Model-Free GPU Online Energy Optimization

Farui Wang, Meng Hao, Weizhe Zhang, *Senior Member, IEEE*, and Zheng Wang

***Abstract*—GPUs play a central and indispensable role as accelerators in modern high-performance computing (HPC) platforms, enabling a wide range of tasks to be performed efficiently. However, the use of GPUs also results in significant energy consumption and carbon dioxide (CO2) emissions. This article presents MF-GPOEO, a model-free GPU online energy efficiency optimization framework. MF-GPOEO leverages a synthetic performance index and a PID controller to dynamically determine the optimal clock frequency configuration for GPUs. It profiles GPU kernel activity information under different frequency configurations and then compares GPU kernel execution time and gap duration between kernels to derive the synthetic performance index. With the performance index and measured average power, MF-GPOEO can use the PID controller to try different frequency configurations and find the optimal frequency configuration under the guidance of user-defined objective functions. We evaluate the MF-GPOEO by running it with 74 applications on an NVIDIA RTX3080Ti GPU. MF-GPOEO delivers a mean energy saving of 26.2% with a slight average execution time increase of 3.4% compared with NVIDIA's default clock scheduling strategy.**

***Index Terms*—GPU, DVFS, performance evaluation, energy efficiency optimization.**

I. Introduction

The general-purpose graph processing units (GPUs) have emerged as the primary computing device in many computing systems, providing acceleration for a wide range of tasks from traditional high-performance computing [1], [2], [3] to emerging workloads such as deep learning models [4]. However, the high power consumption of GPUs significantly impacts their reliability, economic viability, and operational cost, particularly in GPU clouds and high-performance computing systems. The power and cooling infrastructures for GPUs contribute to a large part of the operational cost [5], [6], making it crucial to reduce GPU power consumption without compromising application performance.

Manuscript received 16 June 2023; revised 26 August 2023; accepted 10 September 2023. Date of publication 13 September 2023; date of current version 3 April 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62202123, in part by the Joint Funds of the National Natural Science Foundation of China under Grant U22A2036, in part by the National Key Research and Development Program of China under Grant 2020YFB1406902, in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B0101360001, in part by the Fundamental Research Funds for the Central Universities under Grant HIT.OCEF.2021007, and in part by the GHfund C under Grant 202202033706. Recommended for acceptance by D. Zeng. *(Corresponding author: Weizhe Zhang.)*

Farui Wang, Meng Hao, and Weizhe Zhang are with the School of Cyberspace Science, Harbin Institute of Technology, Harbin 150000, China (e-mail: wangfarui@hit.edu.cn; haomeng@hit.edu.cn; wzzhang@hit.edu.cn).

Zheng Wang is with the School of Computing, University of Leeds, LS2 9JT Leeds, U.K. (e-mail: z.wang5@leeds.ac.uk).

Digital Object Identifier 10.1109/TSUSC.2023.3314916

The optimization of GPU energy efficiency through dynamic voltage and frequency scaling (DVFS) has been extensively studied. Previous studies have developed analytical models to predict energy consumption and performance across various DVFS settings [7], [8], [9], [10]. Additionally, some researchers have employed machine learning methods to create energy and performance prediction models for determining the optimal frequency configuration [10], [11], [12], [13]. However, these existing approaches primarily operate offline, requiring costly profiling and domain expertise to tailor decision models for new applications or different input scenarios. Consequently, their ability to generalize to diverse sets of applications and GPU architectures becomes limited.

Several approaches have been proposed to develop online methods for GPU energy optimization. Majumdar et al. [14] realize online energy saving on the AMD APU platform that supports low-overhead fine-grained profiling functions. ODPP [15] uses coarse-grained features to build prediction models and an online power-performance management framework on the NVIDIA GPU platform. GPOEO [16] utilizes a robust phase detection algorithm, trains prediction models offline, and requires modifying the application source code to optimize GPU energy efficiency. The above three methods are based on predictive models. Collecting data under different clock configurations and training predictive models can be time-consuming, yet the trained model can be outdated for new workloads. DEPO [17] is a mode-free method that exploits CUDA kernel counting to evaluate application performance and save GPU energy consumption. As we will show later in the article, only using CUDA kernel counting is insufficient to support accurate performance evaluations. As such, we need to have a better metric to be used by model-free methods for accurately evaluating application performance.

In this article, we propose a novel model-free GPU online energy efficiency optimization framework called MF-GPOEO. This framework avoids the limitations of our previous GPOEO [16]. MF-GPOEO neither relies on predictive models nor requires modifying source code. After setting several environment variables, MF-GPOEO can automatically collect kernel activity information and find the optimal GPU clock configuration.

MF-GPOEO performs GPU energy optimization using runtime information. First, it measures the kernel activities using NVIDIA's default clock scheduling strategy as a performance baseline. Next, it collects similar data under different clock configurations. The framework then calculates a synthetic relative performance index, which enables the evaluation of application performance degradation associated with a specific clock configuration. This assessment is based on a comparison of GPU kernel execution times and the durations between kernels under different clock configurations. Utilizing the synthetic performance index and the recorded average power consumption, MF-GPOEO employs a proportional-integral-derivative (PID) controller. The PID controller iteratively explores various GPU clock configurations, guided by user-defined objective functions, with the aim of improving GPU energy efficiency. Once the PID controller identifies an optimal configuration, and MF-GPOEO switches to a low-overhead workload change detection mode. If a change in workload is detected, the framework restarts the PID controller and repeats the optimization process until a new optimal configuration is determined. This iterative approach allows MF-GPOEO to adapt to varying workloads and enhance GPU energy efficiency continually.

We evaluate MF-GPOEO by applying it to 74 machine learning and scientific computing applications on an NVIDIA RTX3080Ti GPU. MF-GPOEO can reduce GPU energy consumption by 26.2% on average over NVIDIA's default GPU scheduling policy. Importantly, this energy reduction comes with minimal impact on execution time, as all observed increments remain below 5% (with an average of 3.4%) when compared to NVIDIA's default clock scheduling strategy.

This article makes the following contributions:

- It presents a novel model-free online GPU application performance evaluation method for DVFS. The method utilizes GPU kernel activity information measured online to derive a synthetic performance index and evaluate the application performance under a specific clock configuration.
- It demonstrates how a simple but efficient PID controller can be used for GPU energy efficiency optimization. This controller guides clock adjustment to optimize energy efficiency without compromising the performance constraint.
- It implements a model-free GPU online energy efficiency optimization framework, which does not require offline model training and parameter tuning. The framework further improves the practicability and usability of GPU energy efficiency optimization.

Online Material. The MF-GPOEO framework is publicly available at https://github.com/ruixueqingyang/MF-GPOEO.

II. Background and Motivation

A. Background

1) Kernel Activity Profiling: Our work utilizes the NVIDIA CUPTI [18] library to profile and collect kernel activity information, including start timestamp, end timestamp, kernel name, stream ID, grid size, block size, etc. The kernel activity information is defined as the structure `CUpti_ActivityKernel7` (CUDA 11.7 [19]) in the header file `cupti_activity.h`. With the information, we can calculate the execution time of each kernel and the gap duration times between successive kernels. We can also distinguish and classify different kinds of kernels and gaps. These can help us directly know application performance and derive a synthetic performance index.

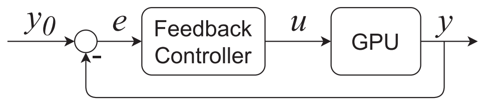


Fig. 1. Feedback controller.

TABLE I
Comparison Against the Existing Studies

Study	Work Mode	Model	Open Source
Wang [9]	offline	analytical model	√
Fan [12]	offline	SVR	×
Guerreiro [8]	offline	analytical model	√
Guerreiro [13]	offline	LSTM, FNN	√
Majumdar [14]	online	random forest	×
ODPP [15]	online	MLP	×
GPOEO [16]	online	XGBoost	√
DEPO [17]	online	model-free	× *
Ours	online	model-free	√

* Only the code of DEPO for CPU [20] is available, but not the code of DEPO for GPU [17] that was compared in our paper. when this paper was submitted.

2) Feedback Controller: Feedback controllers are widely used to make the system state reach and maintain a particular reference value. As shown in Fig. 1, a feedback controller calculates the error e between the setpoint value y_0 and measured process value y, then uses the error e to calculate a new process input u to adjust the measured process value y back to the desired setpoint y_0. We use a PID controller, a commonly used feedback controller, to control performance loss and optimize GPU energy efficiency. The PID controller requires an accurate, direct, and comparable performance index as the setpoint value and the measured process value.

B. Motivation

1) Complex Model Construction: Most existing GPU energy optimization methods construct predictive or analytical models to estimate energy consumption and performance under different DVFS configurations. We compare some recent methods with our work in Table I. Model retraining or tuning is inevitable in some scenarios for these model-based methods. For example, users migrate models to new GPU platforms, or the training data set does not cover new kinds of applications well. Collecting energy consumption and performance data with DVFS and retraining or tuning models are tedious and require much domain-specific knowledge, even if some methods are open source. Our model-free method may be a good alternative or supplement to these existing methods.

2) Using Kernel Activity Information: DEPO [17] evaluates application performance by CUDA kernel counting. DEPO treats all kernels are the same, regardless of differences in kernel name, input data size, stream ID, grid size, block size, and other factors. However, this assumption is inconsistent with reality, and sometimes DEPO shows large evaluation errors. In Fig. 2, we provide examples of three AIBench applications. On AI_I2IP and AI_OBJ, DEPO fails to reflect the trend of actual performance loss under varying GPU power caps, which leads to non-optimal solutions. On AI_TS, DEPO grossly underestimates

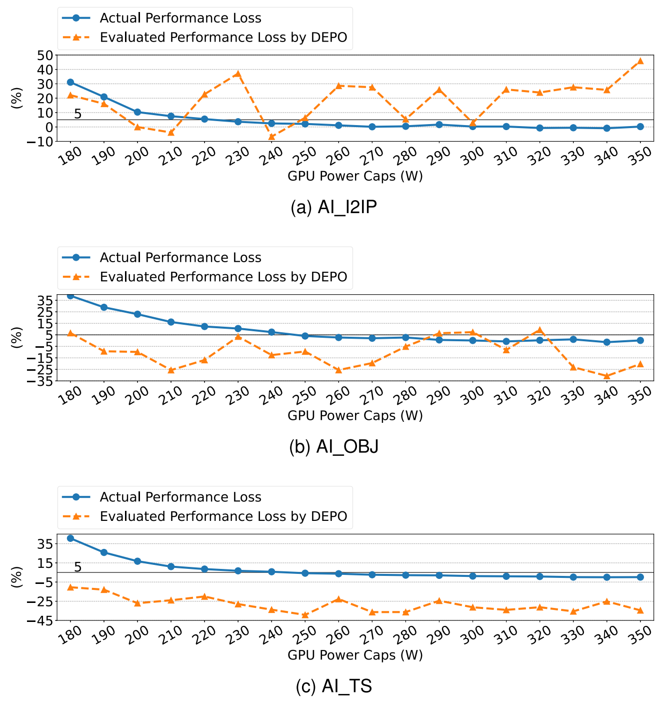


Fig. 2. Performance losses evaluated by DEPO under varying GPU power caps on three AIBench applications.

performance losses under all GPU power caps, leading to the performance loss exceeding the user-specified constraint. These significant errors result in poor energy efficiency optimization results, which we analyze in Section V-C1.

Analyzing kernel activity data, we have identified several inconsistencies with DEPO's assumption that cause the above huge errors. ① There are various types of kernels in these three applications: over 330, 990, and 480, separately. ② The average execution times of various types of kernels span multiple orders of magnitude (10^{-6} to 10^{-2} second). ③ No type of kernel can dominate the application running time in terms of total execution time. ④ Furthermore, the breakdowns of various types of kernels are different in multiple measurements. Hence, we will exploit kernel activity information and consider kernel name, input data size, stream ID, grid size, block size, and other interference factors to design a more accurate synthetic GPU performance index.

3) Satisfying Performance Loss Constraints: The accuracy of model-based methods depends on the training process and model design. In general, such methods hardly achieve low prediction errors on all applications. These methods also cannot use runtime information to discover and eliminate possible model errors. So, they may not be able to satisfy performance loss constraints strictly. For example, ODPP [15] and GPOEO [16] cannot satisfy the performance loss constraint on some applications. Based on kernel counting, DEPO [17] cannot strictly obey the performance loss constraint either. With kernel activity information and feedback control, our energy optimization method can dynamically use runtime information and reduce errors to satisfy the performance loss constraint on all applications. Improved performance loss control enhances the practicality of energy efficiency optimization.

TABLE II
DESCRIPTION OF PARAMETERS

Notation	Description
f_{obj}	objective function
$index_{perf}$	synthetic performance index
$index_{cover}$	database coverage index
$Power$	measured average GPU power
W	total application workload
$Time$	application execution time
Con_{perf}	performance loss constraint
$SMGear_i$	one SM gear represents an SM clock frequency
$MemGear_j$	one memory gear represents a memory clock frequency
K_P	proportional gain of PID controller
K_I	integral gain of PID controller
K_D	derivative gain of PID controller
Th_{exe}	distribution threshold for kernel execution time
Th_{gap}	distribution threshold for gap duration time
Th_{cover}	performance database coverage threshold
Th_{ctrl}	control error threshold of PID controller
Th_{EV}	distance threshold between execution vectors

III. Overview

A. Problem Formalization

Our work aims to improve GPU energy efficiency within a given performance loss constraint by adjusting the SM and memory clock frequency configurations. Our objectives can be formulated as:

$$\begin{aligned} \min \quad & F = f_{obj}\left(Power, index_{perf}\right) \\ \text{s.t. } & index_{perf} < 1 + Con_{perf} \\ & SMGear_i \in \{SMGear_1, \ldots, SMGear_m\} \\ & MemGear_j \in \{MemGear_1, \ldots, MemGear_n\} \end{aligned} \tag{1}$$

$$index_{perf} = \frac{Time}{Time_{default}} \tag{2}$$

Table II lists the notations and parameters used in this article. The $index_{perf}$ is defined as the ratio of the application running time under a specific clock configuration ($Time$) to the running time under the default clock configuration ($Time_{default}$), as shown in (2). For a particular workload, $Time_{default}$ is constant. Thus, $Time$s under varying clock configurations are comparable once we determine $index_{perf}$s with Algorithm 3 in Section IV-B. We obtain $Power$ through measurement. Users can arbitrarily determine an objective function, such as average $Power$, total $Energy = Power \times Time$, $EDP = Energy \times Time$, and $ED^2P = Energy \times Time^2$, to guide energy-efficiency optimization. We select the total $Energy$ consumption as the objective function.

Tables III and IV define two fundamental data units used in this work. We use the CUPTI [18] library and the NVML [21] library to collect raw data of GPU kernels and gaps between kernels. We gather kernels with the identical $KnlID$, defined as (3), to get the data contained within one $KnlData$, shown in Table III. $Name$, $StreamID$, $Block$, and $Grid$ represent

TABLE III
THE DATA CONTAINED WITHIN ONE $KnlData$

Notation	Description
$KnlID$	the ID of one kind of kernel
$count_{exe}$	number of such kernels
$avgT_{exe}$	average execution time of such kernels
$sumT_{exe}$	total execution time of such kernels
$stdT_{exe}$	standard deviation of kernel execution time
Stb_{exe}	stability index of such kernels
$Power$	measured average GPU power

TABLE IV
THE DATA CONTAINED WITHIN ONE $GapData$

Notation	Description
$GapID$	the ID of one kind of gap
$count_{gap}$	number of such gaps
$avgT_{gap}$	average gap time of such gaps
$sumT_{gap}$	total gap time of such gaps
$stdT_{gap}$	standard deviation of gap duration time
Stb_{gap}	stability index of such gaps

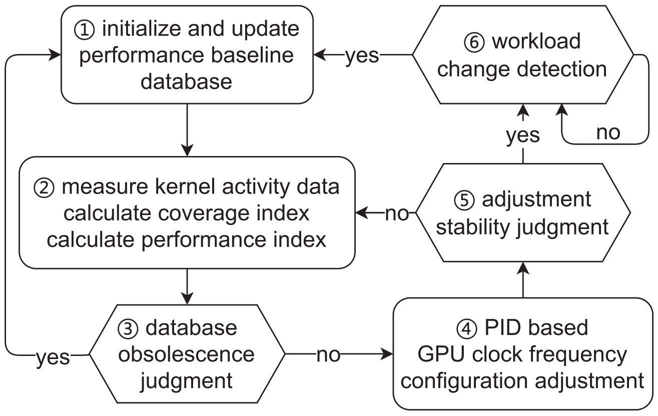

① initialize and update performance baseline database
⑥ workload change detection
yes
no
yes
② measure kernel activity data calculate coverage index calculate performance index
⑤ adjustment stability judgment
no
③ database obsolescence judgment
yes
no
④ PID based GPU clock frequency configuration adjustment

Fig. 3. Overview of MF-GPOEO.

a kernel's name, GPU stream ID, block size, and rounded grid size (7), respectively. Similarly, we gather gaps with the identical $GapID$, defined as (4), to compute the data contained within one $GapData$, shown in Table IV. The lower subscripts $Prev$ and $Back$ represent the two GPU kernels before and after the gap. The $KnlData$s and $GapData$s collected in one measurement form a data set $MeasureData$, defined as (5). The performance baseline database ($DataBase$) is a fusion of multiple $MeasureData$s.

$$KnlID = \{Name, StreamID, Block, Grid\} \tag{3}$$

$$GapID = \{StreamID, Name_{Prev}, Block_{Prev}, \\ Grid_{Prev}, Name_{Back}, Block_{Back}, Grid_{Back}\} \tag{4}$$

$$MeasureData = \{KnlData_i, GapData_j\} \tag{5}$$

B. Framework Overview

The framework of MF-GPOEO is shown in Fig. 3. We measure and collect several $MeasureData$s under NVIDIA's default clock scheduling strategy, merge these data, and update the performance baseline $Database$ with Algorithm 2 (①). Meanwhile, we also determine the measurement duration. Then, we collect the $MeasureData$ under the specific GPU clock configuration given by the PID controller, match these data within the performance baseline $Database$, and calculate the $index_{cover}$ and $index_{perf}$ with Algorithms 1 and 3 (②). If $index_{cover}$ is less than Th_{cover} or after every few adjustments, we think the $Database$ is outdated and update it (③). Otherwise, we feed the $index_{perf}$ into the PID controller, get a new clock configuration with Algorithm 4 (④), and judge the stability of recent clock configurations with Algorithm 5 (⑤). If unstable, we turn to a new round of PID control (②③④). Otherwise, we monitor low-overhead execution vector (EV) data with Algorithm 6 (⑥) until we detect a workload change and then restart our MF-GPOEO system from ①.

IV. DESIGN AND IMPLEMENT

MF-GPOEO targets repeatedly running kernels that are commonly found in scientific computing and artificial intelligence applications. We measure and compare the corresponding $KnlData$s and $GapData$s under different GPU clock configurations to determine the relative performance loss. Then we use a PID controller to find the optimal configuration and improve energy efficiency within the performance loss constraint.

A. Performance Data Measurement and Baseline Database Construction

To compare and obtain relative performance loss, we first collect $MeasureData$s under NVIDIA's default clock scheduling strategy and use these data to build the $DataBase$ as the performance baseline.

1) Performance Data Collection: We should collect the same kind of kernels or gaps together to compare the relative performance under different configurations. Naturally, these kernels or gaps should have the same name, GPU stream ID, block size, and grid size.

We also consider other attributes, such as the input data volumes of kernels. In GPU programming practice, developers usually use different grid and block sizes rather than loops to handle different input data sizes. The block size is typically an integer multiple of the scheduled thread group size (32 threads in the NVIDIA CUDA warp [19] or 32/64 threads in the AMD OpenCL wavefront [22]). The block size is not flexible, and the grid size is variable. Therefore, kernels with the same grid size, block size, name, and stream ID are very likely to have the same input data size and similar execution time.

Further, considering the SIMT architecture of GPUs, we can relax the requirement for the grid size. According to the previous paragraph, we can approximate the input data size as $Size_{in} = N \times GridSize \times BlockSize$, where N is the amount of data processed by each GPU core. $Size_{in}$ may not be evenly divided by $N \times NumCores$ (number of GPU cores). Thus, when the GPU executes the remainder workload ($Size_{in} \mathrm{mod}(N \times NumCores)$), part of GPU cores will be idle and wait for other cores if we temporarily ignore the concurrent kernel execution. Regardless of the number of idle cores, the execution times of the remainder workloads of the same GPU kernel are very close to each other. Therefore, the grid sizes of two kernels only need to satisfy (6) to ensure that the two kernels' execution times are comparable. $Grid_i$ and $Grid_j$ are determined by (7).

$$Grid_i = Grid_j \tag{6}$$

$$Grid = \left\lceil \frac{GridSize \times BlockSize}{NumCores} \right\rceil \times \frac{NumCores}{BlockSize} \tag{7}$$

Other factors also affect the kernel execution time, such as concurrent kernel execution, dependencies between kernels, dependencies between the CPU and GPU, and hardware resource competition within GPU. We cannot directly handle these factors because the kernel activity data collected with CUPTI do not contain the relevant information. For simplicity, we think these interference factors are negligible if the dispersion of execution time or gap duration is concentrated enough. So we derive the stability indexes Stb_{exe} and Stb_{gap} from the standard deviations $stdT_{exe}$ and $stdT_{gap}$, as shown in (8). If Stb_{exe} or Stb_{gap} is less than the threshold Th_{exe} or Th_{gap}, we believe the interference factors are negligible, and $avgT_{exe}$ or $avgT_{gap}$ is comparable, respectively.

$$\begin{aligned} Stb_{exe} &= stdT_{exe}/avgT_{exe} \\ Stb_{gap} &= stdT_{gap}/avgT_{gap} \end{aligned} \tag{8}$$

Consequently, we collect kernels or gaps with the same $KnlID$ or $GapID$ to build comparable $KnlData$ or $GapData$, respectively. A $MeasureData$ consists of $KnlData$s and $GapData$s collected in one measurement. In our implementation, we pause the measurement when the GPU is idle to improve the data stability.

2) Coverage Index: We expect each measured $KnlData_i$ to have a corresponding $KnlData_j$ in $DataBase$, satisfying $KnlID_i = KnlID_j$. A similar logic applies to $GapData$. According to our experimental results, when most $KnlData_i$s have corresponding $KnlData_j$s, most $GapData_i$s also have corresponding $GapData_j$s. Thus, we design $index_{cover}$ derived from Algorithm 1 to evaluate the ratio of $KnlData_i$s with corresponding $KnlData_j$s. The $index_{cover}$ can represent the completeness of $DataBase$ and guide the construction of $DataBase$. For each $KnlData_i$ in $MeasureData$, if we can find a corresponding $KnlData_j$ in $DataBase$, we add a $sumT_{exe,i}$ to T_{cover} (line 2-5). Each $sumT_{exe,i}$ is added to T_{total} (line 6). $index_{cover}$ is the quotient of T_{cover} and T_{total} (line 9).

3) Performance Baseline Database Initialization and Update: We design Algorithm 2 to initialize and update the performance baseline $DataBase$ and determine the measurement $Duration$. If $index_{cover}$ is less than Th_{cover}, we measure and collect $MeasureData$ under the default GPU clock strategy within $Duration$ (line 2-3). Then we update $index_{cover}$ with Algorithm 1 (line 4). For each $KnlData_i$ in $MeasureData$, if we find a corresponding $KnlData_j$, we update $KnlData_j$ with $KnlData_i$, otherwise directly store $KnlData_i$ in $DataBase$ (line 5-11). We

Algorithm 1: Coverage Index Calculation Algorithm.

```
Input: performance baseline $DataBase$,
  $MeasureData$ collected in one measurement
Output: coverage index $index_{cover}$
1: $T_{cover} = 0$; $T_{total} = 0$
2: for each $KnlData_i \in MeasureData$ do
3:   if $\exists KnlData_j \in DataBase$ such that
     $KnlID_i = KnlID_j$ then
4:     $T_{cover} += sumT_{exe,i}$
5:   end if
6:   $T_{total} += sumT_{exe,i}$
7: end for
8: if $T_{total} > 0$ then
9:   $index_{cover} = T_{cover}/T_{total}$
10: else
11:   $index_{cover} = 0$
12: end if
13: return $index_{cover}$
```

also use a similar logic to process $GapData$s (line 12-18). If $index_{cover}$ is less than Th_{cover}, we increase $Duration$ (19-21). When Th_{cover} is satisfied, we think $DataBase$ has covered enough $KnlData$s and return $DataBase$ and $Duration$.

B. Synthetic Performance Index

Now we can derive the synthetic performance index ($Index_{perf}$) by comparing $KnlData$s and $GapData$s under a specific GPU clock configuration to the corresponding data in the performance baseline $DataBase$. We consider the running time of the measured workloads as a combination of kernel execution times and gap duration times in multiple streams. We can find the matching $KnlData_m$s and $GapData_n$s in $DataBase$ for the $KnlData_j$s and $GapData_k$s in $MeasureData$ under a specific GPU clock configuration. If the $KnlData_m$ and $KnlData_j$ satisfy (9), we think they are matched and stable enough for comparison. A similar logic applies to the $GapData_n$ and $GapData_k$ (10). Then, we derive the running time baseline of the same workload under the default clock strategy by calculating the weighted sum of all kernel execution times of $KnlData_j$s and gap duration times of $GapData_k$s. The relative performance index is the ratio of two running times under different configurations.

$$\begin{aligned} &KnlID_m = KnlID_j \\ &\quad Stb_{exe,m} < Th_{exe} \quad Stb_{exe,j} < Th_{exe} \end{aligned} \tag{9}$$

$$\begin{aligned} &GapID_n = GapID_k \\ &\quad Stb_{gap,n} < Th_{gap} \quad Stb_{gap,k} < Th_{gap} \end{aligned} \tag{10}$$

Based on the above ideas, we design the synthetic performance index algorithm (Algorithm 3) in detail. First, we classify $MeasureData$ by streams (line 1). We process kernel execution time data through line 6 to 12. In each stream, we only consider matched and stable $KnlData_m$s and $KnlData_j$s (line 7). We

Algorithm 2: Performance Baseline Database Update Algorithm.

```
Input: empty or outdated performance baseline $DataBase$, initial value of measurement $Duration$, $Th_{cover}$
Output: updated performance baseline $DataBase$, measurement $Duration$
1: $index_{cover} = 0$
2: while $index_{cover} < Th_{cover}$ do
3:   Measure and collect $MeasureData$ within $Duration$ under the default GPU clock strategy
4:   $index_{cover} =$ Algorithm 1($DataBase$, $MeasureData$)
5:   for each $KnlData_i \in MeasureData$ do
6:     if $\exists KnlData_j \in DataBase$ such that $KnlID_i = KnlID_j$ then
7:       $KnlData_j = KnlData_i$
8:     else
9:       Store $KnlData_i$ into $DataBase$
10:    end if
11:  end for
12:  for each $GapData_i \in MeasureData$ do
13:    if $\exists GapData_j \in DataBase$ such that $GapID_i = GapID_j$ then
14:      $GapData_j = GapData_i$
15:    else
16:      Store $GapData_i$ into $DataBase$
17:    end if
18:  end for
19:  if $index_{cover} < Th_{cover}$ then
20:    $Duration = 2 \times Duration$
21:  end if
22: end while
23: return $DataBase$, $Duration$
```

Algorithm 3: Synthetic Performance Index Algorithm.

```
Input: data collected in one measurement $MeasureData$, baseline $DataBase$, $Th_{exe}$, $Th_{gap}$
Output: synthetic performance index $index_{perf}$
1: Classify $MeasureData$ into subsets $\{StreamData_i = \{KnlData_j, GapData_k\}\}$ by different $StreamID$s
2: for each $StreamData_i \in \{StreamData_i\}$ do
3:   $accuExeT_i = 0;\ accuGapT_i = 0$
4:   $wghtExeT_{base,i} = 0;\ wghtExeT_{mesr,i} = 0$
5:   $wghtGapT_{base,i} = 0;\ wghtGapT_{mesr,i} = 0$
6:   for each $KnlData_j \in StreamData_i$ do
7:     if $\exists KnlData_m \in DataBase$ satisfies Equ. 9 then
8:       $wghtExeT_{base,i} += count_{exe,j} \times avgT_{exe,m}$
9:       $wghtExeT_{mesr,i} += count_{exe,j} \times avgT_{exe,j}$
10:    end if
11:    $accuExeT_i += sumT_{exe,j}$
12:  end for
13:  for each $GapData_k \in StreamData_i$ do
14:    if $\exists GapData_n \in DataBase$ satisfies Equ. 10 then
15:      $wghtGapT_{base,i} += count_{gap,k} \times avgT_{gap,n}$
16:      $wghtGapT_{mesr,i} += count_{gap,k} \times avgT_{gap,k}$
17:    end if
18:    $accuGapT_i += sumT_{gap,k}$
19:  end for
20: end for
21: $sumT_{base} = 0;\ sumT_{mesr} = 0$
22: for each $i \in \{1, 2, \ldots, |\{StreamData_i\}|\}$ do
23:   $BaseT_{exe,i} = accuExeT_i \times \frac{wghtExeT_{base,i}}{wghtExeT_{mesr,i}}$
24:   $BaseT_{gap,i} = accuGapT_i \times \frac{wghtGapT_{base,i}}{wghtGapT_{mesr,i}}$
25:   $sumT_{base} += BaseT_{exe,i} + BaseT_{gap,i}$
26:   $sumT_{mesr} += accuExeT_i + accuGapT_i$
27: end for
28: return $index_{perf} = sumT_{mesr}/sumT_{base}$
```

compute cumulative weighted execution times (line 8-9). The weight is kernel $count_{exe,j}$ in $KnlData_j$. These two accumulated values are comparable and reflect the performance difference between the default clock strategy and the specific clock configuration. Then we add all kernel execution times together (line 11). We also use a similar logic to handle $GapData$s (line 13-19).

Later, we calculate the relative performance index of each stream and the synthetic performance index of all streams (line 21-28). For each stream, we derive the execution time baseline ($BaseT_{exe,i}$) under the default clock strategy of the measured workloads (line 23). We derive the gap time baseline ($BaseT_{gap,i}$) similarly (line 24). We sum the data of all streams to obtain the sum running time baseline ($sumT_{base}$) under the default clock strategy and the sum running time ($sumT_{mesr}$) under the specific clock configuration (line 25-26). Finally, the synthetic relative performance index ($index_{perf}$) is the ratio of $sumT_{mesr}$ and $sumT_{base}$. We can get the synthetic relative performance loss by $index_{perf} - 1$.

C. PID Controller

We use a PID controller to find the optimal clock configuration while ensuring the performance loss constraint. This work aims to save energy, so we avoid using heavy-overhead learning-based methods such as DDQN. The execution time of the application is usually monotonically non-increasing as the SM or memory clock gear decreases [23]. In this scenario, the PID controller is suitable for controlling the performance loss. According to our experimental results and the related work [23], the relationship between commonly used energy optimization objectives (average $power$, total $energy$, EDP, and ED^2P) and clock gears is generally a convex function. Thus, we can find the optimal one and ensure the performance loss constraint when the PID controller decreases the clock gear.

Equation (11) defines the classic PID controller in the time domain [24], where $u(t)$ is the controller output, $e(t)$ is the control error, K_P is the proportional gain, K_I is the integral gain, and K_D is the derivative gain. In the scenario of GPU energy

Algorithm 4: PID Control Algorithm.

```
Input: performance loss constraint $Con_{perf}$, synthetic
  performance index $index_{perf}$, measured average
  $Power$
Output: clock configuration $gear_k$, percentage control
  error $err_{ctrl}$
1: calculate $gear_k$ and $err_k$ with (13)
2: calculate $f_{obj}$ with (1)
3: if $err_k > 0$ and $f_{obj} > f_{obj,prev}$ then
4:   if $gear_k \neq gear_{k-1}$ then
5:     $gear_k = gear_{k-1}$
6:   else
7:     $gear_k + = 1$
8:   end if
9: end if
10: $err_{ctrl} = err_k / Con_{perf}$
11: $err_{k-1} = err_k$; $gear_{k-1} = gear_k$; $f_{obj,prev} = f_{obj}$
12: return $gear_k$ and $err_{ctrl}$
```

optimization, we discretize (11) because the clock adjustment is discrete. The integral and derivative terms are approximated with (12), where k is the number of clock adjustments.

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau)\, d\tau + K_D \frac{de(t)}{dt} \tag{11}$$

$$K_I \int_0^t e(\tau)\, d\tau \approx K_I \sum_{i=1}^{k} err_i$$

$$K_D \frac{de(t)}{dt} \approx K_D (err_k - err_{k-1}) \tag{12}$$

$$\begin{aligned} u_k &= K_P err_k + K_I \sum_{i=1}^{k} err_i + K_D (err_k - err_{k-1}) \\ err_k &= (index_{perf} - 1) - Con_{perf} \\ gear_k &= gear_{k-1} + u_k \end{aligned} \tag{13}$$

Then, we get the discrete PID controller as (13). The output is the clock gear increment (u_k) for the next clock gear ($gear_k$). The performance loss control error (err_k) is the difference between the evaluated performance loss $index_{perf} - 1$ and the performance loss constraint Con_{perf}. Based on (13), we design Algorithm 4. We start with a high clock gear. The PID controller decreases the clock gear to save energy and approach Con_{perf}. If the performance loss violates the constraint or the objective function value increases, we roll back or increase the current clock $gear_k$ (line 3-9). We initialize err_{k-1}, $gear_{k-1}$, and $f_{obj,prev}$ with zero, a high gear, and a big value, respectively, when we call the algorithm for the first time.

D. Stability Judgment

We can reduce the profiling overhead by stopping profiling and controlling when the control is stable. We will restart profiling and controlling when the application workload changes. Thus, we design Algorithm 5 to judge control stability and Algorithm 6 to use low overhead features to detect workload change.

Algorithm 5: Control Stability Judgment Algorithm.

```
Input: control error threshold $Th_{ctrl}$, recent control
  error trace $Tc_{err} = \{err_{ctrl,i}\}$, recent energy gear trace
  $Tc_{gear} = \{gear_i\}$
Output: stable energy configuration gear $gear_{stb}$
1: $avgLoss_{perf} = mean(\{err_{ctrl,i}\})$
2: $avgErr_{ctrl} = mean(\{abs(err_{ctrl,i})\})$
3: $range = max(\{gear_i\}) - min(\{gear_i\})$
4: if $avgLoss_{perf} < 0$ and $avgErr_{ctrl} < Th_{ctrl}$ then
5:   $gear_{stb} = mean(\{gear_i\})$
6: else
7:   $gear_{stb} = -1$
8: end if
9: return $gear_{stb}$
```

1) Control Stability Judgment: The control stability judgment algorithm (Algorithm 5) is simple but effective. We calculate the average value of recent clock gears (line 5) and return it as the optimal configuration (line 9) if the following two conditions are met (line 4). ① The mean percentage error of recent controls should be less than zero, meaning that the performance loss satisfies Con_{perf}. ② The mean absolute percentage error of recent controls should be less than Th_{ctrl}. Otherwise, we return -1 (line 7).

2) Workload Change Detection: While stopping kernel activity profiling and controlling, we monitor application workload change using the phase change detection algorithm (Algorithm 6). We use NVML [21] to measure execution vectors (EVs) and compare two successive EVs to detect phase change. The EV consists of features (ev_i) with negligible profiling overhead, such as power, SM utilization, memory utilization, SM frequency, and memory frequency. We calculate the percentage distance between each pair of features and accumulate it (line 1-4). Next, we calculate the mean distance ($dist$) and compare it with the distance threshold (Th_{EV}) (line 5-11). If $dist > Th_{EV}$, we think phase and workload changes happen and restart the PID controller. Otherwise, we repeat the phase detection.

E. Algorithm Framework

Based on previous algorithms, we design the MF-GPOEO algorithm framework (Algorithm 7). The MF-GPOEO framework finds optimal GPU SM and memory clock configurations in a similar way. We only introduce the optimization process of the SM clock configurations for convenience. First, we initialize $DataBase$ and determine measurement $Duration$ using Algorithm 2 (line 1). We initialize $gear_{opt}$ with a high clock gear (line 2). Then we optimize energy efficiency during application execution (line 3-29). We set $gear_{opt}$, collect $MeasureData$ within $Duration$ and calculate $index_{cover}$ using Algorithm 1 (line 4-6). If $index_{cover} > Th_{cover}$, we calculate $index_{perf}$ with Algorithm 3 (line 8). The PID controller gives a new $gear_{opt}$ and a control error (line 9). We use $gear_{opt}$ and err_{ctrl} to

Algorithm 6: Phase Change Detection Algorithm.

```
Input: execution vector distance threshold $Th_{EV}$, two execution vectors measured successively $EV_1$, $EV_2$
Output: phase change flag $flag_{phase}$
1: $sum = 0$
2: for each $i \in \{1, 2, \ldots, |EV_1|\}$ do
3:    $sum+ = abs(ev_{1,i} - ev_{2,i})/(ev_{1,i} + ev_{2,i})$
4: end for
5: $dist = sum/|EV_1|$
6: if $dist > Th_{EV}$ then
7:    $flag_{phase} = true$
8: else
9:    $flag_{phase} = false$
10: end if
11: return $flag_{phase}$
```

Algorithm 7: MF-GPOEO Algorithm Framework.

```
Input: the running target application, parameters in Table II
1: initialize performance baseline $DataBase$ and measurement $Duration$ with Algorithm 2
2: $gear_{opt} = gear_{high}$
3: while the target application is running do
4:    Set SM clock gear to $gear_{opt}$
5:    Measure kernel and gap performance data within $Duration$ and collect $MeasureData$
6:    $index_{cover} =$ Algorithm1$(DataBase, MeasureData)$
7:    if $index_{cover} > Th_{cover}$ then
8:       calculate $index_{perf}$ of $MeasureData$ with Algorithm 3
9:       calculate $gear_{opt}$ and $err_{ctrl}$ with Algorithm 4
10:      save $gear_{opt}$, $err_{ctrl}$ into two recent traces $Tc_{gear}$, $Tc_{err}$ and remove early $gear_{opt}$, $err_{ctrl}$ from $Tc_{gear}$, $Tc_{err}$, respectively
11:   else
12:      update performance baseline $DataBase$ and measurement $Duration$ with Algorithm 2
13:      clear $Tc_{gear}$ and $Tc_{err}$
14:      continue
15:   end if
16:   if $length(Tc_{err}) == len_{max}$ then
17:      calculate stable $gear_{opt}$ with Algorithm 5
18:      if $gear_{opt} \geq 0$ then
19:         set SM clock gear to $gear_{opt}$
20:         measure and collect $EV_i$ within $Duration$
21:         $flag_{phase} = false$
22:         while $flag_{phase} == false$ do
23:            measure and collect $EV_j$ within $Duration$
24:            calculate $flag_{phase}$ with Algorithm 6
25:            $EV_i = EV_j$
26:         end while
27:      end if
28:   end if
29: end while
```

update two traces with fixed length (line 10). If $index_{cover}$ does not satisfy Th_{cover}, we update $DataBase$ and $Duration$, clear two traces, and jump to the next while loop (line 11-14). After each control, if there are enough data in two traces, we judge control stability using Algorithm 5 (line 16-17). If $gear_{opt} > 0$, we think control is stable, set $gear_{opt}$, and monitor EVs until a phase change is detected (line 18-27). Upon encountering a phase change, we enter the next round of energy optimization.

In real GPUs, the number of memory clock frequencies is far less than the number of core clock frequencies. Low memory frequencies can lead to severe slowdowns. Therefore, we first determine the optimal memory clock and then the optimal SM clock in our implementation. If there are only a few memory clock frequencies, the trial-and-error method can also be used to replace the PID controller.

V. Evaluation

We evaluate our MF-GPOEO system in this section. We first introduce the experimental procedure. Next, we evaluate the accuracy and sensitivity of the synthetic performance index. Finally, we analyze the online energy efficiency optimization effect on various benchmarks.

A. Experimental Setup

Our experimental platform is a GPU server equipped with one NVIDIA RTX3080Ti GPU, one AMD 5950X CPU, and 64 GB of memory. The software environment is NVIDIA Driver 520.56, CUDA 11.7, and Linux 5.10. The RTX3080Ti supports discretely adjustable SM clock frequency upper limits, ranging from 210 MHz to 2160 MHz, and the step is 15 MHz. Some high frequencies are not achievable or stable. We exclude these high SM clock limits. We treat each SM clock frequency as an SM clock limit gear ($SMGear_i = 210 + 15i$ MHz), ranging from $SMGear_0 = 210$ MHz to $SMGear_{123} = 2055$ MHz. The RTX3080Ti supports five global memory clock frequencies: 405, 810, 5001, 9251, and 9501 MHz, denoted as $MemGear_0$ to $MemGear_4$. The RTX3080Ti supports continuously adjustable power caps, ranging from 100 W to 350 W, and we set the power cap step to 5 W. MF-GPOEO can also be ported to other platforms, such as AMD GPU, as long as the profiling tool can provide kernel activity information like grid size, block size, timestamp, etc.

We select the total $Energy$ consumption as the objective function and set a 5% performance loss constraint. We evaluate four methods, ODPP [15], GPOEO [16], DEPO [17], and our MF-GPOEO, on the AIBench suite [25], benchmarking-gnns suite [26], GROMACS (GR) [1], and Quantum ESPRESSO (QE) [2]. The medium-sized AIBench contains 14 AI applications, including image recognition, natural language processing, and recommender system applications. The large-sized benchmarking-gnns contains 57 GNN applications, including nine networks and seven datasets. GR is a software suite for high-performance molecular dynamics simulations. QE is a software suite for electronic-structure calculations and materials

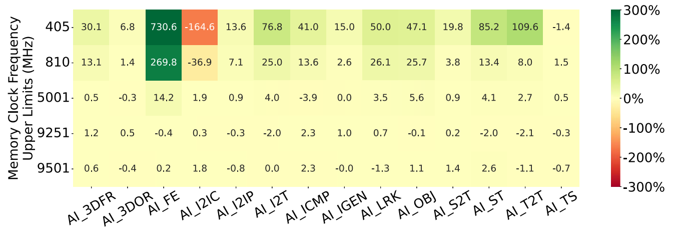


Fig. 4. Heat map of performance loss evaluation errors under varying GPU memory clock frequency upper limits.

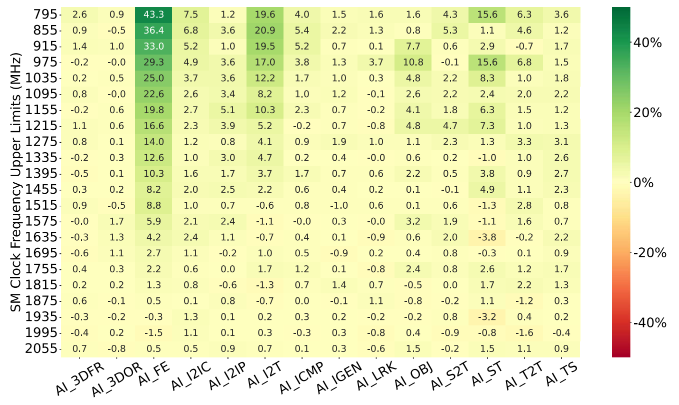


Fig. 5. Heat map of performance loss evaluation errors under varying GPU SM clock frequency upper limits.

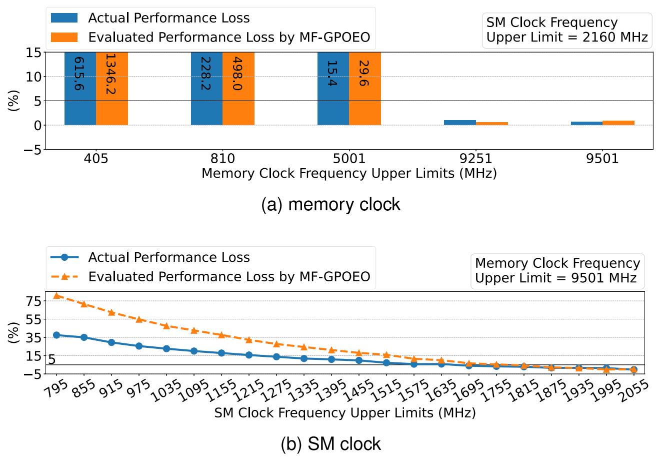


Fig. 6. Performance losses evaluated by MF-GPOEO under varying GPU memory and SM clock frequency upper limits on AI_FE.

modeling. We use the 0768 and 1536 inputs of the water_bare_hbonds model for GR and the AUSURF112 input for QE.

We determine the parameters in Table II by applying a grid-based parameter search on AIBench. We used the same suite of parameter values throughout our evaluation: $K_P = 2.8$, $K_I = 0.002$, $K_D = 0.9$, $Th_{exe} = 10.0$, $Th_{gap} = 2.5$, $Th_{cover}, = 0.75$, $Th_{ctrl} = 0.20$, $Th_{EV} = 0.1$. We found that the parameters perform well, suggesting a good generalization ability of these parameters. Users can also configure and tune these parameters as needed.

B. Accuracy and Sensitivity of Synthetic Performance Index

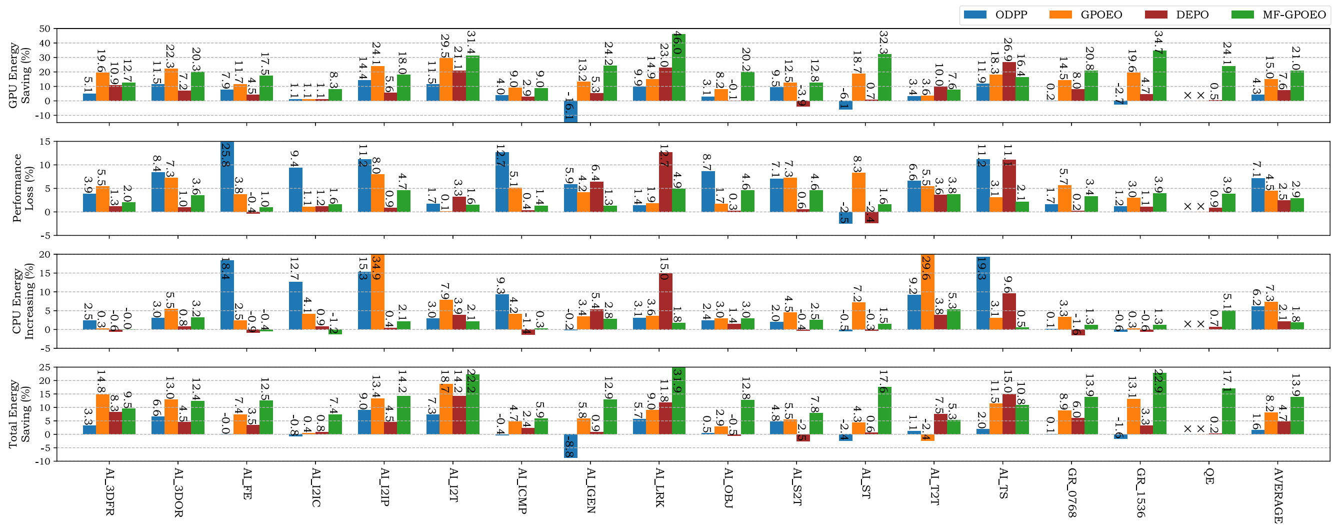
Our synthetic performance index $index_{perf}$ is designed to evaluate the application performance loss under different clock configurations. The evaluated performance loss is derived by $index_{perf} - 1$. We first analyze the performance loss evaluation error under different GPU memory clock upper limits on AIBench applications, as shown in Fig. 4. Under high memory clock limits (9251 and 9501 MHz), $index_{perf}$ values are accurate (errors $< 2.3\%$) and sensitive enough to support the PID controller to find optimal configurations within the performance loss constraint.

Performance loss evaluation errors are relatively large for medium and low memory clock limits ($\leq$5001 MHz) because some $GapData_k$s cannot be matched to the corresponding $GapData_n$s in the $DataBase$ (line 14 of Algorithm 3). Specifically, we find that the gap time distribution may be highly dispersed under low memory clock limits, and the values of $Stb_{gap,k}$s may exceed the threshold Th_{gap}. Consequently, Algorithm 3 does not count these unmatched $GapData_k$s into the weighted gap time (line 15-16) and omits the impact of these $GapData_k$s while calculating the $index_{perf}$ (line 24). Time changes of unmatched gaps may not be consistent with time changes of other gaps and kernels, so the $index_{perf}$ may overestimate or underestimate performance losses.

Most applications suffer severe performance losses that users cannot tolerate under relatively low memory clock limits ($\leq$5001 MHz). Even considering performance loss errors, $index_{perf}$s can still reflect these severe performance losses and support the PID controller to exclude these low clock configurations that violate the performance loss constraint. For example, the actual performance losses of AI_FE, AI_I2IC, and AI_T2T are 615.6%, 1035.9%, and 1109.7% under 405 MHz. The corresponding evaluated performance losses are 1346.2%, 871.3%, and 1219.3%. Both actual and evaluated performance losses far exceed practical performance loss constraints.

Then, we analyze the performance loss evaluation error under different GPU SM clock upper limits, as shown in Fig. 5. $index_{perf}$s are accurate (errors $\leq 2\%$) and sensitive for most applications under most clock limits. Errors slowly increase as the SM clocks decrease on six applications for the same reason mentioned above. These positive errors overestimate performance losses and make the PID controller only find higher suboptimal clock configurations within the performance loss constraint. Even so, our framework can still save significant energy consumption. We will discuss this in the following section.

To make it more intuitive, Fig. 6 shows the trend of actual and evaluated performance losses under varying memory and SM clock limits on AI_FE, which shows significant errors in Figs. 4 and 5. With Fig. 6(a), we can find acceptable memory clock limits (9251 and 9501 MHz) within the 5% performance loss constraint (the solid line). In Fig. 6(b), the evaluated performance loss curve follows the actual performance loss curve, especially under high SM limits. We find the lowest acceptable SM limit (1755 MHz) with the evaluated performance loss curve, while the optimal SM limit is 1680 MHz within the constraint.



* ODPP and GPOEO do not support QE, coded in Fortran language. We mark unsupported data with "×". Average values exclude these data.

Fig. 7. Energy saving and performance loss relative to NVIDIA's default clock strategy on AIBench, GR, and QE applications.

The found SM limit of 1755 MHz can still save much energy consumption. To sum up, the synthetic performance index is accurate and sensitive enough to guide the energy efficiency optimization process and ensure the performance loss constraint.

C. Results of Online Optimization

We use ODPP [15], GPOEO [16], DEPO [17], and our MF-GPOEO system to optimize energy efficiency online, respectively. ODPP only supports adjusting the SM clock, and DEPO only supports adjusting the power cap. GPOEO and MF-GPOEO adjust SM and memory clock. We use the energy consumption and execution time under NVIDIA's default clock strategy as the baseline.

1) Medium-Sized Benchmark Suite: Fig. 7 shows the online energy optimization results of the seventeen AIBench, GR, and QE applications. We evaluate these four methods with four metrics: GPU energy saving, performance loss (execution time increasing), CPU energy increasing, and total (CPU and GPU) energy saving. MF-GPOEO shows the best average results: 21.0% GPU energy saving and 2.9% execution time increase. MF-GPOEO gains notable GPU energy savings ($\geq$7.6%) on all seventeen applications, and only MF-GPOEO can satisfy the performance loss constraint (5%) strictly among these four methods on all applications. MF-GPOEO achieves the greatest GPU energy savings on fifteen applications compared to the other three methods within the performance loss constraint. Especially, MF-GPOEO achieves 46.0% GPU energy saving on On AI_LRK. GPOEO saves the most GPU energy on AI_3DFR, AI_3DOR, and AI_I2IP. However, the performance losses (5.5% to 8.0%) of AI_3DFR, AI_3DOR, and AI_I2IP violate the constraint (5%). DEPO saves the most GPU energy on AI_TS with an unacceptable performance loss (11.1%), consistent with our analysis of Fig. 2(c) in Section II-B2. As our discussion about Fig. 2(a) and (b), DEPO cannot find optimal configurations and saves little GPU energy on AI_I2IP and AI_OBJ. ODPP performs worst on all applications with less GPU energy saving or much higher performance loss.

We also consider the CPU energy overhead of GPU energy optimization systems. MF-GPOEO leads to minimal CPU energy increases on four applications and gains the slightest average CPU energy increase (1.8%). DEPO gains the lowest CPU energy increases on eleven applications but causes massive CPU energy increases on AI_LRK (15.0%) and AI_TS (9.6%). Thus, the average CPU energy increase of DEPO is slightly higher than MF-GPOEO. The average CPU energy increases of ODPP (6.2%) and GPOEO (7.3%) are much higher than MF-GPOEO and DEPO. Generally, these four methods can gain positive total energy savings (CPU and GPU) on most applications. MF-GPOEO achieves the most significant total energy savings on fifteen applications among the four methods within the performance loss constraint. GPOEO and DEPO save the greatest total energy within the constraint on AI_TS and AI_T2T, respectively. ODPP saves the least total energy on almost all applications. Generally, the total energy saving is less than the corresponding GPU energy saving for two reasons. The first apparent reason is the CPU energy increase. The second important reason is that adding the CPU energy into the denominator (the total energy consumption) leads to a larger denominator and a lower energy-saving percentage. Even so, MF-GPOEO can still achieve the most significant average total energy saving (13.9%).

Table V shows the optimal upper limits of the SM clock gear, memory clock frequency, and power found by these four methods to minimize GPU energy consumption within the 5% performance loss constraint. The oracle limits of the SM clock, memory clock, and power are obtained by traversing the search space (all supported SM, memory, and power caps). The SM gear errors are calculated with $Gear_{err} = Gear_{found} - Gear_{oracle}$. ODPP, GPOEO, and MF-GPOEO support SM gear

TABLE V
ONLINE OPTIMAL UPPER LIMITS OF SM CLOCK GEAR, MEMORY CLOCK FREQUENCY, AND POWER FOUND BY FOUR METHODS ON AIBENCH APPLICATIONS

		3DFR	3DOR	FE	I2IC	I2IP	I2T	ICMP	IGEN	LRK	OBJ	S2T	ST	T2T	TS
Oracle SM Gear Limit		104	102	98	101	95	80	103	66	69	95	108	39	100	101
SM Gear Error	ODPP	10	-9	-13	-9	-17	22	-19	48	37	-16	-9	72	-5	-12
	GPOEO	-2	-3	3	4	-6	14	2	0	17	5	-7	4	-2	5
	MF-GPOEO	0	1	5	2	6	3	2	27	4	1	0	43	0	2
Mem Clock Limit (MHz)	Oracle	9501	9251	9501	9501	9251	9251	9251	810	5001	9251	9251	810	9251	9501
	GPOEO	9501	9501	9501	9501	9501	9251	9251	810	5001	9251	9251	810	9251	9501
	MF-GPOEO	9501	9501	9501	9501	9501	9501	9501	5001	5001	9501	9501	5001	9501	9501
Power Cap (W)	Oracle	255	235	210	270	225	180	260	175	180	250	245	180	260	245
	DEPO	270	300	305	320	300	225	305	130	120	350	335	255	280	205
Optimization Duration (s)	ODPP	293	459	999	697	633	190	307	292	463	317	202	187	188	287
	GPOEO	158	207	326	242	630	499	566	243	297	286	235	166	174	138
	DEPO	86	49	58	49	53	49	49	53	70	62	50	53	49	94
	MF-GPOEO	54	60	107	60	99	62	97	76	69	70	98	138	74	60

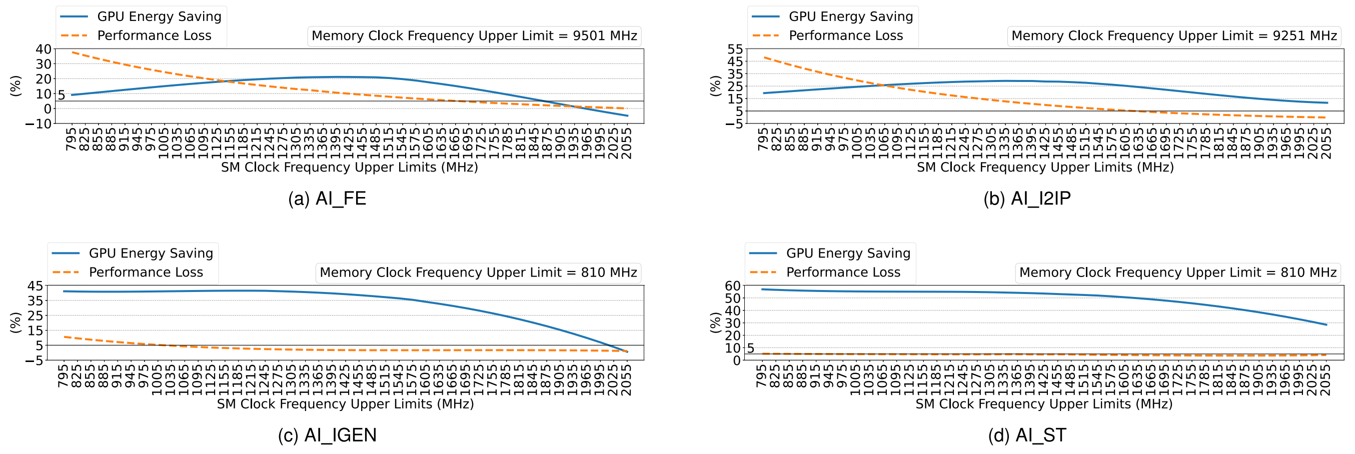


Fig. 8. GPU energy savings and performance losses under varying SM clock frequency upper limits.

adjustment. MF-GPOEO achieves the lowest SM gear errors among these three methods on eleven applications. GPOEO gains the lowest ones on five applications. ODPP has the highest SM gear errors on all applications. GPOEO finds twelve oracle memory clock limits. MF-GPOEO finds five oracle ones and nine suboptimal ones. ODPP does not support memory clock adjustment. DEPO supports power cap adjustment and only finds two passable solutions (errors ≤ 20 W) on AI_3DFR and AI_T2T. On AI_IGEN, AI_LRK, and AI_TS, DEPO gives too low power caps and causes performance loss constraint violation. For other applications, DEPO sets relatively high power caps and saves less GPU energy. ODPP and GPOEO spend too much time during the optimization process. Although SM and memory configurations of GPOEO are reasonable, GPU energy and performance overheads caused by long optimization durations seriously damage its final energy optimization results. MF-GPOEO achieves shorter optimization durations than ODPP and GPOEO.

MF-GPOEO underutilizes the performance loss constraint (5%) on some applications, such as AI_FE, AI_I2T, and AI_GEN. For AI_FE and AI_IGEN, MF-GPOEO misses some energy-saving potential for two reasons. First, the synthetic performance index (Algorithm 3) may overestimate performance losses. Second, in our implementation, the actual performance loss constraint (4.7%) is slightly stricter than 5% to offset possible performance overhead introduced by measurement. On AI_I2T, the SM gear (80) achieving minimal energy is higher than the SM gear (45) achieving the 5% performance loss. MF-GPOEO finds a suboptimal SM gear (83). Thus, the fact that the actual performance loss is less than the constraint does not always indicate that the energy-saving potential is not fully utilized.

For AI_FE, AI_I2IP, AI_IGEN, and AI_ST, the SM gear errors of MF-GPOEO are greater than five gears. To analyze these four applications, we visualize their search space to show their GPU energy saving and performance loss trend under varying SM clock limits and the oracle memory clock limit, as shown in Fig. 8. Solid lines represent the 5% performance loss constraint. On AI_FE and AI_I2IP, MF-GPOEO's suboptimal SM gears (1755 and 1725 MHz) can also exploit obvious energy-saving potentials, as shown in Fig. 8(a) and (b), respectively. Benefiting from a much shorter optimization time, MF-GPOEO is superior to GPOEO in both energy saving and performance loss on AI_FE. With a negative SM gear error, GPOEO saves more energy than MF-GPOEO but exceeds the 5% constraint on AI_I2IP.

As shown in Fig. 8(c) and (d), the energy-saving curves of AI_IGEN and AI_ST both have a plateau, where their

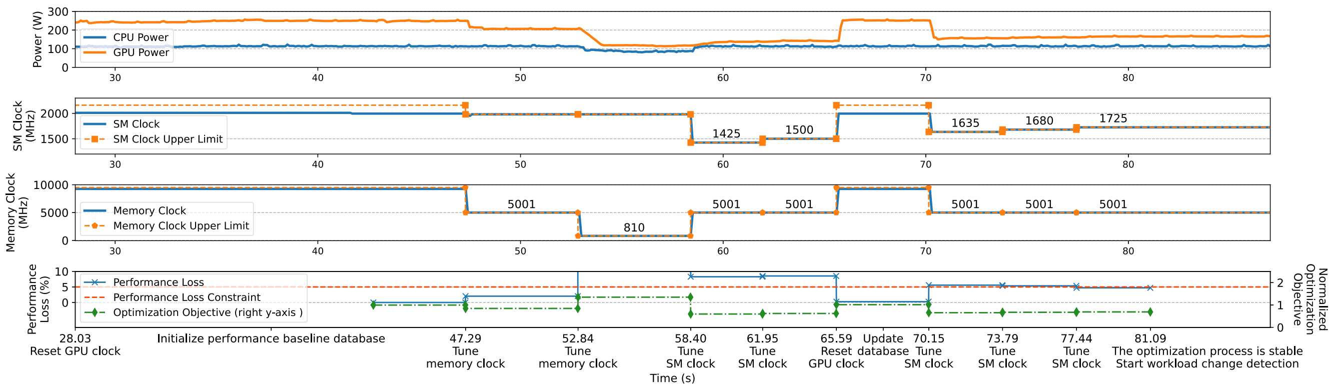


Fig. 9. Visualization of MF-GPOEO's energy-efficiency optimization process on the GR_1536 application.

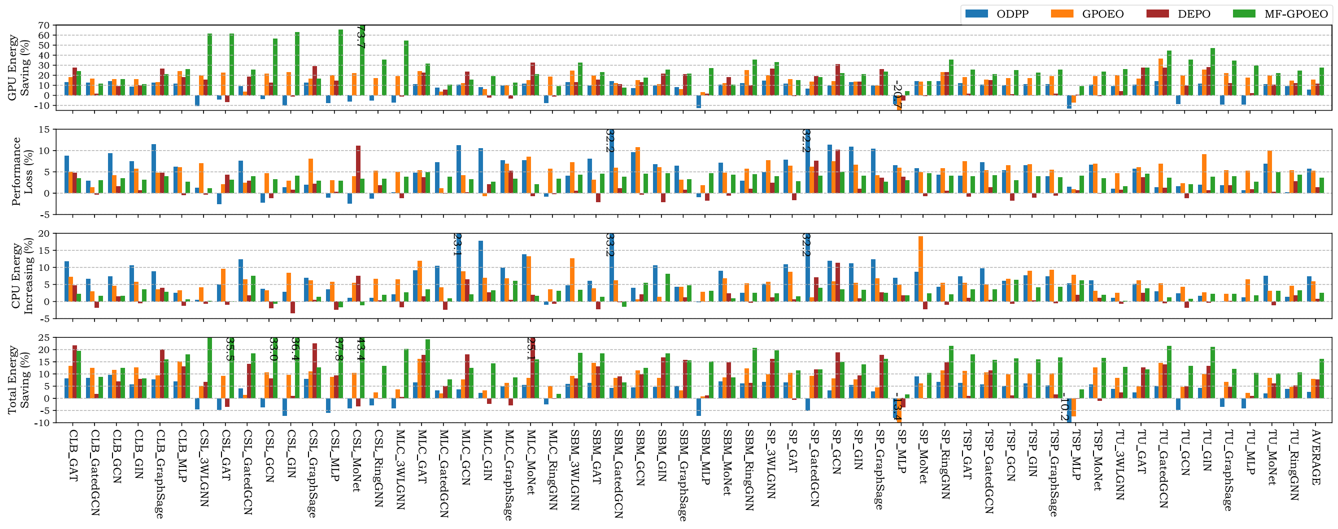


Fig. 10. Energy saving and performance loss relative to NVIDIA's default clock strategy on the benchmarking-gnns suite.

energy-saving results are great, and their performance losses are within the 5% constraint. Although the SM gear errors of AI_IGEN and AI_ST are large, their SM clock limits (1605 and 1440 MHz) given by MF-GPOEO are also on the plateaus, respectively. Therefore, MF-GPOEO still shows great energy-saving results on AI_IGEN and AI_ST and satisfies the performance loss constraint.

Fig. 9 visualizes the trace of GPU/GPU power, SM/memory clock (upper limit), calculated performance loss, and calculated optimization objective during the optimization process of MF-GPOEO on GR_1536. Before starting the optimizer (ODPP, GPOEO, DEPO, or MF-GPOEO), we use a simple logic to skip the unstable phase at the beginning of the application. Thus, MF-GPOEO's optimization process begins at 28.03 s. MF-GPOEO first collects kernel activity data under the default clock limit and initializes the performance baseline database ($28.03 \sim 47.29\ s$). Then, it tries two memory clock limits (5001 and 810 MHz) during $47.29 \sim 52.84\ s$ and $52.84 \sim 58.40\ s$, respectively. The 5001 MHz has a lower optimization objective (lower is better) within the 5% constraint, and the performance loss of 810 MHz (190.50%) seriously violates the constraint.

Under the optimal memory clock limit (5001 MHz), MF-GPOEO searches the optimal SM clock limit starting from a relatively low frequency (1425 MHz). Then, MF-GPOEO gradually increases the SM clock limit ($58.40 \sim 65.59\ s$, $70.15 \sim 81.09\ s$) and finally finds the optimal SM clock limit (1725 MHz). The performance loss of 1725 MHz is just below the constraint, and its optimization objective and GPU power are low. During $65.59 \sim 70.51\ s$, MF-GPOEO measures kernel activity data and updates the performance baseline database. After $81.09\ s$, MF-GPOEO determines that the optimization process is stable and begins to monitor phase changes. If a phase change occurs, MF-GPOEO will restart energy-efficiency optimization.

2) *Large-Sized Benchmark Suite:* Fig. 10 shows the online energy optimization results of 57 benchmarking-gnns applications. MF-GPOEO still shows the best average results: 27.7% GPU energy saving and 3.6% execution time increase. MF-GPOEO satisfies the performance loss constraint (5%) on all 57 applications. ODPP, GPOEO, and DEPO can only meet the constraint on 27, 22, and 54 applications, respectively. MF-GPOEO gains the most GPU energy savings on 46

applications within the constraint. Especially, MF-GPOEO achieves significant GPU energy savings ($\geq 20\%$) on 41 applications and huge GPU energy savings ($\geq 30\%$) on 17 applications. GPU energy savings even exceed 50% on CSL_3WLGNN, CSL_GAT, CSL_GCN, CSL_GIN, CSL_MLP, CSL_MoNet, and MLC_3WLGNN. These applications do not require high GPU memory frequencies. Only MF-GPOEO finds the low optimal GPU memory clock limit (810 MHz). GPOEO and DEPO save the most GPU energy on two and nine applications, respectively, within the constraint. ODPP performs poorly on all applications.

Considering CPU energy, these four methods show different results. DEPO exploits the energy-saving potential poorly and increases average execution time slightly (1.3%). Thus, DEPO introduces the lowest average CPU energy increase (0.8%). MF-GPOEO's average CPU energy increase (2.6%) is less than its performance loss (3.6%). The CPU energy increase is reasonable and acceptable. GPOEO and ODPP cause larger CPU energy increases than DEPO and MF-GPOEO. MF-GPOEO also gains the highest average total energy saving (CPU and GPU) (16.2%) among the four methods. GPOEO and DEPO perform similarly in terms of average total energy saving (7.8%). ODPP can hardly save the average total energy (2.5%). In summary, MF-GPOEO is the only one of these four methods that satisfies the performance loss constraint on all applications. It also saves more energy than the other three methods (twice or more).

VI. Related Work

Many research efforts have been devoted to improving GPU energy efficiency. Several papers survey these research efforts [5], [6], [27], [28], [29]. We classify the related work, which improves GPU energy efficiency via DVFS technology, into two kinds: offline and online.

Some studies collect offline profiling information under different frequency configurations to explore the optimization space of GPU energy efficiency [5], [30], [31], [32], [33]. These works prove the feasibility of GPU energy conservation through DVFS. Many studies build prediction models with machine learning methods to get optimal configurations [10], [11], [12], [13], [34]. These studies usually use data collected offline to train different machine learning models. Other studies build analytical models to predict energy consumption and performance [7], [8], [9], [10], [35]. These studies often analyze different components in GPUs and design analytical models for each component. Analytical models also need offline-measured data as input. Generally, these offline works require tedious profiling and analysis for new applications or even different inputs. Thus, these analytical models are inconvenient to use.

Several studies [14], [15], [16], [17] have realized online GPU energy efficiency optimization. Majumdar et al. [14] collect the stream of energy, execution time, throughput, and performance counter metrics at kernel granularity via the low-overhead fine-grained profiling functions supported by the AMD APU platform. Therefore, this work can catch energy-saving chances at kernel granularity and save considerable energy with low overhead. However, considering the high overhead of continuous profiling, it cannot be transplanted to the NVIDIA GPU platform. ODPP [15] is a lightweight GPU energy efficiency optimization framework. It only measures low-overhead coarse-grained features to predict the optimal frequency configuration. ODPP shows poor energy efficiency optimization results in our evaluation for two reasons. Coarse-grained features cannot provide enough information to model energy consumption and performance accurately. Its period detection algorithm is unstable and error-prone. GPOEO [16] collects hardware performance counter metrics and designs a robust period detection algorithm to tackle the above two weaknesses. GPOEO utilizes XGBoost models and an online local search method to find the optimal frequency configuration. GPOEO only profiles performance counters in one iteration period, so its overhead was acceptable. The above three works need well-trained or well-tuned models.

DEPO [17] is a model-free method to optimize GPU energy efficiency with power capping. It measures the kernel count and energy consumption within the fixed interval to evaluate and choose power capping configurations. Unfortunately, only using CUDA kernel counting is insufficient to support accurate performance evaluations. In addition, other defects also make DEPO impractical, such as the manually set measurement interval, tuning phase time, and kernel reporting frequency.

VII. Conclusion and Future Work

We have proposed MF-GPOEO, a novel model-free online GPU energy efficiency optimization framework. MF-GPOEO collects GPU kernel execution times and gap times between kernels under different energy configurations, computes synthetic performance indexes for different energy configurations, and finds the optimal energy configuration with a PID controller. We evaluate MF-GPOEO on 74 applications running on an NVIDIA RTX3080Ti GPU. MF-GPOEO achieves a mean energy saving of 26.2% with a minor performance loss of 3.4% compared with NVIDIA's default clock scheduling strategy. We will improve the stability and accuracy of gap time matching and comparing in future work.

References

[1] M. J. Abraham et al., "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1, pp. 19–25, 2015.

[2] P. Giannozzi et al., "Quantum espresso: A modular and open-source software project for quantum simulations of materials," *J. Phys. Condens. Matter*, vol. 21, no. 39, 2009, Art. no. 395502.

[3] J. Fang, C. Huang, T. Tang, and Z. Wang, "Parallel programming models for heterogeneous many-cores: A comprehensive survey," *CCF Trans. High Perform. Comput.*, vol. 2, no. 4, pp. 382–400, 2020.

[4] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "AI and ML accelerator survey and trends," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2022, pp. 1–10.

[5] X. Mei, Q. Wang, and X. Chu, "A survey and measurement study of GPU DVFS on energy conservation," *Digit. Commun. Netw.*, vol. 3, no. 2, pp. 89–100, 2017.

[6] W. Gao et al., "Deep learning workload scheduling in GPU datacenters: Taxonomy, challenges and vision," 2022, *arXiv:2205.11913*.

[7] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas, "GPGPU power modeling for multi-domain voltage-frequency scaling," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 789–800.

[8] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas, "Modeling and decoupling the GPU power consumption for cross-domain DVFS," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 11, pp. 2494–2506, Nov. 2019.
[9] Q. Wang and X. Chu, "GPGPU performance estimation with core and memory frequency scaling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 12, pp. 2865–2881, Dec. 2020.
[10] M. Amaris, R. Camargo, D. Cordeiro, A. Goldman, and D. Trystram, "Evaluating execution time predictions on GPU kernels using an analytical model and machine learning techniques," *J. Parallel Distrib. Comput.*, vol. 171, pp. 66–78, 2023.
[11] Q. Wang and X. Chu, "GPGPU power estimation with core and memory frequency scaling," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 45, no. 2, pp. 73–78, 2017.
[12] K. Fan, B. Cosenza, and B. Juurlink, "Predictable GPUs frequency scaling for energy and performance," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–10.
[13] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas, "GPU static modeling using PTX and deep structured learning," *IEEE Access*, vol. 7, pp. 159150–159161, 2019.
[14] A. Majumdar, L. Piga, I. Paul, J. L. Greathouse, W. Huang, and D. H. Albonesi, "Dynamic GPGPU power management using adaptive model predictive control," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 613–624.
[15] P. Zou, L. Ang, K. Barker, and R. Ge, "Indicator-directed dynamic power management for iterative workloads on GPU-accelerated systems," in *Proc. 20th IEEE/ACM Int. Symp. Cluster Cloud Internet Comput.*, 2020, pp. 559–568.
[16] F. Wang, W. Zhang, S. Lai, M. Hao, and Z. Wang, "Dynamic GPU energy optimization for machine learning training workloads," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2943–2954, Nov. 2022.
[17] A. Krzywaniak, P. Czarnul, and J. Proficz, "Dynamic GPU power capping with online performance tracing for energy efficient GPU computing using DEPO tool," *Future Gener. Comput. Syst.*, vol. 145, pp. 396–414, 2023.
[18] NVIDIA Corporation, "Cupti documentation," 2021. Accessed: Aug. 19, 2021. [Online]. Available: https://docs.nvidia.com/cupti/Cupti/index.html
[19] NVIDIA, "Cuda toolkit documentation," 2021. Accessed: Aug. 19, 2021. [Online]. Available: https://docs.nvidia.com/cuda/index.html
[20] A. Krzywaniak, P. Czarnul, and J. Proficz, "DEPO: A dynamic energy-performance optimizer tool for automatic power capping for energy efficient high-performance computing," *Softw., Pract. Experience*, vol. 52, no. 12, pp. 2598–2634, 2022.
[21] NVIDIA, "NVML API reference manual," 2021. Accessed: Aug. 19, 2021. [Online]. Available: https://docs.nvidia.com/deploy/nvml-api/index.html
[22] AMD, "Rnda architecture white paper," Oct. 2022. Accessed: Oct. 03, 2022. [Online]. Available: https://www.amd.com/system/files/documents/rdna-whitepaper.pdf
[23] J. Schwarzrock, C. C. de Oliveira, M. Ritt, A. F. Lorenzon, and A. C. S. Beck, "A runtime and non-intrusive approach to optimize EDP by tuning threads and CPU frequency for OpenMP applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1713–1724, Jul. 2021.
[24] G. F. Franklin, J. D. Powell, A. Emami-Naeini, and J. D. Powell, *Feedback Control of Dynamic Systems*, vol. 4. Upper Saddle River, NJ, USA: Prentice Hall, 2002.
[25] F. Tang et al., "AIBench training: Balanced industry-standard ai training benchmarking," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2021, pp. 24–35.
[26] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson, "Benchmarking graph neural networks," 2020, *arXiv: 2003.00982*.
[27] K. O'brien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou, "A survey of power and energy predictive models in HPC systems and applications," *ACM Comput. Surv.*, vol. 50, no. 3, pp. 1–38, 2017.
[28] S. Pagani, P. S. Manoj, A. Jantsch, and J. Henkel, "Machine learning for power, energy, and thermal management on multicore processors: A survey," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 1, pp. 101–116, Jan. 2020.
[29] G. Xie, X. Xiao, H. Peng, R. Li, and K. Li, "A survey of low-energy parallel scheduling algorithms," *IEEE Trans. Sustain. Comput.*, vol. 7, no. 01, pp. 27–46, First Quarter 2022.
[30] X. Mei, L. S. Yung, K. Zhao, and X. Chu, "A measurement study of GPU DVFS on energy conservation," in *Proc. Workshop Power-Aware Comput. Syst.*, 2013, pp. 1–5.
[31] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas, "DVFS-Aware application classification to improve GPGPUS energy efficiency," *Parallel Comput.*, vol. 83, pp. 93–117, 2019.
[32] Y. Arafa et al., "Verified instruction-level energy consumption measurement for NVIDIA GPUs," in *Proc. 17th ACM Int. Conf. Comput. Front.*, 2020, pp. 60–70.
[33] A. Krzywaniak, P. Czarnul, and J. Proficz, "GPU power capping for energy-performance trade-offs in training of deep convolutional neural networks for image recognition," in *Proc. 22nd Int. Conf. Comput. Sci.*, Springer, 2022, pp. 667–681.
[34] C. Marantos, L. Papadopoulos, C. P. Lamprakos, K. Salapas, and D. Soudris, "Bringing energy efficiency closer to application developers: An extensible software analysis framework," *IEEE Trans. Sustain. Comput.*, vol. 8, no. 2, pp. 180–193, Second Quarter 2023.
[35] S. Hajiamini, B. Shirazi, and H. Dong, "A fast heuristic for improving the energy efficiency of asymmetric VFI-based manycore systems," *IEEE Trans. Sustain. Comput.*, vol. 7, no. 2, pp. 358–370, Second Quarter 2022.

Farui Wang received the MS degree in aeronautical and astronautical science and technology from the Harbin Institute of Technology, China, in 2017. He is currently working toward the PhD degree with the School of Cyberspace Science. His research interests include high-performance computing, source-to-source translation for accelerators, and performance-energy optimization for parallel applications.

Meng Hao received the B.S. degree in computer science and engineering from Harbin Institute of Technology in 2014, and the Ph.D. degree in cyberspace science from Harbin Institute of Technology in 2020. He is currently an assistant professor in the School of Cyberspace Science, Harbin Institute of Technology. His research interests include high-performance computing, performance modeling, and parallel optimization.

Weizhe Zhang (Senior Member, IEEE) received the BEng, MEng, and PhD degrees of engineering in computer science and technology from the Harbin Institute of Technology, in 1999, 2001, and 2006 respectively. He is currently a professor with the School of Computer Science and Technology, Harbin Institute of Technology, China, and director with the Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen, China. His research interests are primarily in parallel computing, distributed computing, cloud and grid computing, and computer network. He has published more than 100 academic papers in journals, books, and conference proceedings.

Zheng Wang is a professor of intelligent software technology with the University of Leeds. His research focuses on parallel computing, compilation, and systems security.