and user-transparent runtime for automated uncore frequency tuning in heterogeneous CPU-GPU systems. Efficient uncore frequency tuning is inherently challenging. It requires dynamic detection of application phases and adaptability to diverse workload behaviors, all while ensuring minimal runtime overhead. The design of MAGUS incorporates two primary techniques. First, it provides a lightweight yet effective method for quickly detecting execution phases that affect uncore utilization. Instead of relying on multiple hardware counters, MAGUS uses a single metric, memory throughput, to minimize runtime overhead. Second, inspired by the prior work [18], we introduce the concept of *memory dynamics*, defined by (a) the first derivative of memory throughput and (b) the frequency of memory throughput changes. These features enable MAGUS to predict near-future memory throughput trends and identify frequency memory throughput fluctuations, thereby more effectively guiding uncore frequency scaling decisions. MAGUS is designed as a generic framework, making it applicable to a wide range of GPU workloads and architectures.

We evaluate MAGUS using a suite of representative GPU-dominant benchmarks and applications, including the widely used GPU benchmarks from Altis [32, 66], ECP proxy applications [1], two real-world HPC applications (LAMMPS and GROMACS), and three ML training workloads (UNet, ResNet50, and BERT) from the MLPerf benchmark [21]. MAGUS is compared against two baselines: Intel's default uncore frequency setting and UPS [24], a state-of-the-art uncore frequency scaling method. Our experiments examine application performance, power and energy savings, and runtime overhead in both single- and multi-GPU configurations. Results demonstrate that MAGUS achieves up to 27% energy savings compared to the baseline approaches, maintains performance degradation below 5%, and introduces minimal overhead (less than 1%). Overall, our work offers the following key contributions:

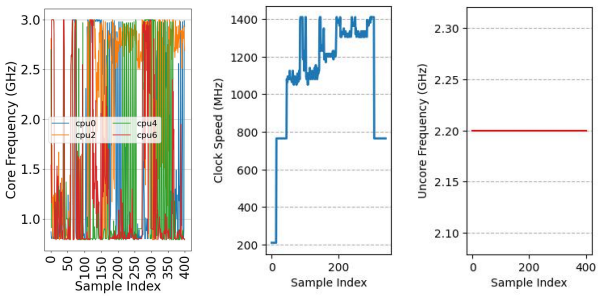
- We analyze the effects of uncore frequency scaling on GPU-dominant workloads in heterogeneous systems, examining power consumption and execution time to demonstrate the necessity and energy-saving potential of our approach (§2).
- We develop a model-free, lightweight, user-transparent runtime for uncore frequency scaling, leveraging memory dynamics to predict near-future memory throughput trends and detect frequent fluctuations (§3).
- We extensively evaluate MAGUS on various heterogeneous systems across various HPC benchmarks and applications. Our results indicate that MAGUS outperforms existing methods by achieving substantial energy savings (e.g., up to 27%) while maintaining minimal performance loss (§5-6).

2 Motivation and Challenges

In modern processors, the CPU core frequency and GPU streaming multiprocessor (SM) clock speed are dynamically adjusted by the hardware to adapt to workload intensity, thereby avoiding unnecessary power waste. However, uncore frequency is dynamically tuned *only when CPU power approaches the thermal design power (TDP) limit* [5].

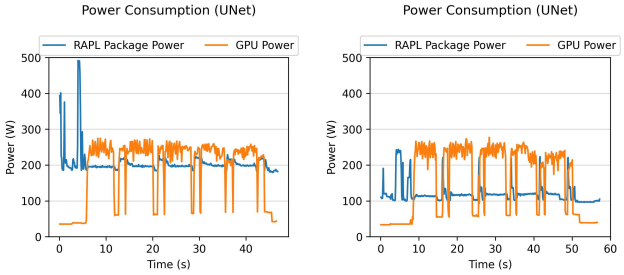
To validate this behavior, we analyze several applications on a Chameleon system equipped with an Intel Xeon Platinum 8380 CPU (uncore ranges from 0.8 to 2.2 GHz) and an NVIDIA A100 40GB GPU [37]. We focus on GPU-dominant workloads. Figure 1a-1c presents a case study of UNet training, a convolutional neural network for image segmentation [59]. As depicted, CPU core frequency and GPU SM clock speed are tuned dynamically by default hardware settings based on workload demands; however, the uncore frequency consistently remains at its maximum.

We observe the same phenomenon on other Intel systems, including Intel Xeon Platinum and Intel Xeon Max processors. According to [5], the uncore frequency is reduced only when the CPU's power approaches its thermal design power (TDP). In practice, CPU package power rarely approaches TDP when running GPU-dominant applications, as these workloads are typically not as CPU-intensive as traditional CPU-only HPC applications.



(a) CPU freq. (b) GPU clock speed (c) Uncore freq.

Figure 1: UNet profiling on a heterogeneous Intel Xeon CPU–A100 GPU node. CPU core frequency and GPU clock speed are dynamically adjusted by default; however, the uncore frequency remains at its maximum. Each socket contains 40 hardware cores; for readability, we plot the core frequency of only four of these cores in (a). Sampling at (c) occurs at 0.5-second intervals, with each sample index representing one monitoring event.



(a) Max Uncore Freq. (2.2 GHz) (b) Min Uncore Freq. (0.8 GHz)

Figure 2: Power profiles of UNet training under different uncore frequencies: max (2.2 GHz) versus min (0.8 GHz). Reducing the uncore frequency results in (i) an 82-watt reduction in CPU power, from 200 watts (blue curve on the left) to 120 watts (blue curve on the right), and (ii) an increase in runtime, from 47 seconds (left) to 57 seconds (right).

Next, we investigate *two important questions*: (i) how significant is uncore power consumption in contemporary processors, and (ii) how does uncore frequency scaling affect application performance? Figure 2 presents our experimental results, showing power profiles during UNet training under different uncore frequency settings. First, tuning the uncore frequency from its maximum (2.2 GHz) to its minimum (0.8 GHz) resulted in a CPU power reduction of up to 82 W, indicating that the uncore subsystem can account for *as much as 40%* of total CPU power consumption during UNet training. Similar behavior was consistently observed across other GPU applications examined in this study. Second, this aggressive uncore scaling increases application runtime by 21%, indicating *a key trade-off* between power savings and performance. While lowering the uncore frequency reduces power consumption, setting it to the minimum without considering workload demands can significantly impact performance, especially for memory-intensive tasks. These observations suggest we need a dynamic approach to scale the uncore frequency based on workload demands.

Previous work, such as the UPS method by Gholkar et al. [24], dynamically scales the uncore frequency based on changes in DRAM power and IPC. However, it mainly targets traditional CPU-only applications and systems. Additionally, UPS relies on active monitoring of IPC, which requires reading instructions retired and CPU cycles through MSRs (Model-Specific Registers) for each core, introducing considerable runtime overhead (details in §6.5).

These gaps motivate the design of MAGUS, a lightweight uncore frequency scaling runtime specifically designed for emerging heterogeneous computing environments. Uncore tuning is a complex process that requires dynamic phase detection and careful frequency adjustments to balance energy savings and performance.

Specifically, we identify the following key challenges:

(1) *Heterogeneity in GPU workloads.* GPU workloads frequently alternate between memory accesses and GPU computations at fine-grained intervals. They also differ significantly in their memory access and computation patterns, especially when comparing scientific simulations with machine learning models. Developing a uniform phase detection mechanism that effectively adapts to such diverse behaviors is difficult, requiring tailored strategies to accommodate the unique behaviors of each application.

(2) *Selection of uncore metrics.* Modern processors provide multiple uncore metrics, such as IPC, DRAM power, CPI, flops, and LLC_misses, that can guide uncore frequency scaling. However, monitoring numerous hardware counters can introduce substantial runtime overhead, particularly in systems with large core counts, where accessing Model-Specific Registers (MSRs) at scale becomes resource-intensive. For instance, active IPC monitoring requires accessing Model-Specific Registers (MSRs) to read instructions retired and CPU cycles for each core. This process becomes increasingly resource-intensive as the number of CPU cores increases. Identifying a minimal yet sufficiently informative metric is crucial for achieving low-overhead data collection and making effective tuning decisions.

(3) *Frequent phase changes.* In GPU workloads, the transition between compute- and memory-intensive phases can occur at millisecond timescales [43]. Capturing these rapid transitions and responding to them without adding significant overhead is challenging.

3 MAGUS Design

MAGUS leverages Intel's Performance Counter Monitor (PCM) API [34] for monitoring system metrics and Model-Specific Registers (MSRs) for hardware control. While currently implemented using these Intel-specific interfaces, MAGUS's design principles can be adapted to other processors, such as AMD and ARM, provided they offer interfaces for reading memory throughput data and enabling uncore frequency scaling.

To address the challenges listed in §2, we adopt *memory throughput* as our primary metric because it directly reflects the activity intensity of uncore components, making it a reliable and effective

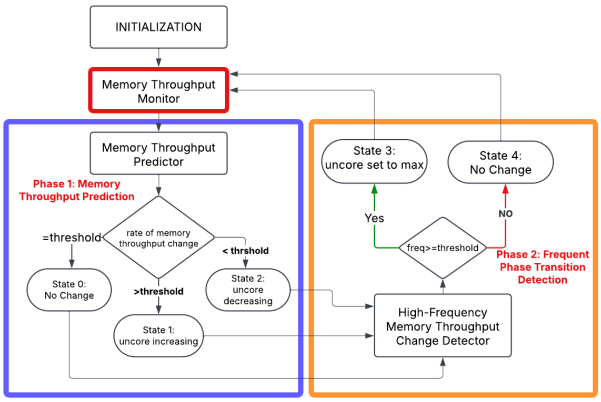


Figure 3: MAGUS Overview. MAGUS comprises three main components: (1) Memory Throughput Monitor, (2) Memory Throughput Predictor, and (3) High-Frequency Memory Throughput Changing Detector, each being highlighted in a different color.

indicator. Monitoring memory throughput over time provides critical insights for dynamically adjusting uncore frequency, thereby improving energy efficiency. Furthermore, unlike reading IPC data via MSRs from each CPU core, obtaining memory throughput data through Intel's PCM API introduces significantly lower runtime overhead, thereby improving the overall efficiency of the process.

To overcome the significant hurdles identified earlier, MAGUS operates through two key phases, *memory throughput prediction* and *frequent phase transition detection*. First, MAGUS employs the concept of *memory dynamics* rather than merely detecting phase transitions between memory-intensive and compute-intensive phases. Specifically, MAGUS utilizes the first derivative of memory throughput data to predict near-future trends, as illustrated in Algorithm 1. This simple yet effective approach allows the runtime to adapt to a broad spectrum of workload behaviors without the overhead of multiple hardware counters. Second, MAGUS incorporates a lightweight algorithm, as listed in Algorithm 2, to automatically detect high-frequency changes in memory throughput. This ensures timely adjustments to uncore frequency and addresses the challenge of high-frequency phase changes. Figure 3 presents the MAGUS flowchart, and the following subsections describe each phase in detail.

3.1 Phase 1: Memory Throughput Prediction

Memory throughput can fluctuate dramatically, either increasing or decreasing sharply. These fluctuations indicate that while applications periodically require maximum uncore frequency for optimal performance, maintaining such a high frequency is often unnecessary during less intensive periods. Anticipating these shifts enables adaptive adjustments to uncore frequency, allowing for a match with upcoming memory throughput demands. Offline performance profiling methods are impractical, as each application exhibits unique memory access patterns, and previously unseen applications introduce further variability. To address this, we introduce the concept of *memory dynamics*, which includes both *the first derivative* and *the frequency of changes* in memory throughput. Specifically, we leverage the first derivative of memory throughput over short intervals to anticipate near-future demands. This predictive approach enables timely and efficient uncore frequency scaling, ensuring responsiveness to workload variations while maintaining energy efficiency. The frequency component will be discussed in Phase 2.

The detailed procedure is outlined in Algorithm 1. We maintain a fixed-size first-in, first-out queue to record memory throughput history. The function MEM_THT_TREND_PREDICTION is invoked periodically to compute the first derivative of memory throughput over a specified interval. If the first derivative exceeds the inc_threshold, it indicates that memory throughput is likely to increase sharply in the near future. Conversely, if the derivative falls below the dec_threshold, it suggests that memory throughput demands are expected to decrease significantly. The function returns 1 or -1 to signal the runtime to either increase or decrease the uncore frequency accordingly. If the derivative falls between these thresholds, the uncore frequency remains unchanged, ensuring stability and avoiding unnecessary adjustments. This adaptive mechanism enables timely and predictive adjustments to uncore frequency, balancing power usage with application performance.

The uncore frequency scaling decision generated by the prediction phase is initially recorded as a temporary decision and is not executed immediately. Depending on the outcome of the frequent phase change detection phase (Section 3.2), this decision may be overwritten or left unchanged.

Algorithm 1: Memory Throughput Trend Prediction

Input: window length L, increase threshold τ_{inc}, decrease threshold τ_{dec}, list *mem_throughput_ls*
Let $n \leftarrow |mem\_throughput\_ls| - 1$
Let $d \leftarrow \dfrac{mem\_throughput\_ls[n] - mem\_throughput\_ls[0]}{L}$
if $d > \tau_{\text{inc}}$ **then**
  return 1
else if $d < \tau_{\text{dec}}$ **then**
  return -1
else
  return 0
end if

Algorithm 2: High Frequency Detection

Input: threshold t_{hi}, binary list *uncore_tune_ls*
$n \leftarrow |uncore\_tune\_ls|$
$s \leftarrow \sum_{i=1}^{n} uncore\_tune\_ls[i]$
$f \leftarrow s \,/\, n$
if $f \geq t_{\text{hi}}$ **then**
  return TRUE
else
  return FALSE
end if

3.2 Phase 2: Frequent Phase Change Detection

Workloads can experience fluctuating and dramatic changes in memory throughput over short time intervals, leading to frequent uncore scaling. Frequent uncore scaling in this scenario can degrade application performance for two key reasons. First, frequent uncore scaling incurs excessive MSR accesses, adding overhead. Second, when memory throughput fluctuates rapidly and substantially, neither software nor hardware can fully adapt in real-time, limiting the system's ability to meet instantaneous throughput demands.

To address frequent changes in memory throughput, we developed a simple yet effective detection algorithm (Algorithm 2) that identifies periods of high-frequency fluctuations. During these periods, MAGUS maintains the maximum uncore frequency to ensure consistent access to maximum memory bandwidth, preventing performance degradation from constant frequency adjustments. We maintain a first-in-first-out queue called *uncore_tune_ls*, which uses a binary flag (0 or 1) to record whether a potential uncore frequency scaling event should occur based on the uncore frequency scaling decision made by the prediction phase (see Algorithm 1). If the rate of triggered UFS events (either an increase or decrease in uncore frequency) exceeds a threshold, it indicates that memory throughput is fluctuating frequently, which we classify as a high-frequency status. When this status is detected, MAGUS overrides the temporary decision from the prediction phase and sets the uncore frequency to its maximum to stabilize performance. Otherwise, MAGUS retains and executes the decision made in the prediction phase. Even if the application remains in a high-frequency state, MAGUS continues the prediction phase in each decision round to estimate memory throughput and log potential uncore scaling events. While not executed during high-frequency states, these logs inform future high-frequency detection.

In short, this detection algorithm ensures that applications experiencing frequent memory throughput fluctuations consistently receive the maximum memory bandwidth, mitigating performance degradation during periods of high variability.

3.3 Putting It All Together

Algorithm 3 outlines the core logic of MAGUS. MAGUS operates transparently once the user application starts, with the runtime periodically reading memory throughput. Initially, uncore frequency is set to the maximum supported by the CPU, ensuring peak uncore performance during periods of rapidly increasing or frequently changing memory throughput. The MDFS function is invoked after 2.0 seconds (i.e., 10 monitoring cycles) from the start of the application. Memory throughput values are collected and appended to the *mem_throughput_ls* list during this initialization period. Meanwhile, *uncore_tune_ls* is initialized as a list of 10 zeros, indicating that no uncore tuning actions occur during this initial phase.

The high-frequency detector evaluates whether the application experiences frequent uncore frequency changes by calculating the rate of frequency scaling events. If frequent changes are detected, MAGUS maintains the uncore frequency at its maximum, overriding any temporary decisions made during the prediction phase. Once the application no longer exhibits frequent frequency changes, the detection phase approves and executes the temporary decision made in the prediction phase. This detection mechanism ensures the maximum memory bandwidth availability during high-frequency phases, thus mitigating performance loss from frequent uncore scaling.

MAGUS incorporates three thresholds for detecting and predicting memory dynamics. Through extensive testing, these thresholds consistently demonstrate effectiveness across various workloads and hardware platforms. Consequently, we recommend default threshold values: *inc_threshold* to 200, *dec_threshold* to 500, and *high_frequency_threshold* to 0.4, with a monitoring frequency of 0.2 seconds. All tested systems use the same thresholds mentioned above. A detailed sensitivity analysis of these thresholds is presented in §6.4.

Algorithm 3: Memory Throughput Based Dynamic Uncore Frequency Scaling (MDFS)

```
1:  global inc_threshold, dec_threshold, uncore_tune_ls,
    high_freq_threshold, direv_length, high_freq_status,
    mem_throughput_ls, tune_uncore_freq
2:
3:  uncore_freq_upper ← max
4:  uncore_freq_lower ← min
5:
6:  mem_throughput_ls.push_back(system_mem_throughput)
7:  mem_throughput_ls.erase(mem_throughput_ls.begin())
8:
9:  if high_freq_detection(high_freq_threshold, uncore_tune_ls)
    then
10:   high_freq_status ← 1
11:   uncore_freq ← uncore_freq_upper
12:   tune_uncore_freq(uncore_freq)
13: else
14:   high_freq_status ← 0
15: end if
16: if mem_throughput_trend_prediction(inc_threshold,
    dec_threshold, mem_throughput_ls, direv_length) == 1
    then
17:   uncore_tune_ls.push_back(1)
18:   if high_freq_status = 0 then
19:     uncore_freq ← uncore_freq_upper
20:     tune_uncore_freq(uncore_freq)
21:   end if
22: else if mem_throughput_trend_prediction(inc_threshold,
    dec_threshold, mem_throughput_ls, direv_length) == -1
    then
23:   uncore_tune_ls.push_back(1)
24:   if high_freq_status = 0 then
25:     uncore_freq ← uncore_freq_lower
26:     tune_uncore_freq(uncore_freq)
27:   end if
28: else
29:   uncore_tune_ls.push_back(0)
30: end if
31: uncore_tune_ls.erase(uncore_tune_ls.begin())
```

4 MAGUS Implementation

We implement MAGUS in C++ with approximately 400 lines of code. MAGUS is a user-transparent runtime that requires elevated privileges only once during installation and launch, which is handled by the system administrator. Afterward, it runs as a background process and operates transparently to users, requiring no user intervention or elevated privileges. The code is available as open-source on GitHub [1].

The default uncore frequencies of compute nodes are set to their minimum values to conserve power when the nodes are idle. Upon the arrival of an application, MAGUS periodically monitors memory throughput using Intel's PCM API [34] and dynamically adjusts uncore frequency based on real-time workload analysis.

To adjust the uncore frequency, MAGUS modifies the maximum frequency bits of the Model-Specific Register (MSR) located at 0x620 for Intel processors, while leaving the minimum frequency bits unchanged. For example, to set the maximum uncore frequency to 1.5 GHz on socket 0, the following command can be executed: `sudo wrmsr -p 0 0x620 0x0F001200`. Since MSR writes are direct register modifications at the hardware level, they incur negligible computational cost.

5 Experimental Configuration

We select a representative suite of GPU-dominant applications for our evaluation. (1) **GPU benchmark suite Altis:** This suite includes fundamental parallel algorithms widely used in parallel computing and real-world applications [32]. Specifically, we utilize 14 benchmarks from Level 1 and Level 2, excluding Level 0 benchmarks due to their short execution times. (2) **ECP proxy applications:** The miniGAN is a GAN-based benchmark for deep learning in HPC [1]. CRADL [47] integrates adaptive learning with computational science for surrogate modeling. Laghos [48] is a high-order Lagrangian hydrodynamics solver for gas dynamics. SW4lite [49] is a lightweight seismic wave propagation solver for earthquake simulations. (3) **AI-enabled applications:** We employ two widely-used open-source molecular dynamics (MD) simulation packages, GROMACS [61] and LAMMPS [9]. (4) **MLPerf benchmarks:** Due to the growing adoption of deep neural network training in HPC, we also include ResNet50, UNet, and BERT from the MLPerf benchmark suite [21] to represent deep learning training workloads.

Three heterogeneous systems are used in our experiments:

- **Intel+A100:** A Chameleon Cloud [38] system featuring two Intel(R) Xeon(R) Platinum 8380 processors paired with a single NVIDIA A100-40GB GPU. The system supports uncore frequencies from 0.8 GHz to 2.2 GHz and runs Ubuntu 22.04 with CUDA 12.6.
- **Intel+4A100:** It has the same architecture and software environment as the first, except it is equipped with four NVIDIA A100-80GB GPUs interconnected via PCIe.
- **Intel+Max1550:** It features the Intel(R) Xeon(R) CPU Max 9462, a Sapphire Rapids architecture processor comprising eight compute tiles Intel(R) Data Center GPU Max 1550 based on the Ponte Vecchio architecture and featuring 128 GB of HBM2e memory. The system supports an uncore frequency range of 0.8 GHz to 2.5 GHz and operates on Ubuntu 24.04.1 LTS with oneAPI DPC++/C++ Compiler 2024.2.1. It is the base unit deployed in the exascale system Aurora [42].

For Intel+A100 and Intel+4A100 systems, the applications utilize CUDA for GPU computation. In contrast, for the Intel+Max1550 system, we utilize 11 applications from the Altis-SYCL benchmark suite [66], excluding those that cannot be compiled. The other applications mentioned above currently do not have SYCL versions available.

We compare MAGUS against two existing methods: **Default uncore frequency scaling (baseline)** and **Uncore Power Scavenger (UPS)**. With the default settings of the Intel Xeon Platinum 8380 and Intel Xeon CPU Max 9462 processors, the uncore frequency is reduced only when the CPU power (package and DRAM) approaches the Thermal Design Power (TDP). UPS is a model-free runtime method that dynamically adjusts the uncore frequency based on changes in DRAM power and instructions per cycle (IPC) for HPC applications[24]. Since an open-source UPS implementation was unavailable, we implemented UPS following the methodology described in the original paper. We evaluate each method using three key metrics:

- **Performance Loss.** Defined as the percentage increase in execution time compared to the baseline, capturing the performance impact of uncore frequency scaling.
- **Power Saving.** Defined as the average reduction in CPU package and DRAM power consumption relative to the baseline. The CPU package encompasses the entire CPU socket, including core and uncore components.
- **Energy Saving:** Defined as the reduction in energy consumption, including CPU package, DRAM, and GPU board energy, compared to the baseline. GPU board includes GPU cores, GPU memory, and GPU onboard components (VRM, fans, PCIe logic). The primary goal of MAGUS is to minimize the total energy required to complete an application.

We employ Intel's RAPL [17] to measure CPU package and DRAM power consumption, and leverage NVIDIA's NVML [53] and Intel oneAPI [33] for monitoring power usage on NVIDIA and Intel GPUs, respectively. Notably, our approach is not limited to RAPL, NVML, or oneAPI and is compatible with any platform that offers equivalent power monitoring capabilities.

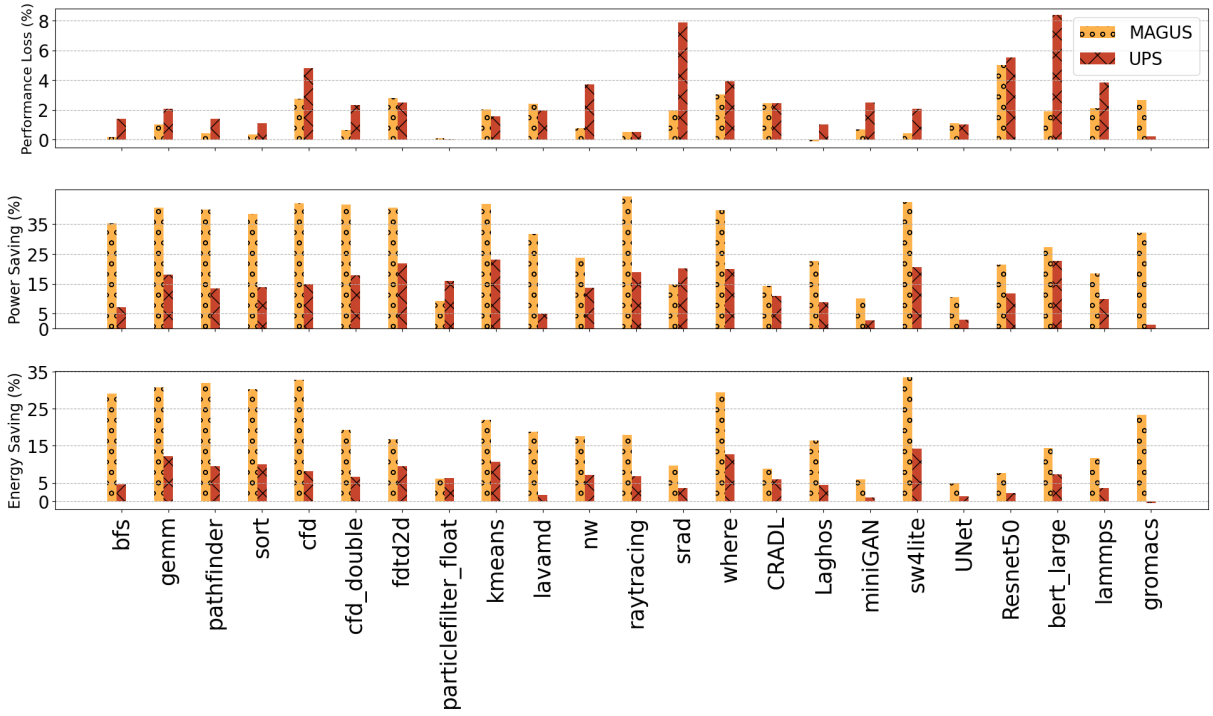
6 Results

Our experiments evaluate uncore frequency scaling across single-GPU workloads on Intel+A100 and Intel+MAX1550, as well as multi-GPU workloads on Intel+4A100 systems. Each experiment was repeated at least five times to account for performance variance and outliers when running applications on real systems. Outliers were removed, and the average of the remaining results was calculated to ensure reliability.

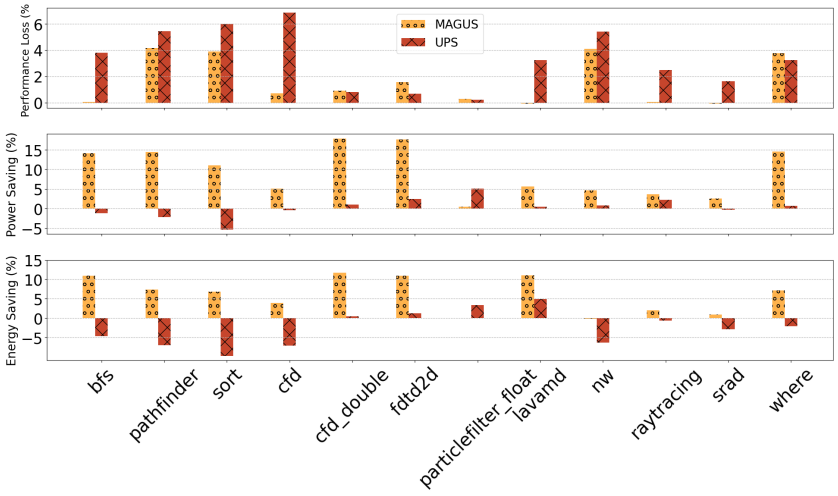
Our evaluation aims to answer the following key questions:

- Q1: what is the impact of uncore frequency scaling on application performance and energy consumption (§6.1)?

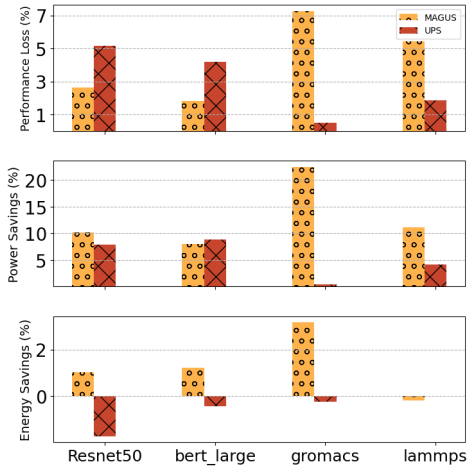
[1] https://github.com/SPEAR-UIC/MAGUS/tree/main



(a) Overall performance on Intel+A100.



(b) Overall performance on Intel+MAX1550.



(c) Overall performance on Intel+4A100.

Figure 4: Application performance on Intel+A100, Intel+MAX1550, and Intel+4A100. The X-axis lists the benchmarks and applications, while the Y-axis shows the corresponding metrics achieved by MAGUS and UPS compared to the baseline.

- Q2: how do different uncore frequency scaling methods perform under highly dynamic memory conditions (§6.2)?
- Q3: how accurately does MAGUS predict memory usage trends (§6.3)?
- Q4: how sensitive is MAGUS to the threshold values used in its uncore frequency scaling decisions (§6.4)?
- Q5: how much overhead introduced by MAGUS (§6.5)?

6.1 End-to-End Performance

Intel+A100. Figure 4a compares different uncore frequency scaling methods against the baseline in terms of performance loss, power savings, and energy savings on the Intel+A100 system. As shown in the top plot of Figure 4a, MAGUS consistently limits performance loss to below 5%. In the default uncore frequency scaling setting, as described in the previous section, the uncore frequency

is reduced only when the CPU package power approaches the thermal design power (TDP). However, CPU package power rarely reaches TDP during GPU-enabled applications, as these workloads offload most of the computational tasks to the GPU, resulting in low CPU utilization. Despite this, GPU-enabled workloads still require high uncore frequency during periods of data movement and memory control operations. MAGUS continuously monitors memory throughput and dynamically adjusts the uncore frequency to minimize uncore power consumption. We observe that specific applications (including but not limited to BFS, GEMM, and Pathfinder) achieve higher CPU package power savings than others (i.e., particlefilter_naive, srad). This is because less memory-intensive applications spend less time in high uncore frequency states, allowing for more frequent uncore downscaling and greater energy efficiency. Reducing instantaneous power consumption helps prevent the aggregate power consumption of all applications from exceeding the system's total power budget if one is in place. The energy savings in our experiments include both CPU energy consumption and GPU power consumption. With MAGUS, all workloads achieve positive energy savings compared to the baseline, with savings of up to 27%.

Intel+Max1550. In Figure 4b, we make several observations regarding the Intel+Max1550 system. First, MAGUS maintains a performance loss of below 4% while achieving up to 10% energy savings, outperforming both the baseline and UPS. Second, for applications like fdtd2d, where MAGUS results in a higher performance loss than UPS, it achieves significantly greater energy savings of up to 10%. This is because MAGUS applies a more aggressive uncore frequency tuning, reducing the frequency directly to the lower bound instead of gradually decreasing it. Third, UPS leads to negative energy savings for some applications because it introduces a 7.9% increase in power consumption. Thus, UPS's power savings are outweighed by the overhead it incurs.

We observe both similarities and differences between the results on the two systems. MAGUS consistently achieves positive energy savings across all applications on both systems. However, for some applications on Intel+Max1550, UPS fails to achieve positive energy savings, unlike on Intel+A100. This discrepancy arises because UPS incurs a higher overhead in power consumption on Intel+Max1550 (7.9%) compared to Intel+A100 (4.9%) (Table 2).

Intel+4A100. We extend our evaluation to multi-GPU scenarios using the Intel+4A100 system, focusing on AI-enabled applications and MLPerf benchmarks that effectively utilize multiple GPUs. Figure 4c presents the performance and energy efficiency results for these workloads. We highlight several key observations. First, although MAGUS introduces a 7% and 5.2% performance loss for GROMACS and LAMMPS, respectively, it achieves CPU power savings of approximately 21% and 10%. Second, MAGUS provides greater or comparable energy savings than UPS for workloads such as ResNet50, BERT, and GROMACS. Third, unlike the single-GPU experiments, we observe modest energy savings for MAGUS and UPS. The decrease in energy savings as the number of GPUs increases (with a fixed number of CPUs) is expected. This occurs primarily because the idle power consumption of a multi-GPU system significantly exceeds that of a single-GPU setup. Specifically, in the Intel+4A100 system, the idle power for four A100-80GB GPUs is approximately 200 W, considerably amplifying the energy penalty associated with performance loss compared to the single-GPU configuration, where a single A100-40GB GPU has an idle power around 30 W. Since this idle power represents a fixed operational cost rather than a workload-specific consumption, including it in the total energy calculation can obscure the actual energy savings achieved by uncore frequency scaling.

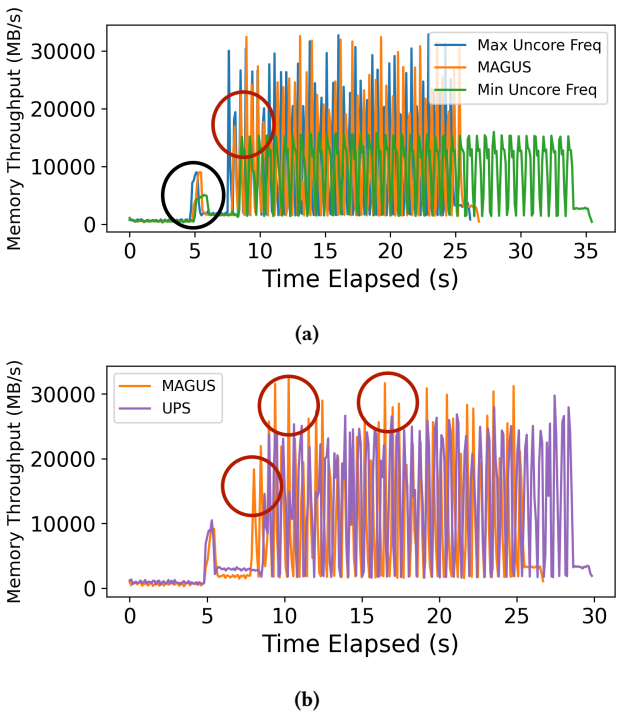


Figure 5: Memory throughout of *SRAD*. The top plot compares MAGUS with the minimum and maximum uncore frequency settings, while the bottom plot highlights the differences between MAGUS and UPS. The embedded circles highlight these differences.

6.2 A Case Study for Detailed Analysis

In this set of experiments, we conduct an analysis to examine how different uncore frequency scaling methods behave using a case study of the *SRAD* application on Intel+A100. This benchmark experiences high-frequency fluctuations in memory throughput, making it a particularly challenging case for uncore frequency tuning. By analyzing *SRAD*, we gain deeper insights into how MAGUS dynamically adjusts uncore frequency compared to other methods.

In Figure 5 (a), three memory throughput plots are presented for the *SRAD* application under three scenarios: maximum uncore frequency (2.2 GHz), minimum uncore frequency (0.8 GHz), and MAGUS. Around the 5-second mark, the memory throughput under the minimum uncore frequency fails to match the level achieved by the maximum uncore frequency. In contrast, MAGUS successfully predicts changes in memory throughput trends, allowing it to reach comparable levels. Overall, MAGUS achieves memory throughput similar to the maximum uncore frequency while delivering an 8.68%

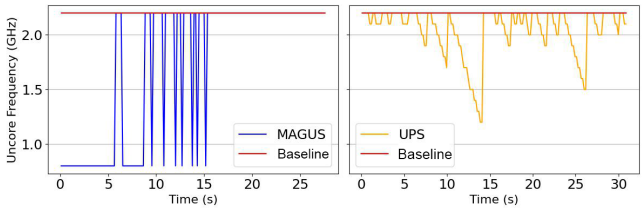


Figure 6: Uncore frequency of *SRAD* under the baseline, UPS, and MAGUS. MAGUS effectively identifies rapid phase shifts and adjusts uncore frequency accordingly.

energy savings compared to Intel's default settings, with only a 3% performance loss.

Next, we compare the memory throughput patterns of maximum uncore frequency, UPS, and MAGUS as shown in Figure 5 (b). We can observe that UPS fails to achieve the high memory throughput levels sustained by MAGUS (as indicated by the read cycles). For workloads with frequent changes in memory throughput demand, response delays due to inherent hardware or software latencies often result in unmet throughput demands. To address this, MAGUS automatically detects phases with high-frequency changes in memory throughput and temporarily sets the uncore frequency to its maximum level to prevent performance loss.

Figure 6 illustrates how MAGUS leverages a high-frequency memory change detector to mitigate performance loss. MAGUS identifies high-frequency phases, such as between seconds 10 to 12.5 and after second 15, and locks the uncore frequency at its maximum (2.2 GHz) during these intervals to maintain performance stability. In contrast, UPS lacks this capability and continues to lower the uncore frequency after second 15, resulting in performance degradation. For the SRAD application, MAGUS achieves a 14% reduction in CPU power consumption compared to UPS's 20%. However, MAGUS incurs only a 3% slowdown, significantly lower than the 7.9% slowdown observed with UPS. Consequently, MAGUS achieves 8.68% energy savings, outperforming UPS, which achieves only 3.5%. By integrating high-frequency memory detection, MAGUS minimizes performance loss when workloads exhibit frequent changes in memory throughput.

6.3 Prediction Accuracy

We assess the accuracy of our algorithm in predicting memory throughput trends by comparing burst patterns observed under MAGUS with those from the baseline configuration, which utilizes the maximum uncore frequency. Specifically, we employ the Jaccard index [35] to quantify alignment between memory throughput bursts from the two configurations. The Jaccard index is selected because it directly quantifies the overlap between memory throughput burst intervals under the maximum uncore frequency case and MAGUS, which adaptively predicts memory throughput trends.

In our analysis, we identify burst intervals, defined as periods when memory throughput exceeds a predefined threshold, and convert them into binary sequences. The Jaccard score is then calculated as the ratio of overlapping burst intervals to the total number of unique burst intervals across MAGUS and baseline runs.

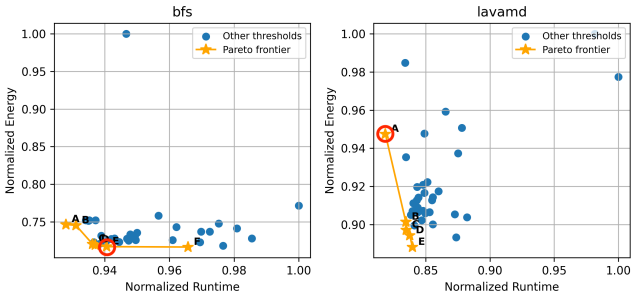


Figure 7: Pareto frontiers of energy consumption and runtime under different threshold configurations. The red-circled configuration represents the common threshold set observed across all applications tested in our experiments, on or close to the Pareto frontier.

Table 1 summarizes these results. The Jaccard score ranges from 0 to 1, with the value of 1 indicating a perfect prediction. We observe scores as high as 0.99, indicating near-perfect burst prediction for many cases. However, applications *fdtd2d*, *cfd_double*, *gemm*, and *particlefilter_float* show lower scores, primarily due to multiple brief bursts during the initialization phase of MAGUS after application launch, before MAGUS starts uncore scaling. Despite their short duration (less than one second), these missed bursts result in only modest performance losses, such as a 3% loss for *fdtd2d*, as shown in Figure 6.1. Conversely, a high Jaccard score (e.g., 0.99) does not always guarantee minimal performance loss. Missing even a single burst with very high memory throughput can significantly impact application performance despite overall accurate burst timing.

Table 1: Jaccard similarity for memory throughput trend

Application	Jaccard	Application	Jaccard
bfs	0.99	gemm	0.71
pathfinder	0.98	sort	0.96
cfd	0.94	cfd_double	0.63
fdtd2d	0.40	kmeans	0.97
lavamd	0.92	nw	0.98
particlefilter_float	0.67	raytracing	0.87
where	0.94	Laghos	0.99
miniGAN	0.98	sw4lite	0.87
UNet	0.99	Resnet50	0.96
bert_large	0.84	lammps	0.99
gromacs	0.99		

6.4 Sensitivity Analysis

Since MAGUS relies on several thresholds to make uncore frequency scaling decisions, we perform a sensitivity analysis to demonstrate that the selected thresholds are broadly effective across all representative workloads used in this work. Specifically, we examine three thresholds: *inc_threshold*, *dec_threshold*, and *high_freq_threshold*.

In the analysis, we fix two thresholds and vary the third, resulting in 40 combinations. Figure 7 illustrates the Pareto frontiers that

capture the trade-off between energy consumption and runtime. We observe that there are multiple sets of thresholds that appear on the Pareto frontier. Notably, a specific set of thresholds (inc_threshold = 300, dec_threshold = 500, and high_freq_threshold = 0.4, circled in red) consistently appears on or close to the Pareto frontier across all representative applications. Due to space constraints, we only present results for two applications in Figure 7, but similar results are observed across the full set of workloads examined.

The Pareto frontier identifies threshold combinations where neither runtime nor energy savings can be improved without hurting the other. Based on the above analysis, we select the common Pareto frontier as the default thresholds. Additionally, MAGUS is invoked every 0.2 seconds, an interval empirically chosen to strike a balance between overhead and responsiveness. Sampling intervals shorter than 0.2 seconds introduce noticeable overhead, whereas longer intervals compromise the timeliness of memory throughput monitoring and uncore frequency adjustment. Hence, we adopt a 0.2-second sampling interval.

Table 2: Overheads by MAGUS and UPS on different systems.

System	Power Overhead (%)		Invocation Overhead (s)	
	MAGUS	UPS	MAGUS	UPS
Intel + A100	1.1%	4.9%	0.1s	0.3s
Intel + Max1550	1.16%	7.9%	0.1s	0.31s

6.5 Runtime Overhead

A runtime inevitably introduces overheads. To quantify runtime overheads, we run MAGUS and UPC individually for 10 minutes without executing any applications, during which we measure the power consumption and the time taken for each invocation (including hardware counter monitoring and phase detection, while excluding uncore scaling). Table 2 shows the results. *Power overhead* is calculated as the relative increase in power consumption introduced by each method. *Invocation overhead* refers to the time taken to retrieve hardware counters and execute the algorithm. The sampling frequency is 0.2 seconds.

MAGUS incurs only a 1% power overhead and requires just 0.1 seconds per invocation. In contrast, UPS introduces up to 7.9% power overhead on Intel+Max1550 and takes approximately 0.3 seconds per invocation. The term "invocation" refers to the measurement and phase detection process performed during each monitoring cycle, which takes approximately 0.1 seconds. MAGUS initiates the next monitoring cycle 0.2 seconds after the previous scaling decision, resulting in a 0.3-second interval between consecutive decisions. This interval is shorter than UPScavenger's 0.5-second (0.3 + 0.2) interval. This efficiency is primarily due to MAGUS retrieving only a single hardware counter (memory throughput). In contrast, UPS accesses each CPU core's model-specific registers (MSRs) to read instructions retired, CPU cycles, and DRAM power data.

6.6 Discussion

Although this study primarily focuses on Intel-based CPUs, the core logic of MAGUS is broadly applicable to various heterogeneous systems. For example, AMD processors (EPYC/Ryzen) include uncore-like components such as the Infinity Fabric, memory controller, and SoC domain. With tools like amd_hsmp [4], it can be used to monitor and, in some cases, adjust SoC/fabric frequencies.

7 Related Work

As large-scale high performance computing (HPC) systems evolve, energy efficiency has become a critical priority [51]. Previous research has explored techniques such as CPU core DVFS, CPU power capping, and memory DVFS to investigate the trade-offs between application performance and energy conservation [10, 22, 23, 31, 44, 57, 62, 63]. For example, Bhalachandra et al. introduced an Adaptive Core-Specific Runtime (ACR) technique that dynamically adjusts CPU core frequencies based on workload characteristics to reduce power consumption while maintaining application performance [11]. Chen et al. showed that jointly applying CPU core DVFS and memory DVFS can further improve energy efficiency [13]. In addition to the RAPL in Intel's family, various other processors have adopted power capping. For example, the IBM Power6 and Power7 architectures provide the power capping feature [12, 50]. The AMD Bulldozer architecture enables users to set a thermal design power (TDP) limit for the processor, allowing for power capping.

Since the Sandy Bridge generation, Intel processors have enabled autonomous DVFS adjustments by modifying clock speeds and voltage dynamically, independent of software-specified settings [16, 19, 55]. Intel's RAPL interface provides mechanisms for monitoring energy consumption and setting power limits in various CPU and DRAM domains[17, 39]. Users can access RAPL data through model-specific registers (MSRs), *sysfs* interface [46], *perf* [45] events, or the PAPI library [65].

Power management has become an active research area with the increasing use of GPUs in HPC systems. NVIDIA provides the NVIDIA Management Library (NVML), which enables users to control GPU power limits, core frequencies, and memory frequencies via the nvidia-smi interface [52, 53]. Significant efforts have been dedicated to modeling power and performance for GPU applications [8, 27, 28, 36, 41]. Research has also explored the impact of GPU core DVFS [2, 20, 28, 40]. GPU memory DVFS [64] on energy efficiency in GPU workloads. Additionally, extensive studies are presented to optimize energy efficiency of deep learning training and inferences [25, 54, 56, 58, 67]. For example, Choi et al. [15] introduced ENVPIPE, which leverages slack time in pipeline parallelism to reduce SM frequency, thereby reducing energy consumption in multi-GPU DNN training without compromising accuracy.

Uncore frequency scaling has received relatively little attention. Existing studies can be broadly classified into model-based and model-free approaches. In model-based approaches, analytical or machine learning models are constructed to predict optimal uncore frequencies by monitoring multiple hardware counters in real time [60, 68]. For instance, Sundriyal et al. develop power and performance models to adjust the uncore frequency for optimal energy efficiency during application execution [60]. Zhang et al.

train a neural network to predict application performance and power consumption, leveraging multi-objective optimization to minimize power usage while limiting performance loss [68]. In contrast, model-free approaches bypass the complex model construction process by dynamically detecting phase transitions between compute-intensive and memory-intensive regions to guide uncore frequency scaling [24, 26]. UPScavenger [24] is a pioneering model-free solution that monitors multiple hardware counters to adjust uncore frequency based on workload phases.

Unlike previous uncore frequency tuning studies focusing on traditional HPC applications running on homogeneous CPU-only systems, this work extends uncore frequency scaling techniques to optimize energy efficiency in various GPU-accelerated HPC environments. Similar to UPScavenger, MAGUS is a model-free runtime. However, MAGUS relies solely on a single hardware counter (memory throughput), which significantly reduces both runtime and power overhead. Additionally, MAGUS employs the concept of memory dynamics, enabling an adaptive and efficient approach to capturing uncore frequency trends based on a workload's real-time memory demands. We implement MAGUS as a user-transparent runtime that requires no code changes, manual intervention, or elevated privileges from users.

8 Conclusion

As HPC systems increasingly adopt heterogeneous architectures, balancing power usage and performance is critical. Our work highlights uncore frequency scaling as a previously underexplored yet impactful strategy for energy efficiency in GPU-accelerated environments. MAGUS addresses this by combining lightweight memory throughput monitoring with dynamic phase detection, achieving up to 27% energy savings while keeping performance loss under 5%. Moreover, MAGUS introduces only 1% overhead in power consumption. Its success in both single- and multi-GPU environments underscores the need for specialized power management strategies that extend beyond traditional CPU-centric approaches. We aim to drive further research and adoption in the HPC community by open-sourcing MAGUS.

Acknowledgments

This work is supported in part by the U.S. National Science Foundation under grants OAC-2402901, CCF-2413597, and CCF-2515009, and by the U.S. Department of Energy under Contract No. DE-SC0024271. Results presented in this paper were obtained in part using the Chameleon testbed supported by the National Science Foundation, as well as resources at the U.S Department of Energy's Office of Science, Argonne Leadership Computing Facility, and Electronic Visualization Laboratory at the University of Illinois Chicago.

References

[1] 2025. ECP proxy apps suite. https://proxyapps.exascaleproject.org/ ecp- proxy-apps- suite/.

[2] Ghazanfar Ali, Mert Side, Sridutt Bhalachandra, Nicholas J Wright, and Yong Chen. 2023. Performance-aware energy-efficient GPU frequency selection using DNN-based models. In *Proceedings of the 52nd International Conference on Parallel Processing*. 433–442.

[3] AMD. 2024. 5TH GEN AMD EPYC™ PROCESSOR ARCHITECTURE. "https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/5th-gen-amd-epyc-processor-architecture-white-paper.pdf".

[4] AMD. 2025. AMD HSMP. "https://github.com/amd/amd_hsmp".

[5] Étienne André, Rémi Dulong, Amina Guermouche, and François Trahay. 2022. DUF: Dynamic Uncore Frequency scaling to reduce power consumption. *Concurrency and Computation: Practice and Experience* 34, 3 (2022), e6580.

[6] Étienne André, Rémi Dulong, Amina Guermouche, and François Trahay. 2022. duf: Dynamic uncore frequency scaling to reduce power consumption. *Concurrency and Computation: Practice and Experience* 34, 3 (2022), e6580.

[7] ARM. 2024. Add an uncore peripheral to Streamline. "https://developer.arm.com/documentation/101814/0904/Customize-your-Streamline-report/Add-an-uncore-peripheral-to-Streamline".

[8] Akhil Arunkumar, Evgeny Bolotin, David Nellans, and Carole-Jean Wu. 2019. Understanding the future of energy efficiency in multi-module gpus. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 519–532.

[9] Large-scale Atomic and Molecular Massively Parallel Simulator. 2013. Lammps. *available at: http:/lammps. sandia. gov* (2013).

[10] Peter E Bailey, Aniruddha Marathe, David K Lowenthal, Barry Rountree, and Martin Schulz. 2015. Finding the limits of power-constrained application performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[11] Sridutt Bhalachandra, Allan Porterfield, Stephen L Olivier, and Jan F Prins. 2017. An adaptive core-specific runtime for energy efficiency. In *2017 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 947–956.

[12] Martha Broyles, Chris Francois, Andrew Geissler, Michael Hollinger, Todd Rosedahl, Guillermo Silva, Jeff Van Heuklon, and Brian Veale. 2010. IBM energyscale for POWER7 processor-based systems. *white paper, IBM* (2010).

[13] Jing Chen, Madhavan Manivannan, Bhavishya Goel, and Miquel Pericàs. 2023. JOSS: Joint Exploration of CPU-Memory DVFS and Task Scheduling for Energy Efficiency. In *Proceedings of the 52nd International Conference on Parallel Processing*. 828–838.

[14] Hsiang-Yun Cheng, Jia Zhan, Jishen Zhao, Yuan Xie, Jack Sampson, and Mary Jane Irwin. 2015. Core vs. uncore: The heart of darkness. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. doi:10.1145/2744769.2647916

[15] Sangjin Choi, Inhoe Koo, Jeongseob Ahn, Myeongjae Jeon, and Youngjin Kwon. 2023. {EnvPipe}: Performance-preserving {DNN} training framework for saving energy. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 851–864.

[16] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R Hanebutte, and Onur Mutlu. 2011. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM international conference on Autonomic computing*. 31–40.

[17] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*. 189–194.

[18] Jianru Ding and Henry Hoffmann. 2023. DPS: Adaptive Power Management for Overprovisioned Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[19] L. K. Documentation. 2024. Intel p-state driver, [Online]. "https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt".

[20] Kaijie Fan, Biagio Cosenza, and Ben Juurlink. 2019. Predictable GPUs frequency scaling for energy and performance. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[21] Steven Farrell, Murali Emani, Jacob Balma, Lukas Drescher, Aleksandr Drozd, Andreas Fink, Geoffrey Fox, David Kanter, Thorsten Kurth, Peter Mattson, et al. 2021. MLPerf™ HPC: A holistic benchmark suite for scientific machine learning on HPC systems. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. IEEE, 33–45.

[22] Vincent W Freeh and David K Lowenthal. 2005. Using multiple energy gears in MPI programs on a power-scalable cluster. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 164–173.

[23] Rong Ge, Xizhou Feng, Wu-chun Feng, and Kirk W Cameron. 2007. Cpu miser: A performance-directed, run-time system for power-aware clusters. In *2007 International Conference on Parallel Processing (ICPP 2007)*. IEEE, 18–18.

[24] Neha Gholkar, Frank Mueller, and Barry Rountree. 2019. Uncore power scavenger: A runtime for uncore power conservation on hpc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–23.

[25] Chengyue Gong, Zixuan Jiang, Dilin Wang, Yibo Lin, Qiang Liu, and David Z. Pan. 2019. Mixed Precision Neural Architecture Search for Energy Efficient Deep Learning. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–7. doi:10.1109/ICCAD45719.2019.8942147

[26] Amina Guermouche. 2022. Combining uncore frequency and dynamic power capping to improve power savings. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1028–1037.

[27] Joao Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomas. 2018. GPGPU power modeling for multi-domain voltage-frequency scaling. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 789–800.

[28] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás. 2019. DVFS-aware application classification to improve GPGPUs energy efficiency. *Parallel Comput.* 83 (2019), 93–117.

[29] Vishal Gupta, Paul Brett, David Koufaty, Dheeraj Reddy, Scott Hahn, Karsten Schwan, and Ganapati Srinivasa. 2012. The Forgotten {'Uncore'}: On the {Energy-Efficiency} of Heterogeneous Cores. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 367–372.

[30] David L Hill, Derek Bachand, Selim Bilgin, Robert Greiner, Per Hammarlund, Thomas Huff, Steve Kulick, and Robert Safranek. 2010. THE UNCORE: A MODULAR APPROACH TO FEEDING THE HIGH-PERFORMANCE CORES. *Intel Technology Journal* 14, 3 (2010).

[31] Chung-hsing Hsu and Wu-chun Feng. 2005. A power-aware run-time system for high-performance computing. In *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, 1–1.

[32] Bodun Hu and Christopher J Rossbach. 2020. Altis: Modernizing gpgpu benchmarks. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 1–11.

[33] Intel. 2025. Intel oneAPI. "https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html".

[34] Intel. 2025. Intel Performance Counter Monitor. "https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html".

[35] Paul Jaccard. 1901. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bull Soc Vaudoise Sci Nat* 37 (1901), 547–579.

[36] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G Rogers, Tor M Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A power modeling framework for modern GPUs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 738–753.

[37] Keahey Kate, Jason Anderson, Paul Ruth, Jcob Colleran, Cody Hammock, Joe Stubbs, and Zhou Zhen. 2019. Operational lessons from Chameleon. In *Proceedings of the Humans in the Loop: Enabling and Facilitating Research on Cloud Computing*.

[38] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S Gunawi, Cody Hammock, et al. 2020. Lessons learned from the chameleon testbed. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 219–233.

[39] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. 2018. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3, 2 (2018), 1–26.

[40] Karlo Kraljic, Daniel Kerger, and Martin Schulz. 2022. Energy Efficient Frequency Scaling on GPUs in Heterogeneous HPC Systems. In *International Conference on Architecture of Computing Systems*. Springer, 3–16.

[41] Adam Krzywaniak and Paweł Czarnul. 2020. Performance/energy aware optimization of parallel applications on gpus under power capping. In *Parallel Processing and Applied Mathematics: 13th International Conference, PPAM 2019, Bialystok, Poland, September 8–11, 2019, Revised Selected Papers, Part II 13*. Springer, 123–133.

[42] Argonne National Laboratory. 2025. Aurora. "https://www.alcf.anl.gov/aurora".

[43] Yuzhuo Li and Yunwei Li. 2025. AI Load Dynamics–A Power Electronics Perspective. *arXiv preprint arXiv:2502.01647* (2025).

[44] Min Yeol Lim, Vincent W Freeh, and David K Lowenthal. 2006. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. 107–es.

[45] Linux. 2024. perf — Linux manual page. "https://man7.org/linux/man-pages/man1/perf.1.html".

[46] Linux. 2024. sysfs — Linux manual page. "https://man7.org/linux/man-pages/man5/sysfs.5.html".

[47] LLNL. 2025. CRADL. "https://github.com/LLNL/CRADL".

[48] LLNL. 2025. Laghos. "https://github.com/CEED/Laghos".

[49] LLNL. 2025. sw4lite. "https://github.com/geodynamics/sw4lite".

[50] H-Y McCreary, Martha A Broyles, Michael S Floyd, Andrew J Geissler, Steven P Hartman, Freeman L Rawson, Todd J Rosedahl, Juan C Rubio, and Malcolm S Ware. 2007. EnergyScale for IBM POWER6 microprocessor-based systems. *IBM Journal of Research and Development* 51, 6 (2007), 775–786.

[51] Paul Messina. 2017. The USDOE Exascale Computing Project–Goals and Challenges.

[52] NVIDIA. 2024. nvidia-smi. "https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf".

[53] Nvidia. 2025. NVML. "https://developer.nvidia.com/management-library-nvml".

[54] Priyadarshini Panda, Abhronil Sengupta, and Kaushik Roy. 2016. Conditional deep learning for energy-efficient and enhanced pattern recognition. In *2016 design, automation & test in europe conference & exhibition (DATE)*. IEEE, 475–480.

[55] N. Pitre. 2024. Teaching the scheduler about power management, [Online]. "https://lwn.net/Articles/603254/".

[56] Saurabhsingh Rajput and Tushar Sharma. 2024. Benchmarking emerging deep learning quantization methods for energy efficiency. In *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 238–242.

[57] Srinivasan Ramesh, Swann Perarnau, Sridutt Bhalachandra, Allen D Malony, and Pete Beckman. 2019. Understanding the impact of dynamic power capping on application progress. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 793–804.

[58] Álvaro Domingo Reguero, Silverio Martínez-Fernández, and Roberto Verdecchia. 2025. Energy-efficient neural network training through runtime layer freezing, model quantization, and early stopping. *Computer Standards & Interfaces* 92 (2025), 103906.

[59] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*. Springer, 234–241.

[60] Vaibhav Sundriyal, Masha Sosonkina, Bryce Westheimer, and Mark Gordon. 2018. Core and uncore joint frequency scaling strategy. *Journal of Computer and Communications* 6, 12 (2018), 184–201.

[61] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E Mark, and Herman JC Berendsen. 2005. GROMACS: fast, flexible, and free. *Journal of computational chemistry* 26, 16 (2005), 1701–1718.

[62] Matthew Walker, Sascha Bischoff, Stephan Diestelhorst, Geoff Merrett, and Bashir Al-Hashimi. 2018. Hardware-validated CPU performance and energy modelling. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 44–53.

[63] Sean Wallace, Zhou Zhou, Venkatram Vishwanath, Susan Coghlan, John Tramm, Zhiling Lan, and Michael E Papka. 2016. Application power profiling on IBM Blue Gene/Q. *Parallel Comput.* 57 (2016), 73–86.

[64] Qiang Wang and Xiaowen Chu. 2020. GPGPU performance estimation with core and memory frequency scaling. *IEEE Transactions on Parallel and Distributed Systems* 31, 12 (2020), 2865–2881.

[65] Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. 2012. Measuring energy and power with PAPI. In *2012 41st international conference on parallel processing workshops*. IEEE, 262–268.

[66] Christoph Weckert, Leonardo Solis-Vasquez, Julian Oppermann, Andreas Koch, and Oliver Sinnen. 2023. Altis-SYCL: Migrating Altis Benchmarking Suite from CUDA to SYCL for GPUs and FPGAs. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 547–555.

[67] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. 2023. Zeus: Understanding and optimizing {GPU} energy consumption of {DNN} training. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 119–139.

[68] Hongjian Zhang, Akira Nukada, and Qiucheng Liao. 2024. FCUFS: Core-Level Frequency Tuning for Energy Optimization on Intel Processors. In *2024 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 214–225.