RESEARCH-ARTICLE

Improving GPU Energy Efficiency through an Application-transparent Frequency Scaling Policy with Performance Assurance

YIJIA ZHANG, Peng Cheng Laboratory, Shenzhen, Guangdong, China

QIANG WANG, Harbin Institute of Technology, Harbin, Heilongjiang, China

ZHE LIN, Sun Yat-Sen University, Guangzhou, Guangdong, China

PENGXIANG XU, Peng Cheng Laboratory, Shenzhen, Guangdong, China

BINGQIANG WANG, Peng Cheng Laboratory, Shenzhen, Guangdong, China

Published: 22 April 2024

Citation in BibTeX format

EuroSys '24: Nineteenth European Conference on Computer Systems
April 22 - 25, 2024
Athens, Greece

Conference Sponsors:
SIGOPS

EuroSys '24: Proceedings of the Nineteenth European Conference on Computer Systems (April 2024)
https://doi.org/10.1145/3627703.3629584
ISBN: 9798400704376

Improving GPU Energy Efficiency through an Application-transparent Frequency Scaling Policy with Performance Assurance

Yijia Zhang
Peng Cheng Laboratory
Shenzhen, China
zhangyj01@pcl.ac.cn

Qiang Wang*
Harbin Institute of Technology
Shenzhen, China
qiang.wang@hit.edu.cn

Zhe Lin
Sun Yat-sen University
Shenzhen, China
linzh235@mail.sysu.edu.cn

Pengxiang Xu
Peng Cheng Laboratory
Shenzhen, China
xupx@pcl.ac.cn

Bingqiang Wang*
Peng Cheng Laboratory
Shenzhen, China
wangbq@pcl.ac.cn

Abstract

Power consumption is one of the top limiting factors in high-performance computing systems and data centers, and dynamic voltage and frequency scaling (DVFS) is an important mechanism to control power. Existing works using DVFS to improve GPU energy efficiency suffer from the limitation that their policies either impact performance too much or require offline application profiling or code modification, which severely limits their applicability on large clusters. To address this issue, we propose a novel GPU DVFS policy, *GEEPAFS*, which improves the energy efficiency of GPUs while providing performance assurance. *GEEPAFS* is application-transparent as it does not require any offline profiling or code modification on user applications. To achieve this, *GEEPAFS* models application performance online based on our quantitative analysis of a correlation between performance and GPU memory bandwidth utilization. Based on their relationship, *GEEPAFS* builds a fold-line frequency-performance model for applications being executed, and it applies the model to guide the setting of GPU frequency to maximize energy efficiency while ensuring the performance loss is bounded. Through experiments on NVIDIA V100 and A100 GPUs, we show that *GEEPAFS* is able to improve the energy efficiency by 26.7% and 20.2% on average. While achieving this improvement, the average performance loss is only 5.8%, and the worst-case performance loss is 12.5% among all 33 tested applications.

*Corresponding authors.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04.
https://doi.org/10.1145/3627703.3629584

CCS Concepts: • **Hardware → Power estimation and optimization**; • **Computing methodologies → Modeling methodologies**; • **Computer systems organization → Parallel architectures**.

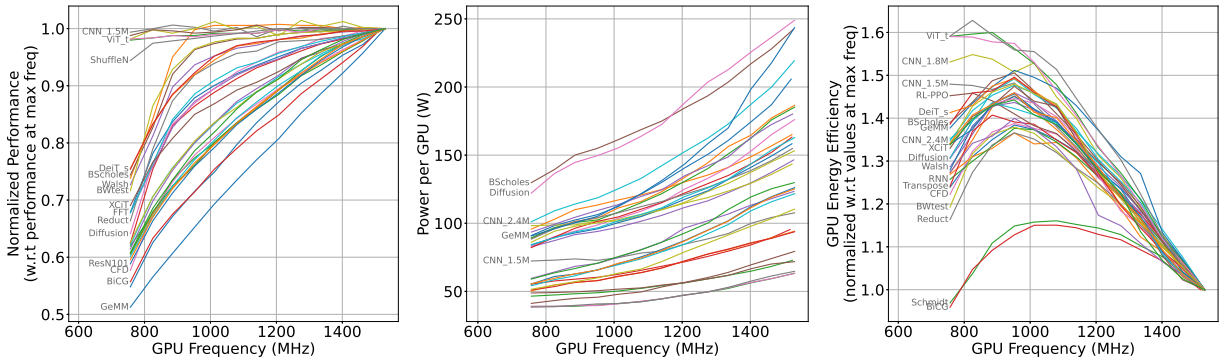
Keywords: Energy Efficiency, Performance Assurance, GPU, DVFS, HPC System, Data Center

ACM Reference Format:
Yijia Zhang, Qiang Wang, Zhe Lin, Pengxiang Xu, and Bingqiang Wang*. 2024. Improving GPU Energy Efficiency through an Application-transparent Frequency Scaling Policy with Performance Assurance. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3627703.3629584

1 Introduction

The rising power consumption has become one of the top limiting factors in high-performance computing (HPC) systems and data centers. The current Top-1 HPC system, Frontier, consumes a peak power of 22.7 MW [46]. It is also estimated that all data centers in the world consume 1% of global energy production [20]. To reduce the electricity bill and environmental impacts, it is critical to improve the energy efficiency of large computing systems. Since the rapid growth of artificial intelligence, graphics processing units (GPU) are playing an important role in computing clusters, but recent generations of GPUs are giant power consumers, as the thermal design power (TDP) of NVIDIA GPUs goes from 300 W on GPU V100, 400 W on A100, and up to 700 W on H100 [28]. In high-end GPU servers, the total power of all GPUs usually surpasses the total power of the remaining part. As a result, improving GPU's energy efficiency is of central importance.

Dynamic voltage-frequency scaling (DVFS) is an approach to regulate the power consumption of a processor by increasing/decreasing its frequency f and voltages V. DVFS is a key control knob in the power management of computing systems because of the following reasons: first, DVFS can modulate power in wide ranges since the dynamic power of



(a) Frequency-performance relation. **(b)** Frequency-power relation. **(c)** Frequency-efficiency relation.

Figure 1. The impact of GPU (V100) frequency on performance, power, and energy efficiency. Overlapping labels are omitted.

a processor is directly proportional to V^2f [15, 21]; second, DVFS in a proper range only affects the execution time of applications, and it does not affect their results (except for the undervoltage scenario [23]); third, the response time of DVFS is small, usually within 100 milliseconds or less [35, 48]; finally, DVFS is generally considered mature on modern processors and it has been widely applied in many systems from personal computers to smartphones.

Given these advantages of DVFS, many works have experimented with GPU DVFS, and they find that tuning voltage and frequency properly improves energy efficiency [1–3, 7, 8, 10–15, 17–19, 21, 22, 26, 27, 37, 38, 42, 43, 45, 47–50, 52]. Figure 1 shows our experiments on NVIDIA V100 GPUs, where we measure the performance[1], power, and energy efficiency of 33 applications (more information in Section 5). Each curve represents one application running at different GPU frequencies[2]. Figure 1(a) shows that the performance changes with frequency, and the highest performance is reached at the max frequency. Figure 1(b) shows that the GPU power increases superlinearly with frequency. This is because the dynamic power is proportional to V^2f, and the voltage V is positively correlated with f. Figure 1(c) shows that the maximal energy efficiency is not reached at the max frequency. The fact that max-efficiency frequency does not align with the max frequency offers the opportunity to improve energy efficiency by tuning frequency properly.

Given this opportunity, previous works have proposed some frequency scaling policies for a GPU server to apply to improve energy efficiency [1, 2, 8, 11, 12, 14, 17–19, 22, 26, 37, 38, 47, 51, 52]. However, as far as we know, production systems rarely adopt these policies, and the common concern is performance loss and application-intrusiveness. Our experiments confirm that maximizing energy efficiency leads to significant performance loss. As shown in Fig. 1, executing on V100 GPUs at the max-efficiency frequency (around 950 MHz) reduces application performance by up to 34%. This amount of performance loss could significantly delay jobs on a cluster. Although some of those works also tried to bound the performance loss, their approaches require offline profiling or code modification on user applications. Given the diversity of applications running on large clusters, offline profiling or code modification is impractical for many systems. Therefore, to design a frequency scaling policy practical for production systems, it is important to offer performance assurance and design the policy in an application-transparent way.

To address this challenge, we propose *GEEPAFS*: a GPU Energy-Efficient and Performance-Assured Frequency Scaling policy. *GEEPAFS* improves energy efficiency while providing performance assurance. Here, performance assurance is quantified as the goal that the performance loss of running any application with *GEEPAFS* is bounded by a preset percentage (such as 10%). *GEEPAFS* is also application-transparent as it does not require any offline profiling or code modification on user applications. It achieves this goal by modeling performance online using GPU hardware counters. Based on our quantitative analysis of a correlation between GPU memory bandwidth utilization and application performance, *GEEPAFS* builds a fold-line model to predict application performance at different frequencies, and applies the model to set the GPU frequency to improve energy efficiency while ensuring the performance loss is bounded.

To evaluate our policy, we implement *GEEPAFS* and compare it with four other frequency scaling policies on three types of GPU servers, including a DGX station with 4×GPU V100, a DGX-1 server with 8×GPU V100-MaxQ (a power-limited version of V100), and an A100 server with 1×GPU A100. We experiment with 33 workloads, including single-GPU applications and parallel applications running on 4 to 8 GPUs. Our results show that, on V100 and A100 GPUs, *GEEPAFS* is able to improve the energy efficiency by 26.7% and 20.2% on average. While achieving this improvement, the average performance loss is only 5.8% and 5.7%, and

[1]We quantify the performance of running an application as the total amount of work done divided by the total execution time, and we quantify energy efficiency as performance divided by average power consumption.

[2]NVIDIA GPUs support adjusting frequency by NVML [32]. Voltage control is managed by internal logics correlated with frequency and is not exposed to users on GPU V100/A100, so we only control frequency directly.

Table 1. Comparison with GPU DVFS policies proposed in related works. Other works [3, 7, 10, 13, 15, 21, 22, 27, 42, 43, 45, 48–50] not proposing online DVFS policies are not listed. Most of the previous works require offline application profiling. The "1 × 4" in "Experiment Scale" column means 4 types of GPU are tested, but only using single-GPU applications.

Related Works	Performance Awareness	No Change to Codes /Binaries	No Offline Application Profiling	No Limitation on Workloads	No Offline Model Training	Experiment Scale (GPUs)	Platform
Ma, 2012 [18]	✓	✓		✓	✓	1	NVIDIA
Komoda, 2013 [17]				✓	✓	1	NVIDIA
Paul, 2013 [38]		✓		✓	✓	1	AMD
Abe, 2014 [1]	✓	✓		✓	✓	1×4	NVIDIA
Paul, 2015 [37]	✓	✓		✓	✓	1	AMD
Guerreiro, 2015 [11]		✓		✓	✓	1×4	NVIDIA
Majumdar, 2017 [19]	✓	✓		✓	✓	1	AMD
Fan, 2019 [8]	✓	✓		✓		1×2	NVIDIA
Guerreiro, 2019 [12]	✓	✓		✓	✓	1×5	NVIDIA
Ilager, 2020 [14]	✓	✓		✓		1	NVIDIA
Zou, 2020 [52]	✓	✓	✓			1	NVIDIA
Nabavinejad, 2022 [26]	✓				✓	1	NVIDIA
Wang, 2022 [47]	✓		✓	✓		1	NVIDIA
Ali, 2022 [2]	✓	✓		✓	✓	1	NVIDIA
Ours	✓	✓	✓	✓	✓	8/4/1	NVIDIA

the worst-case performance loss is 12.5% and 8.9% on V100 and A100 GPUs. The source codes of our work are publicly available at https://github.com/zyjopensource/geepafs.

In summary, this work makes the following contributions:

- We propose a GPU frequency scaling policy, *GEEPAFS*, which improves energy efficiency while ensuring meeting a performance target in terms of execution time. *GEEPAFS* is application-transparent as it does not require any offline profiling or code modification on user applications.
- Based on our analysis of a correlation between performance and GPU memory bandwidth utilization, *GEEPAFS* builds a fold-line model to quantify the frequency-performance relation online, and applies the model to set GPU frequency under performance constraints.
- Through experiments on NVIDIA V100 and A100 GPUs, we show that *GEEPAFS* improves the energy efficiency by 26.7% and 20.2% on average, respectively, with an average performance loss of 5.8%, and the worst-case performance loss is 12.5%. *GEEPAFS* also achieves lower ED2P (energy-delay-square product) than the baseline policies.
- We show that the accuracy of our performance modeling is bottlenecked by the latency of frequency tuning and the resolution of hardware counters on NVIDIA GPUs.

2 Related Works

The impact of GPU frequency on power and performance has been analyzed in many works [3, 10, 13, 21, 22, 36, 45, 48], and models to predict GPU power are built [3, 7, 10, 13, 15, 21, 44, 48–50]. However, these works only focus on measurement and offline modeling, and they have not proposed frequency scaling policies that a GPU server can apply.

Some GPU frequency scaling policies are proposed in previous works with the aim of improving energy efficiency [1, 2, 8, 11, 12, 14, 17–19, 26, 37, 38, 47, 51, 52], and they are summarized in Table 1. However, the majority of these policies require offline profiling or code modification on user applications to obtain the frequency-performance relations of applications. These requirements severely limit their applicability on production systems. On the contrary, our work proposes an approach to avoid these limitations.

Table 1 also lists other limitations of the previous works. For example, some earlier works did not consider performance when improving energy efficiency [11, 17, 38]. Some works are limited to certain workloads [26, 51, 52]. For example, Zou et al. base their approach on capturing the periodic patterns in GPU metrics, and they models performance by detecting the shrinking and expanding of the periodic patterns under frequency scaling [52], which requires the existence of detectable periodic patterns longer than the sampling latency. On the other hand, our work does not suffer from these limitations.

Some works applied reinforcement learning (RL) in GPU DVFS on mobile devices [5, 6, 16]. However, the environments of mobile devices are very different from computing clusters. For example, a smartphone is used by a single person and usually equipped with tens of applications, while a large cluster usually serves thousands of users and runs an enormous number of workloads in its lifetime. As a result, it will be hardly affordable to apply RL-based approaches to learn workload characteristics in a large computing cluster. Besides, a few other works also explored DVFS using GPU simulators [27, 42, 43]. Although simulators enable them to tune frequency in a finer way, their policies require hardware counters that are not available to be accessed online on current GPUs.

In addition, none of the previous works in Table 1 have evaluated their work on applications running with multiple GPUs in parallel, as shown in the "Experiment Scale" column.

"1×4" means they experimented with 4 types of GPU, but only single-GPU applications were tested. Our work evaluates our policy both on single-GPU workloads and multi-GPU workloads up to 8 GPUs running in parallel.

3 Design Principles

Our goal is to design an application-transparent GPU frequency scaling policy applicable on production systems to improve energy efficiency with performance assurance, this section elaborates our design principles towards this goal.

Design Principle 1: *The policy will capture workload characteristics and build frequency-performance models online.*

In order to improve energy efficiency through GPU frequency scaling and at the same time avoid large performance loss, it is necessary to know the relation between frequency and performance. However, Fig. 1(a) shows that there is not a unique freq-perf relation[3] on a certain GPU, but the freq-perf relations vary from one application to another significantly. In Fig. 1(a), for some GPU-compute-intensive applications such as GeMM, BiCG, and CFD, their performance at 800 MHz is about 40% lower than the performance at 1530 MHz. On the other hand, the performance of applications CNN_1.5M, ViT_t, and ShuffleN are almost not affected by GPU frequency. As a result, it is necessary to build freq-perf models online according to the application being executed.

Design Principle 2: *The policy cannot modify user applications' source codes and cannot assume the existence of application execution progress report.*

A key challenge in providing performance assurance is to build freq-perf models online. In this work, we quantify performance as "the total amount of work divided by total execution time". This definition of performance is directly related to user experience. However, the difficulty of modeling performance online comes from the fact that the amount of work done by an application is not accessible in general before its entire execution is finished.

Some previous works assume that their application's runtime progress is accessible. With that assumption, they build freq-perf relations online by comparing the progress at different frequencies [26, 42]. However, that approach is not generalizable to all applications because, on a large cluster with thousands of users and applications, it is hardly possible to ask all users to add formatted progress report to their codes, and it is also hardly possible to build a tool to automatically retrieve the progress of all applications. For the same reason, it is not practical to rely on application profiling tools that require code modification, although the metrics collected by those profiling tools (such as CUPTI [30]) can be helpful in performance modeling.

Design Principle 3: *The frequency scaling policy cannot rely on offline application profiling.*

[3]We use "freq" and "perf" as abbreviations for frequency and performance.

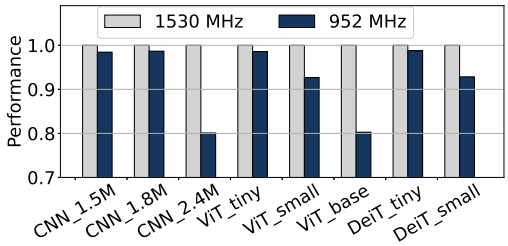


Figure 2. Performance sensitivity to GPU frequency varies with inputs. All performance values are normalized.

To build performance models without modifying user applications, many previous works rely on offline application profiling (see Table 1). Those works profile an application by recording low-level metrics while running the application, and they use the metrics to build freq-perf models. For example, Guerreiro et al. train machine learning models to predict the performance's sensitivity to frequency based on several metrics [12], including the number of reads and writes to DRAM / shared memory / L2 cache, and the utilization of single/double precision units or special function units, etc. However, because most of those metric values are only accessible post-execution on current GPUs (using the nvprof tool [34] for example), those works require running each application at least once to finish offline profiling, which is not practical on many systems.

Design Principle 4: *The frequency scaling policy cannot require repetitively running an application.*

Another way to build performance models without code modification is to repetitively run an application at different GPU frequencies and measure their performance. Some previous works require applying this approach to obtain the optimal frequency [11, 18, 51]. However, as machine hours are valuable resources, repetitively running each application multiple times is not acceptable. In addition, launching every job more than once complicates the workflow of users, so it will not be welcomed by HPC and data center users.

Design Principle 5: *The policy will capture an application's runtime behavior instead of simply relying on its category.*

Since profiling all applications is not practical, some may wonder whether it is possible to profile a few typical applications and apply the same frequency to every application in the same category. The problem is, when running even the same application script, simply changing some parameters could dramatically change its freq-perf relation. Figure 2 shows our experimental results on a DGX station with V100 GPUs. CNN_* are three applications that train a convolutional neural network with different sizes from 1.5 to 2.4 million parameters, ViT_* are training a vision transformer model with different sizes, and DeiT_* are another image transformer model with different sizes.

In Fig. 2, when the model size of CNN increases from 1.8M to 2.4M, the application goes from memory-bound to GPU compute-bound, so CNN_2.4M is more sensitive to GPU frequency than CNN_1.8M. In the case of ViT, when

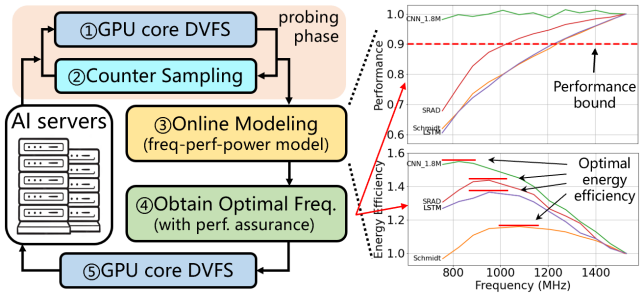


Figure 3. Overview of the *GEEPAFS* policy.

its model size increases, the application goes from CPU compute-bound to GPU compute-bound on our server, so ViT_base is more sensitive to GPU frequency than ViT_tiny. These examples prove that the performance sensitivity of an application cannot be easily predicted in advance, so profiling typical applications cannot serve the purpose.

Design Principle 6: *The frequency scaling policy will rely on models with explainability.*

Some previous works apply machine learning models to build performance models from hardware metrics and to conduct GPU frequency tuning [8, 14, 22, 47, 52]. However, the lack of explainability in some machine learning models could raise concerns when applied to low-level system controls such as GPU DVFS. The predictability of a controller's behavior becomes uncertain when it relies on machine learning models. Additionally, it is unclear if models trained on a limited number of applications can generalize well to thousands of applications in a large cluster. Such concerns could limit the adoption of these methods on production systems.

4 The *GEEPAFS* Policy

Following the design principles discussed above, we propose our GPU frequency scaling policy in this section.

4.1 Overview of *GEEPAFS*

An overview of our *GEEPAFS* policy is shown in Fig. 3. The policy executes Steps ①-⑤ iteratively on each GPU, with the aim of tuning the GPU frequency to achieve an energy efficiency as high as possible while providing performance assurance. Here, performance assurance is quantified as the goal that the performance loss of running any application with *GEEPAFS* is bounded by a percentage of δ (the value is adjustable, and $5\% \leq \delta \leq 30\%$ is suggested). In other words, the performance of running any application with *GEEPAFS* cannot be lower than the performance of running at the maximal GPU frequency by a percentage of δ.

To achieve this goal, online modeling of the freq-perf relation is critical. This online modeling is Step ③, and exemplar freq-perf curves and freq-efficiency curves are on the right side of Fig. 3. Because the freq-perf and freq-power relations vary across applications, *GEEPAFS* samples hardware counters in Step ② to model the relations for an application being executed on a GPU. These counters include GPU utilization, GPU memory bandwidth utilization, and power usage. In *GEEPAFS*, the GPU memory bandwidth utilization metric is the key, which is elaborated in Section 4.3.

GEEPAFS not only sample hardware counters at a certain GPU frequency. Instead, it requires sampling counters at several different GPU frequencies to effectively model performance, which is elaborated in Section 4.4. To achieve this goal, Step ① and Step ② form a small loop as the probing phase in Fig. 3 to tune the frequency and sample the counters at several different frequencies.

Once the perf-freq-power model is established from Step ③, *GEEPAFS* searches for the frequency which improves energy efficiency while ensuring performance loss is less than δ. This forms Step ④. Finally, Step ⑤ tunes the GPU to that optimal frequency found in Step ④, and *GEEPAFS* finishes this execution cycle of the policy and waits for the next execution cycle. Note that this work focuses on the frequency scaling of GPU compute cores. GPU memory frequency scaling is not discussed as it is not supported on GPU V100/A100.

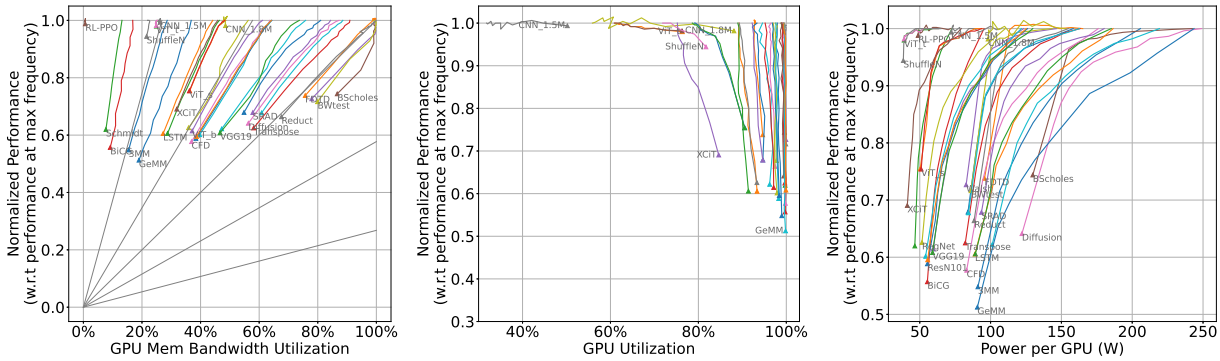
4.2 Hardware counter sampling

Multiple tools are available for sampling hardware counters on NVIDIA GPUs. These tools includes NVML [32], nvprof [34], Nsight [33], CUPTI [30], DCGM [31], etc. However, to follow our design principles, we should not rely on tools that cannot access values during job execution or tools that require editing application source codes. As a result, we do not use nvprof or Nsight because their sampling values are not accessible in the middle of job execution. Also, profiling using nvprof / Nsight usually leads to extension of job execution time [48]. Besides, we do not use CUPTI because CUPTI requires editing application source codes.

Based on the discussions above, we use NVML to read hardware counters on GPUs. DCGM is similar to NVML and can be used as an alternative. The NVML API and its executable, nvidia-smi, allow us to tune the GPU frequency and sample metrics including GPU utilization, GPU memory bandwidth utilization, GPU power usage, etc. NVML supports sampling these metrics at a minimal interval of a few milliseconds on V100/A100 GPUs. Compared with other tools, NVML's number of sampled metrics are limited, but it has an advantage that its values are accessible during job execution and it does not require code modification. In addition, NVML API can be accessed as long as the NVIDIA GPU driver and the CUDA Toolkit are installed, and no installation of additional packages are required, which significantly facilitates applying our policy on production systems.

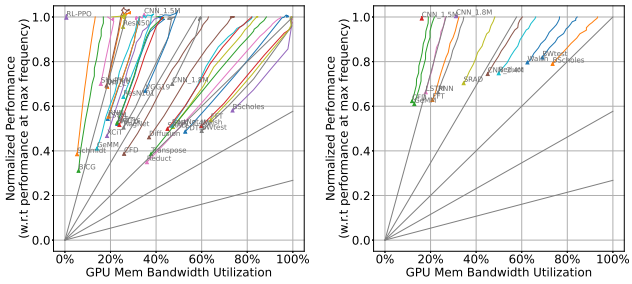
4.3 The GPUbwUt-performance correlation

Among all metrics collected by NVML, we discover that the *GPU memory bandwidth utilization* metric, abbreviated as *GPUbwUt* in the following, is the key to performance modeling. Figure 4 shows our experimental results on a DGX station with 4×V100 GPUs, where each line represents one



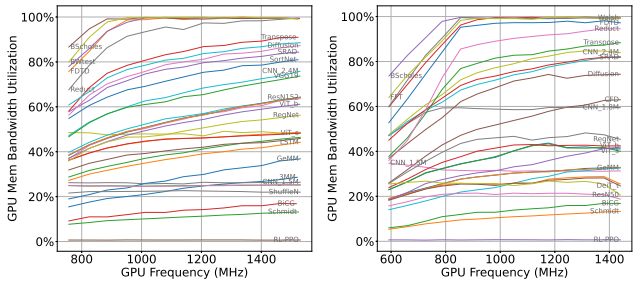
(a) Bandwidth Utilization - Performance. **(b)** GPU Utilization - Performance. **(c)** GPU Power - Performance.

Figure 4. Correlation between performance and GPU metrics on V100 GPUs. Each line represents an application running at different frequencies. Triangle marks the side with low frequency (757 MHz). Overlapping labels are omitted.



(a) On V100-MaxQ server. **(b)** On A100 server.

Figure 5. The GPUbwUt metric shows a directly proportional relation with application performance.



(a) On V100 server. **(b)** On V100-MaxQ server.

Figure 6. Frequency-GPUbwUt relations of 30+ applications. Some overlapping application labels are omitted.

application from Table 3. We repetitively run each application with unchanged inputs at different GPU frequencies. The metric values are collected by NVML and averaged over an application's execution time. For multi-GPU applications, the values are also averaged over all GPUs.

As shown in Fig. 4(a), GPUbwUt is highly correlated with application performance under frequency scaling. Furthermore, for each application, its performance is directly proportional to GPUbwUt. The auxiliary lines in pale grey show the trends of perfectly proportional relations, and the GPUbwUt-perf relations follow those lines closely. Figure 5 shows that these proportional relations also persist on A100 GPU and V100-MaxQ GPU. This proportional relation between performance and GPUbwUt under frequency scaling has not been reported in previous works [1–3, 7, 8, 10–15, 17–19, 21, 22, 25–27, 36–38, 42–45, 47–52].

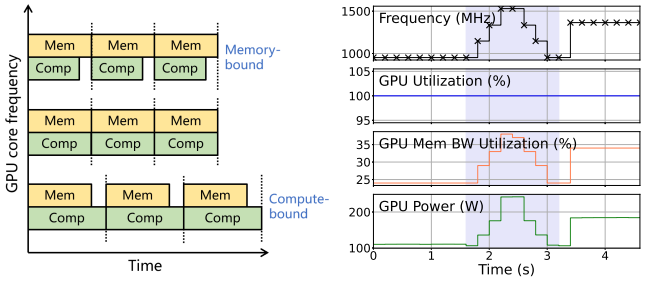
This GPUbwUt-perf relation in directly proportional forms enable us to model performance in an application-transparent way by online measurement of GPUbwUt. To be specific, if we first sample GPUbwUt at the max frequency f_{max} to obtain its value $U(f_{max})$, and sample at another frequency f to obtain $U(f)$. Then, the performance at frequency f_{max}, denoted as $Q(f_{max})$, and the performance at frequency f, denoted as $Q(f)$, should satisfy $Q(f)/Q(f_{max}) = U(f)/U(f_{max})$ due to the directly proportional GPUbwUt-perf relation. As our work is focusing on providing assurance to the normalized performance $Q(f)/Q(f_{max})$, we can use $U(f)/U(f_{max})$ as an estimation of the performance at any frequency f.

For a GPU application, because the amount of work done by the application is generally proportional to its read/write to the GPU memory, we expect GPUbwUt to be directly proportional to performance under frequency scaling. On the other hand, because there are no simple relations between performance and GPU power or GPU utilization, as shown in Fig. 4(b)(c), we cannot use those two metrics to model performance in an application-transparent way.

4.4 Fold-line model of frequency-GPUbwUt relation

With the GPUbwUt-perf relation discussed above, we also need a freq-GPUbwUt relation. Then, combining the two relations gives us the desired freq-perf relation.

Figure 6 shows the measured GPUbwUt values (averaged over execution time) when running applications at different GPU frequencies on V100 and V100-MaxQ GPUs. For each application, its freq-GPUbwUt curve is relatively simple, so if we sample GPUbwUt at a few frequencies, it is possible to fit a freq-GPUbwUt curve and use it to predict the GPUbwUt at other frequencies. From Fig. 6, we see the freq-GPUbwUt relations are nonlinear in many cases. If we simply fit a linear model, it could be significantly inaccurate. It is this



(a) Compute model. **(b)** Metric sampling.

Figure 7. Compute model and *GEEPAFS* metric sampling.

nonlinearity that drives us to include a probing phase in *GEEPAFS* to sample metrics at different frequencies.

Based on Fig. 6, we use fold-line models to fit the freq-GPUbwUt relations, and we assume a freq-GPUbwUt curve can be approximated by two connected line segments. To be specific, we assume there is a critical frequency f_c (which may vary across applications) separating two segments. The GPUbwUt value $U(f)$ follows different linear models when the GPU frequency is beyond or below f_c, i.e.,

$$U(f) = \begin{cases} a_1 \cdot f + b_1, & f < f_c, \\ a_2 \cdot f + b_2, & f \geq f_c. \end{cases} \quad (1)$$

Our modeling of the freq-GPUbwUt relation using a fold-line can be explained by the compute model in Fig. 7(a). The execution of a GPU application is typically composed of iterative phases, where each phase includes both computation (Comp) and memory access (Mem). The sensitivity of an application's performance on GPU core frequency depends on how much memory access latency can be hidden by the computation time [12, 27]. In general, when frequency is low enough, the execution is bounded by computation, and performance is sensitive to core frequency. On the other hand, when frequency is high enough, the execution is bounded by memory access, and performance is insensitive to core frequency. The transition between these two situations under frequency scaling explains why we can model the freq-GPUbwUt relation as a fold-line with one turning point.

To fit the fold-line model for an application, we first sample GPUbwUt values at three or more different frequencies while running the application. Then, we apply the least squares method to fit the fold-line model Eq. (1) using the samples.

4.5 Algorithms of the *GEEPAFS* policy

Our *GEEPAFS* policy follows the procedures in Fig. 3. It first samples GPU counters in Steps ①② by looping the two steps for several times so that metric values at several GPU frequencies are sampled. This is called the *probing phase*. Next, Step ③ applies the fold-line modeling method described in Section 4.4 to obtain the freq-perf model of the application. Then, given a performance constraint δ, Step ④ calculates a frequency lower bound based on the freq-perf model, and beyond that lower bound, it finds the optimal frequency that maximizes energy efficiency. Finally, Step ⑤ sets the GPU to that optimal frequency using NVML.

Algorithm 1 is the pseudocode for *GEEPAFS*, which runs on each GPU in a server. The algorithm takes several inputs. Performance constraint δ is a percentage set by the system administrator. T_1 sets *GEEPAFS* to repeatedly execute Steps ①-⑤ every T_1 seconds, and we use $T_1 = 15$ s in our experiments. T_2 sets the time interval of sampling different frequencies in the probing phase, and we use $T_2 = 200$ ms in our experiments. F is the list of all the probing frequencies. M is the total number of samples. Section 4.7 further discusses how to set these parameters.

Algorithm 1: *GEEPAFS* Policy

Input: performance constraint δ, global loop interval T_1, probing phase interval T_2, probing frequency list F, total number of samples M

```
1  while True do
2      t ←current time;
3      if isProbingPhase(t) then
4          G, U, P ←Collect GPU utilization, GPU memory
             bandwidth utilization, and power readings;
5          SetNextProbingFreq();
6          wait(T2);
7      else if isEndOfProbingPhase(t) then
8          Model←FitFreqPerfModel(U, F);
9          f_bound ←CalcPerfAssuredFreq(Model,δ);
10         f_eff ←GetEnergyEfficientFreq(Model,P);
11         f_cap ←CalcGpuFreqCap(G, F, δ); // get frequency
             upper limit according to Eq. (2).
12         f_opt ←min(max(f_bound, f_eff),f_cap);
13         SetGpuFreq(f_opt);
14         wait(T1 − M · T2);
15     end
16 end
```

At Line 3 of Algorithm 1, function *isProbingPhase(t)* determines whether the current loop is in the probing phase or not. When in the probing phase, it executes Lines 4-6. Figure 7(b) shows a real example of probing when applying *GEEPAFS* on a V100 GPU while running GeMM. The shaded region shows the probing phase, and the sampled metrics are drawn. We set *GEEPAFS* to sample four frequencies, 952 MHz, 1147 MHz, 1335 MHz, 1530 MHz. Each frequency is sampled twice, first by going from 952 to 1530 MHz, then by going from 1530 to 952 MHz, as shown in the first row of Fig. 7(b). The total number of samples is $4 \times 2 = 8$. Two adjacent samples are separated by a $T_2 = 200$ ms interval, so the duration of the probing phase is $8 \times 0.2 = 1.6$ s.

When the probing phase reaches the end, Lines 7-14 of Algorithm 1 are executed to fit the performance model and obtain the optimal frequency. Line 8 fits the freq-perf model as described in Sections 4.3 and 4.4. We fit both a fold-line model and a linear model, because linear model is a special

case of fold-line models when the turning point falls outside the range of sampled frequencies. Either the fold-line or the linear model that has a smaller regression error is selected. With the model and δ, Line 9 calculates the lower bound of frequency, f_{bound}, and the pseudocode is Algorithm 2.

Line 10 of Algorithm 1 obtains the max-efficiency frequency f_{eff} by comparing the energy efficiency at different frequencies. Given the power reading samples, a typical way to fit a freq-power model is using a third-order polynomial [15], and then we can derive the freq-efficiency model. In our experiments, since power modeling is not the focus of this work, we simplify this step by only comparing the energy efficiency at the sampled frequencies, and we set f_{eff} as the one with the highest energy efficiency, where we calculate energy efficiency as the predicted performance (using the fold-line model) divided by the average power reading at a frequency. This simplified way works well because f_{eff} is usually lower than f_{bound}, and in that case, f_{bound} will be chosen to assure performance and f_{eff} will be ignored. The third-order polynomial power model can optionally be selected in the source code of *GEEPAFS*. If a high-accuracy power model is needed, more advanced power models based on GPU counters may also be considered [13, 15].

Line 11 adds an upper bound on GPU frequency using

$$f_{cap} = \max_i (1-\delta) \cdot f_i \cdot G(f_i). \tag{2}$$

This step is not critical to *GEEPAFS*. We only add it to further help bound the frequency from above in case GPU utilization is low. In Eq. (2), δ is the preset performance constraint, f_i are different sampling frequencies, and $G(f_i)$ stands for the corresponding GPU utilization sampled at each f_i.

We set this frequency cap by the assumption that, for an application underutilizing the GPU (i.e., $G << 100\%$), the existence of idle cycles allows reducing frequency, and the effect of reducing frequency can be compensated by an increase in GPU utilization without losing performance. This can be seen from Fig. 4(b), where applications CNN_1.5M, CNN_1.8M, ShuffleN, and ViT_t have relatively low GPU utilization. When frequency decreases, the utilization increases (triangle marks the lowest frequency tested) and the performance maintains at a similar level. As $G(f_i)$ are sampled at different frequencies, Eq. (2) selects the maximum calculated from all samples to make f_{cap} a conservative bound. Finally, Line 12 sets the optimal frequency f_{opt} to make it satisfy $f_{opt} \le f_{cap}$ and $f_{opt} \ge f_{bound}$ and close to f_{eff} if possible.

4.6 Tuning frequency with performance assurance

Algorithm 2 implements Line 9 of Algorithm 1 by calculating the frequency bound f_{bound} given the performance constraint δ. This algorithm uses the fold-line model of the freq-GPUbwUt relation to find f_{bound}. In principle, it first finds the maximal GPUbwUt value (denoted as U_{max}), and then, it sets f_{bound} to be the minimal frequency whose corresponding GPUbwUt value is $(1-\delta)U_{max}$. Since GPUbwUt is directly proportional to performance (Section 4.3), having a GPUbwUt value of $(1-\delta)U_{max}$ means the performance at that frequency is lower than the maximal performance by δ. If we set $\delta = 10\%$, then $(1-\delta)U_{max} = 0.9U_{max}$.

Algorithm 2: Calculate frequency bound with performance-assurance (Line 9 of Algorithm 1)

Input: frequency-performance model parameters, performance constraint δ
Output: performance-assured frequency f_{bound}

1 **if** *isLinearModel()* **then**
2 // The model has slope a and intercept b.
3 **if** *slope* $a > 0$ **then**
4 Calculate f_{bound} by equation:
  (3) $a \cdot f_{bound} + b = (1-\delta)(a \cdot f_{max} + b)$
5 **else if** *slope* $a \le 0$ **then**
6 $f_{bound} \leftarrow$ the lowest sampled frequency f_{low}
7 **end**
8 **else if** *isFoldLineModel()* **then**
9 // The low-freq line has slope a_1 and intercept b_1.
10 // The high-freq line has slope a_2 and intercept b_2.
11 **if** *slope* $a_1 < a_2$ **then**
12 Model not accepted. Use linear model.
13 **else if** $a_1 > 0$ *and* $a_2 > 0$ **then**
14 Calculate f_{bound} by equation:
  (4) $a_i \cdot f_{bound} + b_i = (1-\delta)(a_2 \cdot f_{max} + b_2)$
  // i =1 or 2 if f_{bound} falls in low- or high-freq part.
15 **else if** $a_1 > 0$ *and* $a_2 \le 0$ **then**
16 Calculate f_{bound} by equation:
  (5) $a_1 \cdot f_{bound} + b_1 = (1-\delta)(a_1 \cdot f_{turn} + b_1)$
  // f_{turn} defined by $a_1 \cdot f_{turn} + b_1 = a_2 \cdot f_{turn} + b_2$.
17 **else if** $a_1 \le 0$ **then**
18 $f_{bound} \leftarrow$ the lowest sampled frequency f_{low}
19 **end**
20 **end**

Algorithm 2 distinguishes between two cases, a linear model and a fold-line model, using functions *isLinearModel()* and *isFoldLineModel()* at Line 1 and Line 8. Whether the linear or fold-line model is adopted is determined by Line 8 of Algorithm 1 as discussed in Section 4.5. When the linear model is adopted, assume the model is $U(f) = a \cdot f + b$. Then, following Lines 3-7 of Algorithm 2, if $a > 0$, $U_{max} = a \cdot f_{max} + b$, so we can obtain f_{bound} by Eq. (3). If $a \le 0$, the maximal U we know is at the lowest sampled frequencies, f_{low}, then we simply set f_{bound} as f_{low}.

When the fold-line model is adopted, assume the low-frequency segment is $U(f) = a_1 \cdot f + b_1$, and the high-frequency segment is $U(f) = a_2 \cdot f + b_2$. At Line 11, if $a_1 < a_2$, the model is theoretically impossible and it must be caused by large variations. In this case, this model is not accepted and the linear model is adopted. If in the case of Line 13, then $U_{max} = a_2 \cdot f_{max} + b_2$, and we solve Eq. (4) to obtain f_{bound}.

Similarly, if in the case of Line 15, then $U_{max} = a_1 \cdot f_{turn} + b_1$, and we solve Eq. (5) to obtain f_{bound}, where f_{turn} is the turning point of the fold-line model. In the case of Line 17, we set f_{bound} as f_{low} because f_{low}'s GPUbwUt value is the highest.

4.7 Discussion on parameter selection

When applying *GEEPAFS* on a GPU server, the parameters of the policy should be chosen based on the specific GPU type. The frequency range for probing phase should at least cover the max frequency and the max-efficiency frequency (see Table 2). Probing more frequencies inside that range is necessary since the freq-GPUbwUt relation is nonlinear as shown in Fig. 6. Since *GEEPAFS* relies on the assumption that the probing can capture the typical behavior of the application being executed, increasing the number of samples is helpful to mitigate noise, and our experiments probe each frequency twice. However, too much sampling also impacts performance as frequency is reduced during probing, so the number of samples should be set moderately. *GEEPAFS* in our experiments probes six frequencies in the range of 720-1440 MHz on V100-MaxQ, four frequencies in 952-1530 MHz on V100, and four frequencies in 1110-1410 MHz on A100.

The global loop interval T_1, the probing interval T_2, and the number of samples M should meet the following constraint: $M \cdot T_2/T_1 << 1$. This is because the frequency is reduced during probing phase, so the portion of time spent in probing, $M \cdot T_2/T_1$, should be small enough to reduce performance impact. In addition, every time after setting frequency in the probing phase, the policy needs to wait for the tuning to accomplish and for the counter values to stabilize, so T_2 should be larger than the latency of frequency tuning and metric sampling. In our experiments on V100, we use $M = 8$, $T_2 = 200$ ms, $T_1 = 15$ s, so $M \cdot T_2/T_1 = 8 \times 0.2/15 = 10.7\%$. We did not set a larger T_1 because the application execution time in our experiments is ~2 minutes. When deploying *GEEPAFS* on a production system where applications' execution time is longer, a larger T_1 can be used, and the number of samples M can also be increased to help mitigate sampling noise.

GEEPAFS allows us to set a performance loss threshold δ. On production systems with V100/A100 GPUs, we suggest setting a δ in the range of 5%-30%. As shown in Fig. 1(a), setting $\delta = 5\%$ offers opportunities to reduce the frequency of many applications, but setting $\delta < 5\%$ is not recommended because sampling noise and execution variations can usually reach 3% or more. On the other hand, when running at the max-efficiency frequency, the performance loss of most applications are smaller than 30% (see EfficientFix in Fig. 9), so applying *GEEPAFS* with $\delta > 30\%$ is not much better than simply applying the EfficientFix policy. In addition, setting δ should consider the overall utilization level of the cluster. For GPU clusters with a high utilization above 90%, a small $\delta < 10\%$ should be chosen to avoid delaying the job queue.

Table 2. GPU servers in our experiments.

GPU specifications	DGX station	DGX-1 MaxQ	A100
Number of GPUs	4	8	1
GPU type	V100	V100-MaxQ	A100
GPU max power	300 W	163 W	400 W
GPU idle power	~39 W	~41 W	~60 W
Base frequency	1297 MHz	817 MHz	1155 MHz
Max frequency	1530 MHz	1440 MHz	1410 MHz
Max-efficiency freq.	~952 MHz	~855 MHz	~1110 MHz
GPU memory freq.	877 MHz	810 MHz	1593 MHz
GPU HBM size	32 GB	32 GB	80 GB
GPU bandwidth	898 GB/s	829.44 GB/s	2039 GB/s

5 Experimental Methodology

For evaluation, we conduct experiments on three different GPU servers, including a DGX station with 4×GPU V100, a DGX-1 MaxQ server with 8×GPU V100-MaxQ, and an A100 server with 1×GPU A100. Table 2 lists their specifications. The V100-MaxQ is a power-limited version of V100. When the power of V100-MaxQ tends to exceed its 163 W limit, its frequency is tuned down compulsorily.

Workloads. We experiment with 33 workloads listed in Table 3. These workloads belong to the following suites: NVIDIA CUDA Code Samples [29], PolyBench/GPU [9], Rodinia Benchmark Suite v3.1 [4], RL Baselines3 Zoo [40], Stable Diffusion [41], Pytorch Examples [39], Self-Supervised Vision Transformers with DINO [24]. We use these workloads to cover a wide range of GPU computing behaviors. They include GPU computing kernels (GeMM, FFT, Transpose, etc.), scientific applications (BiCG, FDTD, CFD, etc.), and deep learning applications (ViT, ResNet, Diffusion, etc.).

We adjust the loop number or problem size in these applications so that their typical execution time is about 2 minutes. Some other applications (not listed) whose execution time cannot be easily extended to 2 minutes are not used. When we compare the energy efficiency of an application running with different frequencies or policies, the loop number and problem size are kept unchanged. For applications "CNN_*", we change the size of a convolutional layer in the original implementation to adjust its total number of parameters to be 1.5, 1.8, and 2.4 million.

Among all 33 applications, 21 of them are run with a single GPU, and the other 12 are run with multiple GPUs. We run each multi-GPU application using 4 GPUs on the DGX station, and using 8 GPUs on the DGX-1 MaxQ server. The multi-GPU applications are not tested on the A100 server.

Baseline Policies. We compare *GEEPAFS* with the following baseline policies:

- **MaxPerf** simply sets a GPU to always run at the max frequency f_{max}. The performance of each application when applying the MaxPerf policy is normalized as 1.
- **EfficientFix** sets a GPU to always run at the fixed max-efficiency frequency listed in Table 2. This frequency is

Table 3. List of workloads in our experiments.

Suite	Application	Label	GPUs
NVIDIA CUDA Code Samples	BlackScholes	BScholes	1
	bandwidthTest	BWtest	1
	convolutionFFT2D	FFT	1
	cudaTensorCoreGemm	GeMM	1
	fastWalshTransform	Walsh	1
	reductionMultiBlockCG	Reduct	1
	sortingNetworks	SortNet	1
	transpose	Transpose	1
PolyBench/GPU	3MM	3MM	1
	BICG	BiCG	1
	FDTD-2D	FDTD	1
	GRAMSCHMIDT	Schmidt	1
Rodinia Suite 3.1	CFD	CFD	1
	SRAD	SRAD	1
RL Baselines3	Space Invader + PPO	RL-PPO	1
Stable Diffusion	Image Generation	Diffusion	1
Pytorch Examples MNIST	CNN (1.5M parameters)	CNN_1.5M	1
	CNN (1.8M parameters)	CNN_1.8M	1
	CNN (2.4M parameters)	CNN_2.4M	1
Pytorch Examples Language Model	LSTM	LSTM	1
	RNN	RNN	1
Pytorch Examples ImageNet	ResNet50	ResN50	4 or 8
	ResNet101	ResN101	4 or 8
	ResNet152	ResN152	4 or 8
	VGG19	VGG19	4 or 8
Self-Supervised Vision Transformers with DINO	DeiT_small	DeiT_s	4 or 8
	DeiT_tiny	DeiT_t	4 or 8
	RegNet_y_8gf	RegNet	4 or 8
	ShuffleNet_v2_x2_0	ShuffleN	4 or 8
	ViT_base	ViT_b	4 or 8
	ViT_small	ViT_s	4 or 8
	ViT_tiny	ViT_t	4 or 8
	XCiT_tiny_24_p16	XCiT	4 or 8

determined as the frequency with the highest energy efficiency averaged across applications. EfficientFix would be a good policy if performance can be ignored.

- **UtilizScale** dynamically tunes the frequency to be linearly proportional to GPU utilization. In our implementation, every 4 seconds, it temporarily sets the frequency to the maximum. After maintaining at max frequency for 200 milliseconds, it reads the GPU utilization $G(0 < G < 100\%)$ at that moment, and then sets the frequency to be $f_{max} \cdot G$. The theoretical background of this policy follows the same discussion on Eq. (2).
- **NVboost** stands for the default frequency scaling policy on NVIDIA V100/A100 GPUs. We set GPUs to apply this policy by `nvidia-smi -rac` and `nvidia-smi -rgc` commands. The details of this policy are not publicly disclosed. Roughly speaking, NVboost sets a GPU to the base frequency in Table 2 when the GPU utilization is below a threshold, and boosts it to maximum otherwise. When applying NVboost, most of our applications run at the max frequency owing to their high GPU utilization.

Previous works in Table 1 are not used for comparison because those policies mostly rely on offline profiling and code modification and do not meet our design principles.

Implementation. To evaluate *GEEPAFS* and baseline policies, we implement these policies as daemons written in C language. The daemons call NVML to set GPU frequency and read GPU counters. When deploying *GEEPAFS* on a server with multiple GPUs, we let the probing phases of *GEEPAFS* on different GPUs start and end synchronously.

To measure the performance and energy-related metrics of a frequency scaling policy, we first start the daemon which applies the policy in the background, and then run all applications one by one. When one pass of running all applications finishes, we repeat the running by 10 times and take the average. Our frequency scaling daemon is decoupled from the job scheduling script in our experiments. The daemon cannot access any information about which application is running on the server, but only has access to hardware counters.

Evaluation Metrics. The performance, average power, and energy efficiency of different policies are evaluated. The performance of an application is measured at the end of its execution, calculated as the total amount of work divided by the total execution time. The unit of performance varies across applications. Since we always compare the normalized performance, the unit is not an issue. GPU power is continuously measured during execution every $T_2 = 200$ ms using NVML. The reported power is averaged over all samples (and averaged over all GPUs for multi-GPU applications). Energy efficiency is defined as the performance divided by the average power.

We also evaluate the energy-delay product (EDP) and the energy-delay-square product (ED2P) metrics, where $EDP = E \times D$ and $ED2P = E \times D^2$. Delay D is the application execution time, and energy E equals to the average power multiplied by execution time. Both EDP and ED2P are popular metrics used in related works on energy efficiency including Refs. [2, 37, 38, 44, 47]. In Section 6.2, we show that ED2P is a better metric than EDP, because minimizing EDP cannot avoid large performance loss.

6 Results and Discussions

This section presents our results. Sections 6.1-6.2 compare different policies in detail and compare with an Oracle policy. Section 6.3 shows results on different machines. Section 6.4 discusses how a GPU sampling and tuning tool with higher resolution and lower latency will be helpful. Section 6.5 discusses the overhead and limitations of *GEEPAFS*.

6.1 Compare frequency scaling policies

Figure 8 compares the energy efficiency of applying different GPU frequency scaling policies on the DGX station with 4×V100 GPUs. The average of all workloads is on the right side of the figure. For each workload, the values are normalized by setting MaxPerf's result as 1. On average, *GEEPAFS* (with performance constraint $\delta = 10\%$) improves the energy efficiency by 26.7% compared with MaxPerf.

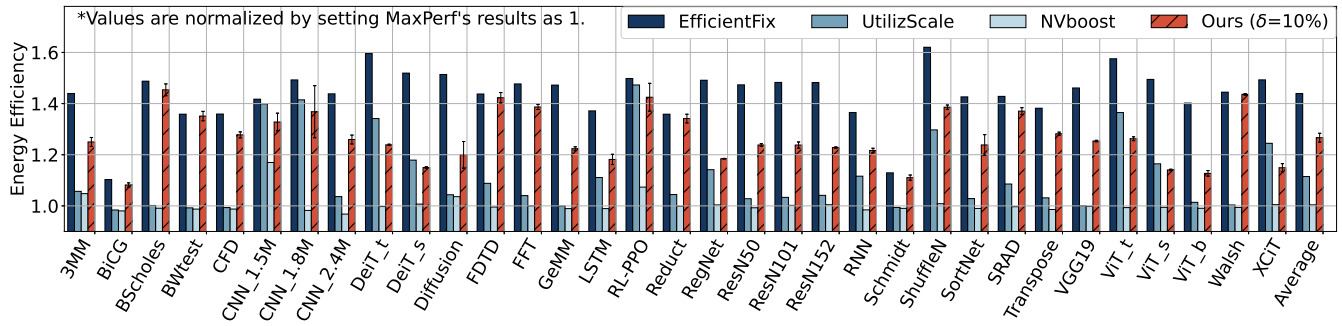


Figure 8. Energy efficiency of different GPU frequency scaling policies on a DGX station with 4×GPU V100. Our *GEEPAFS* policy with a performance constraint of 90% is shown in red, and errorbars show the standard deviation from multiple runs.

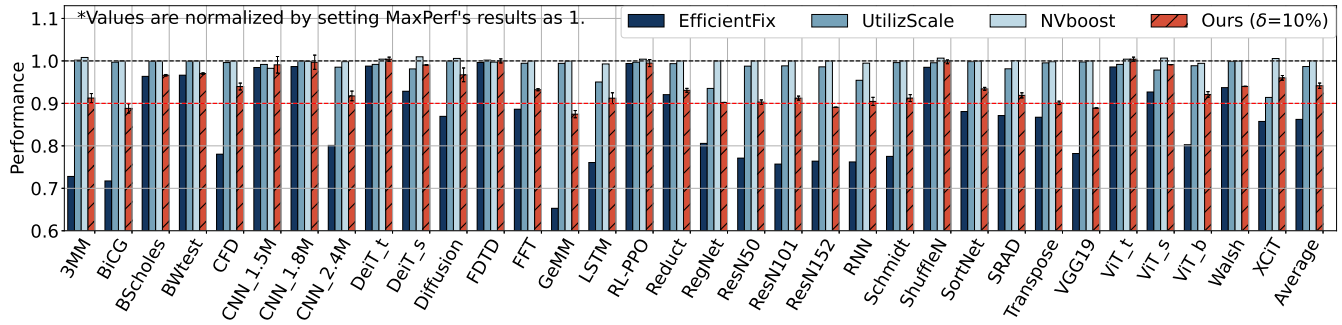


Figure 9. Application performance of different GPU frequency scaling policies on a DGX station with 4×GPU V100. Our *GEEPAFS* policy is shown by the red bars. The red dashed line shows the 90% performance constraint used in experiments, and the black dashed line shows the performance of running at max frequency (normalized as 1).

The EfficientFix policy improves energy efficiency by 43.9% on average, which is the largest among these policies. However, EfficientFix suffers from significant performance loss, which is why this policy is seldom applied in production systems. In Fig. 9, EfficientFix's average performance is lower than MaxPerf by 13.8%. Moreover, when applying EfficientFix, 21 out of 33 applications lose performance by more than 10%, 11 of them lose more than 20%, 3 of them lose more than 25%, and the largest performance loss among all applications is 34.7% when running GeMM.

On the contrary, *GEEPAFS* shows much smaller performance loss than EfficientFix. In Fig. 9, the average performance loss of *GEEPAFS* (with $\delta = 10\%$) is 5.8%, and the performance loss of almost all applications (29 out of 33) is within 10%. A few cases (BiCG, GeMM, ResN152, VGG19) show a performance loss slightly larger than 10%, which are caused by execution variations and inaccuracy in counter sampling and performance modeling. As small variations and sampling inaccuracy usually happen on servers, these cases may not be avoided completely, and Section 6.4 further discusses how this situation can be improved by faster and more accurate sampling tools. Nonetheless, the largest performance loss of *GEEPAFS* in Fig. 9 is 12.5%, which is close to δ and much lower than the 34.7% loss of EfficientFix.

The results of NVIDIA GPU's default policy NVboost are also compared in Fig. 8 and Fig. 9. Because most of the tested applications' GPU utilization is high, NVboost runs them at a frequency close to maximum, so the energy efficiency and performance are very close to 1. The average energy efficiency improvement is only 0.4%. CNN_1.5M is a case where the GPU frequency can be reduced because CNN_1.5M trains a neural network which is too small to utilize all compute units of GPU V100. In this case, our *GEEPAFS* policy sets a frequency around 1070 MHz and improves the energy efficiency by 32.8% while the performance loss is 1%. On the other hand, NVboost still sets a high frequency of 1297 MHz, and its energy efficiency improvement is only 17.0%. This shows that NVboost is far from energy-efficient.

UtilizScale performs better than NVboost. In Fig. 8, UtilizScale achieves an average energy efficiency improvement of 11.5%, with an average performance loss of 1.3%, and the largest performance loss is 8.6% when running XCiT. Although UtilizScale meets the $\delta = 10\%$ performance constraint, its average energy efficiency is lower than *GEEPAFS* by 15.2%. Especially, because UtilizScale sets frequency based on the GPU utilization metric, it cannot identify many opportunities for improving energy efficiency in high-utilization applications. For example, when running BScholes at max frequency, both GPU utilization and GPUbwUt are above 99%, but the freq-GPUbwUt curve of BScholes in Fig. 6 indicates that BScholes' performance can be maintained even if running at a low frequency. As a result, *GEEPAFS* is able to improve the energy efficiency of BScholes by 45.4% with a performance loss of only 3.4%, while UtilizScale only improves

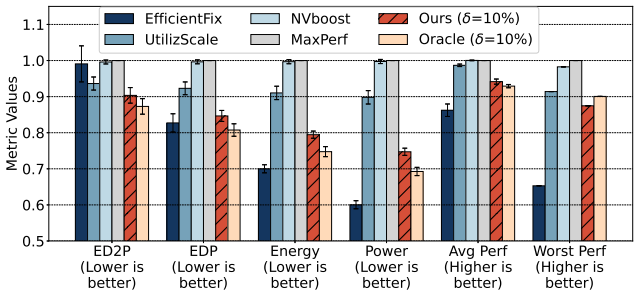


Figure 10. Summarized results of different DVFS policies on GPU V100. Each bar shows the value averaged over all applications, and errorbars show the standard error of average.

its energy efficiency by 0.2%. Similarly, for CFD, CNN_2.4M, FDTD, FFT, SRAD, VGG19, Walsh, etc., *GEEPAFS*'s improvement on energy efficiency is larger than UtilizScale's by more than 20%, as shown in Fig. 8.

6.2 Compare EDP, ED2P, and Oracle

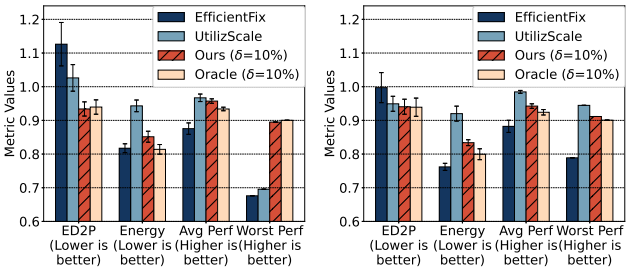
We compare the ED2P and the EDP metrics in Fig. 10. Each bar's value is averaged over all 33 applications. All values are normalized with reference to MaxPerf. In Fig. 10, EfficientFix is better than *GEEPAFS* in terms of EDP, Energy, Power. If the goal is to optimize EDP, then EfficientFix will be selected. However, we have seen that EfficientFix significantly impacts performance. In comparison, *GEEPAFS* can improve energy efficiency and reduce average power with a much better worst-case performance. Also, *GEEPAFS*'s ED2P is the lowest among all baseline policies, lower than UtilizScale's ED2P by 3.5%, and lower than EfficientFix's ED2P by 8.9%. This shows that ED2P is a better metric than EDP, as EDP does not put enough weights on delay and leads to inappropriate policy selection.

The Oracle policy is also compared in Fig. 10. The Oracle is defined as the policy that improves the energy efficiency as much as it can while ensuring the performance loss does not exceed δ, and we assume the Oracle can access any information including the freq-perf and freq-power relations in Fig. 1 even though these information can only be acquired through expensive offline profiling. Therefore, the Oracle shows the ideal improvement that could be achieved by *GEEPAFS*. As shown in Fig. 10, the worst-case performance of Oracle (with $\delta = 10\%$) is 0.9 as expected. The Oracle achieves the lowest ED2P, and it is also better than *GEEPAFS* in terms of energy by 5%. In other words, *GEEPAFS* is not far away from the ideal, but there are space for improvement.

6.3 Compare different machines and thresholds

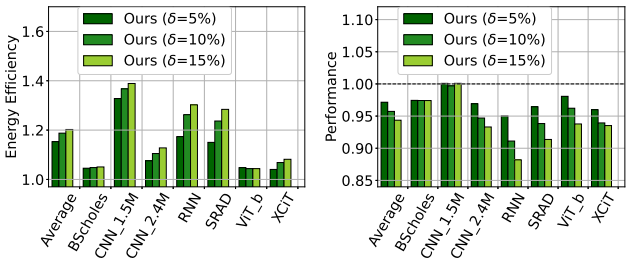
Section 6.1 presents results on the DGX station with V100 GPUs. This section shows results on the other machines.

Figure 11(a) and (b) show the averaged results of applying each policy on the DGX-1 MaxQ server and the A100 server. On both machines, *GEEPAFS* achieves the target of providing



(a) DGX-1 MaxQ server. **(b)** A100 server.

Figure 11. Summarized results on DGX-1 MaxQ and A100.



(a) Energy-efficiency. **(b)** Performance.

Figure 12. *GEEPAFS* with different performance constraints.

performance assurance and the worst-case performance loss is around 10%. *GEEPAFS* also improves energy efficiency by 18.8% on DGX-1 MaxQ and 20.2% on A100, with average performance loss of 4.3% and 5.7%, respectively. On the other hand, EfficientFix significantly reduces performance, and its worst-case performance loss is 32.4% on DGX-1 MaxQ and 21.1% on A100. UtilizScale's average performance is higher than *GEEPAFS* but it is not as energy-efficient as *GEEPAFS*. NVboost's metric values are very close to 1 and are omitted from the figure.

On both machines, *GEEPAFS* achieves lower ED2P than the baselines, and the ED2P of *GEEPAFS* on both machines are very close to the ED2P of the Oracle. In Fig. 11(a), the average ED2P of *GEEPAFS* is even slightly smaller than that of the Oracle, which is because this Oracle policy defined in Section 6.2 was not designed to optimize ED2P. This Oracle is designed to maximize energy efficiency under the constraint that application performance does not drop below 90%, but the ED2P may not be minimized at this constraint boundary. For example, from our profiling data in Fig. 1, GeMM reaches the lowest ED2P at a normalized performance of 88%, while BWtest reaches the lowest ED2P at a performance of 97%, significantly different from 90%. On the other side, due to *GEEPAFS*'s limited accuracy in online probing and modeling, *GEEPAFS* may not find the GPU frequency that exactly meets an application's 90% performance constraint (e.g., see Diffusion in Fig. 9), which in some cases can lead to a lower ED2P than the Oracle.

Figure 12 shows the energy efficiency improvement and the performance of applying *GEEPAFS* on the DGX-1 MaxQ

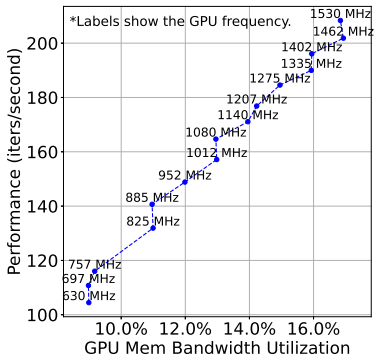


Figure 13. Artifacts appear in the GPUbwUt-performance relation of BiCG due to the 1% granularity of GPUbwUt.

server, where different performance constraints δ are compared. All values are normalized with reference to MaxPerf. "Average" shows the average value of all 33 applications, and the results for some applications are also drawn. When δ increases from 5% to 15%, application performance decreases as expected, and the energy efficiency improvement increases from 15.0% to 19.8%. On V100-MaxQ, the energy efficiency improvement is lower than the 26.7% improvement on V100 (with $\delta = 10\%$) because V100-MaxQ GPUs have a lower TDP of 163 W. As a result, V100-MaxQ encounters power throttling much more easily than V100, which reduces V100-MaxQ GPUs' performance running at the max frequency.

6.4 Demands for faster and more accurate GPU sampling/tuning tools

As mentioned in Section 6.1, our performance modeling approach is not perfect due to variations and inaccuracy in counter sampling. Part of the inaccuracy originates from the resolution of the GPU counters. In our experiments with NVIDIA V100/A100 GPUs, NVML allows us to read the GPU memory bandwidth utilization (GPUbwUt) with an accuracy of 1%. As a result, our modeling based on GPUbwUt (Section 4.3) can achieve an accuracy no better than 1%. Furthermore, because we model the performance using the ratio of two GPUbwUt values, if both two GPUbwUt values are small, the predicted performance's precision further decreases.

Figure 13 shows the GPUbwUt-performance relation of running BiCG at different GPU frequencies. The x-coordinate of a point shows the GPUbwUt value of running BiCG at a certain frequency, and we see that these values are close to integer percentages (each value is not strictly integer as it is averaged over all readings of a run). Because the granularity of GPUbwUt is 1%, the relation in the figure is a zigzag curve, which is an artifact due to the limited resolution of NVML. The actual relation should be much smoother and close to a line if the resolution of GPUbwUt is higher. Therefore, we believe that improving the accuracy of the GPUbwUt counter from the current 1% to 0.1% could be helpful in modeling performance more accurately.

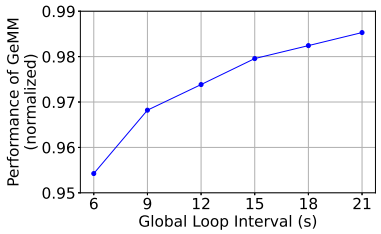


Figure 14. The impact of global loop interval on the overhead of *GEEPAFS* quantified by GeMM performance.

As mentioned in Section 4.7, we set the time interval of probing as $T_2 = 200$ ms. This T_2 cannot be reduced due to the latency in GPU frequency tuning. Our measurements on the DGX station with V100 GPUs show that the latency of changing GPU frequency using NVML is ~25 ms per GPU. Decreasing the time interval T_2 will reduce the accuracy of the counter readings, which has been observed by us in experiments. Therefore, we believe that reducing the latency in GPU frequency tuning could be much helpful in improving the performance modeling. If that latency can be reduced, both T_1 and T_2 of *GEEPAFS* can be shortened, which would enable *GEEPAFS* to react faster to changes in workload characteristics and improve the performance of *GEEPAFS*.

In addition, NVML suffers from the limitation that very few GPU metrics are supported to be sampled in an online way, and our performance modeling relies only on the GPUbwUt metric. If more metrics are supported in future online sampling tools, that could open up opportunities to design more accurate online performance modeling approaches.

6.5 Overhead and limitations of *GEEPAFS*

The overhead of *GEEPAFS* comes from actively reducing the GPU frequency during the probing phase, and the impact of this overhead depends on the ratio of time spent on probing. In Fig. 14, we evaluate the impact of the global loop interval T_1 on the overhead of *GEEPAFS* quantified by GeMM performance. In this experiment, we measure the performance of GeMM while running *GEEPAFS* in the background to activate the probing phase periodically but keeping the frequency to be maximal outside the probing phase. As a result, the performance degradation shown here is only caused by the probing phase of *GEEPAFS*, which is activated every T_1 seconds. As Fig. 14 shows, when we reduce this global loop interval from 21 to 6 seconds, the performance loss of GeMM increases gradually from 1.5% to 4.5%, which is expected because a shorter interval leads to a longer portion of time spent on probing ($M \times T_2/T_1$). Although this performance loss is inevitable because the GPU frequency needs to be reduced during the probing phase so that an application's performance sensitivity to frequency can be evaluated, the values of these parameters should be determined appropriately as discussed in Section 4.7.

Finally, it deserves mentioning that one limitation of *GEEPAFS* is that it may not work well for very-short-lived applications. As the probing phase in our experiments takes around 1 second, which may not be much shortened on current GPUs due to the latency of frequency tuning, our solution is currently more suitable to longer jobs than short-lived computations at second or sub-second level. To enable applying *GEEPAFS* on short-lived computations, faster probing is necessary to reduce its overhead. Therefore, our approach will definitely benefit from any future improvements from the hardware side that reduce the latency of the online sampling and frequency tuning of GPUs.

7 Conclusion

In this work, we propose an application-transparent GPU frequency scaling policy to improve the energy efficiency while providing performance assurance. Our policy addresses the limitations of previous works which require offline profiling and code modification on user applications. Our experiments on NVIDIA V100 and A100 GPUs show that *GEEPAFS* improves energy efficiency by 26.7% and 20.2% on average, with an average performance loss of 5.8%, and the worst-case performance loss is 12.5%. We also show that the accuracy of our performance modeling is bottlenecked by the latency of GPU frequency tuning and the resolution of GPU counters.

In future works, we plan to improve our policy for parallel workloads running at larger scales. Whether our performance modeling and frequency scaling policy work well for multi-process service and multi-instance GPU are also interesting topics to be explored. Finally, taking CPU frequency scaling and memory frequency scaling into account and optimizing these components together could open up more opportunities to improve energy efficiency.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Onur Mutlu for their insightful comments. This work was supported by Peng Cheng Laboratory Project under Grant PCL2021A13, in part by the National Natural Science Foundation of China (Grant No. 62302126), and in part by the Shenzhen Science and Technology Program (Grant No. RCBS20221008093125065).

References

[1] Yuki Abe, Hiroshi Sasaki, Shinpei Kato, Koji Inoue, Masato Edahiro, and Martin Peres. 2014. Power and Performance Characterization and Modeling of GPU-Accelerated Systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 113–122.

[2] Ghazanfar Ali, Sridutt Bhalachandra, Nicholas J. Wright, Mert Side, and Yong Chen. 2022. Optimal GPU Frequency Selection using Multi-Objective Approaches for HPC Systems. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[3] Enrico Calore, Alessandro Gabbana, Sebastiano Fabio Schifano, and Raffaele Tripiccione. 2017. Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications. *Concurrency and Computation: Practice and Experience* 29, 12 (2017), e4143.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Accessed: 2023-01. Rodinia Benchmark Suite version 3.1. https://rodinia.cs.virginia.edu/doku.php.

[5] Somdip Dey, Samuel Isuwa, Suman Saha, Amit Kumar Singh, and Klaus McDonald-Maier. 2022. CPU-GPU-Memory DVFS for Power-Efficient MPSoC in Mobile Cyber Physical Systems. *Future Internet* 14, 3 (2022).

[6] Somdip Dey, Amit Kumar Singh, Xiaohang Wang, and Klaus McDonald-Maier. 2020. User Interaction Aware Reinforcement Learning for Power and Thermal Efficiency of CPU-GPU Mobile MPSoCs. In *2020 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 1728–1733.

[7] Bishwajit Dutta, Vignesh Adhinarayanan, and Wu-chun Feng. 2018. GPU Power Prediction via Ensemble Machine Learning for DVFS Space Exploration. In *Proceedings of the 15th ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '18)*. Association for Computing Machinery, New York, NY, USA, 240–243.

[8] 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* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 52, 10 pages.

[9] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Accessed: 2023-01. PolyBench Benchmarks on GPU. https://github.com/socal-ucr/polybench-gpu.

[10] 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)*. 789–800.

[11] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás. 2015. Multi-kernel Auto-Tuning on GPUs: Performance and Energy-Aware Optimization. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 438–445.

[12] 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.

[13] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás. 2019. Modeling and Decoupling the GPU Power Consumption for Cross-Domain DVFS. *IEEE Transactions on Parallel and Distributed Systems* 30, 11 (2019), 2494–2506.

[14] Shashikant Ilager, Rajeev Muralidhar, Kotagiri Rammohanrao, and Rajkumar Buyya. 2020. A Data-Driven Frequency Scaling Approach for Deadline-aware Energy Efficient Scheduling on Graphics Processing Units (GPUs). In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 579–588.

[15] 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* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 738–753.

[16] Seyeon Kim, Kyungmin Bin, Sangtae Ha, Kyunghan Lee, and Song Chong. 2021. ZTT: Learning-Based DVFS with Zero Thermal Throttling for Mobile Devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services* (Virtual Event, Wisconsin) *(MobiSys '21)*. Association for Computing Machinery, New York, NY, USA, 41–53.

[17] Toshiya Komoda, Shingo Hayashi, Takashi Nakada, Shinobu Miwa, and Hiroshi Nakamura. 2013. Power capping of CPU-GPU heterogeneous systems through coordinating DVFS and task mapping. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 349–356.

[18] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Xiaorui Wang. 2012. GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU

Heterogeneous Architectures. In *2012 41st International Conference on Parallel Processing*. 48–57.

[19] Abhinandan Majumdar, Leonardo Piga, Indrani Paul, Joseph L. Greathouse, Wei Huang, and David H. Albonesi. 2017. Dynamic GPGPU Power Management Using Adaptive Model Predictive Control. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 613–624.

[20] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. 2020. Recalibrating global data center energy-use estimates. *Science* 367, 6481 (2020), 984–986.

[21] Xinxin Mei, Qiang Wang, and Xiaowen Chu. 2017. A survey and measurement study of GPU DVFS on energy conservation. *Digital Communications and Networks* 3, 2 (2017), 89–100.

[22] Xinxin Mei, Ling Sing Yung, Kaiyong Zhao, and Xiaowen Chu. 2013. A Measurement Study of GPU DVFS on Energy Conservation. In *Proceedings of the Workshop on Power-Aware Computing and Systems* (Farmington, Pennsylvania) *(HotPower '13)*. Association for Computing Machinery, New York, NY, USA, Article 10, 5 pages.

[23] Francisco Mendes, Pedro Tomás, and Nuno Roma. 2022. Decoupling GPGPU voltage-frequency scaling for deep-learning applications. *J. Parallel and Distrib. Comput.* 165 (2022), 32–51.

[24] Meta Research. Accessed: 2023-01. Self-Supervised Vision Transformers with DINO. https://github.com/facebookresearch/dino.

[25] Seyed Morteza Nabavinejad, Hassan Hafez-Kolahi, and Sherief Reda. 2019. Coordinated DVFS and Precision Control for Deep Neural Networks. *IEEE Computer Architecture Letters* 18, 2 (2019), 136–140.

[26] Seyed Morteza Nabavinejad, Sherief Reda, and Masoumeh Ebrahimi. 2022. Coordinated Batching and DVFS for DNN Inference on GPU Accelerators. *IEEE Transactions on Parallel and Distributed Systems* 33, 10 (2022), 2496–2508.

[27] Rajib Nath and Dean Tullsen. 2015. The CRISP Performance Model for Dynamic Voltage and Frequency Scaling in a GPGPU. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 281–293.

[28] NVIDIA. Accessed: 2023-01. NVIDIA H100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h100/.

[29] NVIDIA Corporation. Accessed: 2023-01. NVIDIA CUDA Code Samples. https://github.com/nvidia/cuda-samples.

[30] NVIDIA Corporation. Accessed: 2023-01. NVIDIA CUDA Profiling Tools Interface (CUPTI). https://developer.nvidia.com/cupti.

[31] NVIDIA Corporation. Accessed: 2023-01. NVIDIA Data Center GPU Manager (DCGM). https://developer.nvidia.com/dcgm.

[32] NVIDIA Corporation. Accessed: 2023-01. NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml.

[33] NVIDIA Corporation. Accessed: 2023-01. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.

[34] NVIDIA Corporation. Accessed: 2023-01. NVIDIA profiling tools (nvprof). https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[35] Sangyoung Park, Jaehyun Park, Donghwa Shin, Yanzhi Wang, Qing Xie, Massoud Pedram, and Naehyuck Chang. 2013. Accurate Modeling of the Delay and Energy Overhead of Dynamic Voltage and Frequency Scaling in Modern Microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 5 (2013), 695–708.

[36] Tapasya Patki, Zachary Frye, Harsh Bhatia, Francesco Di Natale, James Glosli, Helgi Ingolfsson, and Barry Rountree. 2019. Comparing GPU Power and Frequency Capping: A Case Study with the MuMMI Workflow. In *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. 31–39.

[37] Indrani Paul, Wei Huang, Manish Arora, and Sudhakar Yalamanchili. 2015. Harmonia: Balancing Compute and Memory Power in High-Performance GPUs. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 54–65.

[38] Indrani Paul, Vignesh Ravi, Srilatha Manne, Manish Arora, and Sudhakar Yalamanchili. 2013. Coordinated energy management in heterogeneous processors. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.

[39] PyTorch Developers. Accessed: 2023-01. PyTorch Examples. https://github.com/pytorch/examples.

[40] Antonin Raffin. Accessed: 2023-01. RL Baselines3 Zoo: A Training Framework for Stable Baselines3 Reinforcement Learning Agents. https://github.com/DLR-RM/rl-baselines3-zoo.

[41] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. Accessed: 2023-01. Latent Diffusion Models. https://github.com/CompVis/latent-diffusion.

[42] Muhammad Husni Santriaji and Henry Hoffmann. 2016. GRAPE: Minimizing energy for GPU applications with performance requirements. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13.

[43] Ankit Sethia and Scott Mahlke. 2014. Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 647–658.

[44] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W. Cameron. 2013. A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 673–686.

[45] Zhenheng Tang, Yuxin Wang, Qiang Wang, and Xiaowen Chu. 2019. The Impact of GPU DVFS on the Energy and Performance of Deep Learning: An Empirical Study. In *Proceedings of the Tenth ACM International Conference on Future Energy Systems* (Phoenix, AZ, USA) *(e-Energy '19)*. Association for Computing Machinery, New York, NY, USA, 315–325.

[46] TOP500. Accessed: 2023-05. TOP500 List (June 2023). https://www.top500.org/lists/top500/2023/06/.

[47] Farui Wang, Weizhe Zhang, Shichao Lai, Meng Hao, and Zheng Wang. 2022. Dynamic GPU Energy Optimization for Machine Learning Training Workloads. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2022), 2943–2954.

[48] 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.

[49] Qiang Wang, Chengjian Liu, and Xiaowen Chu. 2020. GPGPU Performance Estimation for Frequency Scaling Using Cross-Benchmarking. In *Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit* (San Diego, California) *(GPGPU '20)*. Association for Computing Machinery, New York, NY, USA, 31–40.

[50] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. 2015. GPGPU performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 564–576.

[51] Junyeol Yu, Jongseok Kim, and Euiseong Seo. 2023. Know Your Enemy To Save Cloud Energy: Energy-Performance Characterization of Machine Learning Serving. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 842–854.

[52] Pengfei Zou, Ang Li, Kevin Barker, and Rong Ge. 2020. Indicator-Directed Dynamic Power Management for Iterative Workloads on GPU-Accelerated Systems. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 559–568.

A Artifact Appendix

A.1 Abstract

This appendix provides additional information of our work. The source codes of *GEEPAFS* as well as examplar scripts to launch experiments and to post-process results are available in a publicly accessible GitHub repository. We describe the steps using our codes to launch the *GEEPAFS* policy and reproduce our experimental results.

A.2 Description & Requirements

A.2.1 How to access. Source codes are publicly available at https://github.com/zyjopensource/geepafs.

A.2.2 Hardware dependencies. Results in this work are produced with three types of servers, including a DGX station with 4×GPU V100, a DGX-1 MaxQ server with 8×GPU V100-MaxQ, and an A100 server with 1×GPU A100. Table 2 lists their GPU specifications.

A.2.3 Software dependencies. Results in this work are produced on Linux platforms. NVIDIA CUDA should be fully installed. Root privileges are needed in order to apply GPU frequency setting. Python3 is used for launching experiments and post-processing results. The specific system versions are as follows: the DGX station is installed with Ubuntu 18.04, CUDA 11.2; the DGX-1 MaxQ server is installed with Ubuntu 16.04, CUDA 10.2; the A100 server is installed with Ubuntu 20.04, CUDA 11.7.

A.2.4 Benchmarks. The workloads used in our experiments are listed in Table 3. The references to the workloads can be found in Section 5. For all the workloads, we have made small changes to extend or limit their typical execution time to be around 2 minutes.

A.3 Set-up

The following instructions run the *GEEPAFS* policy alone:

1. Open `dvfs.c` and select the correct GPU type by editing the `MACHINE` variable at the front.
2. Compile `dvfs.c` by executing `make`. The `CUDA_PATH` in the `Makefile` may need to be changed if CUDA cannot be found in its default place.
3. To run *GEEPAFS* with default settings, use the command `sudo ./dvfs mod Assure p90`, which starts the *GEEPAFS* policy with a performance constraint of 90%. This program runs endlessly by default. Press ctrl-c to stop.
4. To run a baseline policy, use the command `sudo ./dvfs mod MaxFreq`, where the name `MaxFreq` can also be replaced by `NVboost`, `EfficientFix`, or `UtilizScale`. The baseline policies are explained in Section 5.

The following instructions run *GEEPAFS* together with workloads for evaluation:

1. Edit and compile `dvfs.c` following the previous instructions.
2. Go to folder `cuda_samples/benchmarks/` and execute `make` in the subfolders to compile each applications one by one. The files in the `cuda_samples` folder are from NVIDIA CUDA Code Samples and are copied here only to facilitate the running of experiments. We have made minor changes to the application source codes to extend their execution time.
3. After compilation, launch experiments by the command `sudo python3 runExp.py`. It automatically launches the *GEEPAFS* daemon and the applications as subprocesses. Experimental results are saved into the `./output/` folder.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): *GEEPAFS* improves the energy efficiency of V100 GPUs by 26.7% on average, and the worst-case performance loss is 12.5%. This is proven by the experiment (E1) described in Section 6.1 whose results are illustrated in Figs. 8,9.
- (C2): The EfficientFix policy improves energy efficiency of V100 GPUs by 43.9% on average, and the largest performance loss of EfficientFix is 34.7%. The average energy efficiency improvement of NVboost and UtilizScale are 0.4% and 11.5%. These are proven by the experiment (E2) described in Section 6.1 whose results are illustrated in Figs. 8,9.
- (C3): *GEEPAFS* improves energy efficiency by 18.8% on DGX-1 MaxQ and 20.2% on A100, with average performance loss of 4.3% and 5.7%, respectively. This is proven by the experiment (E3) described in Section 6.3 whose results are illustrated in Fig. 11.
- (C4): When δ increases from 5% to 15%, application performance decreases as expected, and the energy efficiency improvement of V100-MaxQ increases from 15.0% to 19.8%. This is proven by the experiment (E4) described in Section 6.3 whose results are illustrated in Fig. 12.

A.4.2 Experiments.

Experiment (E1): [10+ compute-hour]: Compare the energy efficiency of *GEEPAFS* and MaxFreq.

[Preparation] For the first-time running, it takes less than 0.5 human-hour to compile *GEEPAFS* and the CUDA sample applications in the repository. More efforts are needed if workloads other than CUDA samples are to be experimented. For those workloads, please refer to their public repositories for instructions on compilation.

[Execution] In the main function of `runExp.py`, keep the policy as `policy='Assure'`, edit the experiment iteration

to be `iterations=10`, and save the file. Next, launch experiments by `sudo python3 runExp.py` and wait until it finishes. Then, in the main function of `runExp.py`, change the policy to be `policy='MaxFreq'` and launch experiments again to collect results under the MaxFreq policy.

[Results] Run `python3 postprocessing.py` to process the raw data generated by experiments. By default, it reads the examplar files `allApps_Assure_p90_2iter_demo.out`, `dvfs_Assure_p90_demo.out`, and output processed files in CSV format, which contains the average performance, average power usage, average energy efficiency values for each application from the experiments. Simply change the input file name in the script to post-process a different file. Divide the energy efficiency value while running *GEEPAFS* by the energy efficiency value while running MaxFreq, and we get the normalized energy efficiency in Fig. 8.

Experiment (E2): [10+ compute-hour]: Evaluate the performance and energy efficiency of baseline policies including EfficientFix, NVboost, UtilizScale.

[Preparation] In the main function of `runExp.py`, change the policy name, such as `policy='EfficientFix'`, and set the iteration to be `iterations=10`.

[Execution] Execute `sudo python3 runExp.py` to launch experiments and wait until it finishes. Then, repeat experiments for another policy.

[Results] Use `postprocessing.py` to process the raw data generated by the experiments. Comparing the performance and energy efficiency of the baselines with the results of MaxPerf (from E1) leads to our claim.

Experiment (E3): [10+ compute-hour]: Evaluate different policies on DGX-1 MaxQ and A100 servers.

[Preparation] Update the `MACHINE` variable in `dvfs.c` according to the GPU type selected, and compile `dvfs.c`. Update `runExp.py` to launch certain experiments. Multiple `allApps` functions can be called to run a set of experiments in a sequence.

[Execution] Execute `sudo python3 runExp.py` and wait until it finishes.

[Results] Post-process the results in similar ways as E1.

Experiment (E4): [10+ compute-hour]: Compare different performance constraints δ.

[Preparation] In the main function of `runExp.py`, use the parameters `policy='Assure'`, `iterations=10`, and set the parameter `assurance` to be one of 85, 90, and 95.

[Execution] Execute `sudo python3 runExp.py` and wait until it finishes. Repeat experiments after changing the parameter `assurance`.

[Results] Post-process the results in similar ways as E1.

A.5 Notes on Reusability

GEEPAFS can work for GPU types other than V100/A100 in principle. To work for another NVIDIA GPU, the constants in `dvfs.c` defining certain frequency values need to be updated. To work for AMD/Intel/... GPUs, the NVML API calls for metric reading and frequency tuning should also be replaced by corresponding API calls.

Our policy code `dvfs.c`, experiment script `runExp.py`, and the post-processing script `postprocessing.py` can also be reused for other experiments in measuring energy efficiency or evaluating new DVFS policies.