DEFT: Joint Task Placement and DVFS for Energy-Efficient Multi-GPU Runtimes

DEFT: Joint Task Placement and DVFS for Energy-Efficient Multi-GPU Runtimes

Jing Chen
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
chjing@chalmers.se

Miquel Pericàs
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
miquelp@chalmers.se

Abstract

Energy efficiency has become a first-order concern in modern high-performance computing systems, as it directly determines achievable throughput under fixed power budgets. Although Dynamic Voltage and Frequency Scaling (DVFS) provides an effective mechanism for reducing GPU energy consumption, existing runtime systems decouple DVFS from task placement and inter-GPU communication, focus on single-GPU execution, or cannot adapt frequency to task granularity and runtime contention in multi-GPU environments. Consequently, current schedulers fail to capture the tight coupling between task placement, frequency selection, and inter-GPU data movement that fundamentally governs energy–performance trade-offs on multi-GPU systems.

This paper presents DEFT, an energy-aware scheduling framework that jointly optimizes task-to-device assignment and per-GPU DVFS configuration for task-based multi-GPU applications. DEFT employs a cost-model–driven strategy that integrates slack awareness, throughput awareness, and explicit modeling of task execution cost, inter-GPU data movement, and DVFS transition overheads, enabling coordinated placement and frequency decisions at task granularity under dynamic runtime conditions. We prototype DEFT within the CUDASTF runtime and demonstrate its effectiveness across five optimization objectives. The evaluation shows that DEFT reduces energy consumption by 14.8% and 4.8% on average on NVIDIA L40S and L4, and reduces EDP by 9.9% and 3.7%, respectively, while maintaining performance within 1.5% of the fastest baseline.

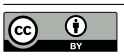
CCS Concepts

• **Software and its engineering → Development frameworks and environments**; • **Hardware → Power and energy**.

Keywords

Energy Efficiency, Task Graphs, Runtime Systems, DVFS, Multi-GPU Scheduling

ACM Reference Format:
Jing Chen and Miquel Pericàs. 2026. DEFT: Joint Task Placement and DVFS for Energy-Efficient Multi-GPU Runtimes. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3797905.3807847



ICS '26, Belfast, United Kingdom

ACM ISBN 979-8-4007-2522-7/26/07
https://doi.org/10.1145/3797905.3807847

1 Introduction

As High Performance Computing (HPC) systems advance beyond the Exascale barrier, energy efficiency has emerged as a primary constraint limiting further scalability [16, 24, 28, 38]. In large-scale deployments, improved energy efficiency directly translates into higher aggregate throughput. Contemporary HPC nodes increasingly rely on dense multi-GPU integration to deliver massive computational capability, yet translating this capability into energy-efficient execution remains challenging. In practice, GPUs are often overprovisioned relative to application-level parallelism [37], and default power management policies tend to maintain peak operating frequencies whenever devices are allocated, even when utilization is low [35], resulting in substantial energy waste.

Energy optimization poses different challenges compared to performance optimization in task-based systems. While energy is an execution-wide, cumulative metric that depends on the collective effect of all scheduling decisions, task scheduling decisions are made online and sequentially as tasks become ready. As a result, locally optimal choices, such as executing a task at the frequency that minimizes its standalone energy consumption, may increase global energy usage by delaying dependent tasks, prolonging idle periods on other GPUs, and ultimately extending the overall makespan, thereby offsetting per-task energy savings. For example, our characterization shows that an NVIDIA L4 GPU consumes approximately 27 W when idle and waiting for tasks, accounting for nearly 38% of its peak power (72 W). Moreover, task scheduling decisions are irreversible once dispatched, allowing early suboptimal choices to propagate through the dependency graph and constrain future decisions. Together, these factors make energy-aware scheduling in multi-GPU task graphs substantially more challenging.

DVFS offers a compelling mechanism to improve energy efficiency by trading performance slack for reduced power consumption. Modern GPUs expose fine-grained DVFS controls, enabling adaptive optimization based on workload heterogeneity. Different kernels exhibit distinct utilization patterns across GPU functional units, making the optimal frequency highly kernel-dependent [2, 21, 29]. Prior work has demonstrated energy savings through DVFS-based scheduling, but primarily targets single-GPU execution [3, 4, 7, 19, 26, 39, 46, 50].

In addition, frequency reconfiguration incurs non-trivial overhead, ranging from hundreds of microseconds to milliseconds depending on the transition magnitude and direction [42]. This overhead makes fine-grained per-kernel tuning counterproductive for short-running tasks, while coarse-grained application-level policies often overlook substantial optimization opportunities by treating heterogeneous kernels uniformly. To amortize this overhead,

some prior work adopts static or phase-based frequency selection strategies [8, 10], which limits the ability to adapt to variable task granularity and dynamic runtime conditions. A robust solution must make DVFS decisions online, determining when and how to adjust frequencies based on task granularity, computational characteristics, reconfiguration costs, and evolving system state within a unified optimization framework.

The challenge intensifies in multi-GPU environments, where scheduling decisions span three interdependent dimensions. First, *frequency selection*: each GPU may operate at a different frequency, with optimal settings varying based on the task's computational characteristics and the current device state. Second, *task placement*: tasks must be mapped to GPUs considering not only device availability but also data locality and device-specific performance characteristics. Third, *cost awareness*: both inter-GPU data transfers and DVFS transitions impose latency and energy costs that can dominate savings for fine-grained operations. These dimensions are inherently interdependent: the optimal frequency for a task depends on where it executes, placement must account for both DVFS reconfiguration and data movement costs, and data movement costs vary with both placement and device frequency. This interdependence makes isolated optimization of any single dimension suboptimal, necessitating a holistic scheduling approach. However, prior work addresses these challenges largely in isolation. Approaches that focus on task placement target throughput improvement or load balancing across devices without integrating DVFS [11, 27, 41, 43, 47], while DVFS-focused techniques are commonly designed for single-device execution and do not account for the joint impact of task placement, inter-GPU data movement, and frequency transition overheads in multi-GPU systems.

To address the multi-dimensional challenges of energy-efficient task-graph execution on multi-GPU systems, we propose **DEFT** (**D**ynamic **E**nergy-aware **F**requency – **T**ask scheduling), a scheduler that jointly determines task-to-device placement and per-GPU DVFS configuration to achieve user-specified energy-performance trade-offs. Users specify optimization objectives through a generalized energy-delay product metric $ED^{\beta}P$, where the exponent β controls the relative importance of execution time versus energy.

DEFT adopts a two-phase decision strategy that hierarchically structures task placement and DVFS selection. The first phase selects the execution device that yields the earliest feasible start time for each ready task, based on device availability and data readiness, and independently of DVFS, since frequency does not affect start-time feasibility. In the second phase, DEFT determines the DVFS configuration on the selected device to optimize the energy–performance objective while preserving global progress. DEFT integrates: (i) *slack awareness*, derived from critical-path analysis, to bound execution delay without extending the application makespan; (ii) *throughput awareness* to prevent excessive slowdowns under resource contention when the number of ready tasks exceeds the available GPUs; and (iii) *cost awareness*, which explicitly models inter-GPU data transfer and DVFS transition overheads to avoid counterproductive decisions when these costs outweigh potential savings. This unified design enables DEFT to adapt its optimization strategy to both task granularity and dynamic system conditions. For short-lived tasks where reconfiguration or data movement overheads would dominate potential benefits, the scheduler favors local execution at the current frequency. For longer-running or slack-rich tasks, DEFT explores a broader configuration space, trading execution time for improved energy efficiency.

In summary, this work makes the following contributions:

- **Energy-Aware Multi-GPU Scheduling Framework:** We present a scheduling framework for task-graph execution on multi-GPU systems that enables users to specify optimization objectives and automatically orchestrates task placement and DVFS configuration. We prototype DEFT on top of the task-based CUDA runtime CUDASTF [5]; however, the methodology generalizes to other task-based runtimes.
- **Cost-Model–Driven Hierarchical Scheduling Algorithm:** We develop a two-phase scheduling algorithm that prioritizes device selection to preserve parallelism and then optimizes the DVFS configuration on the selected device. The algorithm is guided by a unified cost model that explicitly captures task execution cost, inter-GPU data movement, DVFS transition overheads, and slack/throughput constraints when evaluating configurations. To our knowledge, DEFT is the first runtime framework to provide integrated, cost-aware coordination between placement and DVFS at task granularity in multi-GPU task-based runtimes.
- **Predictive Models:** We develop three complementary predictive models to support efficient runtime scheduling decisions across the device–frequency configuration space. Specifically, we (i) introduce an XGBoost-based performance model that predicts task execution time using hardware utilization features; (ii) adopt and extend an analytical power model that estimates task-level GPU power consumption based on component-level hardware utilization; and (iii) propose an empirical DVFS transition cost model that captures asymmetric and non-linear reconfiguration overheads through piecewise functions.
- **Evaluation:** We evaluate DEFT on NVIDIA L4 and L40S multi-GPU nodes under five optimization objectives (Energy-only, E^2DP, EDP, ED^2P, and Time-only). DEFT reduces energy by 14.8% and 4.8% on average on L40S and L4, respectively (9.9% and 3.7% in EDP), compared to native CUDASTF scheduling, while maintaining performance within 1.5% of the fastest baseline.

2 Background

2.1 Task Graph

Task-based parallel programming models provide a natural abstraction for expressing and optimizing dynamic parallelism in heterogeneous systems [6, 15, 20, 34, 36]. These models decompose applications into units of work (tasks) connected by data dependencies, forming a directed acyclic graph (DAG) that the runtime schedules automatically. The runtime leverages schedulers to make resource allocation and hardware configuration decisions based on task characteristics, device capabilities, and optimization objectives.

In a DAG, the critical path is the longest dependency chain in the graph, which defines a lower bound on the application makespan, since any delay along this path directly extends total execution time. Tasks that do not lie on the critical path may tolerate limited execution delays without affecting the overall completion time. This tolerance is referred to as *task slack*. Task slack can be derived through static analysis of the DAG by examining the longest paths

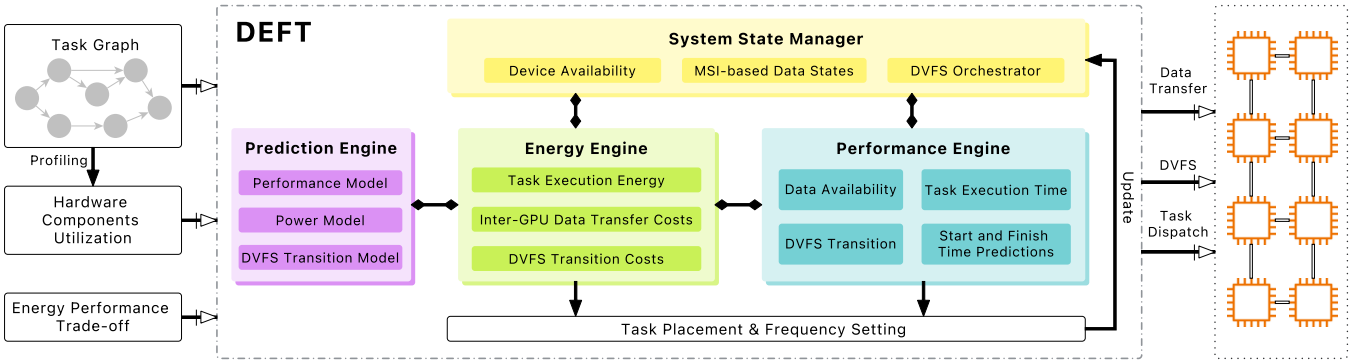


Figure 1: Overview of the proposed energy-aware scheduling framework.

from the entry to a task and from the task to an exit node under baseline execution assumptions. Conceptually, slack represents the degree of scheduling flexibility available for a task. This notion is especially important in runtime scheduling, as it enables the scheduler to distinguish between time-critical tasks that must be executed promptly and slack-rich tasks that can be delayed or slowed down to optimize secondary objectives, such as reducing energy consumption or avoiding resource contention.

2.2 CUDASTF

CUDASTF [5] extends this task-based paradigm for multi-GPU CUDA applications and implements the Sequential Task Flow programming model, allowing applications to be expressed as a sequence of tasks that are automatically transformed into a DAG based on data dependencies inferred from task declarations. The runtime automatically manages critical low-level operations, including data transfers, synchronization, and scheduling across multiple GPUs. A core abstraction is logical data, which represents data as an abstract entity without associating it explicitly with specific storage locations; instead, it tracks multiple coherent replicas (data instances) across distinct physical memories. We leverage CUDASTF's task-graph abstraction and flexible design to transparently integrate energy-aware scheduling into task-based runtimes without application code modification. CUDASTF supports both stream-based and CUDA Graph–based execution backends. We build DEFT on the stream backend, which enables the eager task execution and runtime flexibility necessary for per-task DVFS decisions and dynamic device assignment.

3 Methodology

In this section, we first provide an overview of the proposed framework (§ 3.1). We then detail the scheduling algorithm and cost model (§ 3.2), followed by a discussion of the MSI-based coherence protocol for tracking data residency across GPUs (§ 3.3). Finally, we describe the profiling methodology for collecting per-task hardware utilization metrics (§ 3.4), which serve as inputs to the performance and power models (§ 4).

3.1 Overview

DEFT is guided by a unified cost model that accounts for task performance and power, inter-GPU data transfer, DVFS transition overheads, task slack, and throughput constraints. Figure 1 illustrates the architectural overview of the DEFT framework. It operates on three inputs: (1) an application expressed as a DAG with explicit data dependencies; (2) per-task hardware utilization metrics collected via lightweight one-time profiling (detailed in § 3.4); (3) a user-specified optimization objective of the form $ED^{\beta}P$. Our framework consists of four tightly coupled components that collaboratively make scheduling decisions. Throughout this section, we use the following notation: t_i denotes task i in the task graph, d denotes a GPU device index, and $f = (f_{\text{gpu}}, f_{\text{mem}})$ denotes a DVFS configuration comprising GPU core and memory frequencies. The components are:

- **Prediction Engine**: Comprises three models (detailed in § 4) that predict task execution time $T_{i,d,f}$, average power consumption $P_{i,d,f}$, and DVFS reconfiguration latency, respectively. The performance and power models leverage hardware utilization metrics to extrapolate $T_{i,d,f}$ and $P_{i,d,f}$ across all device-frequency pairs (d, f) without exhaustive runtime profiling.
- **Energy Engine**: Computes the total energy cost $E_{i,d,f}$ of scheduling task t_i on device d at frequency f, decomposed into three components: (i) task execution energy $E_{\text{exec}} = T_{i,d,f} \times P_{i,d,f}$, (ii) inter-GPU data transfer energy required to satisfy input dependencies, and (iii) DVFS reconfiguration energy incurred when transitioning device d from its current frequency to f.
- **Performance Engine**: Computes task earliest start time T_{start} and expected finish time T_{finish} on each device by analyzing device and data availability, accounting for inter-GPU transfer latency, DVFS transition overhead, and task execution time.
- **System State Manager**: Maintains runtime state comprising (i) per-device availability times, (ii) MSI-based coherence metadata tracking data residency and validity across devices (described in § 3.3), and (iii) a DVFS orchestrator that checks for pending frequency change requests and idle GPU conditions. When a GPU remains idle beyond a predefined temporal threshold, the orchestrator proactively transitions it to a low-power state.

Upon selecting configuration (d^*, f^*), the System State Manager atomically updates: (i) device d^*'s workload and availability time, (ii) MSI coherence states for task t_i's input and output data, and (iii) device d^*'s scheduled frequency to f^*. The runtime subsequently initiates inter-GPU data transfers for missing dependencies,

issues DVFS reconfiguration commands to set frequency f^*, and dispatches the task to device d^* for execution.

3.2 Scheduling Algorithm

The scheduler processes tasks in topological order, ensuring all dependencies are satisfied before scheduling any dependent task. For each ready task t_i, rather than exhaustively enumerating all $(d, f) \in \mathcal{D} \times \mathcal{F}$ pairs (where $\mathcal{D}$ is the set of available GPUs and $\mathcal{F}$ is the discrete DVFS configuration space), DEFT enables a hierarchical decision structure in which device selection is performed first based on task earliest start time, while DVFS configurations are evaluated conditionally on the selected device using the full cost model.

3.2.1 Design Rationale. Prioritizing device selection based on earliest start time serves three critical objectives: (i) minimizing task completion time to unblock dependent tasks as early as possible, thereby preserving task-level parallelism in the DAG; (ii) reducing system-wide idle energy by avoiding unnecessary delays that force other devices to wait idly for data dependencies; and (iii) maintaining temporal locality to reduce inter-GPU communication overhead. Crucially, the earliest time at which task t_i can begin execution on device d depends only on device availability, data availability, and the completion of its parent tasks—factors that are independent of DVFS configuration—enabling the two-phase decomposition.

Scheduling decisions that achieve the lowest objective locally by aggressively reducing frequency may delay dependent tasks, prolong overall execution, and increase idle periods on other GPUs. To bound this effect, we leverage task slack, defined as the difference between the application's critical path length and the longest path passing through the task, which quantifies how much execution delay can be tolerated without extending overall makespan. Task slack is precomputed via static analysis of the task graph using predicted execution times at maximum frequency. When multiple tasks are ready, we prioritize tasks with lower slack to avoid extending the critical path and increasing overall makespan.

However, dependency-aware slack alone is insufficient under resource contention, when task-level parallelism exceeds the number of available GPUs. In such throughput-bound scenarios, aggressively slowing down slack-rich tasks risks delaying critical tasks, reducing overall system throughput, and prolonging the makespan despite preserving the critical path. We therefore monitor runtime contention by tracking the instantaneous number of ready tasks relative to the number of available GPUs, and activate a throughput-aware refinement when contention arises. This refinement caps per-task execution-time inflation, preventing slack-rich tasks from monopolizing shared GPU resources and bounding slowdown to preserve overall throughput.

3.2.2 Two-Phase Algorithmic Design.

Phase 1: Device Selection. For each device $d \in \mathcal{D}$, we first compute when task t_i's input data will be available on device d:

$$T^d_{\text{data}} = \max_{p \in \text{parents}(t_i)} \left(T^p_{\text{finish}} + \frac{|\text{data}_p|}{BW} \right) \tag{1}$$

where T^p_{finish} is the scheduled finish time of parent task p, $|\text{data}_p|$ is the size of data produced by p that t_i requires, and BW is the peer-to-peer bandwidth. Inter-GPU transfers are handled by dedicated copy engines (e.g., NVLink or PCIe DMA) whose throughput is independent of GPU DVFS states and operates at fixed interconnect bandwidth. If data resides on device d, the transfer time is zero.

The frequency-independent earliest start time for task t_i on device d is then:

$$T^d_{\text{earliest}} = \max \left(T^d_{\text{available}}, T^d_{\text{data}} \right) \tag{2}$$

where $T^d_{\text{available}}$ is the time when device d completes its currently scheduled workload. This represents the earliest moment at which task t_i can begin execution on device d, determined by the later of device availability or data readiness.

We also compute the transfer energy E^d_{transfer} in this phase. When data arrives before the GPU becomes available ($T^d_{\text{data}} \le T^d_{\text{available}}$), the GPU is busy and incurs no additional energy cost during the transfer (computation-transfer overlap). When data arrives after the GPU is ready ($T^d_{\text{data}} > T^d_{\text{available}}$), the GPU idles during the transfer gap, consuming idle power:

$$E^d_{\text{transfer}} = \begin{cases} 0 & \text{if } T^d_{\text{data}} \le T^d_{\text{available}} \\ (T^d_{\text{data}} - T^d_{\text{available}}) \times P_{\text{idle}}(f^d_{\text{curr}}) & \text{otherwise} \end{cases} \tag{3}$$

where $P_{\text{idle}}(f^d_{\text{curr}})$ is the idle power of device d at its current frequency configuration.

The device with the minimum T^d_{earliest} is selected:

$$d^* = \arg\min_{d \in \mathcal{D}} T^d_{\text{earliest}} \tag{4}$$

This selection prioritizes temporal availability, ensuring tasks are scheduled on the device that can start execution earliest, thereby minimizing data dependency stalls and critical path delays.

Phase 2: Frequency Optimization. Having selected device d^*, we now evaluate all frequency configurations $f \in \mathcal{F}$ to find the best DVFS state. For each candidate frequency $f = (f_{\text{gpu}}, f_{\text{mem}})$, we perform the following analysis:

Step ❶: DVFS Transition Cost. The scheduler computes the cost of transitioning device d^* from its current frequency configuration f_{curr} to the candidate configuration f. The transition latency T_{trans} is predicted using the empirical DVFS transition model described in § 4.4. During the transition, the device operates in an idle state, consuming:

$$E_{\text{trans}} = T_{\text{trans}} \times (P_{\text{idle}}(f_{\text{curr}}) + P_{\text{idle}}(f))/2 \tag{5}$$

where we approximate the average power during the transition as the mean of the idle power at the initial and target frequencies.

Step ❷: Task Timing. We assume that task t_i cannot begin execution until both the DVFS transition completes and data is available. The actual start time is:

$$T_{\text{start}} = \max(T^{d^*}_{\text{available}} + T_{\text{trans}}, T^{d^*}_{\text{data}}) \tag{6}$$

Note that DVFS reconfiguration and data transfers occur in parallel on independent hardware (power management unit and copy engines, respectively); the start time is determined by whichever completes last. The projected finish time is $T_{\text{finish}} = T_{\text{start}} + T_{i,d^*,f}$, where $T_{i,d^*,f}$ is the predicted execution time at frequency f (from the Performance Model in § 4.3).

Step ❸: Slack Feasibility Analysis. Given a candidate frequency configuration f on the selected device d^*, we first compute the additional execution delay relative to the baseline execution at maximum frequency:

$$\Delta T(t_i, d^*, f) = T_{i,d^*,f} - T_{i,d^*}^{\text{baseline}} \tag{7}$$

We define a configuration as *slack-feasible* if the delay can be absorbed without extending the application makespan: $\Delta T(t_i, d^*, f) \leq \text{Slack}(t_i)$, where $\text{Slack}(t_i)$ represents the maximum tolerable delay of task t_i derived from static critical-path analysis.

When the number of ready tasks exceeds the number of available GPUs, we tighten the effective slack budget under resource contention by restricting the allowable DVFS-induced slowdown to the minimum of the task slack and a small fraction of its baseline execution time:

$$\Delta T(t_i, d^*, f) \leq \min\{\text{Slack}(t_i), \kappa \times T_{i,d^*}^{\text{baseline}}\} \tag{8}$$

This constraint jointly enforces slack feasibility and throughput preservation under contention. Configurations that violate either bound are considered infeasible and penalized in the cost function. In our experiments we use a fixed, conservative cap of $\kappa = 2\%$, selected via preliminary sensitivity testing to bound per-task slowdown under contention.

Step ❹: Energy Cost Evaluation. The execution energy for task t_i at configuration f is: $E_{\text{exec}} = T_{i,d^*,f} \times P_{i,d^*,f}$. The total incremental energy cost is

$$E_{i,d^*,f} = E_{\text{exec}} + E_{\text{trans}} + E_{\text{transfer}}^{d^*} \tag{9}$$

Step ❺: Configuration Selection. The scheduler selects the frequency that minimizes a generalized cost function:

$$f^* = \arg\min_{f \in \mathcal{F}} \left[E_{i,d^*,f} \times (T_{i,d^*,f})^{\beta} \right] \tag{10}$$

Step ❻: State Update. Upon selecting (d^*, f^*), the System State Manager performs the following updates: (i) $T_{\text{available}}^{d^*} \leftarrow T_{\text{finish}}$; (ii) MSI coherence states are updated to reflect that t_i's output data will be in Modified state on device d^* upon completion; (iii) device d^*'s scheduled frequency set to f^*.

The two-phase process repeats for each subsequent task in topological order until the entire task graph is scheduled. We describe DEFT in the context of multi-GPU systems; however, the same scheduling procedure applies to single-GPU execution, where device selection is degenerate and only DVFS decisions remain.

3.3 MSI-Based Data Residency Tracking

To accurately model data transfer and avoid unnecessary inter-GPU communication, we maintain per-data-instance coherence state using a protocol inspired by the Modified-Shared-Invalid (MSI) cache coherence abstraction [5]. Unlike hardware MSI implementations that operate at cache-line granularity with synchronous runtime invalidations, the protocol operates at task-output granularity and maintains logical coherence through scheduler-driven state updates at scheduling time. Each logical data object (tasks' inputs and outputs) is associated with one of three per-device coherence states: (1) **Modified (M):** Device d holds the exclusive, up-to-date copy; all other devices are Invalid. (2) **Shared (S):** Device d holds a valid, read-only copy; other devices may also hold Shared copies. (3) **Invalid (I):** Device d does not have a valid copy; data must be transferred before use.

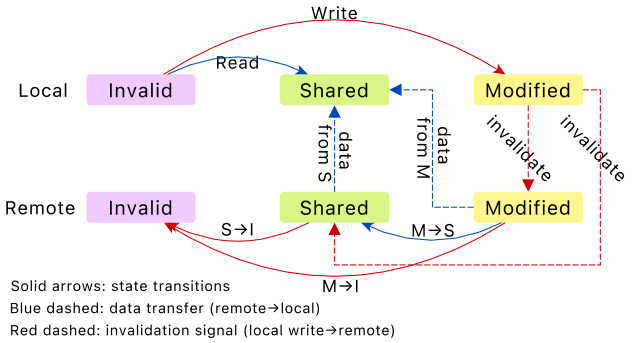


Figure 2: MSI coherence protocol for data instance state transitions in local and remote devices.

Figure 2 summarizes the state transitions and data flow. When task t_i executes on device d and produces output o:

- Write (produce): Device d's state for o transitions to Modified; all other devices' states transition to Invalid.
- Read (consume): If device d has state Invalid, it transitions to Shared and data is transferred from a device in state Modified or Shared; the source device's state remains Shared or is downgraded from Modified to Shared.

This protocol allows DEFT to maintain an up-to-date logical view of data residency and coherence across devices, enabling determination of data transfer requirements via simple state queries and accurate estimation of inter-GPU data transfer latency and energy.

3.4 Lightweight One-Time Profiling

DEFT relies on accurate task performance and power predictions to guide scheduling decisions. The Prediction Engine leverages models trained on hardware utilization metrics to estimate execution time and power consumption for each task across all $(d, f) \in \mathcal{D} \times \mathcal{F}$. Rather than exhaustive offline profiling or continuous runtime monitoring, we employ a lightweight profiling approach based on a single Nsight Compute [31] characterization run. We execute the application once and collect a fixed set of hardware performance counters for all target kernels at the default frequency, using a bounded number of launches per kernel to limit profiling overhead.

The profiling captures utilization rates across major GPU functional units, as described in § 4.1 and summarized in Table 1. Each metric is normalized as a fraction of the hardware's theoretical peak throughput. The resulting utilization vectors serve as input to the Prediction Engine.

In most cases, a task corresponds to an invocation of a single kernel, and the collected kernel-level metrics are directly used to characterize the task. In cases where a task comprises multiple kernel launches, we aggregate the kernel-level metrics using an execution-time weighted average to derive a single utilization vector that represents the task's overall hardware behavior. Because profiling is performed once per application and on a bounded number of launches per kernel, the cost scales with the number of distinct task types and profiled launches rather than DAG size, is

Table 1: Methodology for calculating hardware utilization ratios and required NCU metrics (L40S GPU example).

Component	Utilization Formula / Method	NCU Metrics Required	Peak Rate Constant
INT32 (ALU)	$\frac{\text{smsp\_\_thread\_inst\_executed\_pipe\_alu\_pred\_on.sum}}{\text{smsp\_\_cycles\_active.sum} \times \text{Peak\_Rate}_{\text{ALU}}}$	`smsp__thread_inst_executed_pipe_alu_pred_on.sum` `smsp__cycles_active.sum`	16 inst/cycle/SMSP
FP32 (FMA)	$\frac{\text{smsp\_\_thread\_inst\_executed\_pipe\_fma\_pred\_on.sum}}{\text{smsp\_\_cycles\_active.sum} \times \text{Peak\_Rate}_{\text{FMA}}}$	`smsp__thread_inst_executed_pipe_fma_pred_on.sum` `smsp__cycles_active.sum`	32 inst/cycle/SMSP
FP16	$\frac{\text{smsp\_\_thread\_inst\_executed\_pipe\_fp16\_pred\_on.sum}}{\text{smsp\_\_cycles\_active.sum} \times \text{Peak\_Rate}_{\text{FP16}}}$	`smsp__thread_inst_executed_pipe_fp16_pred_on.sum` `smsp__cycles_active.sum`	32 inst/cycle/SMSP
FP64	$\frac{\text{smsp\_\_thread\_inst\_executed\_pipe\_fp64\_pred\_on.sum}}{\text{smsp\_\_cycles\_active.sum} \times \text{Peak\_Rate}_{\text{FP64}}}$	`smsp__thread_inst_executed_pipe_fp64_pred_on.sum` `smsp__cycles_active.sum`	0.5 inst/cycle/SMSP
SFU (XU)	$\frac{\text{smsp\_\_thread\_inst\_executed\_pipe\_xu\_pred\_on.sum}}{\text{smsp\_\_cycles\_active.sum} \times \text{Peak\_Rate}_{\text{XU}}}$	`smsp__thread_inst_executed_pipe_xu_pred_on.sum` `smsp__cycles_active.sum`	4 inst/cycle/SMSP
Tensor (Half)	$\frac{\text{smsp\_\_pipe\_tensor\_op\_hmma\_cycles\_active.sum}}{\text{smsp\_\_cycles\_active.sum}}$	`smsp__pipe_tensor_op_hmma_cycles_active.sum` `smsp__cycles_active.sum`	N/A
Tensor (Int)	$\frac{\text{smsp\_\_pipe\_tensor\_op\_imma\_cycles\_active.sum}}{\text{smsp\_\_cycles\_active.sum}}$	`smsp__pipe_tensor_op_imma_cycles_active.sum` `smsp__cycles_active.sum`	N/A
DRAM	$\frac{\text{Achieved BW}}{\text{Peak BW}}$, where Achieved BW = $\frac{\text{dram\_\_bytes.sum}}{\text{gpu\_\_time\_duration.sum}}$	`dram__bytes.sum` `gpu__time_duration.sum`	864 GB/s
L2 Cache	$\frac{\text{Achieved BW}}{\text{Peak BW}}$ where Achieved BW = $\frac{\texttt{lts\_\_t\_bytes.sum}}{\texttt{gpu\_\_time\_duration.sum}}$	`lts__t_bytes.sum` `gpu__time_duration.sum`	45740 GB/s
L1 Cache	$\frac{\text{l1tex\_\_t\_sector\_hit\_rate.pct}}{100}$ (Hit Rate)	`l1tex__t_sector_hit_rate.pct`	N/A
Shared Memory	$\frac{\text{smsp\_\_sass\_l1tex\_pipe\_lsu\_wavefronts\_mem\_shared.sum}}{\text{l1tex\_\_data\_pipe\_lsu\_wavefronts.sum}}$	`smsp__sass_l1tex_pipe_lsu_wavefronts_mem_shared.sum` `l1tex__data_pipe_lsu_wavefronts.sum`	N/A

on the order of minutes for our benchmarks, and is amortized across subsequent executions. We leave applications with input-dependent task behavior or dynamically generated task graphs to future work.

4 Models

This section describes the three models comprising the Prediction Engine. We first present the offline profiling methodology used to construct the training datasets (§ 4.1), then detail the power model and performance model that predict task average power consumption and execution time across DVFS configurations (§ 4.2–4.3). Finally, we describe the DVFS transition cost model that quantifies frequency reconfiguration overhead (§ 4.4).

4.1 Training Dataset Generation

To capture task behavior across DVFS configurations, our power and performance models require hardware utilization metrics that characterize workload demands on GPU functional units. We adapt the microbenchmark-based characterization methodology from prior work [21, 23], extending it to support modern GPU architectures and the runtime prediction requirements of our scheduler. The training dataset is constructed using a suite of synthetic microbenchmarks (244 in total), each designed to exercise specific GPU functional units, including INT32 (ALU), FP32 (FMA), FP16, FP64, SFU (special functions), shared memory, L1 cache, L2 cache, DRAM, and Tensor cores (FP16/FP8 and INT8/INT4), at varying intensities. Specifically, we control the arithmetic intensity of each microbenchmark by changing the number of compute or memory operations performed within tight loops, allowing us to generate a wide spectrum of utilization values for each hardware component.

We begin by establishing a baseline hardware profile for each microbenchmark. Each benchmark executes at the default frequency configuration ($f_{\text{gpu}}^{\text{default}}, f_{\text{mem}}^{\text{default}}$) while we collect detailed hardware performance counters via NVIDIA Nsight Compute (NCU). These metrics (shown in Table 1) capture a kernel's resource consumption across the major GPU functional units, forming a hardware utilization signature independent of frequency scaling. From these raw counter values, we compute a normalized utilization vector $U = [U_j]$, where each component $U_j \in [0, 1]$ represents the fraction of peak theoretical throughput achieved by j. For example, integer (INT) pipeline utilization is computed using the number of executed integer ALU instructions:

$$U_{\text{INT}} = \frac{\text{ALU Instructions Executed}}{\text{Active Cycles} \times \text{Peak\_Rate}_{\text{ALU}}}, \tag{11}$$

where $\text{Peak\_Rate}_{\text{ALU}}$ is the architecture-specific throughput (e.g., 16 instructions/cycle/SM partition for L40S, from Table 1 [32]). We compute memory subsystem utilization by normalizing measured throughput against the theoretical peak bandwidth. DRAM theoretical peak bandwidth BW_{DRAM} is derived from the external memory interface specifications: $\text{BW}_{\text{DRAM}} = (\text{R}_{\text{transfer}} \times \text{W}_{\text{bus}})/8$, where $\text{R}_{\text{transfer}}$ is the effective transfer rate (MT/s) and W_{bus} is the aggregate memory interface width (in bits). L2 cache peak bandwidth BW_{L2} represents the aggregate throughput of the on-chip interconnect. It is calculated as the product of the data path width per SM, the total number of SMs, and the GPU's maximum graphics clock:

$$\text{BW}_{\text{L2}} = \left(\frac{\text{Bytes}}{\text{Cycle} \times \text{SM}}\right) \times \text{N}_{\text{SMs}} \times f_{\text{gpu}}^{\max}. \tag{12}$$

We execute each microbenchmark across the full spectrum of supported $(f_{\text{gpu}}, f_{\text{mem}})$, recording the execution time and average steady-state power P. These measurements are combined with the baseline utilization vector U (profiled at the default frequency) to generate training tuples: $(U, f_{\text{gpu}}, f_{\text{mem}}, P)$ for the power model and $(U, f_{\text{gpu}}, f_{\text{mem}}, S)$ for the performance model, where S is the

speedup factor relative to the execution time at the default frequency.

4.2 Power Model

We employ an analytical power model adapted from prior work [21, 23] to predict GPU power consumption $P(U, f_{\text{gpu}}, f_{\text{mem}})$ given workload utilization U and DVFS configuration. The model decomposes total GPU power into three components: static leakage, frequency-dependent idle power, and utilization-dependent dynamic power. Formally, for frequency domains $k \in \{\text{core}, \text{memory}\}$:

$$P_{\text{total}} = \sum_{k \in \text{Domains}} (\gamma_{\text{static},k} \cdot V_k + \gamma_{\text{idle},k} \cdot f_k \cdot V_k^2 + \sum_{j \in C_k} \omega_{j,k} \cdot U_j \cdot f_k \cdot V_k^2) \quad (13)$$

where f_k and V_k are the clock frequency and normalized supply voltage for domain k, and C_k denotes the set of functional units in domain k. The model parameters are: $\gamma_{\text{static},k}$ and $\gamma_{\text{idle},k}$ (static and idle power coefficients), and $\omega_{j,k}$ (dynamic power coefficient per unit utilization for component j).

In the proposed power model, the voltage–frequency relationships and power-model coefficients are jointly estimated through an iterative optimization procedure that alternates between two steps. Given current coefficient estimates $\{\gamma, \omega\}$, the supply voltage V_k at each frequency f_k is obtained by treating the power equation as a nonlinear system in V_k. We apply constrained nonlinear least-squares regression, enforcing monotonic dependence of V_k on f_k, to fit voltage values that minimize prediction error over the training set. This yields a normalized voltage-frequency (V-F) curve satisfying $V_k(f_{\text{ref}}) = 1.0$ for a reference frequency f_{ref}. With $V_k(f_k)$ fixed, the power equation becomes linear in coefficients $\{\gamma, \omega\}$. Non-negative least-squares regression is then used to estimate the coefficients that minimize prediction error. These two steps are repeated iteratively until convergence (i.e., coefficient updates fall below a predefined threshold). The converged parameters define the final power model $P(U, f_{\text{gpu}}, f_{\text{mem}})$ used for runtime prediction.

4.3 Performance Model

Unlike power consumption, task execution time exhibits complex, non-linear dependencies on multiple factors, such as task characteristics, instruction throughput, memory bandwidth, latency hiding efficiency, and inter-component contention. These nonlinear interactions preclude simple analytical formulations, motivating a machine-learning-based approach. An XGBoost gradient-boosted decision tree regressor is employed to learn the mapping $(U, f_{\text{gpu}}, f_{\text{mem}}) \rightarrow S$, where S is the speedup factor relative to execution at the default frequency. XGBoost is well suited to this task due to its ability to capture non-linear feature interactions through tree ensembles, handle high-dimensional utilization features efficiently, and provide robust predictions with limited training data [14].

The model takes as input the normalized utilization metrics from Table 1 along with the normalized GPU and memory frequency ratios $R_{\text{gpu}} = f_{\text{gpu}}/f_{\text{gpu}}^{\text{default}}$ and $R_{\text{mem}} = f_{\text{mem}}/f_{\text{mem}}^{\text{default}}$. Normalizing frequencies allows the model to learn relative scaling trends rather than overfitting to absolute clock values. The model is trained on the full dataset $\{(U, R_{\text{gpu}}, R_{\text{mem}}, S)\}$ via gradient boosting with squared error loss. Training across diverse microbenchmarks and frequency configurations enables accurate speedup prediction for unseen kernels based solely on their utilization profiles at the default frequency setting, eliminating the need for exhaustive per-kernel profiling across the entire DVFS space.

4.4 DVFS Transition Cost Model

Accurate characterization of DVFS transition overhead is a prerequisite for effective cost-aware scheduling in DEFT to distinguish between profitable and counterproductive scaling decisions. Unfortunately, GPU vendors do not publicly disclose DVFS transition mechanisms or timing characteristics. Prior work [42] has shown that transition latency depends on GPU architecture and voltage regulator design, frequency change magnitude, and transition direction (upscaling vs. downscaling). As a result, assuming instantaneous transitions or symmetric, linearly scaled costs leads to inaccurate performance estimates and suboptimal scheduling decisions. The challenge is exacerbated by modern architectures (e.g., Ada Lovelace) that expose fine-grained frequency steps (15 MHz), creating a vast configuration space. To avoid the prohibitively high cost of exhaustive profiling, we develop a lightweight empirical methodology to construct device-specific transition cost models.

4.4.1 Measurement Methodology. Direct measurement of hardware transition latency is challenging because software interfaces such as NVIDIA Management Library (NVML) [33] provide only coarse-grained telemetry (e.g., ∼100 ms sampling resolution on L4 and L40S). We therefore infer transition latency indirectly by observing the transient impact of frequency scaling on a continuous stream of lightweight kernels. The procedure consists of three stages. First, for each target frequency f_{target}, we establish a steady-state baseline by executing a sequence of probe kernels and computing the mean execution time $T_{\text{baseline}}(f_{\text{target}})$. Next, we issue a frequency scaling command ($f_{\text{start}} \rightarrow f_{\text{target}}$) and immediately launch a dense stream of asynchronous probe kernels. Each probe is timestamped at dispatch and completion using CUDA Events, producing a high-resolution execution trace during the transition. Finally, we perform post-execution analysis to identify the convergence point. The transition latency is defined as the elapsed time from the first probe launch to the first probe whose execution time stabilizes within a tolerance band (±5%) of $T_{\text{baseline}}(f_{\text{target}})$.

4.4.2 Model Derivation and Instantiation. Figure 3 shows the measured transition latencies on the L4 and L40S GPUs used in our evaluation. The data points correspond to a selected set of frequency pairs used to train and validate the multi-tiered transition cost model. Each configuration is measured 50 times, and the median value is reported to filter outliers from system jitter. Measurements across the two types of GPUs reveal several key insights. First, transition behavior is strongly dependent on GPU design, reflecting differences in voltage regulator design, thermal limits, and power delivery networks [42]. Second, transitions exhibit clear directional asymmetry: frequency upscaling is consistently faster than downscaling for the same $\Delta f = |f_{\text{target}} - f_{\text{start}}|$. Third, latency does not scale linearly with Δf; instead, it shows piecewise behavior with discrete thresholds. Finally, the relationship is non-monotonic: larger Δf does not always imply higher latency, reflecting complex interactions across multiple voltage domains and power-management

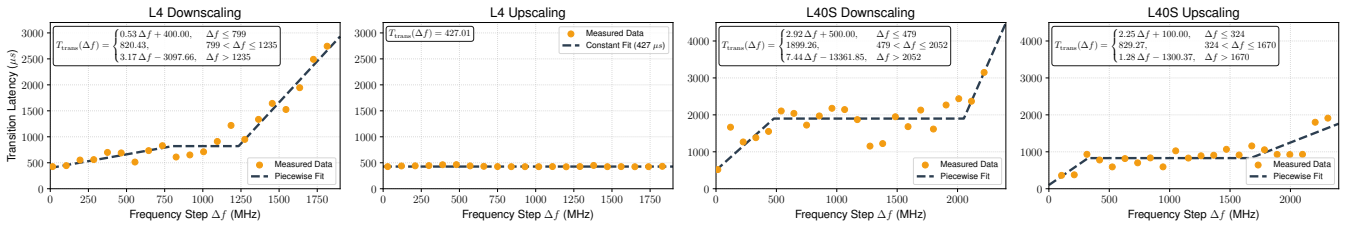


Figure 3: Measured DVFS transition latencies on NVIDIA L4 and L40S GPUs, showing asymmetric and non-monotonic behavior.

firmware. Guided by these observations, we construct empirical, GPU-specific transition models by fitting multi-tier piecewise functions, as illustrated in Figure 3. These models are integrated into the Prediction Engine (§ 3.1) to enable cost-aware evaluation of DVFS decisions during scheduling.

5 Experimental Setup

Hardware Platform. The evaluation was conducted on two single-node multi-GPU systems: (a) a node with eight NVIDIA L4 GPUs, and (b) a node with four NVIDIA L40S GPUs, both hosted on dual-socket Intel Xeon Gold 6548N platforms (2 × 64 cores). Both GPUs expose 15 MHz-granular core-frequency ranges (L4: 210–2040 MHz, L40S: 210–2520 MHz) at default memory frequencies (L4: 6251 MHz, L40S: 9001 MHz). Reducing memory frequency to 405 MHz narrows the core range to 210–645 MHz on L4 and 210–405 MHz on L40S. The 405 MHz memory state is a low-power idle mode that cannot be reliably enforced via the application-clocks API (the memory falls back to its maximum frequency at runtime), so we exclude it from scheduling. By default, both the GPU core and the memory operate at the maximum supported frequencies.

Both systems run Ubuntu 24.04.3 LTS with NVIDIA driver version 565.57.01. CUDASTF is compiled using `nvcc` 12.4 and `g++` 13.3.0. DVFS control is implemented using NVML, specifically through the `nvmlDeviceSetApplicationsClocks()` API call [33]. Empirical characterization shows that the hardware power and energy counters on L4 and L40S GPUs update at ∼100 ms granularity. We therefore sample every 50 ms via `nvmlDeviceGetTotalEnergyConsumption` to capture each increment reliably.

Benchmarks. We evaluate DEFT using four scientific applications with diverse computational characteristics and data access patterns. **Cholesky** factorization [1, 9] performs blocked dense linear algebra using CUBLAS/CUSOLVER routines (N=32768, tile size 2048; 816 tasks in total). Finite-Difference Time-Domain **FDTD** [49] simulates electromagnetic wave propagation via 3D stencil updates (L4: 512^3, L40S: 1024^3; 200 iterations, 1200 tasks in total). Conjugate Gradient **CG** [25] solves large sparse linear systems in CSR format using sparse matrix–vector multiplications, dot products, and vector operations (L4: 210M unknowns, 169 tasks; L40S: 450M, 175 tasks). **miniWeather** [30] models atmospheric dynamics on a 2D domain (L4: 13000×11000, 360 tasks; L40S: 16384×13107, 240 tasks) with heterogeneous domain decomposition, creating irregular workload distribution and asymmetric communication patterns.

6 Evaluation

In this section, we first evaluate how effectively DEFT achieves various energy–performance trade-off objectives across diverse benchmarks and multi-GPU platforms (§ 6.1). We then validate the accuracy of our predictive models (§ 6.2). Finally, we analyze the computational complexity and overhead introduced by the proposed scheduling approach (§ 6.3).

6.1 Effectiveness of DEFT Scheduling

We focus the evaluation on multi-GPU scheduling policies because DEFT is designed to jointly optimize task placement and DVFS across devices. Existing DVFS-based schedulers primarily target single-GPU execution and do not consider task placement, inter-GPU data movement, or multi-device contention, making direct comparison in multi-GPU settings infeasible. In the single-GPU case, DEFT reduces to a DVFS-aware scheduler; however, the primary contribution lies in coordinating DVFS with multi-GPU task placement under explicit slack and throughput constraints.

We evaluate DEFT against three scheduling policies provided by the native CUDASTF runtime: **Random**, which assigns tasks to devices randomly; **Round Robin (RR)**, which distributes tasks cyclically; and **HEFT**, the Heterogeneous Earliest Finish Time algorithm [40], which represents a canonical makespan-oriented task graph scheduler and assigns each task to the device that minimizes its earliest finish time. None of these policies incorporates DVFS control, relying on the hardware default power management. To isolate the contribution of DEFT's joint scheduling, we implement two DVFS-augmented HEFT variants. **HEFT-G** applies a uniform, device-wide frequency that minimizes the target objective across all tasks, representing global frequency tuning without task differentiation. **HEFT-PT** applies per-task optimal DVFS after placement, selecting the best frequency for each task independently but without slack or throughput awareness. Both retain HEFT's placement decisions and ignore DVFS transition overheads. We show that DEFT's gains arise from constrained, globally aware scheduling rather than simply adding DVFS to existing placement strategies.

When DVFS is disabled, DEFT's device selection policy converges to the same task placement as HEFT. This has two important implications: (1) it confirms that DEFT preserves the placement quality of established makespan-oriented heuristics under fixed operating conditions, and (2) it ensures that the comparison between HEFT and DEFT isolates the contribution of DVFS-aware scheduling. When DVFS is enabled, DEFT's scheduling decisions diverge from HEFT because execution times become frequency-dependent, altering device availability and task timing. By coordinating placement feasibility and frequency selection, DEFT consistently outperforms HEFT (the fastest baseline) across multiple objectives. For example, DEFT reduces energy by 14.8% on L40S and 4.8% on L4 and EDP by 9.9% and 3.7% (geometric mean), respectively, while remaining within 1.5% of HEFT's performance.

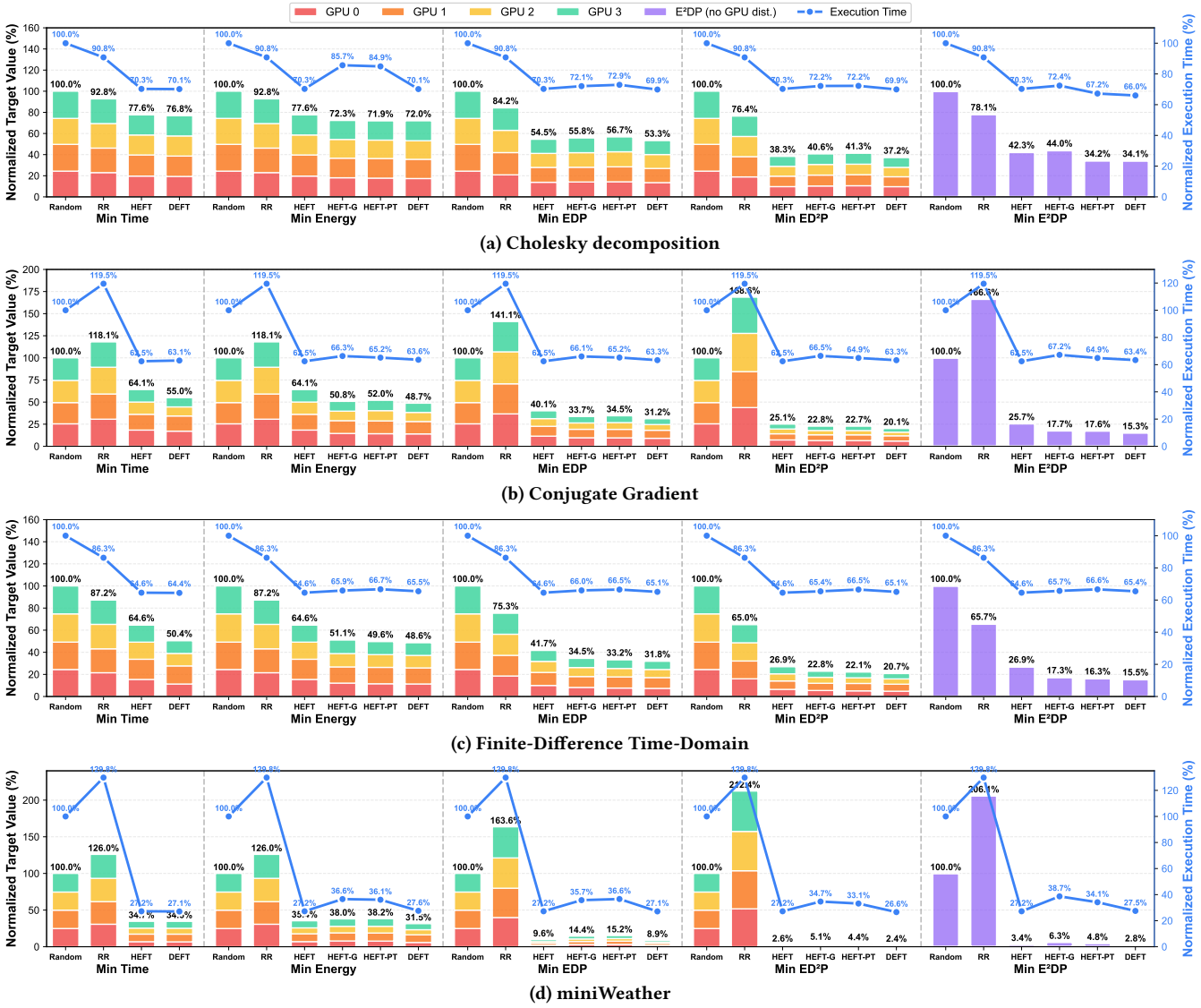


Figure 4: Normalized execution time and energy-performance metrics for the four different schedulers on NVIDIA L40S GPUs (normalized to Random scheduler). In the Min Time case, bars show per-GPU energy consumption and the line shows normalized makespan. For energy-related objectives (Min Energy, EDP, E^2DP, ED^2P**), bars show the normalized objective value. Note that** E^2DP **involves squared energy and is computed from total system energy, thus cannot be decomposed per GPU.**

6.1.1 Results on NVIDIA L40S GPUs. The evaluation is configured under five optimization objectives: Energy-only ($\beta = 0$), E^2DP ($\beta = 0.5$), EDP ($\beta = 1$), ED^2P ($\beta = 2$), and Time-only ($\beta \to \infty$). Figure 4 reports the normalized objective values and makespan for the four applications on the four-GPU NVIDIA L40S node.

Cholesky. Under the time-only objective, DEFT converges to the same decisions as HEFT and achieves comparable performance. Under energy-related objectives, DEFT reduces total energy by 7.2% relative to HEFT (2.3% in EDP, 3.0% in ED^2P, and 3.9% in E^2DP) by exploiting task slack for mild frequency reduction, while maintaining similar execution time. HEFT-G and HEFT-PT reduce more energy than HEFT by exploiting DVFS; however, they incur execution time inflation due to aggressive frequency scaling without slack or throughput awareness, especially under resource contention.

CG. Under the time-only objective, DEFT achieves execution times comparable to HEFT while reducing energy consumption by 14.3%. Under energy-related objectives, DEFT delivers substantial improvements, reducing energy by 24.0%, EDP by 22.3%, ED^2P by 19.7%, and E^2DP by 40.6%, with only a 1.3% increase in execution time compared to HEFT. Due to limited task-level parallelism in CG, some GPUs periodically become idle. DEFT leverages these intervals to transition idle devices into lower-power states, substantially reducing energy consumption on GPUs 2 and 3, as shown in the per-GPU energy breakdown. HEFT-PT performs slightly

worse than HEFT-G despite per-task frequency selection: frequent DVFS transitions across short sparse kernels introduce additional overhead, while HEFT-G avoids transition costs but cannot exploit kernel heterogeneity. DEFT, by contrast, bounds slowdown using slack and contention awareness, achieving balanced energy savings without degrading performance.

FDTD. FDTD is memory-bound and benefits strongly from DVFS. When optimizing for performance, DEFT matches HEFT's execution time while reducing energy consumption by 21.9%. Under energy-related objectives, DEFT achieves 24.7% reduction in energy, 23.6% in EDP, 23.0% in ED^2P, and 42.6% in E^2DP, with negligible (1.1%) slowdown in comparison to HEFT. Although HEFT-G and HEFT-PT improve energy efficiency over HEFT, they incur execution-time penalties. HEFT-PT, for instance, reduces energy by 23.1% but degrades performance by 4%, reflecting the absence of slack and contention awareness. In contrast, DEFT maintains near-optimal performance while achieving additional energy savings of approximately 2–5% over both HEFT variants.

miniWeather features irregular workload distribution due to heterogeneous domain decomposition. Random and RR perform poorly because device-agnostic placement induces excessive inter-GPU communication. HEFT significantly improves performance by prioritizing data locality. When optimizing for performance, DEFT matches HEFT by selecting devices that prioritize earliest start time and locality. Its joint optimization of placement and frequency further reduces energy by 11.1% with only a 1.4% performance overhead. Under energy-related objectives, DEFT delivers 8.0% reduction in EDP, 9.5% in ED^2P, and 18.0% in E^2DP. The irregular workload creates substantial slack, which DEFT selectively exploits by lowering frequency on non-critical tasks while preserving the critical path. In contrast, HEFT-G and HEFT-PT slow both critical and non-critical tasks, thereby significantly extending execution time. This inflated makespan increases system idle energy and reduces overall efficiency. For example, compared to HEFT-G, DEFT improves performance by 17% and reduces energy by 25%.

We also evaluated a Mixture-of-Experts workload; however, sustained high utilization triggers hardware power limiting and GPU Boost, which overrides software DVFS control, restricting DEFT's ability to adjust frequency. This behavior identifies compute-saturated, power-bound workloads as a boundary condition for DVFS-based optimization.

6.1.2 Results on NVIDIA L4 GPUs. Figure 5 summarizes the results for all four benchmarks on the 8-GPU L4 node using scatter plots in normalized target–time space, where points closer to the bottom-left corner indicate better overall trade-offs. Across all benchmarks, Random and Round Robin schedulers consistently occupy upper right regions of the plot, reflecting their inability to account for task characteristics, device heterogeneity, or energy–performance trade-offs. HEFT achieves strong performance-oriented results in most of the benchmarks. In contrast, DEFT consistently achieves improved trade-offs across multiple optimization targets.

For compute-bound workloads such as **Cholesky**, DEFT slightly outperforms HEFT (by ~1%) under energy-related objectives. For **CG**, DEFT significantly improves energy-related metrics (e.g., 22.7% energy reduction with only 1.2% impact on execution time), demonstrating the effectiveness of slack-aware DVFS decisions and idle power management on iterative and memory-sensitive workloads.

Notably, for **FDTD**, DEFT not only improves energy efficiency but also outperforms HEFT when optimizing for execution time, achieving a shorter makespan. This behavior can be attributed to the interaction between workload characteristics and GPU power management dynamics. FDTD is memory-bound and exhibits relatively low and fluctuating compute utilization. As a result, operating persistently at the maximum GPU frequency does not yield optimal performance. Without explicit DVFS control, the GPU's hardware power and thermal management mechanisms autonomously adjust the operating frequency in response to instantaneous load variations, leading to pronounced frequency oscillations (e.g., 531 frequency transitions observed under HEFT vs. 88 under DEFT) and execution-time variability. In contrast, DEFT proactively selects a slightly lower but stable operating frequency based on predictive modeling, suppressing runtime frequency fluctuations and improving execution stability. This highlights an advantage of proactive DVFS management: it uncovers performance opportunities inaccessible when frequency control is left to hardware defaults.

We evaluate **miniWeather** (average task parallelism of 6) across 4-GPU and 8-GPU configurations to examine both resource-constrained and resource-abundant scenarios, as well as the scalability of DEFT under varying hardware availability. With 4 GPUs, the system operates under resource contention, and DEFT tightens slack bounds to limit frequency scaling and prevent excessive execution-time inflation. As a result, DEFT achieves performance and energy efficiency comparable to HEFT. Disabling throughput awareness degrades performance by 9.6% relative to HEFT, confirming that contention-aware slack refinement is essential under resource pressure. With 8 GPUs, the system has sufficient resources to accommodate the workload, enabling DEFT to exploit task slack more effectively. As a result, DEFT achieves 8.4% energy reduction compared to HEFT while maintaining comparable performance.

6.2 Model Accuracy Validation

We validate the proposed power and performance models by comparing the predictions against measured values across the full spectrum of DVFS configurations over the four applications. We use Mean Absolute Percentage Error (MAPE) as the evaluation metric, defined as $\text{MAPE} = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{y_i-\hat{y}_i}{y_i}\right| \times 100\%$, where y_i and $\hat{y}_i$ denote the measured and predicted values. Table 2 reports the per-application MAPE of both models, averaged across the L4 and L40S platforms. The analytical power model maintains an MAPE of 8.1%, while the performance model achieves an MAPE of 4.3% across all benchmarks and GPU platforms, indicating that the proposed models capture the non-linear scaling characteristics of the hardware and provide sufficiently accurate predictions to support effective energy-aware scheduling decisions.

6.3 Scheduling Overhead Analysis

A critical concern for any online scheduling system is the runtime overhead introduced by scheduling decisions. We analyze DEFT's overhead from both theoretical complexity and empirical measurement perspectives.

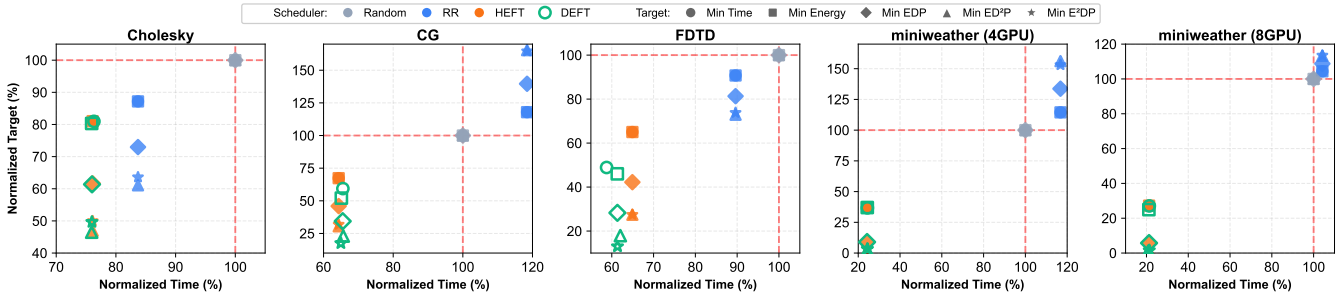


Figure 5: Normalized performance and optimization objectives of four scheduling algorithms on NVIDIA L4 GPUs. For Random, Round-robin (RR), and HEFT, the Min Time and Min Energy configurations produce identical data points because both plot energy (y-axis) against makespan (x-axis); these schedulers operate at fixed frequency and thus have the same energy-time coordinates regardless of the stated objective. Consequently, their markers (circles and squares) overlap in the scatter plots.

Table 2: Per-application MAPE (%) of the power and performance models, averaged across L4 and L40S.

Metric	Cholesky	CG	FDTD	miniWeather
Power	6.7	8.3	9.5	7.9
Performance	3.8	4.2	4.7	4.5

As described in § 3.2, DEFT employs a two-phase approach. Phase 1 iterates over $|\mathcal{D}|$ devices to select the target device, phase 2 evaluates $|\mathcal{F}|$ frequency configurations for the selected device. For a task graph with N tasks, the overall complexity is $O(N \times (|\mathcal{D}| + |\mathcal{F}|))$. In our experimental platforms, $|\mathcal{D}| \in \{4, 8\}$ and $|\mathcal{F}|$ ranges from 122 to 154 discrete DVFS configurations. Since both are determined by the hardware platform and independent of the application, scheduling cost scales linearly with task count for a given system configuration.

We evaluate the runtime overhead of DEFT by measuring the average scheduling time spent per task, rather than reporting a percentage of total execution time. Percentage-based metrics are sensitive to task granularity and can be misleading, where fine-grained workloads amplify relative scheduling costs. On the L4 and L40S, DEFT incurs an average scheduling overhead of 61 μs and 89 μs per task, respectively. It is 2–3 orders of magnitude smaller than typical kernel execution times in our benchmarks (7–112 ms), confirming DEFT's viability for online scheduling. This overhead is incurred only at task submission time and is independent of task execution time, scaling primarily with $|\mathcal{D}|$ and $|\mathcal{F}|$.

7 Related Work

Our work relates to two primary research areas: GPU power and performance modeling and energy-aware runtime systems. We structure the discussion around these topics and highlight how our approach differs from prior work.

GPU Power and Performance Modeling. Several prior works have developed power and performance models to characterize the impact of DVFS on GPU energy consumption and execution time. Empirical studies show that the optimal GPU frequency depends strongly on application characteristics and architecture [29]. Analytical and machine-learning models predict power and performance across the DVFS space from hardware counters collected at a single frequency, using functional-component decomposition [21, 23], neural networks with kernel clustering [48], microbenchmark-calibrated architectural parameters [44], or offline multi-objective frequency selection [2]. To avoid runtime profiling, static approaches derive features directly from kernel code, including OpenCL operations [18], LSTM-encoded PTX sequences [22], and DCGM–PTX fusion for per-application frequency selection [45].

Energy-Aware Runtime Systems. Several works explore integrating DVFS control into GPU programming models and runtime systems to enable energy-aware execution. SYnergy [19] extends SYCL with a per-kernel energy interface that drives frequency selection through compile-time analytical models, and phase-based scaling [10] groups similar kernels under a uniform frequency to amortize DVFS overhead. Complementary efforts target placement rather than DVFS: unbalanced power capping with StarPU [6, 17], genetic-augmented HEFT to mitigate kernel interference [4], SM-level load balancing [26], and core/warp throttling for memory-intensive workloads [39]. For heterogeneous multi-CPU systems, JOSS [12] and its follow-up SWEEP [13] jointly schedule tasks with CPU/memory DVFS under energy–performance constraints.

In contrast to prior work, which confines DVFS and placement to single-GPU settings or treats them as separate concerns, DEFT integrates both within a unified, cost-aware runtime framework for multi-GPU task graphs.

8 Conclusion

Multi-GPU architectures deliver massive computational throughput but pose significant challenges for energy-efficient execution, due to the tight coupling between task placement, DVFS configuration, and inter-GPU data movement. This work presents DEFT, an energy-aware scheduling framework that jointly coordinates task-to-device placement and per-GPU DVFS configuration to achieve user-specified energy–performance trade-offs in task-based multi-GPU applications. By integrating slack and throughput awareness with explicit modeling of execution cost, data movement and DVFS transition cost within a unified cost model, DEFT enables coordinated, task-granularity–aware decisions that preserve task parallelism and data locality while avoiding excessive slowdown under contention. The evaluation on NVIDIA L4 and L40S multi-GPU systems demonstrates that DEFT consistently improves energy efficiency across multiple objectives compared to native CUDASTF scheduling policies, while maintaining high performance.

Acknowledgments

This work was supported in part by the Swedish Foundation for Strategic Research (SSF) through PRIDE (CHI19-0048), the Swedish Research Council (VR) through P4PIM (2020-04892), and the EuroHPC Joint Undertaking through DARE under grant agreement No. 101202459. Computational resources were provided by the Minerva cluster at the Department of Computer Science and Engineering, Chalmers University of Technology.

References

[1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Fast Cholesky factorization on GPUs for batch and native modes in MAGMA. *Procedia Computer Science* 80 (2016), 93–101.

[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. doi:10.1109/HPEC55821.2022.9926317

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

[4] Negar Baradar Alizadeh and Mahmoud Momtazpour. 2024. Multi-Objective Concurrent Kernel Scheduling for Multi-GPU Systems. In *2024 32nd International Conference on Electrical Engineering (ICEE)*. 1–6. doi:10.1109/ICEE63041.2024.10667973

[5] Cédric Augonnet, Andrei Alexandrescu, Albert Sidelnik, and Michael Garland. 2024. CUDASTF: Bridging the Gap Between CUDA and Task Parallelism. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–17. doi:10.1109/SC41406.2024.00049

[6] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1631 doi:10.1002/cpe.1631

[7] Srikant Bharadwaj, Shomit Das, Kaushik Mazumdar, Bradford M Beckmann, and Stephen Kosonocky. 2023. Predict; don't react for enabling efficient fine-grain dvfs in gpus. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 253–267.

[8] Joshua Dennis Booth, Jagadish Kotra, Hui Zhao, Mahmut Kandemir, and Padma Raghavan. 2015. Phase detection with hidden markov models for dvfs on many-core processors. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 185–195.

[9] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* 35, 1 (2009), 38–53. doi:10.1016/j.parco.2008.10.002

[10] Lorenzo Carpentieri, Antonio De Caro, Majid Salimi Beni, Kaijie Fan, and Biagio Cosenza. 2025. Phase-Based Frequency Scaling for Energy-Efficient Heterogeneous Computing. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 824–836. doi:10.1109/IPDPS64566.2025.00078

[11] Chao Chen, Chris Porter, and Santosh Pande. 2022. CASE: a compiler-assisted SchEduling framework for multi-GPU systems. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 17–31. doi:10.1145/3503221.3508423

[12] Jing Chen, Madhavan Manivannan, Bhavishya Goel, and Miquel Pericàs. 2023. JOSS: Joint Exploration of CPU-Memory DVFS and Task Scheduling for Energy Efficiency. In *Proceedings of the 52nd International Conference on Parallel Processing* (Salt Lake City, UT, USA) *(ICPP '23)*. Association for Computing Machinery, New York, NY, USA, 828–838. doi:10.1145/3605573.3605586

[13] Jing Chen, Madhavan Manivannan, Bhavishya Goel, and Miquel Pericàs. 2024. SWEEP: Adaptive Task Scheduling for Exploring Energy Performance Trade-offs. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 325–336. doi:10.1109/IPDPS57955.2024.00036

[14] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. doi:10.1145/2939672.2939785

[15] Gilberto Contreras and Margaret Martonosi. 2008. Characterizing and improving the performance of intel threading building blocks. In *2008 IEEE International Symposium on Workload Characterization*. IEEE, 57–66.

[16] Georges Da Costa and Jean-Marc Pierson. 2015. DVFS Governor for HPC: Higher, Faster, Greener. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 533–540. doi:10.1109/PDP.2015.73

[17] Albert d'Aviau de Piolant, Hayfa Tayeb, Berenger Bramas, Mathieu Faverge, Abdou Guermouche, and Amina Guermouche. 2025. Improving energy efficiency of HPC applications using unbalanced GPU power capping. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 820–829. doi:10.1109/IPDPSW66978.2025.00132

[18] 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 '19)*. Association for Computing Machinery, New York, NY, USA, Article 52, 10 pages. doi:10.1145/3337821.3337833

[19] Kaijie Fan, Marco D'Antonio, Lorenzo Carpentieri, Biagio Cosenza, Federico Ficarelli, and Daniele Cesarini. 2023. SYnergy: Fine-Grained Energy-Efficient Heterogeneous Computing for Scalable Energy Saving. In *SC23: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. doi:10.1145/3581784.3607055

[20] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 212–223. doi:10.1145/277650.277725

[21] 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. doi:10.1109/HPCA.2018.00072

[22] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás. 2019. GPU Static Modeling Using PTX and Deep Structured Learning. *IEEE Access* 7 (2019), 159150–159161. doi:10.1109/ACCESS.2019.2951218

[23] 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. doi:10.1109/TPDS.2019.2917181

[24] Hamblen, Anna-Lena. 2014. Total Cost of Ownership in High Performance Computing. (2014). Presentation/Technical Report, University of Hamburg. Explicitly states "Supercomputer's lifelong energy costs almost equal the investment costs.".

[25] Magnus R Hestenes and Eduard Stiefel. 1952. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Standards* 49, 6 (1952), 409–435.

[26] Yanhui Huang, Bing Guo, and Yan Shen. 2020. GPU Energy optimization based on task balance scheduling. *Journal of Systems Architecture* 107 (2020), 101808. doi:10.1016/j.sysarc.2020.101808

[27] Joseph John, Josh Milthorpe, Thomas Herault, and George Bosilca. 2024. Multi-GPU work sharing in a task-based dataflow programming model. *Future Generation Computer Systems* 156 (2024), 313–324. doi:10.1016/j.future.2024.03.017

[28] 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. arXiv:https://www.science.org/doi/pdf/10.1126/science.aba3758 doi:10.1126/science.aba3758

[29] 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. doi:10.1016/j.dcan.2016.10.001

[30] Matthew R Norman, Jeff Larkin, and Isaac Lyngaas. 2020. miniWeather. https://github.com/mrnorman/miniWeather OSTI ID: 1631691.

[31] NVIDIA Corporation. [n. d.]. NVIDIA Nsight Compute. ([n. d.]). https://docs.nvidia.com/nsight-compute/index.html

[32] NVIDIA Corporation. 2023. NVIDIA ADA GPU ARCHITECTURE - Designed to deliver outstanding gaming and creating, professional graphics, AI, and compute performance. (2023). https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

[33] NVIDIA Corporation. 2025. NVIDIA Management Library (NVML) API Reference Guide. (2025). https://docs.nvidia.com/deploy/pdf/NVML_API_Reference_Guide.pdf Version used: vR580, September 2025.

[34] OpenMP Architecture Review Board. 2018. OpenMP Application Program Interface. Version 5.0.

[35] Pratyush Patel, Zibo Gong, Syeda Rizvi, Esha Choukse, Pulkit Misra, Thomas Anderson, and Akshitha Sriraman. 2023. Towards Improved Power Management in Cloud GPUs. *IEEE Comput. Archit. Lett.* 22, 2 (July 2023), 141–144. doi:10.1109/LCA.2023.3278652

[36] Josep M. Perez, Vicenç Beltran, Jesus Labarta, and Eduard Ayguadé. 2017. Improving the Integration of Task Nesting and Dependencies in OpenMP. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 809–818. doi:10.1109/IPDPS.2017.69

[37] Efe Sencan, Dhruva Kulkarni, Ayse Coskun, and Kadidia Konate. 2025. Analyzing GPU Utilization in HPC Workloads: Insights from Large-Scale Systems. In *Practice and Experience in Advanced Research Computing 2025: The Power of Collaboration (PEARC '25)*. Association for Computing Machinery, New York, NY, USA, Article 14, 8 pages. doi:10.1145/3708035.3736010

[38] Arman Shehabi, Sarah J. Smith, Adam Hubbard, Diana Sartor, et al. 2024. *2024 United States Data Center Energy Usage Report.* Technical Report LBNL-2001637. Lawrence Berkeley National Laboratory. https://eta-publications.lbl.gov/sites/default/files/2024-12/lbnl-2024-united-states-data-center-energy-usage-report_1.pdf

[39] Seokwoo Song, Minseok Lee, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Energy-efficient scheduling for memory-intensive GPGPU workloads. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[40] H. Topcuoglu, S. Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (2002), 260–274. doi:10.1109/71.993206

[41] Ilyas Turimbetov, Mohamed Wahib, and Didem Unat. 2025. A Device-Side Execution Model for Multi-GPU Task Graphs. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 384–396. doi:10.1145/3721145.3730426

[42] Daniel Velicka, Ondrej Vysocky, and Lubomir Riha. 2025. Methodology for GPU Frequency Switching Latency Measurement. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 830–839. doi:10.1109/IPDPSW66978.2025.00133

[43] Jorge Villarrubia, Luis Costero, Francisco D. Igual, and Katzalin Olcoz. 2025. Leveraging Multi-Instance GPUs through moldable task scheduling. *J. Parallel and Distrib. Comput.* 204 (2025), 105128. doi:10.1016/j.jpdc.2025.105128

[44] 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. doi:10.1109/TPDS.2020.3004623

[45] Qiang Wang, Laiyi Li, Weile Luo, Yijia Zhang, and Bingqiang Wang. 2024. DSO: A GPU Energy Efficiency Optimizer by Fusing Dynamic and Static Information. In *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*. 1–6. doi:10.1109/IWQoS61813.2024.10682917

[46] Yidi Wang, Mohsen Karimi, Yecheng Xiang, and Hyoseung Kim. 2021. Balancing energy efficiency and real-time performance in GPU scheduling. In *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 110–122.

[47] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 119–130.

[48] 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. doi:10.1109/HPCA.2015.7056063

[49] Kane Yee. 1966. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE Transactions on Antennas and Propagation* 14, 3 (1966), 302–307. doi:10.1109/TAP.1966.1138693

[50] Hadi Zamani, Laxmi Bhuyan, Jieyang Chen, and Zizhong Chen. 2023. GreenMD: Energy-efficient Matrix Decomposition on Heterogeneous Multi-GPU Systems. *ACM Trans. Parallel Comput.* 10, 2, Article 12 (June 2023), 23 pages. doi:10.1145/3583590