CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Multi-Objective Scheduling Framework for Energy–Performance Optimization in Multi-GPU Task Graphs

Master's thesis in Computer Science and Engineering

Long Cheng, Yicheng Li

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

A Multi-Objective Scheduling Framework for Energy–Performance Optimization in Multi-GPU Task Graphs

Long Cheng, Yicheng Li

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

A Multi-Objective Scheduling
Framework for Energy–Performance
Optimization in Multi-GPU
Task Graphs
Long Cheng, Yicheng Li

Supervisor: Jing Chen, Department of Computer Science and Engineering
Examiner: Miquel Pericas, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

A Multi-Objective Scheduling
Framework for Energy–Performance
Optimization in Multi-GPU
Task Graphs

Long Cheng, Yicheng Li
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Multi-GPU task-graph runtimes expose scheduling information that is not available to external power controllers, including task dependencies, data residency, communication cost, device queues, and execution slack. This thesis uses that information to study energy–performance optimization inside CUDA Sequential Task Flow (CUDASTF). The proposed framework separates the problem into two runtime layers. The first layer is a HEFT-relative residual placement scheduler that keeps earliest-finish-time placement as a conservative baseline and accepts non-HEFT placements only when predicted data-movement savings justify a bounded finish-time penalty. A window-level contextual bandit adapts the aggressiveness of this residual gate. The second layer is a guarded proposal–commit DVFS controller that converts task-level frequency intents into stable per-GPU window-level frequency decisions.

The evaluation uses a single-node eight-GPU NVIDIA L4 platform and five CUDASTF benchmarks. With DVFS disabled, the placement layer reduces geometric-mean latency by 9.99% relative to HEFT. Data-movement evidence shows that the strongest placement improvements coincide with substantial copy-byte reductions, while the near-neutral workload remains close to the baseline. Under fixed Bandit placement, WinDVFS reduces geometric-mean energy by 5.32% with a 0.48% geometric-mean latency overhead and improves geometric-mean EDP by 4.87%. End to end, Bandit+WinDVFS reduces geometric-mean latency by 8.80%, energy by 11.87%, and EDP by 19.63% relative to HEFT. These results show that bounded, explainable runtime decisions can improve locality and energy efficiency without replacing HEFT with an unconstrained learned scheduler.

Keywords: multi-GPU scheduling, CUDASTF, task graphs, HEFT, DVFS, energy efficiency, EDP, data locality.

Acknowledgements

The authors would like to thank Jing Chen for supervision, technical guidance, and constructive feedback throughout this thesis project. The authors are also grateful to Miquel Pericas for serving as examiner and for providing academic oversight and feedback. The authors further thank the Department of Computer Science and Engineering at Chalmers University of Technology for providing the academic environment and infrastructure that supported this work.

Long Cheng, Yicheng Li, Gothenburg, 2026

Contents

List of Figures

List of Tables

1

Introduction

Energy efficiency has become a central constraint in high-performance computing. Modern scientific and data-intensive applications increasingly rely on multi-GPU nodes to obtain high throughput, while the corresponding power consumption, cooling cost, and carbon footprint make performance-oriented execution at maximum frequency difficult to justify in all cases [1]. Dynamic Voltage and Frequency Scaling (DVFS) provides a hardware mechanism for improving energy efficiency by adjusting GPU core and memory frequencies. On contemporary GPUs, this exposes a large energy–performance design space, where different frequency settings may lead to substantially different runtime, power, energy, and energy–delay product (EDP) outcomes [2], [3], [4], [5].

Using GPU DVFS effectively is non-trivial. The best frequency setting depends on the computational and memory-access characteristics of each workload. Compute-bound kernels are often sensitive to core frequency, whereas memory-bound or communication-heavy tasks may tolerate lower core frequencies with limited performance loss. Prior work has therefore studied GPU power modeling, core-memory frequency scaling, predictable frequency selection, and fine-grained DVFS prediction [6], [7], [8], [9], [10]. Other systems have explored phase-level or application-level energy tuning for heterogeneous and GPU-based workloads [5], [11], [12]. However, making frequency decisions independently for each kernel ignores the broader execution context. Frequency transitions may introduce non-negligible latency, and excessive switching can offset the energy savings obtained from lower-frequency execution. Isolated per-kernel decisions also do not reason about task dependencies, data movement, device queues, or critical-path slack.

For multi-GPU task graphs, the placement decision is itself part of the energy–performance problem. Prior multi-GPU and heterogeneous-runtime work has shown that task mapping, work sharing, and energy-aware decomposition can strongly affect both locality and energy behavior [13], [14], [15]. A scheduler that minimizes the predicted finish time of the next ready task may still create future data movement by moving tasks away from the GPUs that already hold their logical data. The extra peer copies increase latency directly and can also reduce the slack that a later DVFS controller would need in order to lower frequency safely. Thus, before frequency control can be evaluated cleanly, the runtime must first provide a placement policy that preserves the latency robustness of a strong baseline while avoiding unnecessary communication.

Task-graph runtimes provide a suitable control point for this problem. Earlier

task-based systems established the value of runtime-managed dependencies and heterogeneous scheduling, while CUDASTF provides a CUDA-oriented task-flow abstraction for modern GPU execution [16], [17]. In the CUDA Sequential Task Flow (CUDASTF) runtime, applications are represented as directed acyclic graphs (DAGs), where tasks encode computation or data movement and edges encode data-driven dependencies. This representation gives the runtime explicit information about dependencies, logical-data residency, synchronization, communication cost, and task placement across GPUs. Such information is not visible to an external DVFS controller or to a kernel-level frequency heuristic. A CUDASTF scheduler can therefore reason about both where a task should execute and how aggressively the selected GPU can trade frequency for energy savings.

This thesis investigates runtime-level scheduling for single-node multi-GPU CUDASTF task graphs. The design is deliberately layered. The first layer is a HEFT-relative residual placement scheduler. HEFT is a widely used list-scheduling heuristic for heterogeneous systems because it provides a low-complexity approximation of earliest-finish-time task placement [18]. The scheduler in this thesis keeps HEFT as a conservative baseline, then permits a non-HEFT placement only when the predicted copy saving is large enough and the normalized finish-time penalty remains bounded. A window-level contextual bandit adapts the aggressiveness of this residual gate according to recent task-graph and runtime context, following the general principle of learning actions from contextual feedback [19]. Importantly, the bandit does not directly choose a GPU for each task; it chooses the gate profile that controls how permissive HEFT-relative deviation should be.

The second layer is a guarded, windowed DVFS controller. It is applied after the placement decision. Instead of changing frequency directly for every task, each task produces a high-, mid-, or low-frequency intent. These intents are aggregated by a per-device, per-window committer, which selects the actual committed frequency while applying guardrails for critical tasks, heavy computation, backlog pressure, and recent slowdown. This proposal–commit structure aims to preserve the task-level information needed for energy-aware control while avoiding unstable per-task frequency switching.

This separation between placement and DVFS is a central design choice. Directly combining GPU placement and frequency selection into one online action space would increase decision complexity, enlarge the exploration problem, and make reward attribution difficult. By contrast, the two-level structure preserves HEFT as a performance anchor, makes placement decisions explainable through copy-saving residual certificates, and lets the DVFS controller make stable frequency decisions at a coarser granularity than individual kernel invocations. The resulting framework is designed to exploit the task-graph information exposed by CUDASTF while remaining lightweight enough for runtime use.

1.1 Problem Statement

The problem addressed in this thesis is how to design an online runtime scheduler for multi-GPU task graphs that improves locality and exposes energy-saving oppor-

tunities without substantially degrading performance. The scheduler must operate under four practical constraints. First, scheduling decisions must be lightweight enough to run inside the CUDASTF runtime for fine-grained tasks. Second, task placement must account for logical-data movement and queueing effects, rather than only the earliest finish time of the current task. Third, any learned policy must remain explainable and must have a safe fallback when the workload offers no useful locality opportunity. Fourth, DVFS control must avoid unstable per-task frequency switching and must protect critical, heavy, or backlogged execution phases from excessive slowdown.

This thesis focuses on the following research questions:

- How can a CUDASTF runtime scheduler preserve the performance advantages of HEFT-style earliest-finish-time placement while exploiting data locality through bounded deviations?
- Can observed latency improvements or regressions be attributed to concrete locality mechanisms such as copy-time reduction, copy-volume reduction, and non-HEFT deviation rate?
- How can GPU DVFS decisions be integrated into the task scheduler without introducing excessive frequency-transition overhead or unstable per-task control?
- To what extent can the resulting two-level runtime policy improve energy or energy-delay metrics while keeping runtime degradation within an acceptable bound?
- What runtime overheads are introduced by the residual gate, window-level adaptation, instrumentation, and DVFS guardrails, and are those overheads small enough for online scheduling?

The scope of this work is single-node multi-GPU execution. The focus is on GPU task placement and GPU core/memory frequency control inside CUDASTF. CPU DVFS, cluster-level scheduling, offline global DAG optimization, and application-specific hand placement are outside the scope of this thesis.

1.2 Approach

The proposed framework decomposes runtime control into two layers. The first layer performs learning-guided residual placement. The second layer performs guarded proposal–commit DVFS control. This separation is designed to retain the stability of a strong scheduling baseline while allowing energy-aware frequency adaptation at a coarser and more robust granularity than individual kernel invocations.

1.2.1 Learning-Guided Residual Placement

The placement layer uses HEFT-style earliest-finish-time scheduling as its performance baseline. For each arriving task, the scheduler estimates the data-ready time, copy time, queueing delay, start time, and finish time on each GPU. The GPU with the earliest estimated finish time is selected as the baseline placement.

The scheduler then searches for a residual alternative: a non-baseline GPU that can reduce data movement while introducing only a bounded finish-time penalty. Such a deviation is accepted only when the expected copy reduction is sufficiently large and the delay relative to the HEFT baseline remains within a configurable threshold. This design treats HEFT as a conservative performance anchor, rather than replacing it with an unconstrained learned policy.

To adapt this residual decision across workloads, the scheduler uses a window-level contextual bandit. Instead of selecting an action independently for every task, the bandit selects a gate profile for a window of tasks. The context summarizes recent runtime behavior, including task criticality, dependency structure, input size, baseline copy cost, task cost, device-load imbalance, and recent deviation rate. The active gate profile determines how aggressively the scheduler may deviate from the HEFT baseline. In the main configuration, the bandit chooses between a default profile and a more aggressive copy-saving profile. The reward balances copy reduction against the finish-time penalty caused by deviation.

1.2.2 Guarded Proposal–Commit DVFS Control

The DVFS layer is applied after placement. Rather than allowing every task to directly change GPU frequency, each task produces only a frequency intent: high, mid, or low. This intent is computed from lightweight runtime features such as task criticality, estimated task cost, backlog pressure, copy/wait ratio, available slack, and whether the task appears heavy or urgent.

Actual frequency changes are made by a per-device, per-window committer. The committer aggregates task-level intents over a window and chooses a committed frequency level for the corresponding GPU. This aggregation reduces unstable per-task switching and allows the controller to reason about the recent execution phase of each device. The committer also applies guardrails. Critical tasks, urgent heavy tasks, high backlog, high critical mass, and recent slowdown signals can force the device back to high frequency. Entering lower-frequency modes requires consecutive eligible windows, whereas leaving them is deliberately conservative and rapid when eligibility disappears.

This proposal–commit structure provides a practical compromise between fine-grained DVFS sensitivity and runtime stability. Task-level proposals retain information about individual tasks, while window-level commits amortize frequency decisions and reduce transition frequency. The evaluation therefore separates placement evidence from DVFS evidence: placement-only runs establish whether the residual scheduler improves latency and locality, while DVFS runs evaluate whether frequency commits can reduce energy or energy-delay metrics under the same placement framework.

1.3 Contributions

This thesis makes the following contributions:

- It presents a runtime-integrated scheduling framework for single-node multi-GPU CUDASTF task graphs, combining HEFT-relative residual placement

with guarded window-level DVFS control.

- It introduces a copy-aware residual placement policy built on top of HEFT. The policy preserves HEFT as a performance baseline while allowing bounded deviations when they reduce predicted data movement.
- It designs a window-level contextual bandit that adapts the aggressiveness of placement deviations using recent task-graph and runtime context, while keeping every GPU choice explainable relative to the HEFT baseline.
- It develops a guarded proposal–commit DVFS controller in which task-level frequency intents are aggregated into stable per-device window decisions with criticality, backlog, heavy-task, and slowdown guardrails.
- It evaluates the framework with a reproducible CUDASTF thesis benchmark suite, separating placement latency, locality attribution, runtime overhead, DVFS energy, and guardrail behavior.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 reviews GPU DVFS, DAG scheduling, energy-aware runtime systems, and CUDASTF. Chapter 3 describes the proposed two-level scheduling framework, including residual placement, contextual-bandit adaptation, proposal–commit DVFS control, and the corresponding CUDASTF runtime integration. Chapter 4 evaluates the framework experimentally, first isolating placement and locality effects and then analyzing DVFS energy and guardrail behavior. Chapter 5 discusses the implications and limitations of the results. Chapter 6 outlines future work, and Chapter 7 concludes the thesis.

2

Theory

Runtime-level energy optimization for multi-GPU task graphs requires a model of both scheduling and frequency control. The target execution model in this thesis is CUDA Sequential Task Flow (CUDASTF), where applications are represented as task graphs with data-driven dependencies. The central optimization goal is to reduce energy consumption and energy–delay product (EDP) while keeping performance loss within a user-defined bound. This chapter provides the theoretical background for that goal by reviewing multi-GPU task graph execution, CUDASTF, GPU dynamic voltage and frequency scaling (DVFS), energy–performance metrics, DAG scheduling, data-locality-aware placement, online bandit scheduling, and windowed proposal–commit DVFS control.

2.1 Multi-GPU Task Graph Execution

Task-graph execution is the foundation of this thesis. Instead of viewing a GPU application as a flat sequence of kernel launches, a task-based runtime represents the application as a graph of computation and data-movement tasks. Prior heterogeneous runtimes and multi-GPU task systems show that this abstraction can expose scheduling, load-balancing, and data-management opportunities that are difficult to express through flat kernel-launch interfaces [14], [15], [16]. This graph exposes dependencies, communication, device placement choices, and possible slack that can be exploited by a scheduler.

2.1.1 Directed Acyclic Graph Representation

A task graph is usually modeled as a directed acyclic graph (DAG):

$$G = (V, E), \tag{2.1}$$

where each vertex $v \in V$ represents a task and each directed edge $(u, v) \in E$ represents a dependency from task u to task v. In a GPU runtime, a task may correspond to a kernel execution, a data transfer, a synchronization operation, or a higher-level runtime operation. A task v can start only when all predecessor tasks in $\text{pred}(v)$ have completed and the required input data are available on the device selected for v.

The acyclic property is important because it gives the runtime a partial order over tasks. Tasks that are not connected by dependencies may execute concurrently,

while dependent tasks must respect the order encoded by the edges. This makes the DAG a natural abstraction for multi-GPU execution, where available parallelism and communication constraints must be considered together.

2.1.2 Device Readiness, Data Readiness, and Finish Time

In a multi-GPU system, scheduling a task requires choosing both a device and a start time. Let d denote a candidate GPU. The data-ready time of task v on device d can be expressed as

$$R_d(v) = \max_{u \in \text{pred}(v)} \left(F(u) + C_{u,v,d} \right), \tag{2.2}$$

where $F(u)$ is the finish time of predecessor u, and $C_{u,v,d}$ is the communication or data-migration cost needed before v can execute on device d. If the required data are already resident on device d, then $C_{u,v,d}$ may be close to zero.

Let A_d be the time at which device d becomes available according to the scheduler. The estimated start time and finish time of task v on device d are then

$$S_d(v) = \max \left(A_d, R_d(v) \right), \tag{2.3}$$

$$F_d(v) = S_d(v) + T_d(v), \tag{2.4}$$

where $T_d(v)$ is the predicted execution time of task v on device d. These equations are the basis for earliest-finish-time scheduling and for HEFT-style placement decisions.

2.1.3 Critical Path, Slack, and Parallelism

The critical path is the longest dependency chain in the DAG, including computation and communication costs. It determines the minimum possible makespan under ideal scheduling assumptions. Tasks on or near the critical path have little scheduling freedom because delaying them is likely to delay the whole application.

Slack is the opposite notion. A task has slack if it can be delayed, placed differently, or executed more slowly without extending the final makespan. Slack can arise from load imbalance, data transfers, synchronization, or non-critical branches of the DAG. In this thesis, slack is important for two reasons. First, non-critical tasks may tolerate energy-saving frequency choices. Second, communication and waiting periods may hide the latency of DVFS transitions.

2.1.3.1 Why Slack Matters for DVFS

DVFS reduces energy only when the saved power outweighs the additional runtime and transition overhead. If a task is on the critical path, lowering its frequency may directly increase the application runtime. If the same task has slack, the runtime increase may be partially or fully hidden. Prior work on energy-aware GPU and DAG scheduling similarly shows that frequency, power-cap, or resource decisions must be interpreted under workload structure, timing requirements, or data-movement behavior [20], [21], [22], [23], [24], [25], [26]. Therefore, a DAG-aware DVFS policy should not make frequency decisions from task cost alone; it should also consider dependency structure, data movement, and criticality.

2.2 CUDASTF Execution Model

CUDASTF is the runtime substrate considered in this thesis. It implements a CUDA-oriented Sequential Task Flow model in which applications are expressed through task submissions with data-access information. The runtime derives data-driven dependencies, manages data placement, and schedules work across GPUs [17].

2.2.1 Sequential Task Flow Model

The Sequential Task Flow model allows the programmer to submit tasks sequentially while the runtime extracts parallelism from data dependencies. The key idea is that the programmer specifies which data objects a task reads or writes. The runtime then determines when the task can execute safely. If two tasks access independent data, they may run concurrently. If one task writes data that another task later reads, the runtime inserts the required dependency.

This model is useful for GPU applications because explicit synchronization is often error-prone and overly conservative. A task-based runtime can avoid unnecessary synchronization while preserving correctness through data-dependency tracking.

2.2.2 Data-Driven Dependencies and Data Residency

In multi-GPU execution, dependencies are not only ordering constraints. They also determine where data are located and whether a task requires data migration. CUDASTF tracks data accesses and can therefore infer when a task needs a local copy, when data must be transferred between GPUs, and when multiple devices may share valid copies.

For the scheduler, this information is essential. A device that appears fast from a compute perspective may be a poor choice if the required data are located elsewhere. Conversely, a slightly slower or more loaded device may be attractive if it already holds the necessary input data. Figure 2.1 summarizes this task, data, and residency view of the runtime.

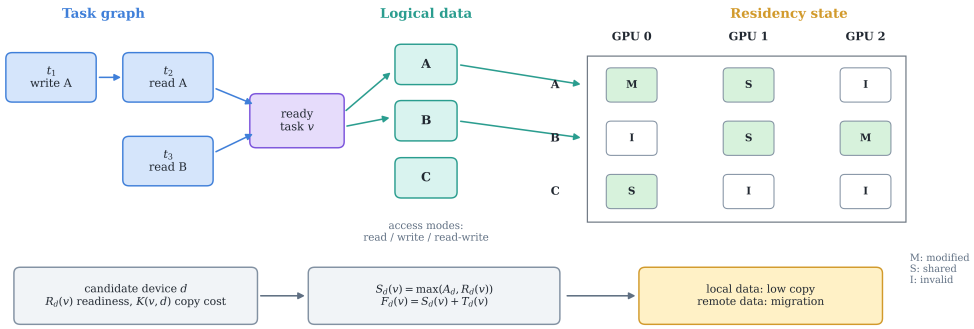

Figure 2.1: CUDASTF task-graph execution model. Tasks form a DAG with data-driven dependencies, while each GPU maintains a changing view of logical data residency. Candidate placement decisions differ in device readiness, data-ready time, copy cost, and predicted finish time.

2.2.3 Runtime Scheduling Opportunities

CUDASTF exposes several opportunities for energy-aware scheduling:

- task dependencies can be used to estimate criticality and slack;
- data residency information can be used to reduce unnecessary transfers;
- communication phases can be used to hide frequency-transition latency;
- repeated task types can be profiled and modeled at runtime;
- task placement and DVFS decisions can be integrated into one runtime-level control pipeline.

These properties make CUDASTF a suitable setting for combining DAG scheduling with GPU DVFS. The thesis does not treat DVFS as an external node-level controller. Instead, DVFS decisions are made with direct access to task-graph information.

2.3 GPU DVFS and Energy–Performance Trade-offs

Dynamic voltage and frequency scaling (DVFS) is a hardware power-management mechanism that changes operating frequencies, and when supported, the corresponding voltages. On GPUs, DVFS is commonly exposed through core and memory frequency controls. Prior GPU DVFS studies show that the best frequency setting depends strongly on workload characteristics, and that maximum frequency is not always energy-optimal [2], [3], [4], [5], [6], [7], [8].

2.3.1 GPU Core and Memory Frequency Domains

Two frequency domains are particularly relevant for this thesis. The GPU core frequency affects arithmetic throughput, instruction issue, and the execution rate of compute pipelines. The memory frequency affects memory bandwidth and, indirectly, the cost of moving data to and from global memory.

A frequency configuration can be written as

$$\phi = (f_g, f_m), \tag{2.5}$$

where f_g is the GPU core frequency and f_m is the memory frequency. A DVFS policy chooses ϕ for a task, a group of tasks, or a larger execution phase.

Although the hardware may expose many supported clock pairs, the runtime policy in this thesis uses a small discrete abstraction:

$$L = \{\text{HIGH}, \text{MID}, \text{LOW}\}. \tag{2.6}$$

The high level represents the fastest available graphics clock, while the mid and low levels are selected by rank among lower supported graphics clocks. The memory clock is treated conservatively and kept at the highest supported level in the thesis design. This discrete-level view keeps the online decision space small enough for a runtime scheduler while still exposing meaningful energy–performance choices.

2.3.2 Workload Sensitivity to Frequency Scaling

The effect of DVFS depends on how a task uses the GPU. The thesis uses the following conceptual classification.

2.3.2.1 Compute-Bound Tasks

A compute-bound task spends most of its time in arithmetic operations or instruction execution. Its runtime is sensitive to GPU core frequency. Lowering the core frequency may reduce power, but it is also likely to increase runtime. For compute-bound tasks on the critical path, aggressive down-scaling can therefore harm EDP.

2.3.2.2 Memory-Bound Tasks

A memory-bound task spends a large fraction of time waiting for memory accesses. Such a task may be less sensitive to core frequency because increasing arithmetic throughput does not remove the memory bottleneck. In these cases, lower core frequency may save energy with limited performance impact. However, memory frequency may still be important.

2.3.2.3 Mixed Tasks

Many GPU tasks are neither purely compute-bound nor purely memory-bound. Their sensitivity depends on input size, data layout, reuse, occupancy, and concurrent runtime activities. Mixed tasks motivate lightweight runtime models rather than fixed, hand-written rules for all kernels.

2.3.3 Energy of a Frequency Configuration

For a task v executed under frequency configuration ϕ, the task energy can be approximated as

$$E_v(\phi) = P_v(\phi) \cdot T_v(\phi), \tag{2.7}$$

where $P_v(\phi)$ is average power and $T_v(\phi)$ is execution time. The energy-optimal configuration is

$$\phi_v^* = \arg\min_{\phi \in \Phi} E_v(\phi), \tag{2.8}$$

where Φ is the set of supported frequency configurations.

However, minimizing per-task energy independently is not necessarily optimal at the application level. A task-level decision may increase the critical path, create queueing effects, or trigger excessive frequency transitions. This distinction is consistent with recent energy-aware scheduling systems that combine frequency, phase, or task information rather than treating each kernel as an isolated optimization target [11], [12], [27], [28]. Therefore, the thesis focuses on runtime-level decisions that consider both task characteristics and DAG structure.

2.3.4 DVFS Transition Latency

Changing GPU frequency is not free. A transition may introduce latency, temporarily stall command execution, or interact with stream scheduling. Fine-grained DVFS mechanisms can be effective when they predict workload behavior accurately, but they also highlight the need to control the cost and timing of frequency changes [10]. If the runtime changes frequency before every short task, the accumulated transition cost can dominate the potential savings. A more practical approach is to amortize transitions over groups of tasks and to place transitions where their latency is least visible, such as during data transfers or non-critical work.

2.4 Energy, EDP, and Bounded Slowdown Objectives

The scheduler in this thesis is multi-objective. It should reduce energy consumption while preserving acceptable performance. This section defines the metrics used to reason about this trade-off.

2.4.1 Runtime, Power, and Energy

The application runtime is the makespan of the task graph:

$$T_{\text{app}} = \max_{v \in V} F(v). \tag{2.9}$$

The total energy is the integral of power over time:

$$E_{\text{app}} = \int_0^{T_{\text{app}}} P(t)\, dt. \tag{2.10}$$

In an experimental system, this integral is usually approximated from sampled power readings:

$$E_{\text{app}} \approx \sum_i P_i \Delta t_i. \tag{2.11}$$

2.4.2 Energy–Delay Product

Energy–delay product combines energy and runtime:

$$\text{EDP} = E_{\text{app}} \cdot T_{\text{app}}. \tag{2.12}$$

EDP is useful when both energy and runtime matter. In this thesis, energy, latency, and EDP are reported separately so that the energy–performance trade-off remains explicit.

2.4.3 Bounded Slowdown

A bounded slowdown constraint compares the optimized runtime to a baseline runtime T_{base}:

$$T_{\text{app}} \leq (1 + \epsilon) T_{\text{base}}, \tag{2.13}$$

where ϵ is the maximum allowed slowdown. The baseline may be fixed maximum-frequency execution or another user-selected reference policy.

Metric	Definition
Runtime	T_{app}
Energy	E_{app}
EDP	$E_{\text{app}} T_{\text{app}}$
Slowdown	$T_{\text{app}} / T_{\text{base}} - 1$

Table 2.1: Energy–performance metrics used in this thesis.

2.5 Online Task Characterization for DVFS

A runtime scheduler cannot rely on expensive offline modeling for every task instance. Offline task-type profiles can be useful background information, but the DVFS path studied in this thesis is designed to operate from information already available inside the scheduler. The characterization problem is therefore not to predict exact power and runtime for every frequency pair. It is to separate tasks that should remain at high frequency from tasks that are plausible mid- or low-frequency candidates under the current DAG and device context.

2.5.1 Observable Runtime Features

For each scheduled task v, the runtime observes or estimates a small set of features:

- task cost $T(v)$;
- dependency fan-in and input-byte volume;
- copy time and queue-wait time under the selected placement context;
- device backlog on the chosen GPU;
- a local criticality proxy derived from cost, dependencies, and input size;
- task-symbol hints for heavy kernels such as dense linear algebra operations.

These features are cheap to compute because they are already needed for HEFT-style candidate evaluation or are available from CUDASTF task metadata.

2.5.2 Copy-Wait Ratio and Slack Proxy

The DVFS model uses copy and waiting time as a proxy for how much execution can hide slower core frequency. Let $W(v)$ denote the predicted copy-plus-queue waiting component and let $T(v)$ denote the predicted compute component. A normalized

copy-wait ratio is

$$\omega(v) = \frac{W(v)}{\max(\varepsilon, W(v) + T(v))}. \tag{2.14}$$

A large $\omega(v)$ indicates that the task is dominated by data movement or queue waiting, so a lower core frequency is less likely to be fully visible in the application makespan. The scheduler also uses a slack proxy

$$s(v) = 1 - \kappa(v), \tag{2.15}$$

where $\kappa(v)$ is the local criticality proxy. This is not an exact critical-path slack computation, but it gives the DVFS layer a conservative signal: high-criticality tasks are treated as having little slack, while low-criticality tasks may be considered for lower frequency when other guard conditions agree.

2.5.3 Runtime Suitability

The model must be inexpensive to evaluate and robust to imperfect estimates. Frequency-selection studies demonstrate that accurate GPU energy tuning can use profiling, analytical models, or learned predictors, but such information is costly to apply for every online task decision inside a fine-grained scheduler [3], [4], [8], [9]. For this reason, the thesis uses online proxies and guardrails rather than a large offline table of per-task frequency sensitivities. A task may request lower frequency only when multiple signals agree: it is not critical, not urgently heavy, not on a backlogged device, and has enough copy-wait time or slack to hide the expected slowdown. This conservative characterization supports the design goal of exposing an energy-control path without turning every task into a separate profiling problem.

2.6 HEFT-Based DAG Scheduling

Heterogeneous Earliest Finish Time (HEFT) is a widely used list-scheduling heuristic for heterogeneous systems [18]. The full HEFT algorithm ranks tasks and then places them on processors according to earliest finish time. This thesis uses HEFT primarily as a baseline placement principle: for each ready task, estimate its finish time on each GPU and select the device with the earliest finish time.

2.6.1 Earliest-Finish-Time Placement

Given the candidate finish time $F_d(v)$ from Equation (2.4), a HEFT-style baseline chooses

$$d_{\text{base}}(v) = \arg\min_{d \in D} F_d(v), \tag{2.16}$$

where D is the set of available GPUs. This decision balances device availability, data-readiness, communication cost, and predicted execution time.

2.6.2 Strengths of HEFT for Multi-GPU Runtime Scheduling

HEFT-style scheduling is attractive because it is simple, fast, and directly connected to makespan minimization. It naturally avoids placing tasks on devices that are busy or require excessive data movement if doing so would delay the finish time. For online runtime scheduling, this makes HEFT a strong baseline.

2.6.3 Limitations for Energy-Aware Scheduling

The main limitation is that HEFT is performance-oriented. It does not directly optimize energy, EDP, or DVFS transition overhead. It may also ignore placement alternatives that slightly delay a non-critical task but save a significant amount of data movement. This motivates a controlled-deviation strategy: keep HEFT as the safe baseline, but allow limited deviations when the expected communication benefit is high and the end-time penalty is small.

2.7 Data Locality, Communication Cost, and Criticality

Multi-GPU applications are often limited not only by computation but also by data movement. A scheduler that ignores data locality may repeatedly move data between GPUs, increasing runtime and energy. This section explains the concepts used to reason about locality-aware placement.

2.7.1 Data Residency and Copy Cost

Data residency describes which GPU currently holds a valid copy of a data object. If a task is placed on a GPU that already contains its required inputs, it can start after dependency completion and device availability. If the data are located elsewhere, the runtime must perform one or more transfers.

For a candidate device d, the total copy cost for task v can be approximated as

$$C_d(v) = \sum_{x \in \text{inputs}(v)} C_{x,d}, \tag{2.17}$$

where $C_{x,d}$ is the estimated cost of making input data object x available on device d.

2.7.2 Queue Delay and Data-Ready Delay

A placement candidate can be delayed for two different reasons. First, the selected device may already be busy. Second, the required data may not yet be available. These two delays can be separated as

$$Q_d(v) = \max\left(0, S_d(v) - R_d(v)\right), \tag{2.18}$$

$$W_d(v) = \max\left(0, R_d(v) - A_d\right), \tag{2.19}$$

where $Q_d(v)$ represents queue delay after data are ready, and $W_d(v)$ represents waiting for data relative to device availability. This distinction helps identify whether a task is delayed primarily by load imbalance or by data movement.

2.7.3 Criticality Proxy

Exact critical-path computation can be expensive or unavailable in online runtime scheduling. A lightweight proxy can estimate whether a task is likely to be important. The current scheduling design uses three normalized features: task cost, dependency count, and input size. Let

$$c_T(v), \qquad c_D(v), \qquad c_B(v) \in [0,1] \tag{2.20}$$

denote the normalized task-cost, dependency-count, and input-byte features for task v. The criticality proxy combines them with tunable non-negative weights:

$$\kappa(v) = \mathrm{clip}_{[0,1]}\left(\beta_C c_T(v) + \beta_d c_D(v) + \beta_B c_B(v)\right), \tag{2.21}$$

where

$$\beta_C, \beta_d, \beta_B \geq 0, \qquad \beta_C + \beta_d + \beta_B = 1. \tag{2.22}$$

The weights describe how much the proxy should trust each signal. A larger β_C makes long-running tasks more critical; a larger β_d emphasizes tasks with many dependencies, which often occur near synchronization points; and a larger β_B emphasizes data-heavy tasks, where poor placement can amplify communication cost. These weights are not part of the theoretical definition itself. They are implementation hyperparameters, and the evaluation defaults are reported in the Methods chapter.

2.8 Controlled Deviation from HEFT Baselines

This thesis does not discard the HEFT baseline. Instead, it uses HEFT as a conservative placement decision and then considers whether a nearby alternative is worthwhile. The goal is to reduce communication and improve locality without significantly extending the task graph's critical path.

2.8.1 Baseline and Alternative Candidates

For each task v, the scheduler first computes the baseline device d_{base} using Equation (2.16). It then evaluates alternative devices $d \neq d_{\text{base}}$. An alternative is interesting when it saves copy cost or queue delay while keeping the predicted finish time close to the baseline.

2.8.2 Copy Gain and End-Time Penalty

Let $F_{\text{base}}(v)$ and $C_{\text{base}}(v)$ be the baseline finish time and copy cost. For an alternative device d, define the relative end-time penalty as

$$\Delta_F(d,v) = \frac{F_d(v) - F_{\text{base}}(v)}{\max(\varepsilon, F_{\text{base}}(v))}, \tag{2.23}$$

where ε avoids division by zero. The copy gain is

$$G_C(d,v) = \frac{C_{\text{base}}(v) - C_d(v)}{\max(C_{\min}, C_{\text{base}}(v))}, \tag{2.24}$$

where $C_{\min}$ is a small denominator floor.

2.8.3 Gate-Based Acceptance

An alternative is accepted only if it satisfies a gate:

$$\Delta_F(d,v) \leq \tau_F, \tag{2.25}$$
$$G_C(d,v) \geq \tau_C, \tag{2.26}$$

where τ_F is the allowed end-time penalty and τ_C is the required copy-gain threshold. Critical tasks should use stricter thresholds than non-critical tasks. Figure 2.2 visualizes the resulting acceptance region.

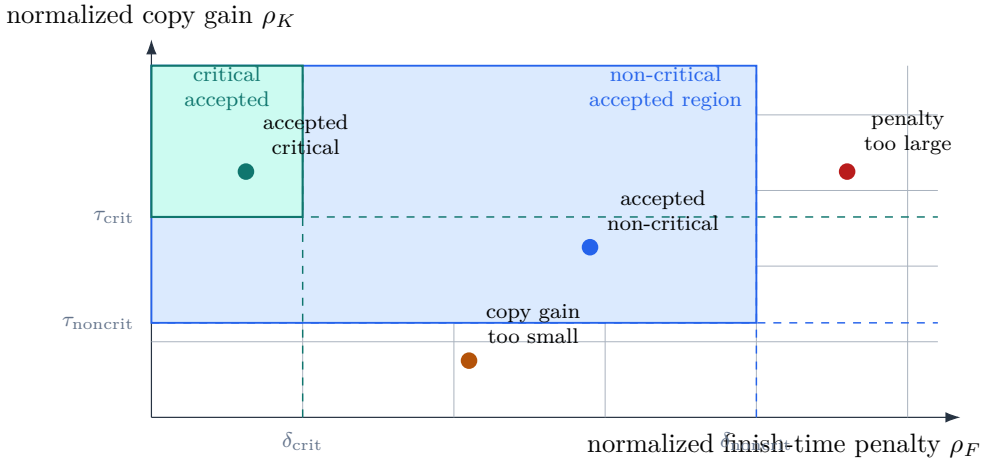

Figure 2.2: Acceptance region of the HEFT-relative residual gate. A non-HEFT candidate is accepted only when its normalized finish-time penalty remains below the active threshold and its normalized copy gain exceeds the required evidence threshold. Critical tasks use stricter gates than non-critical tasks.

2.8.3.1 Critical and Non-Critical Gates

The scheduler can use two sets of thresholds. For critical tasks, the allowed end-time penalty should be small and the copy-gain requirement should be high. For non-critical tasks, the scheduler can be more aggressive because small local delays may be hidden by slack.

2.8.3.2 Gate Profiles

A gate profile is a parameterized policy that controls how willing the scheduler is to deviate from HEFT. Table 2.2 gives a conceptual example.

This profile abstraction is useful because an online scheduler can adapt the profile according to recent workload behavior instead of relying on one fixed policy.

Profile	Behavior	End-time penalty	Copy-gain threshold
Closed	Always follow HEFT	Not applicable	Not applicable
Conservative	Rarely deviate	Very small	High
Default	Balanced deviation	Small	Moderate
Aggressive	More locality-driven	Larger	Lower

Table 2.2: Conceptual gate profiles for controlled deviation from HEFT.

2.9 Contextual Bandits for Online Runtime Scheduling

Runtime scheduling is online: the scheduler must make decisions with incomplete information and adapt as execution progresses. Contextual bandits provide a lightweight framework for choosing among several policies while learning from observed rewards. In this thesis, the bandit does not select a GPU directly. Instead, it selects a gate profile that controls how aggressively the scheduler may deviate from the HEFT baseline.

2.9.1 Exploration and Exploitation

A scheduler faces an exploration–exploitation trade-off. Exploitation means using the currently best-known profile. Exploration means trying another profile to learn whether it performs better under the current workload. Pure exploitation may get stuck in a suboptimal policy, while excessive exploration may harm performance.

2.9.2 Window-Level Context

Instead of choosing a new bandit action for every task, the scheduler can operate at window granularity. A window contains a fixed number of tasks. At the beginning of a window, the scheduler observes a context vector x summarizing recent execution, such as average criticality, average copy cost, average task cost, device load imbalance, and recent deviation rate. It then selects a gate profile for all tasks in that window.

This window-level design reduces noise and overhead. It also makes reward attribution easier because the reward is aggregated over many task decisions.

2.9.3 LinUCB Selection

LinUCB is a contextual bandit algorithm that assumes the expected reward of an action is approximately linear in the context [19]. For each arm a, the algorithm maintains a parameter estimate θ_a. Given context x, it computes

$$\text{score}_a = \theta_a^T x + \alpha\sqrt{x^T A_a^{-1} x}, \tag{2.27}$$

where the first term is the predicted reward, the second term is an uncertainty bonus, A_a^{-1} is the inverse covariance matrix for arm a, and α controls exploration. The scheduler selects the arm with the highest score.

2.9.4 Reward for Placement Profiles

A useful reward should encourage copy savings while penalizing risky delays. For a deviated task, a simple reward can be written as

$$r(v) = G_C(v) - \lambda_F \Delta_F(v), \tag{2.28}$$

where $G_C(v)$ is copy gain, $\Delta_F(v)$ is end-time penalty, and λ_F is larger for critical tasks than for non-critical tasks.

At the end of a window w, the reward can be averaged and penalized by the deviation rate:

$$R_w = \frac{1}{|w|} \sum_{v \in w} r(v) - \eta \rho_w, \tag{2.29}$$

where ρ_w is the fraction of tasks that deviated from HEFT and η is a regularization coefficient. This avoids rewarding a policy that deviates too often without sufficient benefit.

2.9.4.1 Relative Reward

A relative reward compares the selected profile against a default profile on the same window or task. This reduces the effect of workload difficulty. For example, if no profile has good deviation opportunities in a window, the reward should not incorrectly punish the selected profile. Relative reward therefore helps the bandit learn whether a profile is better than the default under the same scheduling conditions.

2.10 Region-Level DVFS and Transition Amortization

DVFS can be applied at different granularities. Very fine-grained control may choose a frequency for every kernel, while coarse-grained control may choose one frequency for a whole application phase. This thesis uses the task graph to define an intermediate granularity: frequency regions or device-level commit windows.

2.10.1 DVFS Granularity

Table 2.3 summarizes common DVFS granularities.

Granularity	Advantage	Limitation
Per-kernel / per-task	Fine control	Many transitions, high overhead
Region / window	Amortizes transitions	Requires grouping and aggregation
Application phase	Low overhead	May miss fine-grained variation
Whole application	Simple and stable	Too coarse for heterogeneous DAGs

Table 2.3: DVFS granularity levels and their trade-offs.

Per-task DVFS can approximate the best frequency for each task but may trigger too many transitions. Phase-level DVFS is stable but may ignore short-term changes in task type, data movement, and GPU load. Region-level DVFS is a compromise: it

allows the runtime to react to task-graph structure while limiting transition overhead. This compromise is related to phase- and region-oriented energy-control approaches for heterogeneous systems, but the runtime window in this thesis is defined by CUDASTF scheduling events rather than by an external phase detector [11], [12].

2.10.2 Frequency Regions in Task Graphs

A frequency region is a group of tasks that share a frequency decision. Regions may be constructed along dependency chains, within a device window, or around repeated task types. The goal is to place tasks with similar frequency preferences into the same region and to avoid unnecessary frequency switches.

For a region R, a frequency decision can be written as

$$\phi_R^* = \arg\min_{\phi \in \Phi} J_R(\phi), \tag{2.30}$$

where J_R is an objective such as energy, EDP, or energy under a bounded slowdown constraint. In practice, the runtime may not solve this optimization exactly; it can use heuristic rules or learned policies that approximate the same goal.

2.10.3 Overlapping Transitions with Communication and Slack

Let L_{tr} be the latency of a frequency transition. If the transition is placed on the critical path, it may increase the makespan by nearly L_{tr}. If it is placed during a data transfer or a period of slack, the visible delay can be smaller:

$$L_{\text{visible}} = \max(0, L_{\text{tr}} - L_{\text{hidden}}), \tag{2.31}$$

where L_{hidden} is the portion overlapped with communication, synchronization, or non-critical work.

This is one of the main reasons to integrate DVFS with task-graph scheduling. A runtime that sees dependencies and data movement can choose transition points more intelligently than an external controller that only observes coarse application phases. Figure 2.3 illustrates this delayed application and transition-hiding argument.

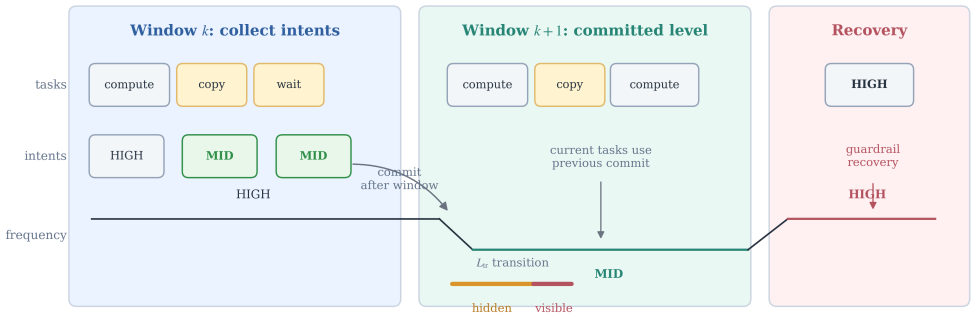

Figure 2.3: Windowed proposal–commit DVFS and transition amortization. Task-level intents are accumulated over a per-GPU window, while the committed frequency level is applied to later tasks. Transition latency can be partially hidden by communication or slack.

2.11 Windowed Proposal–Commit DVFS Control

The DVFS design in this thesis separates task-level frequency intent from device-level frequency commitment. This avoids making the placement bandit responsible for a very large joint action space of devices and frequencies. Instead, placement is the first layer, and DVFS is the second layer, with committed frequency selected at a per-device window granularity rather than independently for every task.

2.11.1 Two-Layer Runtime Control

The runtime control problem is separated into two decisions:

1. **Placement layer**: choose the GPU for each task, using a HEFT baseline and controlled deviation.
2. **DVFS layer**: choose the frequency level for the selected GPU, using proposal–commit control.

This separation simplifies learning and attribution. The placement layer reasons about data locality, queue delay, and finish time. The DVFS layer reasons about criticality, slack, backlog, and frequency risk after the device has been selected. It also makes the decisions easier to audit: placement decisions can be explained relative to HEFT, while frequency decisions can be explained through proposal shares and commit guardrails.

2.11.2 Task-Level Proposer

The proposer assigns each scheduled task a frequency intent, such as high, mid, or low. The intent is not immediately applied to that same task. It is a vote that expresses what the task appears to need under the current context and contributes to the next device-level commit decision. The current task receives the frequency level that was already committed by the previous window for the selected GPU.

A proposer can use the following features:

- criticality proxy $\kappa(v)$;
- estimated task cost $T(v)$;
- backlog pressure on the selected device;
- copy/wait ratio;
- available slack;
- whether the task is heavy or latency-sensitive.

A conceptual scoring function is

$$s(v) = w_\kappa \kappa(v) + w_T c_T(v) + w_B b(v) - w_W \omega(v) - w_S s_{\text{slack}}(v) + w_H h(v), \tag{2.32}$$

where $b(v)$ is backlog pressure, $\omega(v)$ is the copy-wait ratio from Equation (2.14), $s_{\text{slack}}(v)$ is normalized slack, and $h(v)$ indicates a heavy task. Positive terms push the decision toward high frequency, while negative terms indicate that lower frequency may be safe.

2.11.2.1 High-Frequency Conditions

A task should request high frequency if it is critical, heavy with little slack, placed on a backlogged device, or expected to be highly sensitive to frequency reduction. High frequency is also the safe fallback when the runtime lacks enough information.

2.11.2.2 Mid- and Low-Frequency Conditions

A task may request mid frequency when it has sufficient copy/wait time, moderate or low criticality, and no urgent heavy-computation requirement. Low frequency should be more restrictive. It is appropriate only for light, non-critical tasks with substantial slack and low backlog pressure.

2.11.3 Device-Level Window Committer

The committer aggregates task proposals over a device-level window. Only the committer changes the actual committed frequency level, and the new level affects later tasks on the same GPU. This delayed-commit design prevents individual tasks from causing frequent frequency oscillations and amortizes transition overhead across a region of execution.

For a device window W_d, weighted proposal shares can be written as

$$P_{\text{high}} = \frac{\sum_{v \in W_d} \omega_v \mathbb{I}[p_v = \text{high}]}{\sum_{v \in W_d} \omega_v}, \tag{2.33}$$

$$P_{\text{mid}} = \frac{\sum_{v \in W_d} \omega_v \mathbb{I}[p_v = \text{mid}]}{\sum_{v \in W_d} \omega_v}, \tag{2.34}$$

$$P_{\text{low}} = \frac{\sum_{v \in W_d} \omega_v \mathbb{I}[p_v = \text{low}]}{\sum_{v \in W_d} \omega_v}, \tag{2.35}$$

where p_v is the proposal of task v and ω_v is a proposal weight. Critical and expensive tasks can receive larger weights.

Guardrail	Trigger	Effect
Critical-work protection	High criticality mass or urgent heavy tasks in the window	Force the next committed level to HIGH
Backlog protection	High backlog pressure on the selected GPU	Prevent lower-frequency commits while the device is overloaded
Frequency-separation check	Mid or low graphics clocks are too close to the high clock	Treat the lower level as unavailable to avoid ineffective transitions
Eligibility persistence	Mid or low eligibility appears only transiently	Require consecutive eligible windows before entering a lower level
Slowdown recovery	Recent window runtime or busy-tail time exceeds moving-average guard bands	Force recovery windows at HIGH
Low-level cooldown	A slowdown is observed after a low-frequency commit	Temporarily disable LOW while keeping MID available if safe

Table 2.4: Guardrails used by the windowed proposal–commit DVFS controller.

2.11.4 Commit Eligibility and Slack Budget

The committer should enter a lower frequency only when the estimated performance risk fits within the available slack. A simple estimate of extra runtime when moving from high frequency to target frequency ϕ is

$$\Delta T_v(\phi) = \alpha_v T_v \left(1 - \omega(v)\right) \left(\frac{f_{g,\text{high}}}{f_{g,\phi}} - 1 \right), \tag{2.36}$$

where $\omega(v)$ is the copy-wait ratio and α_v increases for critical or heavy tasks. The aggregate extra runtime should satisfy a slack-budget condition:

$$\sum_{v \in W_d} \Delta T_v(\phi) \leq \beta S_{W_d} + B, \tag{2.37}$$

where S_{W_d} is the estimated slack reserve, β controls how much slack can be consumed, and B is a small performance budget.

2.11.5 Guardrails and Stability

A proposal–commit system needs guardrails to avoid performance regressions. Table 2.4 summarizes the guardrails used by the windowed controller.

The purpose of these guardrails is not to maximize energy savings for every local window. Instead, they protect the global runtime objective and make DVFS behavior stable. Recent slowdown is detected from window-level runtime and busy-tail measurements maintained as moving averages. If enough recent windows exceed the allowed slowdown ratios, the committer forces recovery to high frequency. This turns the lower-frequency levels into optimistic choices that must continue to earn their place.

Algorithm 1 Runtime scheduling with windowed proposal–commit DVFS

Require: Ready task v, GPU set D, placement state $\mathcal{S}$, DVFS windows $\{W_d\}_{d \in D}$
Ensure: Selected GPU d_{exec} and frequency annotation for task v

for all $d \in D$ **do**
  $\xi(v, d) \leftarrow \textsc{EvaluateCandidate}(v, d, \mathcal{S})$
end for
$d_{\text{base}} \leftarrow \arg\min_{d \in D} F_d(v)$
$d_{\text{exec}} \leftarrow \textsc{ResidualGate}(v, d_{\text{base}}, \{\xi(v, d)\}_{d \in D})$
$\kappa(v) \leftarrow \textsc{CriticalityProxy}(v)$
$p_v \leftarrow \textsc{ProposeLevel}(v, d_{\text{exec}}, \kappa(v), \xi(v, d_{\text{base}}))$
AccumulateIntent$(W_{d_{\text{exec}}}, p_v, v)$
$\ell_v \leftarrow \textsc{CommittedLevel}(W_{d_{\text{exec}}})$
$\varphi_v \leftarrow \textsc{FrequencyPair}(d_{\text{exec}}, \ell_v)$
$\textsc{AnnotateFrequency}(v, \varphi_v)$
$\mathcal{S} \leftarrow \textsc{UpdatePlacementState}(\mathcal{S}, v, d_{\text{exec}})$
if $\textsc{WindowComplete}(W_{d_{\text{exec}}})$ **then**
  $m \leftarrow \textsc{AggregateWindowMetrics}(W_{d_{\text{exec}}})$
  $\ell_{\text{next}} \leftarrow \textsc{CommitLevelWithGuardrails}(m)$
  ResetWindow$(W_{d_{\text{exec}}}, \ell_{\text{next}})$
end if
return $(d_{\text{exec}}, \ell_v)$

2.11.6 Algorithm

Algorithm 1 formalizes one scheduling step with windowed proposal–commit DVFS.

2.12 Multi-Objective Runtime Scheduling Framework

The preceding sections describe the components required by the thesis framework. This section summarizes how they fit together.

2.12.1 Joint Placement and Frequency Control

The runtime simultaneously considers device placement and frequency control, but it does not merge them into one large monolithic decision. Prior multi-GPU systems have explored compiler-assisted scheduling, dynamic work sharing, and energy-aware decomposition for heterogeneous devices [13], [14], [15]. The design in this thesis follows a more conservative separation: placement determines where the task should execute, using HEFT as a baseline and allowing controlled locality-driven deviations. DVFS then determines how aggressively the selected GPU can reduce frequency without violating criticality, backlog, and slack constraints.

2.12.2 Optimization View

At a high level, the runtime can be viewed as approximating the following constrained optimization objective:

$$\min_{\pi,\Phi} \quad J(E_{\text{app}}, T_{\text{app}}) \tag{2.38}$$

$$\text{s.t.} \quad T_{\text{app}} \leq (1+\epsilon)T_{\text{base}}, \tag{2.39}$$

$$\pi(v) \in D, \quad \forall v \in V, \tag{2.40}$$

$$\phi(v) \in \Phi, \quad \forall v \in V, \tag{2.41}$$

$$\text{all DAG dependencies are respected}, \tag{2.42}$$

where $\pi(v)$ is the placement decision, $\phi(v)$ is the frequency decision, and J may be energy, EDP, or another user-selected objective. This formulation is a conceptual target rather than a claim that the online scheduler solves the constrained problem exactly.

Solving this problem exactly online is impractical. The thesis therefore uses a layered heuristic framework: HEFT-style baseline placement, controlled deviation for data locality, contextual-bandit adaptation of gate profiles, and proposal–commit DVFS for stable frequency control.

2.12.3 Design Implications

The theory motivates four design choices:

1. **Use the task graph.** Dependencies, data movement, and slack are essential for safe energy optimization.
2. **Keep HEFT as a baseline.** A strong performance baseline reduces scheduling risk.
3. **Adapt deviation online.** Workloads differ, so the scheduler should learn how aggressive it can be.
4. **Commit DVFS by window.** Aggregating frequency intent avoids excessive transitions and unstable oscillations.

2.13 Summary

The theoretical foundation of the proposed framework is that a CUDASTF application can be viewed as a DAG in which tasks, dependencies, data residency, and communication costs are visible to the runtime. GPU DVFS introduces an energy–performance trade-off that depends on criticality, copy-wait behavior, backlog, frequency levels, and transition overhead. HEFT-style scheduling provides a strong baseline for performance, but it does not directly optimize energy or data-locality-driven trade-offs. Controlled deviation from HEFT, contextual bandit adaptation, region-level DVFS, and windowed proposal–commit frequency control therefore form the basis for the framework developed in this thesis.

3

Methods

The proposed runtime scheduler targets single-node multi-GPU CUDASTF task graphs and is organized as two cooperating layers. The first layer is a placement scheduler. It uses HEFT as a deterministic earliest-finish-time baseline and then applies a residual locality gate that may redirect a task to a non-HEFT GPU only when the predicted data-movement saving is sufficiently large and the normalized finish-time penalty is bounded. The second layer is an optional DVFS extension. It does not change the placement decision directly; instead, it assigns frequency intents to scheduled tasks, aggregates those intents over per-GPU windows, and commits stable GPU frequency levels with guardrails against slowdown.

The placement policy studied in this work is a two-profile, copy-only, HEFT-relative residual gate with relative reward and an aggressive-profile bias. The term "bandit" should therefore be interpreted carefully: the bandit layer does not select a GPU for each task. GPU selection remains a HEFT-relative deterministic decision. The bandit layer selects the gate profile used over a scheduling window, thereby controlling how permissive the residual deviation from HEFT should be. Figure 3.1 summarizes the complete runtime pipeline.

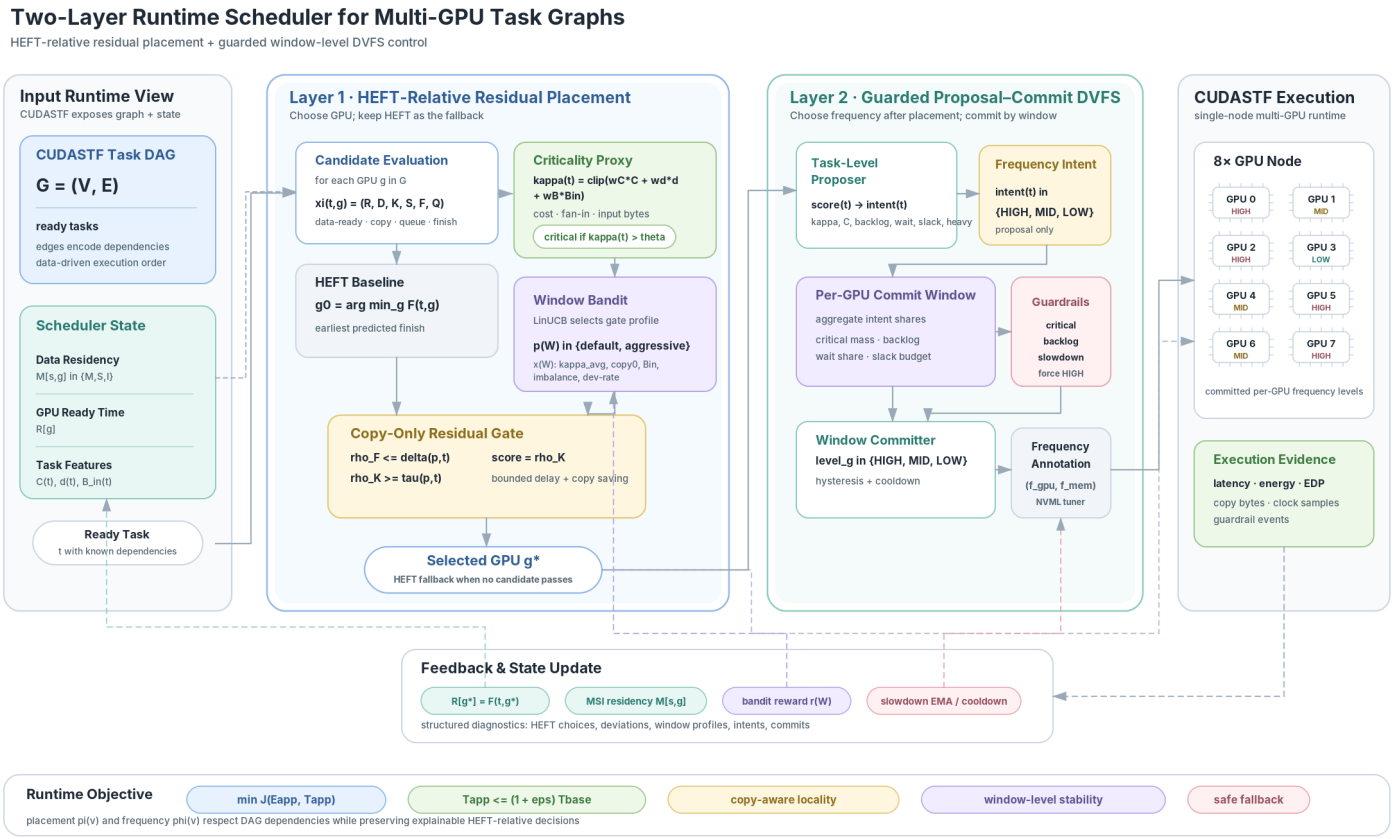

Figure 3.1: Overview of the proposed two-layer runtime scheduler. CUDASTF exposes a task graph and runtime state to the placement layer, which applies a HEFT-relative residual gate. The selected GPU is then passed to the guarded proposal–commit DVFS layer, and execution feedback updates ready times, data residency, bandit rewards, and slowdown guardrails.

3.1 Runtime Model and Scheduler State

Let

$$\mathcal{G} = \{0, 1, \ldots, m-1\} \tag{3.1}$$

denote the set of GPUs available to the runtime. A task t becomes schedulable when all of its CUDASTF dependencies are known to the scheduler. For each task, the scheduler observes a set of dependency descriptors, their access modes, logical data sizes, and an estimated execution cost. The method uses calibrated task statistics when available and falls back to a default task cost otherwise. The estimated cost is denoted by

$$C(t). \tag{3.2}$$

The placement layer maintains two central runtime states. The first is the predicted ready time of each GPU,

$$R[g], \qquad g \in \mathcal{G}, \tag{3.3}$$

which is the scheduler's estimate of when the task queue already assigned to GPU g will finish. The second is a logical-data residency model. For each logical data object s and GPU g, the scheduler tracks a state

$$M[s, g] \in \{\mathrm{M}, \mathrm{S}, \mathrm{I}\}, \tag{3.4}$$

where M means that GPU g holds the modified copy, S means that GPU g holds a valid shared copy, and I means that the copy on GPU g is invalid or absent. This model is lightweight: it is not a full cache-coherence protocol, but it provides enough information for predicting whether a task placement will require a peer copy or can reuse local data.

For a dependency s and candidate GPU g, the residency model exposes a timing query

$$\mathrm{whenAvailable}(s, g) = \left(A(s, g), X(s, g)\right), \tag{3.5}$$

where $A(s, g)$ is the predicted time at which s is available on g, and $X(s, g)$ is the predicted transfer time needed to make s available on g. If g already has a valid copy, then $X(s, g) = 0$. Otherwise, $X(s, g)$ is estimated from the data size and the effective copy bandwidth used by the runtime.

Symbol or state	Meaning
$R[g]$	Predicted ready time of GPU g.
$M[s, g]$	MSI-style residency state of logical data object s on GPU g.
$C(t)$	Estimated execution time of task t.
$B_{\text{in}}(t)$	Total input bytes read by task t.
$d(t)$	Number of dependency descriptors attached to task t across all access modes.
$q(t)$	Local criticality proxy used by placement and DVFS.
p_W	Gate profile selected for the current placement window.
ℓ_g	Committed DVFS level currently assigned to GPU g.

Table 3.1: Main runtime state used by the placement and DVFS layers.

3.2 HEFT Baseline Candidate Evaluation

For every ready task t, the scheduler evaluates all candidate GPUs. Let $\mathcal{I}(t)$ be the set of input dependencies of t, namely dependencies accessed in read or read-write mode. Write-only dependencies are not counted as input data for start-time prediction. For a candidate GPU g, the predicted input data-ready time is

$$D(t, g) = \max\left(\{0\} \cup \{A(s, g) \mid s \in \mathcal{I}(t)\}\right). \tag{3.6}$$

The predicted copy time is the sum of transfer times over the input dependencies:

$$K(t, g) = \sum_{s \in \mathcal{I}(t)} X(s, g). \tag{3.7}$$

Given the device ready time $R[g]$, the task start time and finish time on GPU g are

$$S(t, g) = \max\left(R[g], D(t, g)\right), \tag{3.8}$$

$$F(t, g) = S(t, g) + C(t). \tag{3.9}$$

The queue-delay component is

$$Q(t, g) = \max\left(0, S(t, g) - D(t, g)\right). \tag{3.10}$$

Thus each candidate GPU is represented by

$$\xi(t, g) = \left(g, R[g], D(t, g), K(t, g), S(t, g), F(t, g), Q(t, g)\right). \tag{3.11}$$

The HEFT baseline is the candidate with minimum predicted finish time:

$$g_0(t) = \arg\min_{g \in \mathcal{G}} F(t, g). \tag{3.12}$$

Ties are resolved deterministically; when two candidates have the same finish time, the lower GPU identifier is selected. The corresponding baseline candidate is denoted by

$$\xi_0(t) = \xi(t, g_0(t)). \tag{3.13}$$

HEFT is therefore the latency-oriented reference policy under the current residency state. All residual decisions are expressed relative to this reference rather than as absolute placement scores.

Parameter	Role	Evaluation default
C_{ref}	Cost normalizer	500 scheduler-cost units
K_{ref}	Copy-time normalizer for bandit context	100 scheduler-cost units
d_{ref}	Dependency-count normalizer	16
B_{ref}	Input-byte normalizer	2^{30} bytes
β_C	Task-cost weight	0.55
β_d	Dependency-count weight	0.30
β_B	Input-byte weight	0.15
θ_q	Criticality split for residual gates	0.60

Table 3.2: Criticality, context-normalization, and residual-gate hyperparameters used in the evaluation.

3.3 Local Criticality Proxy

The scheduler does not require an offline critical-path computation. Instead, it uses a tunable local criticality proxy derived from three observable task features: estimated compute cost, dependency fan-in, and input-data volume. Let $B_{\text{in}}(t)$ be the total number of input bytes read by task t, and let $d(t)$ be the total number of dependency descriptors attached to t across all access modes. These features are normalized as

$$\widetilde{C}(t) = \frac{\log(1 + C(t))}{\log(1 + C_{\text{ref}})}, \qquad \widetilde{d}(t) = \min\left(\frac{d(t)}{d_{\text{ref}}}, 1\right), \qquad \widetilde{B}(t) = \frac{\log(1 + B_{\text{in}}(t))}{\log(1 + B_{\text{ref}})}. \tag{3.14}$$

Here C_{ref}, d_{ref}, and B_{ref} are normalization constants. Their evaluation defaults are reported in Table 3.2. The criticality proxy is then defined as

$$q(t) = \text{clip}_{[0,1]}\left(\beta_C \widetilde{C}(t) + \beta_d \widetilde{d}(t) + \beta_B \widetilde{B}(t)\right), \tag{3.15}$$

where β_C, β_d, and β_B are tunable non-negative weights. They control the relative contribution of compute cost, dependency fan-in, and input-data volume, respectively, and satisfy $\beta_C + \beta_d + \beta_B = 1$ in the default configuration. A larger β_C makes the scheduler treat long-running kernels as more critical; a larger β_d emphasizes tasks with many input dependencies, which are more likely to sit at synchronization points; and a larger β_B emphasizes tasks with large input data volumes, where poor placement can amplify copy cost. A larger value of $q(t)$ therefore indicates a task that is locally more likely to affect the critical execution path. The placement gate and the DVFS proposer both use this proxy to become more conservative for high-criticality tasks.

3.4 Copy-Only HEFT-Relative Residual Gate

The residual gate is the core placement mechanism. It asks whether any non-baseline GPU can reduce predicted copy time enough to justify a small local delay relative to HEFT. Figure 3.2 illustrates the motivation. A locally optimal HEFT decision may place the current task on an idle GPU because that device gives the earliest predicted finish time, but the produced data may then be needed by successor tasks on the original GPU. In that case, the local finish-time improvement can create downstream copy traffic. The residual gate therefore treats a non-HEFT placement as acceptable

only when the locality evidence is large enough and the predicted finish-time penalty remains bounded.

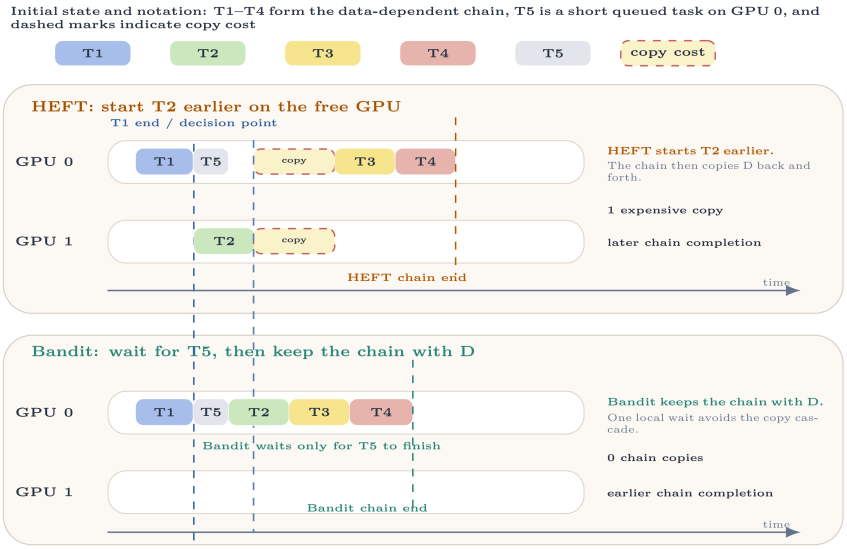

Figure 3.2: Motivating example for HEFT-relative residual placement. A HEFT-local earliest-finish decision can introduce downstream copies, while a bounded residual deviation can keep successor data local.

Let c be a non-baseline candidate for task t, and let c_0 be the HEFT baseline candidate. The normalized finish-time penalty is

$$\rho_F(t,c) = \frac{F(t,c) - F(t,c_0)}{\max(\epsilon, F(t,c_0))}, \tag{3.16}$$

where ϵ is a small positive constant. The normalized copy gain is

$$\rho_K(t,c) = \frac{K(t,c_0) - K(t,c)}{\max(K_{\min}, K(t,c_0))}, \tag{3.17}$$

where $K_{\min}$ prevents unstable ratios when the baseline copy time is nearly zero. The scheduler also computes a normalized queue-delay gain,

$$\rho_Q(t,c) = \frac{Q(t,c_0) - Q(t,c)}{\max(Q_{\min}, Q(t,c_0))}, \tag{3.18}$$

but the active bandit profiles set the queue-gain weight to zero. Therefore, for the method evaluated in this thesis, the effective residual score is determined by copy gain, while queue delay remains part of the HEFT finish-time estimate and of the DVFS wait proxy.

The gate has two profiles,

$$p \in \{\text{default}, \text{aggressive}\}. \tag{3.19}$$

Each profile defines a maximum normalized finish-time penalty and a minimum normalized copy gain. The gate uses separate thresholds for critical and non-critical

Profile	δ_{crit}	δ_{noncrit}	τ_{crit}	τ_{noncrit}
default	0.003	0.015	0.25	0.15
aggressive	0.004	0.020	0.20	0.10

Table 3.3: Default residual-gate profiles used in the evaluation.

tasks. Table 3.3 shows the default parameterization used in the evaluation. In the default parameterization, δ denotes the allowed value of ρ_F, and τ denotes the required value of ρ_K. The aggressive profile is more permissive because it allows a slightly larger finish-time penalty and requires a smaller copy gain.

The numeric thresholds in Table 3.3 are fixed method parameters, not per-benchmark adjustment points. They encode the intended asymmetry of the policy: latency-sensitive tasks receive a tighter finish-time bound and require stronger copy evidence, while non-critical tasks may accept slightly more predicted delay in exchange for locality. The same parameterization is used across the thesis evaluation, with concrete runtime configuration treated as reproducibility metadata.

Let θ_q be the criticality threshold used by the gate. For a selected profile p, the active thresholds for task t are

$$\delta(p,t) = \begin{cases} \delta_{\text{crit}}(p), & q(t) > \theta_q, \\ \delta_{\text{noncrit}}(p), & q(t) \le \theta_q. \end{cases} \tag{3.20}$$

$$\tau(p,t) = \begin{cases} \tau_{\text{crit}}(p), & q(t) > \theta_q, \\ \tau_{\text{noncrit}}(p), & q(t) \le \theta_q. \end{cases} \tag{3.21}$$

A non-baseline candidate c is feasible if and only if

$$\rho_F(t,c) \le \delta(p,t) \tag{3.22}$$

and

$$\rho_K(t,c) \ge \tau(p,t). \tag{3.23}$$

Equations 3.22 and 3.23 form the locality certificate used by the placement rule. Within the scheduler's prediction model, every accepted deviation has bounded normalized finish-time loss and sufficient predicted copy reduction.

Among feasible candidates, the scheduler selects the candidate with maximum residual score

$$H(t,c) = \rho_K(t,c) + \lambda_Q \max(0, \rho_Q(t,c)). \tag{3.24}$$

In the evaluated copy-only configuration, $\lambda_Q = 0$, so

$$H(t,c) = \rho_K(t,c). \tag{3.25}$$

If no non-baseline candidate satisfies the gate, the task is assigned to the HEFT baseline. The resulting placement rule is

$$g^*(t) = \begin{cases} g_0(t), & \mathcal{F}(t,p_W) = \emptyset, \\ \arg\max\limits_{c \in \mathcal{F}(t,p_W)} H(t,c), & \mathcal{F}(t,p_W) \neq \emptyset. \end{cases} \tag{3.26}$$

where p_W is the profile selected for the current window and $\mathcal{F}(t,p_W)$ is the feasible set under that profile. Ties are resolved by earlier predicted finish time and then deterministically by GPU identifier. Figure 3.3 summarizes the decision flow.

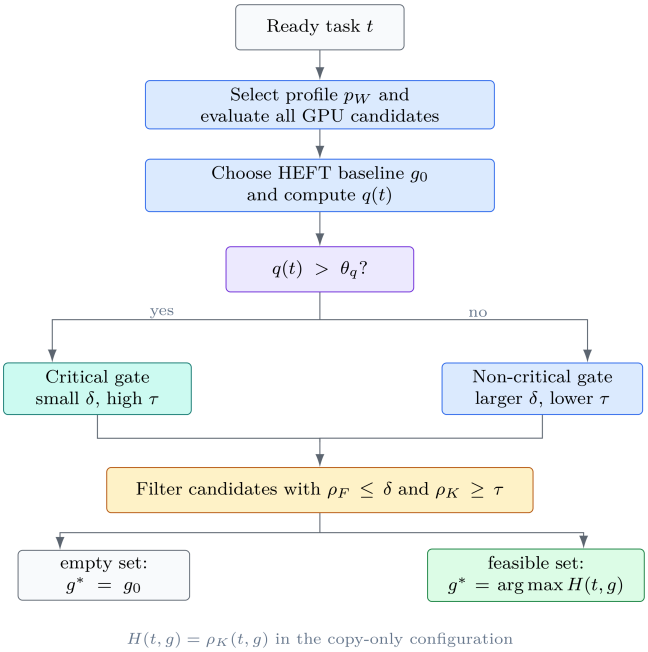

Figure 3.3: HEFT-relative residual placement decision. The scheduler accepts a non-baseline GPU only when the active profile's finish-time and copy-gain gates pass; otherwise it falls back to HEFT.

3.5 Window-Level Bandit Controller

The residual gate profile is selected at the granularity of a scheduling window rather than independently for every task. This design reduces oscillation and lets the scheduler adapt to phases of the task graph. A window contains up to N_W scheduled tasks; N_W is fixed for a run and reported as part of the evaluation metadata.

3.5.1 Context Features

At the end of each window, the scheduler constructs the context vector used by the next window:

$$x(W) = \begin{bmatrix} \overline{q} \\ \phi_{\text{high}} \\ \dfrac{\log(1+\overline{K}_0)}{\log(1+K_{\text{ref}})} \\ \dfrac{\log(1+\overline{B}_{\text{in}})}{\log(1+B_{\text{ref}})} \\ \dfrac{\overline{d}}{d_{\text{ref}}} \\ \dfrac{\log(1+\overline{C})}{\log(1+C_{\text{ref}})} \\ \overline{L}_{\text{imb}} \\ \phi_{\text{dev}} \end{bmatrix}. \tag{3.27}$$

Here $\overline{q}$ is average criticality, ϕ_{high} is the fraction of high-criticality tasks, $\overline{K}_0$ is average HEFT-baseline copy time, $\overline{B}_{\text{in}}$ is average input bytes, $\overline{d}$ is average total dependency count, $\overline{C}$ is average task cost, $\overline{L}_{\text{imb}}$ is average device-load imbalance, and ϕ_{dev} is the deviation rate in the completed window. The first window starts from a zero context because no previous window statistics are available.

3.5.2 Profile Selection

The controller maintains a separate LinUCB state for each profile p:

$$A_p^{-1} = \frac{1}{\lambda} I, \qquad b_p = 0 \tag{3.28}$$

at initialization. At the beginning of a window, the controller computes

$$\widehat{\theta}_p = A_p^{-1} b_p, \tag{3.29}$$

$$\mu_p = \widehat{\theta}_p^T x(W), \tag{3.30}$$

and

$$u_p = \alpha \sqrt{x(W)^T A_p^{-1} x(W)}. \tag{3.31}$$

The nominal score of profile p is

$$\text{score}(p, W) = \mu_p + u_p. \tag{3.32}$$

The default configuration favors the aggressive profile through an exploration-scale adjustment and an additive score prior for the aggressive arm. The first window is also initialized with the aggressive profile. After that, the controller chooses the profile with larger score, subject to a guardrail: the scheduler switches from aggressive back to default only when the default score exceeds the aggressive score by a configured margin. This rule reflects the empirical goal of retaining locality-preserving deviations unless there is clear evidence that the more conservative gate is preferable.

3.5.3 Reward and Update

For a task that deviates from HEFT, the task-level placement reward is

$$r_t = \rho_K(t, g^*) - \kappa(t) \rho_F(t, g^*), \tag{3.33}$$

where

$$\kappa(t) = \begin{cases} \kappa_{\text{crit}}, & q(t) > \theta_q, \\ \kappa_{\text{noncrit}}, & q(t) \leq \theta_q. \end{cases} \tag{3.34}$$

The default values are $\kappa_{\text{crit}} = 4.0$ and $\kappa_{\text{noncrit}} = 2.0$. If no alternative candidate is accepted, the task reward is zero.

The controller uses a relative reward. For each scheduled task, it also evaluates what reward the default profile would have obtained on the same candidate set. The reward credited to the active profile is therefore

$$r_t^{\text{rel}} = r_t^{\text{active}} - r_t^{\text{default}}. \tag{3.35}$$

This makes the default profile the reference behavior and trains the controller primarily on the incremental value of a more permissive gate.

At the end of a window, the reward used for the LinUCB update is

$$r(W) = \frac{1}{|W|} \sum_{t \in W} r_t^{\text{rel}} - \eta \phi_{\text{dev}}, \tag{3.36}$$

where ϕ_{dev} is the deviation rate and η is a small deviation penalty. The selected profile state is then updated using the Sherman–Morrison form of the LinUCB update:

$$z = A_p^{-1} x(W), \tag{3.37}$$

$$A_p^{-1} \leftarrow A_p^{-1} - \frac{z z^T}{1 + x(W)^T z}, \tag{3.38}$$

$$b_p \leftarrow b_p + r(W) x(W). \tag{3.39}$$

Only the active profile is updated for the completed window. Figure 3.4 shows the temporal loop between completed-window statistics, profile selection, execution, and reward update.

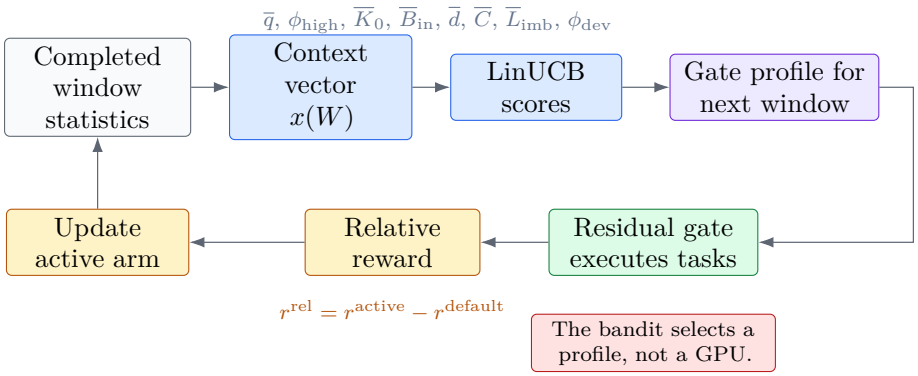

Figure 3.4: Window-level contextual-bandit adaptation. Window statistics form the context for the next scheduling window; LinUCB selects the residual-gate profile, and the selected profile is updated using relative reward after the window completes.

3.6 Placement Procedure and State Update

Algorithms 2 and 3 summarize the placement procedure and the corresponding scheduler-state update. The first algorithm describes the HEFT-relative decision rule, while the second algorithm describes how the selected placement is made visible to subsequent tasks through the ready-time and residency states. The separation is intentional. Algorithm 2 evaluates all candidate GPUs against the same pre-placement state, selects either the HEFT baseline or an accepted residual candidate, and records the information needed for the bandit update. Algorithm 3 is applied only after this decision has been made, so that candidate evaluation is not affected by partially updated residency or ready-time information.

This procedure is executed online for each task that becomes ready in the CUDASTF runtime. The scheduler does not build an offline schedule for the full DAG. Instead,

it repeatedly observes the current ready task, estimates the candidate finish time and copy cost on each GPU, applies the active residual gate, and then commits the selected placement to the runtime state. This online structure is important because both device availability and logical-data residency change after every scheduled task. A placement that is attractive before a write-like task may become unattractive immediately afterward if that task invalidates copies on other GPUs.

Algorithm 2 HEFT-relative placement decision

Require: Ready task t, GPU set $\mathcal{G}$, ready times R, residency model M, active window state W
Ensure: Selected GPU g^*

if there is no active placement window **then**
  $p_W \leftarrow \textsc{SelectProfile}(x(W))$
  $W \leftarrow \textsc{StartWindow}(p_W)$
end if
$C(t) \leftarrow \textsc{EstimateCost}(t)$
$\mathcal{I}(t) \leftarrow \textsc{InputDependencies}(t)$
for all $g \in \mathcal{G}$ **do**
  $\xi(t, g) \leftarrow \textsc{EvaluateCandidate}(t, g, R, M)$ ▷ D, K, S, F, Q
end for
$g_0 \leftarrow \arg\min_{g \in \mathcal{G}} F(t, g)$ ▷ HEFT baseline
$q(t) \leftarrow \textsc{CriticalityProxy}(C(t), d(t), B_{\text{in}}(t))$
$\mathcal{F} \leftarrow \emptyset$
for all $g \in \mathcal{G} \setminus \{g_0\}$ **do**
  compute $\rho_F(t, g)$, $\rho_K(t, g)$, $\rho_Q(t, g)$, and $H(t, g)$
  if $\rho_F(t, g) \leq \delta(p_W, t)$ **and** $\rho_K(t, g) \geq \tau(p_W, t)$ **then**
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{g\}$
  end if
end for
if $\mathcal{F} = \emptyset$ **then**
  $g^* \leftarrow g_0$
else
  $g^* \leftarrow \arg\max_{g \in \mathcal{F}} H(t, g)$ ▷ tie: finish time, GPU id
end if
accumulate task reward and window statistics for t
if the placement window is complete **then**
  update the LinUCB state of the active profile using the window reward
end if
return g^*

The state update after placement is essential because the next ready task observes the new device-ready time and the new data-residency state. Read accesses create or preserve shared valid copies on the executing GPU. Write-like accesses, including write, read-write, and reduction modes, make the executing GPU the modified owner and invalidate stale copies on other GPUs. Thus the residual gate always evaluates candidates against the current predicted data layout rather than against a static placement model.

The update also defines the boundary between the placement layer and the later DVFS layer. Placement decides the GPU and updates the scheduling state; DVFS is then allowed to attach a frequency decision to the already placed task, but it does not rewrite the placement. This ordering keeps the two objectives separable in the evaluation: placement experiments can disable DVFS while still using the same HEFT-relative decision path, and DVFS experiments can reuse a fixed placement

Algorithm 3 Scheduler-state update after placement

Require: Scheduled task t, selected GPU g^*, ready times R, residency model M
Ensure: Updated R and M
for all dependency s of task t **do**
  if s is read or read-write **then**
    record predicted copy bytes and copy time required on g^*, if any
  end if
  if s is read-only **then**
    mark $M[s, g^*] \leftarrow$ S and preserve other valid shared copies
  else if s is write-like **then**
    mark $M[s, g^*] \leftarrow$ M
    mark $M[s, h] \leftarrow$ I for all $h \in \mathcal{G} \setminus \{g^*\}$
  end if
end for
$R[g^*] \leftarrow F(t, g^*)$
return (R, M)

policy while changing only frequency behavior.

Finally, the procedure produces the diagnostic fields used in the Results chapter. For each scheduled task, the runtime can record the HEFT baseline GPU, the selected GPU, whether the decision deviated from HEFT, the predicted copy volume and copy time under the baseline and selected placement, the active window profile, and the task-level reward contribution. These records make it possible to distinguish genuine locality-driven deviations from incidental latency variation, which is why the evaluation reports both performance outcomes and data-movement evidence.

3.7 DVFS Framework

The DVFS layer is implemented as a conservative proposer and per-GPU window committer. It is separate from the placement bandit: placement chooses the GPU, and DVFS chooses the frequency level to use on that GPU. In the thesis experiments, the DVFS configuration reuses the same HEFT-relative placement path and adds only the proposal–commit frequency controller.

When DVFS is active, the runtime first checks that the selected placement scheduler supports per-task frequency annotations and that valid graphics-clock levels can be obtained for all GPUs. The method defines three abstract levels,

$$\ell \in \{\text{HIGH}, \text{MID}, \text{LOW}\}. \tag{3.40}$$

The high level is the highest supported graphics clock. The mid level is a lower supported clock chosen by rank, with rank two as the default. The low level is the next lower rank when low-frequency operation is globally allowed. On the default probed path, the memory clock is kept at the highest supported memory clock. If automatic probing is unavailable, the method can use explicit frequency-pair fallbacks only when sufficient high and mid frequencies are provided.

3.7.1 DVFS Control Flow

DVFS is invoked after the placement decision for a task has been made. For a task t placed on GPU g^*, the scheduler performs four steps:

1. mark the task as requiring a fresh stream, so that frequency changes are not mixed with a reused stream context;
2. compute a task-level DVFS proposal from local scheduling features;
3. accumulate the proposal into the per-GPU DVFS window for g^*;
4. annotate the current task with the frequency level already committed by the previous window of g^*, and then commit a new level if the current window has reached its size threshold.

The key design choice is that a task's proposal does not immediately set the frequency for that same task. Instead, proposals affect future tasks through the next committed window state. This delayed-commit design avoids high-frequency task-by-task oscillation.

3.7.2 Task-Level DVFS Proposer

The proposer maps each scheduled task to a frequency intent. It uses the same criticality proxy as the placement layer and adds runtime pressure features. For a task t placed on GPU g, define the normalized cost

$$c_{\mathrm{norm}}(t) = \mathrm{clip}_{[0,1]}\left(\frac{C(t)}{G_{\mathrm{ref}}}\right), \tag{3.41}$$

where the default reference time is $G_{\mathrm{ref}} = 1.0\,\mathrm{ms}$. Let

$$b_{\mathrm{norm}}(t,g) = \begin{cases} 0, & \max_h R[h] = \min_h R[h], \\ \dfrac{R[g] - \min_h R[h]}{\max_h R[h] - \min_h R[h]}, & \text{otherwise.} \end{cases} \tag{3.42}$$

be the normalized backlog of the selected GPU. The slack proxy is

$$s_{\mathrm{norm}}(t) = 1 - q(t), \tag{3.43}$$

and the wait time relevant to DVFS is

$$W(t) = K(t, g_0(t)) + Q(t, g_0(t)). \tag{3.44}$$

The copy-wait ratio is

$$\omega(t) = \frac{W(t)}{W(t) + C(t)}. \tag{3.45}$$

Large $\omega(t)$ indicates that the task is dominated by copy or queue waiting rather than by compute time, so reducing core frequency is less likely to harm performance.

The proposer also detects heavy kernels. A task is marked heavy if its estimated cost exceeds the heavy-task threshold or if its task symbol matches compute-intensive

Parameter	Role	Evaluation default
G_{ref}	Task-cost normalizer	1.0 ms
λ_q^w	Proposal-weight criticality multiplier	1.5
λ_C^w	Proposal-weight cost multiplier	1.0
γ_q	Diagnostic risk weight for criticality	0.35
γ_C	Diagnostic risk weight for task cost	0.20
γ_b	Diagnostic risk weight for backlog	0.15
γ_ω	Diagnostic risk weight for copy-wait ratio	0.20
γ_s	Diagnostic risk weight for slack	0.10
γ_h	Diagnostic risk weight for heavy tasks	0.15
θ_{force}	Task-level criticality force-high threshold	0.90
$C_{\text{heavy}}^{\text{strict}}$	Strict heavy-task force-high cost	1.0 ms
C_{light}	Maximum task cost for low intent	0.25 ms

Table 3.4: Task-level DVFS proposer hyperparameters.

kernels such as GEMM, SYRK, TRSM, POTRF, or POTRS. A heavy task is marked urgent-heavy when it is heavy and has low slack.

The proposal weight accumulated by the window committer is

$$w_{\text{dvfs}}(t) = 1 + \lambda_q^w q(t) + \lambda_C^w c_{\text{norm}}(t). \tag{3.46}$$

The method also defines a diagnostic DVFS risk score,

$$z_{\text{dvfs}}(t) = \gamma_q q(t) + \gamma_C c_{\text{norm}}(t) + \gamma_b b_{\text{norm}}(t,g) - \gamma_\omega \omega(t) - \gamma_s s_{\text{norm}}(t) + \gamma_h \mathbf{1}\{\text{heavy}(t)\}, \tag{3.47}$$

where the weights are listed in Table 3.4. These coefficients are fixed across experiments and serve only as diagnostic attribution weights; the actual proposal is governed by the threshold rules below.

The deterministic proposal rule is conservative. It returns HIGH if the mid frequency is not meaningfully below high, if

$$q(t) \geq \theta_{\text{force}}, \qquad \text{urgentHeavy}(t) = 1, \qquad C(t) \geq C_{\text{heavy}}^{\text{strict}}, \qquad b_{\text{norm}}(t,g) \geq b_{\text{force}}, \tag{3.48}$$

or if a heavy task is not sufficiently wait-bound and slack-rich. A heavy task is eligible for mid only when

$$\omega(t) \geq \omega_{\text{heavy}}^{\text{mid}}, \qquad s_{\text{norm}}(t) \geq s_{\text{heavy}}^{\text{mid}}, \qquad b_{\text{norm}}(t,g) \leq b_{\text{heavy}}^{\text{mid}}. \tag{3.49}$$

For non-forced cases, the mid-level intent is eligible when

$$\omega(t) \geq \omega_{\text{mid}}, \qquad q(t) \leq q_{\text{mid}}, \qquad \text{urgentHeavy}(t) = 0, \qquad b_{\text{norm}}(t,g) \leq b_{\text{mid}}. \tag{3.50}$$

The low-level intent is eligible only under stricter conditions:

$$q(t) \leq q_{\text{low}}, \qquad s_{\text{norm}}(t) \geq s_{\text{low}}, \qquad \omega(t) \geq \omega_{\text{low}}, \tag{3.51}$$

$$\text{heavy}(t) = 0, \qquad C(t) \leq C_{\text{light}}, \qquad b_{\text{norm}}(t,g) \leq b_{\text{low}}, \tag{3.52}$$

and only when mid is already eligible and low-frequency operation is enabled. The proposer returns LOW when the low rule passes, MID when only the mid rule passes, and HIGH otherwise.

Parameter	Role	Evaluation default
b_{force}	Task-level backlog force-high threshold	0.80
$\omega_{\text{heavy}}^{\text{mid}}$	Heavy-task wait ratio for mid intent	0.60
$s_{\text{heavy}}^{\text{mid}}$	Heavy-task slack for mid intent	0.70
$b_{\text{heavy}}^{\text{mid}}$	Heavy-task backlog limit for mid intent	0.35
ω_{mid}	Wait-ratio threshold for mid intent	0.25
q_{mid}	Criticality limit for mid intent	0.45
b_{mid}	Backlog limit for mid intent	0.60
q_{low}	Criticality limit for low intent	0.25
s_{low}	Slack threshold for low intent	0.50
ω_{low}	Wait-ratio threshold for low intent	0.40
b_{low}	Backlog limit for low intent	0.40

Table 3.5: Task-level DVFS intent thresholds.

3.7.3 Per-GPU Window Committer

Each GPU owns an independent DVFS window state. For every task placed on GPU g, the scheduler accumulates the proposal weight, weighted votes for HIGH, MID, and LOW, estimated runtime, criticality, task cost, copy-wait ratio, slack, backlog, and heavy-task indicators. The window also accumulates a slack reserve

$$\Sigma_{\text{slack}}(g, W) = \sum_{t \in W_g} s_{\text{norm}}(t) C(t), \tag{3.53}$$

where W_g is the subset of window tasks placed on GPU g.

For a lower frequency level ℓ, the committer estimates the extra runtime as

$$\Delta T_\ell(t) = \alpha_t T(t)(1 - \omega(t)) \left(\frac{f_{\text{HIGH}}}{f_\ell} - 1 \right), \tag{3.54}$$

where

$$\alpha_t = 1 + \lambda_q^\alpha q(t) + \lambda_h^\alpha \mathbf{1}\{\text{heavy}(t)\}, \tag{3.55}$$

$T(t)$ is the predicted runtime contribution of the task, and f_ℓ is the graphics clock of level ℓ. The default values of λ_q^α and λ_h^α are given in Table 3.6. Equation 3.54 makes the estimated slowdown larger for compute-bound, critical, or heavy tasks and smaller for copy-wait-bound tasks.

A commit is considered when the per-GPU window reaches the configured number of intents N_{dvfs}. Let V_{HIGH}, V_{MID}, and V_{LOW} be the weighted vote totals and let

$$p_\ell = \frac{V_\ell}{V_{\text{HIGH}} + V_{\text{MID}} + V_{\text{LOW}}}. \tag{3.56}$$

The committer computes the weighted criticality mass,

$$\bar{q}_w = \frac{\sum_{t \in W_g} w_{\text{dvfs}}(t) q(t)}{\sum_{t \in W_g} w_{\text{dvfs}}(t)}, \tag{3.57}$$

the urgent-heavy mass, the wait share

$$\Omega = \frac{\sum_{t \in W_g} W(t)}{\sum_{t \in W_g} W(t) + \sum_{t \in W_g} T(t)}, \tag{3.58}$$

Parameter	Role	Evaluation default
N_{dvfs}	Intents per per-GPU DVFS window	16
λ_q^{α}	Criticality multiplier in slowdown estimate	1.5
λ_h^{α}	Heavy-task multiplier in slowdown estimate	1.0
u_{force}	Urgent-heavy mass force-high threshold	0.15
q_{force}^{W}	Criticality-mass force-high threshold	0.55
b_{force}^{W}	Average-backlog force-high threshold	0.80
Ω_{mid}	Minimum wait share for mid commit	0.25
u_{mid}	Urgent-heavy mass limit for mid commit	0.10
q_{mid}^{W}	Criticality-mass limit for mid commit	0.45
b_{mid}^{W}	Average-backlog limit for mid commit	0.60
λ_{mid}^{S}	Slack-consumption coefficient for mid	0.35
B_{mid}	Additive mid performance budget	0.20 ms
$p_{\text{HIGH}}^{\max}$	High-vote limit for considering low	0.25
$h_{\text{low}}^{\max}$	Heavy-task fraction limit for low	0.20
$b_{\text{low}}^{\text{forbid}}$	Backlog threshold that forbids low	0.70
Ω_{low}	Minimum wait share for low commit	0.40
u_{low}	Urgent-heavy mass limit for low commit	0.0
q_{low}^{W}	Criticality-mass limit for low commit	0.25
b_{low}^{W}	Average-backlog limit for low commit	0.40
λ_{low}^{S}	Slack-consumption coefficient for low	0.25
B_{low}	Additive low performance budget	0.10 ms
n_{mid}	Consecutive eligible windows for mid	2
n_{low}	Consecutive eligible windows for low	3

Table 3.6: Window-level DVFS committer hyperparameters.

the average backlog, and the heavy-task fraction.

The window is forced to high frequency when the mid level is unavailable, a cooldown is active, a critical override has been triggered, the urgent-heavy mass exceeds u_{force}, the weighted criticality mass exceeds q_{force}^{W}, or the average backlog exceeds b_{force}^{W}. If none of these conditions holds, the mid level is eligible only when

$$\Omega \geq \Omega_{\text{mid}}, \qquad \text{urgentHeavyMass} \leq u_{\text{mid}}, \qquad \bar{q}_w \leq q_{\text{mid}}^{W}, \qquad \bar{b}_{\text{norm}} \leq b_{\text{mid}}^{W}, \tag{3.59}$$

and

$$\sum_{t \in W_g} \Delta T_{\text{MID}}(t) \leq \lambda_{\text{mid}}^{S} \Sigma_{\text{slack}}(g, W) + B_{\text{mid}}. \tag{3.60}$$

The low level is considered only when it is not disabled by global settings or guardrails, no cooldown is active, $p_{\text{HIGH}} < p_{\text{HIGH}}^{\max}$, the heavy-task fraction is below $h_{\text{low}}^{\max}$, and the average backlog is below $b_{\text{low}}^{\text{forbid}}$. It is then eligible only when mid is eligible, the window contains enough intents, no recent slowdown guardrail is active, and

$$\Omega \geq \Omega_{\text{low}}, \qquad \text{urgentHeavyMass} \leq u_{\text{low}}, \qquad \bar{q}_w \leq q_{\text{low}}^{W}, \qquad \bar{b}_{\text{norm}} \leq b_{\text{low}}^{W}, \tag{3.61}$$

with

$$\sum_{t \in W_g} \Delta T_{\text{LOW}}(t) \leq \lambda_{\text{low}}^{S} \Sigma_{\text{slack}}(g, W) + B_{\text{low}}. \tag{3.62}$$

To avoid oscillation, the committer requires consecutive eligible windows before entering a lower level. The required counts are n_{mid} for MID and n_{low} for LOW. If a forced-high condition is present, the committed level is HIGH. If the previous level is LOW and low is no longer eligible, the committer returns to HIGH. If the previous level is MID and mid is no longer eligible, it also returns to HIGH. Otherwise, the committer enters LOW or MID only after the required consecutive eligibility count has been reached. Figure 3.5 summarizes the proposer–committer state machine.

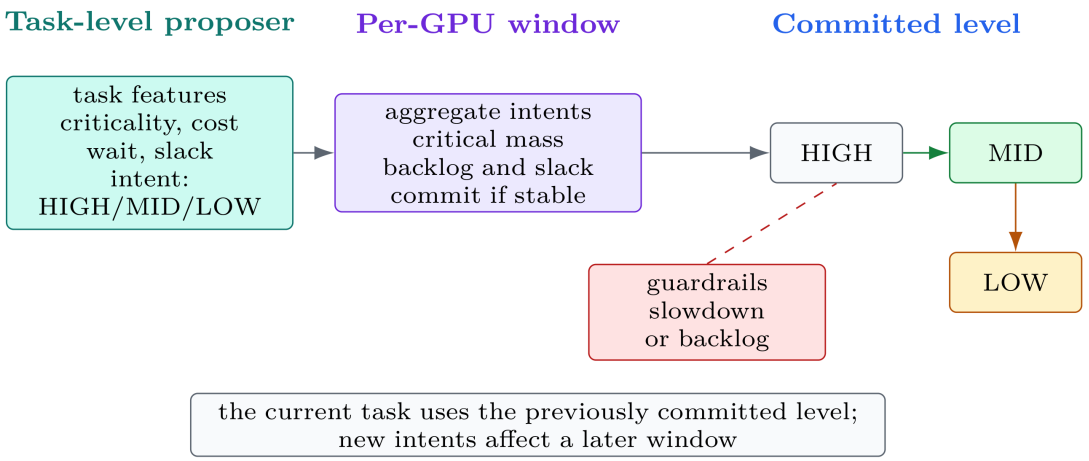

Figure 3.5: Guarded proposal–commit DVFS state machine. Task-level proposals are accumulated in per-GPU windows, while committed frequency levels change only when lower-frequency eligibility persists and no guardrail forces recovery to high frequency.

Parameter	Role	Evaluation default
r_{win}	Window-runtime slowdown threshold	2%
r_{tail}	Busy-tail slowdown threshold	3%
a_{ema}	Exponential moving-average coefficient	0.2
n_{recent}	Number of recent windows tracked	4
n_{trigger}	Recent slowdown indicators needed for recovery	2

Table 3.7: DVFS slowdown-guardrail hyperparameters.

3.7.4 Performance Guardrail and Frequency Application

The guardrail is not an additional frequency-optimization objective. Its role is to make the proposal–commit DVFS layer fail-safe. The proposer and committer decide from predicted task features, while the guardrail observes the realized behavior of recent windows and revokes lower-frequency operation when those decisions appear to increase runtime.

After each per-GPU commit, the runtime updates exponential moving averages for two window-level measurements using coefficient a_{ema}: the total window runtime and the busy-tail duration between the first and last task endpoints in the window. A slowdown indicator is raised when the current window runtime exceeds its moving average by more than r_{win} or when the busy-tail duration exceeds its moving average by more than r_{tail}. Recent indicators are stored in a ring buffer of n_{recent} windows. If at least n_{trigger} of the recent windows indicate slowdown, the next windows are forced to HIGH frequency for a recovery interval. If the slowdown follows a LOW commit, LOW is disabled for a longer interval. This mechanism prevents an unsafe lower-frequency choice from persisting across multiple scheduling windows. Table 3.7 reports the default guardrail values used in the evaluation.

Frequency application is also delayed by design. The committed level ℓ_g for GPU g

Algorithm 4 Windowed DVFS proposal and commit

Require: Scheduled task t, selected GPU g^*, current committed level ℓ_{g^*}, per-GPU DVFS window W_{g^*}

Ensure: Frequency annotation for t and updated committed level for future tasks

$(\ell_H, \ell_M, \ell_L) \leftarrow (\text{HIGH}, \text{MID}, \text{LOW})$
$\mathcal{L}_\downarrow \leftarrow \{\ell_M, \ell_L\}$ ▷ levels below high frequency
$u_t \leftarrow \textsc{MakeProposal}(t, g^*)$ ▷ $u_t \in \{\ell_H, \ell_M, \ell_L\}$
accumulate u_t, $w_{\text{dvfs}}(t)$, $q(t)$, $\omega(t)$, backlog, slack, heavy flags, and slowdown estimates into W_{g^*}
annotate t with $\text{freq}(\ell_{g^*})$
if W_{g^*} has fewer than N_{dvfs} intents **then**
    return
end if
compute vote shares, criticality mass, urgent-heavy mass, wait share, backlog, and slack budget for W_{g^*}
$h \leftarrow \textsc{ForceHigh}(W_{g^*})$ ▷ critical, heavy, backlog, cooldown, or slowdown guardrail
$\mathcal{E} \leftarrow \textsc{EligibleLowerLevels}(W_{g^*}, \mathcal{L}_\downarrow)$
update consecutive-window counters for each $\ell \in \mathcal{L}_\downarrow$
$\mathcal{E} \leftarrow \{\ell \in \mathcal{E} : \textsc{HysteresisSatisfied}(\ell)\}$
if h or ($\ell_{g^*} \in \mathcal{L}_\downarrow$ and $\ell_{g^*} \notin \mathcal{E}$) **then**
    $\ell_{g^*} \leftarrow \ell_H$
else if $\mathcal{E} \neq \emptyset$ **then**
    $\ell_{g^*} \leftarrow \textsc{LowestFrequency}(\mathcal{E})$ ▷ prefer ℓ_L over ℓ_M when both are safe
else
    $\ell_{g^*} \leftarrow \ell_H$
end if
update slowdown guardrail state and reset W_{g^*}

is applied to later tasks by annotating each task with

$$(f_{\text{gpu}}, f_{\text{mem}}) = \text{freq}(\ell_g), \tag{3.63}$$

where f_{mem} remains at the highest supported memory clock on the probed path. The stream-side runtime submits an asynchronous application-clock request before task execution, and a background tuner applies the requested clocks through NVML. DVFS-marked tasks require fresh streams so that a task is not launched on a reused stream whose execution context may correspond to a previous frequency level. The result is a window-level control loop: tasks generate intents, the committer chooses the next level, and the guardrail restores high frequency when measured behavior suggests performance risk.

3.8 Implementation and Instrumentation

The method is implemented inside the CUDASTF scheduler rather than as an external controller. The placement-only configuration uses the HEFT-relative residual

gate described above. The DVFS configuration reuses the same placement path and enables the per-device proposal–commit frequency controller. This keeps the experimental configurations comparable: the DVFS experiments evaluate the additional frequency-control layer on top of the same placement mechanism, rather than mixing placement changes with frequency changes.

The runtime also records structured diagnostic information used by the evaluation. For placement, the records include HEFT baseline choices, accepted residual deviations, copy estimates, window rewards, and selected gate profiles. For DVFS, they include task-level frequency intents, per-window vote summaries, committed frequency levels, and guardrail events. The concrete encoding of these records is treated as reproducibility metadata rather than as part of the scheduling method itself. In the thesis, these records are used only to connect observed performance differences to concrete mechanisms: baseline HEFT placement, copy-saving deviations, window-level profile selection, and DVFS frequency commits.

3.9 Method Summary

The method implemented in this thesis is a residual, locality-aware extension of HEFT. HEFT first supplies a safe minimum-finish-time baseline under the current GPU ready times and data-residency state. The residual gate then considers non-baseline GPUs only when they reduce predicted copy time by a sufficient normalized amount and increase the current task's predicted finish time by no more than a profile-dependent bound. The bandit component adapts this gate at window granularity, using interpretable workload statistics and a relative reward that measures improvement over the default gate.

The DVFS extension follows the same conservative design philosophy. It assigns per-task frequency intents based on criticality, compute cost, backlog, slack, copy-wait ratio, and heavy-kernel detection. These intents are not applied immediately; they are accumulated in per-GPU windows and committed only when there is enough evidence that a lower frequency is safe. Consecutive-window requirements, cooldowns, and slowdown guardrails force recovery to high frequency when performance risk appears. Together, the placement and DVFS layers form a runtime system that improves data locality and exposes an energy-control path while preserving HEFT as the fallback behavior.

4

Results

The evaluation studies the CUDASTF runtime scheduler on a single-node multi-GPU platform. It first describes the experimental platform and measurement methodology, then separates placement from DVFS before reporting the end-to-end behavior of the complete runtime. Data-movement, clock, and power measurements are used to explain the observed trends rather than to replace the primary latency, energy, and EDP results.

4.1 Experimental Setup

All experiments are conducted on a single Slurm-managed node with eight NVIDIA L4 GPUs. The application-visible GPU set contains all eight devices, and power sampling is performed over the same device set. This configuration aligns the placement and energy measurements: the scheduler can place work on any visible GPU, and the energy measurement covers all GPUs that may execute benchmark work.

Table 4.1 summarizes the hardware and software configuration. The benchmark executable and the power-sampling process run inside the same Apptainer environment, which keeps the compiler, CUDA runtime, and library configuration fixed across all measurements.

Component	Configuration
Node manager	Slurm-managed single node
GPUs	8 NVIDIA L4 GPUs
Application-visible GPUs	0–7
GPU power limit	72 W per GPU
Container runtime	Apptainer
Build type	Release
CUDA version	CUDA 12.4.1
NVCC version	12.4.131
GCC version	11.4
CUDA architecture	sm_89
High SM clock	approximately 2040 MHz
Middle SM clock target	1875 MHz
Memory clock	6251 MHz
Power sampling	NVML samples across all eight visible GPUs

Table 4.1: Hardware and software configuration used for the evaluation.

The NVIDIA L4 platform exposes application-clock control through NVML. The

evaluated DVFS policy changes the SM clock level while keeping the memory clock fixed at 6251 MHz. The evaluation therefore studies a conservative core-frequency control path rather than a joint core-and-memory frequency policy.

Table 4.2 summarizes the benchmark inputs and the observed task-graph scale. The benchmark suite includes both copy-sensitive task graphs and lower-opportunity task graphs, allowing the evaluation to test whether the scheduler improves favorable cases while keeping low-opportunity cases close to the baseline.

Benchmark	Problem size / input	Tasks	Baseline copy bytes
FDTD	3840, 288, 288, 288	491712	353009664
CG	16777216, 64, 4, 1e-10	34	654311738
MiniWeather	1600, 2048, 1536	57600	241833733942
Cholesky	49152, 2048	2600	15221178776
LU decomposition	25088, 256	318549	22258961026

Table 4.2: Benchmark characteristics from the placement raw data.

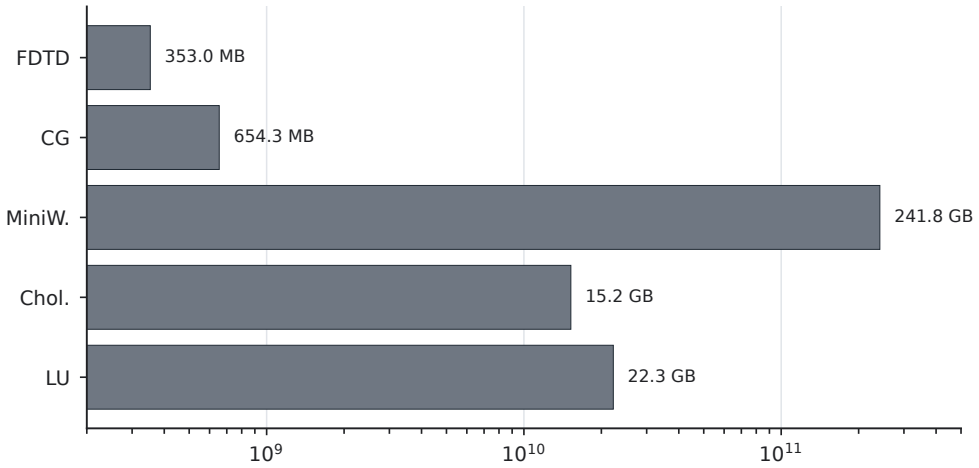

Figure 4.1: Baseline copy volume for the benchmark suite. The log-scale x-axis separates absolute copy traffic from residual-placement opportunity: MiniWeather has large baseline copy volume, but the placement measurements show little reducible copy traffic under the residual policy.

Figure 4.1 visualizes the baseline copy volume column on a logarithmic scale. This figure is included before the main result tables to avoid conflating raw copy volume with scheduler opportunity. The later locality evidence focuses on copy traffic that the residual scheduler actually reduces relative to HEFT.

4.2 Metrics and Measurement Methodology

Latency: Latency is the primary performance metric. It measures the duration of the timed benchmark execution window and is used for both placement and DVFS performance comparisons. This metric excludes process startup, final output, and unrelated host-side bookkeeping, so it focuses on the execution interval that can be affected by the runtime scheduler.

Energy consumption: Energy consumption is the primary energy metric for DVFS. At each NVML sampling timestamp, the result scripts record power readings for

the eight visible GPUs. Energy is computed by trapezoidal integration over the benchmark execution window. The final measurements have an observed sampling interval of approximately 100 ms. The resulting estimate is therefore limited by the NVML sampling resolution, especially when clock transitions are short relative to the sampling period.

Power: Power is the average NVML power over the same benchmark execution window. It is reported to explain energy changes, not as the primary objective by itself. A lower average power only improves energy if the corresponding runtime increase is small enough.

Whole-process energy: Whole-process energy is measured as an auxiliary diagnostic but is not used for the main claims. It includes startup, initialization, host-side work, and final output handling, whereas both the placement scheduler and the DVFS controller act on the timed benchmark execution window.

Speedup and geometric means: Placement speedup is computed as HEFT latency divided by scheduler latency. Suite-level summaries use geometric means so that each benchmark contributes multiplicatively and no single large absolute runtime dominates the aggregate.

Relative deltas: Relative deltas are computed against the baseline named in the corresponding section. For placement, the baseline is HEFT with DVFS disabled. For DVFS, the baseline is no-DVFS execution under the same placement layer. Negative deltas in latency, energy, power, and copy traffic indicate reductions relative to that baseline.

EDP: Energy–delay product is reported for DVFS as $E \times T$. This ratio shows whether energy savings still hold after accounting for runtime changes, especially for latency-sensitive workloads.

Data-movement and DVFS evidence: Copy-byte reduction is used as placement locality evidence. Average SM clock, middle-clock sample share, and commit or guardrail counters are used as DVFS evidence. These supporting measurements explain the observed latency and energy changes, but the primary claims are based on latency, energy consumption, and EDP.

Evidence roles: The measurements in this chapter serve different evidential roles. Placement performance provides the primary latency result; locality measurements provide diagnostic attribution; DVFS measurements provide energy–performance evidence under the evaluated configuration; and the overhead measurements are diagnostic checks rather than production-overhead estimates.

4.3 Evaluation Plan

The evaluation is organized as a sequence of controlled comparisons. Placement is evaluated first, with DVFS disabled, so that the effect of the residual scheduler can be measured against HEFT without frequency-control effects. The locality analysis then explains the placement result through copy traffic. Frequency control is evaluated in the next step by holding the placement layer fixed and comparing WinDVFS with a no-DVFS execution of the same scheduler. The final end-to-end comparison combines

the two layers and compares the full Bandit+WinDVFS configuration against HEFT.

Table 4.3 summarizes how the sections in this chapter connect to these comparisons. The purpose of the table is not to define separate benchmarks, but to keep the baseline of each result explicit. This is important because placement, data movement, DVFS, end-to-end behavior, and overhead diagnostics answer different parts of the evaluation.

Section	Role in the evaluation	Comparison	Reported evidence
Placement performance	Scheduler effect	HEFT vs Bandit, DVFS disabled	latency
Placement locality evidence	Attribution of scheduler gains	HEFT vs Bandit, DVFS disabled	copy footprint and latency
DVFS trade-off	Frequency-control effect	Bandit no-DVFS vs Bandit+WinDVFS	latency, energy, and EDP
End-to-end comparison	Combined system effect	HEFT vs Bandit+WinDVFS	latency, energy, and EDP
Overhead diagnostic	Control-path sanity check	Dynamic scheduling vs static replay	latency, energy, and EDP

Table 4.3: Evaluation structure and baselines used in the Results chapter.

The figure that follows gives the high-level outcome of the same evidence chain. It reports the normalized placement latency, the normalized DVFS energy, and the normalized end-to-end EDP. The detailed sections then expand each aggregate number with per-benchmark measurements and supporting diagnostics.

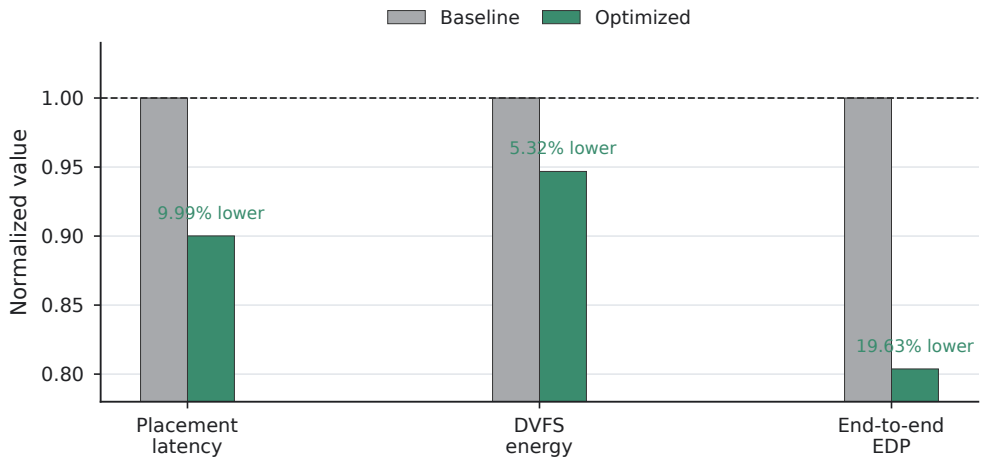

Figure 4.2: Overview of the main evaluation outcomes. Placement reports normalized latency relative to HEFT, DVFS reports normalized energy relative to no-DVFS execution under the Bandit placement layer, and the end-to-end result reports normalized EDP relative to HEFT. The y-axis is truncated to make the normalized differences visible.

4.4 Placement Performance

Figure 4.3 visualizes the placement-only latency result by normalizing HEFT to 100% for each benchmark and plotting the corresponding Bandit latency. Table 4.4 reports the absolute latency measurements. The result separates the benchmark suite into three behaviors. CG, Cholesky, and LU decomposition show strong placement benefits; FDTD shows a moderate positive effect; and MiniWeather remains near the baseline. This separation is important because the scheduler is expected to exploit locality only when the task graph exposes useful placement alternatives.

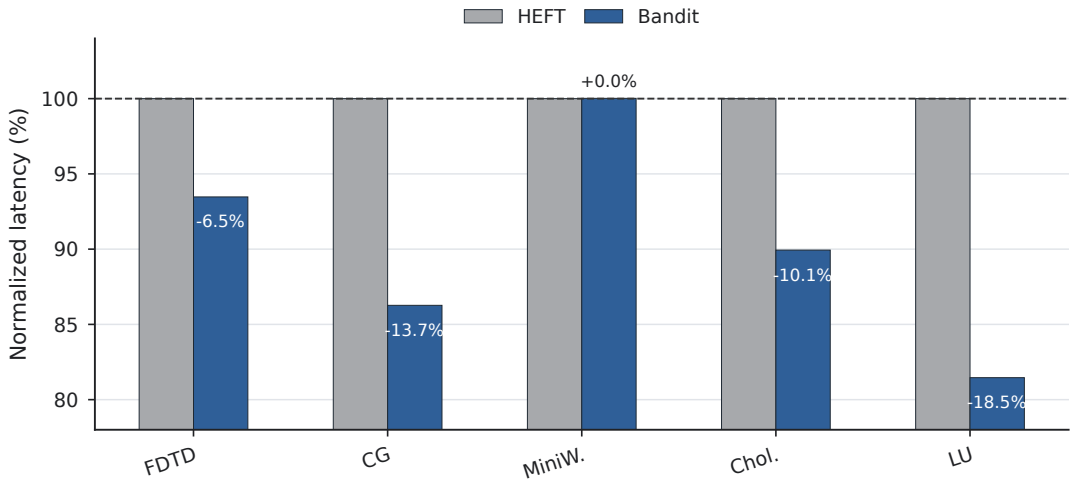

Figure 4.3: Normalized placement latency relative to HEFT. HEFT is fixed at 100% for each benchmark, and the Bandit bars show the corresponding normalized latency reduction.

Benchmark	HEFT latency	Bandit latency	Latency Δ
FDTD	6784 ms	6341 ms	−6.53%
CG	14301 ms	12337 ms	−13.73%
MiniWeather	15112 ms	15114 ms	+0.02%
Cholesky	20673 ms	18593 ms	−10.06%
LU decomposition	18592 ms	15145 ms	−18.54%

Table 4.4: Placement-only results relative to HEFT. DVFS is disabled.

The five-benchmark geometric-mean speedup is 1.111×, equivalent to a 9.99% normalized latency reduction. This aggregate value should be read together with the per-benchmark distribution. LU decomposition has the largest gain, followed by CG and Cholesky. These three workloads dominate the placement benefit because their task graphs expose useful alternatives to HEFT placement. FDTD is a moderate positive case. MiniWeather is effectively unchanged, indicating that the scheduler remains close to the baseline when there is little locality benefit.

The geometric mean is therefore a suite-level summary rather than evidence that every benchmark benefits equally. The stronger placement evidence comes from CG, Cholesky, and LU decomposition, where the effects are substantially larger. FDTD still contributes a clear positive result, while MiniWeather serves as the near-neutral case.

The placement claim is therefore based on latency and locality evidence rather than on frequency-control effects. The following locality section explains why the large latency reductions in CG, Cholesky, and LU decomposition are consistent with reduced data movement.

4.5 Placement Locality Evidence

The copy-traffic measurements are used as explanatory evidence rather than as a separate performance objective. Table 4.5 uses the same latency deltas as the final placement result in Table 4.4, and adds copy-traffic measurements for attribution. The purpose is to check whether the direction of the latency improvement is consistent with reduced data movement.

Benchmark	HEFT copy bytes	Bandit copy bytes	Copy bytes Δ	Latency Δ
FDTD	353,009,664	310,542,336	−12.03%	−6.53%
CG	654,311,738	603,980,114	−7.69%	−13.73%
MiniWeather	241,833,733,942	241,833,733,942	0.00%	+0.02%
Cholesky	15,221,178,776	9,403,679,128	−38.22%	−10.06%
LU decomposition	22,258,961,026	16,938,027,650	−23.91%	−18.54%

Table 4.5: Data-movement evidence for the placement result. Copy-byte counts are measured run means, and negative deltas indicate reductions relative to HEFT.

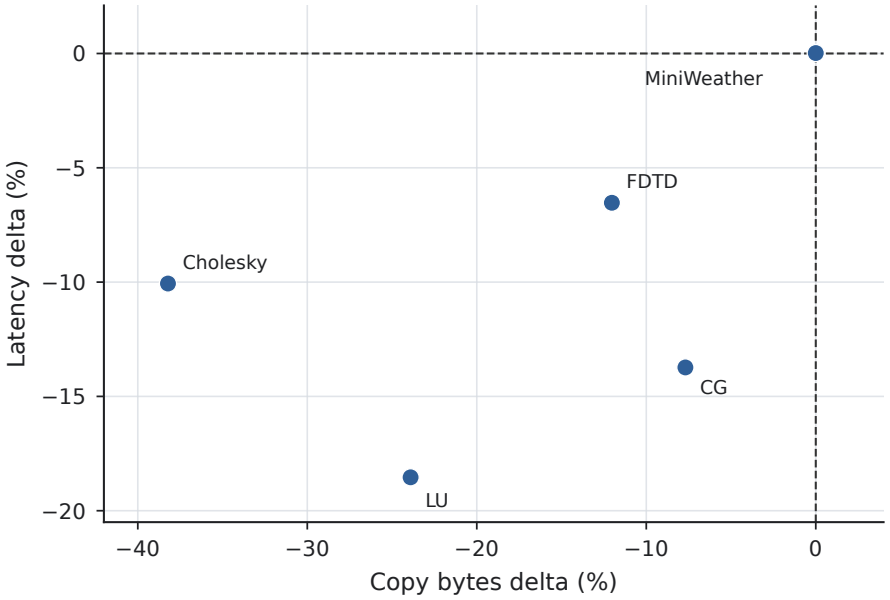

Figure 4.4: Relationship between copy-byte reduction and placement latency reduction. The latency deltas match the final placement table, while the copy-byte deltas provide attribution evidence.

The locality data separates the benchmarks into copy-saving and near-neutral cases. LU decomposition, Cholesky, FDTD, and CG all reduce copy volume, and all four also reduce latency in the final placement table. Figure 4.4 visualizes this relationship: the copy-saving benchmarks lie in the latency-improvement region, while MiniWeather stays close to zero on both axes.

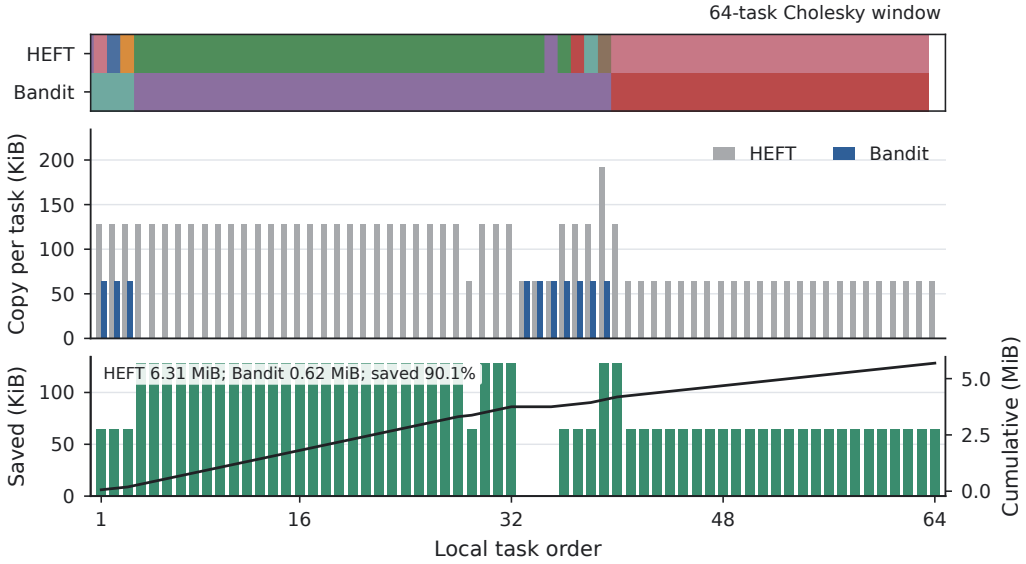

Figure 4.5: Representative local copy footprint from a Cholesky copy-attribution run. The selected window contains 64 consecutive tasks. The figure shows the device selected by each scheduler, the per-task copy volume, and the cumulative copy traffic removed by Bandit.

Figure 4.5 gives a local view of the same effect for Cholesky. In this representative 64-task window, HEFT produces 6.31 MiB of copy traffic, while Bandit produces 0.63 MiB. The resulting 90.1% local reduction illustrates how a sequence of placement changes can remove repeated tile transfers within a dense linear-algebra wavefront. This window-level example is used as attribution evidence, not to recompute the aggregate placement speedup.

MiniWeather provides the contrast case. Copy traffic is unchanged, and latency remains effectively at the HEFT baseline. This contrast supports the interpretation that the residual policy is selective: it changes placements when copy-saving opportunities are visible and otherwise remains close to HEFT-like behavior.

4.6 Trace-Based Explanation of Placement Gains

The previous section shows that the placement gains are aligned with reduced copy traffic, but copy bytes alone do not explain the full latency effect. CG is the most diagnostic case in this dataset. In the main placement result, CG reduces copy bytes by 7.69%, while its latency decreases by 13.73%. This gap does not mean that measured transfer time decreased by the same amount as latency. Copy-byte accounting is a locality-volume measurement: it indicates that less copy volume is observed, but it does not reveal overlap, queueing, readiness delay, or whether a transfer lies on the execution tail.

To separate these effects, a single-run targeted CG diagnostic compares three configurations: HEFT, the full residual Bandit placement policy, and a no-deviation control. The no-deviation row keeps the Bandit scheduler path but disables accepted non-HEFT residual placements. It therefore isolates whether the observed improve-

ment comes from residual placement decisions rather than from the surrounding scheduler code path.

Configuration	Latency	Latency Δ	Copy bytes	Copy bytes Δ	Accepted deviations
HEFT	14885 ms	0.00%	654,311,738	0.00%	0
Full residual	12624 ms	−15.19%	603,980,114	−7.69%	128
No deviation	14789 ms	−0.65%	654,311,762	0.00%	0

Table 4.6: Single-run CG no-deviation diagnostic. The no-deviation row keeps the Bandit scheduler path but disables accepted residual placement deviations. The table is used only for mechanism analysis and does not recompute the main placement result.

Table 4.6 indicates that the large diagnostic gain appears only when residual placements are allowed. With deviations disabled, copy volume is unchanged and latency remains close to HEFT, improving by only 0.65%. With deviations enabled, 128 accepted residual placements reduce copy bytes by 7.69% and latency by 15.19% in the same diagnostic setting. The mechanism question is therefore not whether copy traffic is lower, but how a small number of locality-preserving deviations changes the realized schedule.

For this purpose, a second diagnostic records CUDA-event start and end times for individual tasks. This task trace is intrusive because event timing adds synchronization, so it is not used as a headline latency measurement. It is used only to decompose the observed schedule into measured task work and idle or readiness gaps. The trace does not provide a direct memcpy timeline. For each GPU, the gap is computed from measured task timestamps as:

$$\text{gap}_d = (\text{last task end}_d - \text{first task start}_d) - \text{busy task time}_d.$$

Trace quantity	HEFT	Full residual	Delta
Total traced task duration	3899.3 ms	3937.1 ms	+37.9 ms
GPU 0 last task end	16460.9 ms	13590.1 ms	−2870.8 ms
GPU 0 busy task time	2186.0 ms	2858.0 ms	+672.0 ms
GPU 0 idle/readiness gap	14274.9 ms	10732.1 ms	−3542.8 ms

Table 4.7: Representative CG task-trace decomposition. The trace uses task start and end events, so the quantities are measured task timing and observed idle/readiness gaps, not modeled copy time.

Table 4.7 provides the trace-level explanation. The full residual run does not make the traced task bodies shorter. The total traced task duration slightly increases from 3899.3 ms to 3937.1 ms. The finishing GPU also receives more measured busy work. What changes is the non-busy part of the schedule: on GPU 0, the idle/readiness gap falls by 3542.8 ms, while busy task time increases by 672.0 ms. The net effect is a 2870.8 ms earlier last task end.

This trace explains why, in this diagnostic, the latency reduction can be larger than the copy-byte reduction. The residual scheduler is not merely removing a proportional amount of transfer volume. It is changing where data is resident and

when later dependent tasks become ready. In the CG trace, these changes repeat across the iterative task structure. Grouping matched timed tasks into recurring blocks, the full residual policy gains about 45.7 ms of additional end-time advantage per block, whereas the no-deviation control gains only about 1.4 ms per block. The latency improvement is therefore amplified through schedule compression along the repeated dependency chain. Copy bytes provide the directional locality evidence; the task trace shows that the runtime benefit is realized primarily as reduced readiness gaps on the execution tail.

4.7 DVFS Energy–Performance Trade-off

DVFS is evaluated as an energy–performance trade-off rather than as a pure power-reduction objective. Lower average power does not necessarily imply lower energy when runtime increases. Unlike the placement result, the DVFS result is interpreted as a final-configuration energy–performance measurement rather than as a repeated-run statistical comparison. The analysis therefore emphasizes multi-percent energy changes and treats sub-percent changes as near-neutral unless they are supported by power, clock, or trade-off evidence. Figure 4.6 reports latency, energy, and EDP together, and Table 4.8 gives the corresponding relative deltas. The absolute values used to compute the deltas are reported in Appendix A.

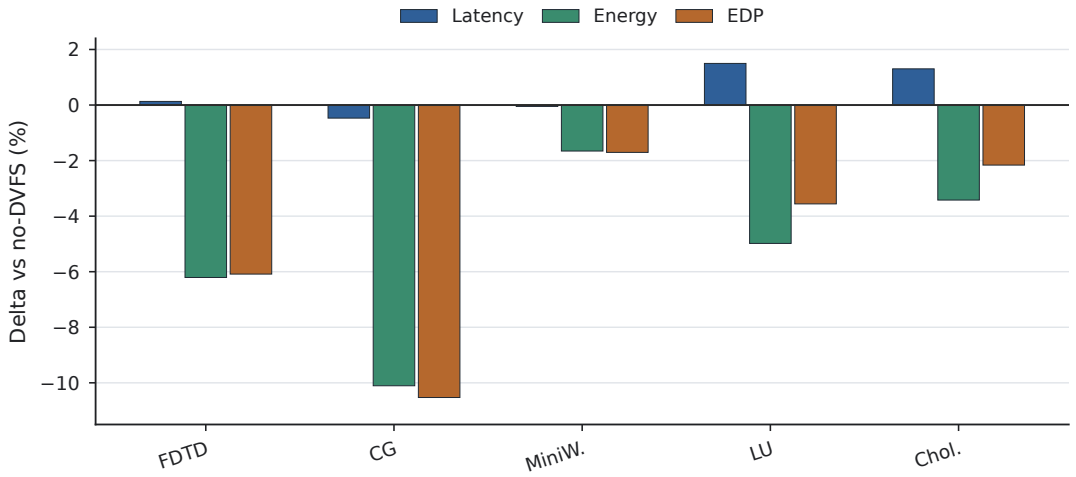

Figure 4.6: Per-benchmark DVFS latency, energy, and EDP deltas relative to no-DVFS execution. Negative values indicate improvement for each metric.

Benchmark	Latency Δ	Energy Δ	EDP Δ
FDTD	+0.13%	−6.21%	−6.09%
CG	−0.47%	−10.11%	−10.53%
MiniWeather	−0.05%	−1.66%	−1.70%
LU decomposition	+1.50%	−4.98%	−3.56%
Cholesky	+1.30%	−3.42%	−2.16%

Table 4.8: Final WinDVFS result relative to no-DVFS execution under the same placement layer. Negative values indicate reductions relative to the no-DVFS baseline.

Across the five benchmarks, the geometric-mean energy ratio is 0.947, corresponding

to 5.32% lower normalized energy. The geometric-mean latency ratio is 1.005, corresponding to a 0.48% latency overhead. The geometric-mean EDP ratio is 0.951, corresponding to a 4.87% improvement. This is the main DVFS result: guarded window-level frequency control reduces energy and EDP on this benchmark set while keeping the aggregate latency cost below one percent.

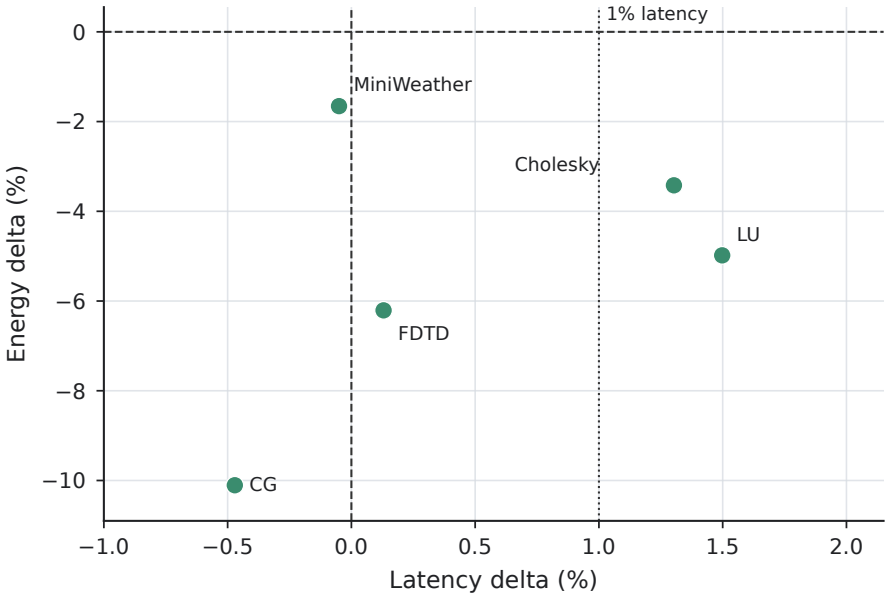

Figure 4.7: Energy–latency trade-off of WinDVFS. The dotted vertical line marks a 1% latency-overhead reference. Points below the horizontal axis reduce energy; points to the left of the vertical axis also reduce latency.

Figure 4.7 shows why the aggregate result needs to be read as a trade-off. CG provides the strongest row, with slightly lower latency and a 10.11% energy reduction. FDTD is close to latency-neutral while still reducing energy by 6.21%. MiniWeather is also nearly latency-neutral, but its energy reduction is smaller. LU decomposition and Cholesky accept modest latency overheads of 1.50% and 1.30%, respectively, while still improving both energy and EDP.

Benchmark	EDP ratio
FDTD	0.939
CG	0.895
MiniWeather	0.983
LU decomposition	0.964
Cholesky	0.978
Geometric mean	0.951

Table 4.9: DVFS energy–delay ratios computed from the energy and latency ratios in Table 4.8. Values below 1.0 indicate improvement.

The EDP ratios support the same conclusion. All five benchmarks improve after runtime is accounted for, and the largest EDP improvement appears on CG. The smaller EDP improvements for MiniWeather and Cholesky show the boundary of the effect: the policy is most useful when the energy reduction is large enough to remain visible after the latency term is included.

Figure 4.8: DVFS EDP ratios relative to no-DVFS execution. Values below the dashed 1.0 line indicate improvement after accounting for runtime.

4.8 End-to-End Performance and Energy

The previous sections isolate the two components of the system: placement is measured against HEFT with DVFS disabled, and frequency control is measured against the same placement layer without DVFS. Table 4.10 combines the two effects by comparing HEFT directly with the final Bandit+WinDVFS configuration. The absolute values behind this end-to-end comparison are reported in Appendix A.

Benchmark	Latency Δ	Energy Δ	EDP Δ
FDTD	−5.56%	−8.36%	−13.45%
CG	−13.48%	−20.13%	−30.89%
MiniWeather	+0.05%	−4.04%	−3.99%
LU decomposition	−15.14%	−20.37%	−32.42%
Cholesky	−9.04%	−4.98%	−13.56%

Table 4.10: End-to-end comparison between HEFT and the final Bandit+WinDVFS configuration. Negative values indicate reductions relative to HEFT.

The end-to-end comparison gives a geometric-mean latency ratio of 0.912, which corresponds to an 8.80% latency reduction relative to HEFT. The geometric-mean energy ratio is 0.881, corresponding to an 11.87% energy reduction. Because both latency and energy improve on most benchmarks, the EDP effect is stronger: the geometric-mean EDP ratio is 0.804, or a 19.63% reduction.

The per-benchmark distribution is also important. CG and LU decomposition are the strongest end-to-end cases because they benefit from both placement and frequency control. FDTD and Cholesky also improve substantially, although the source of the benefit differs: FDTD combines moderate latency reduction with a larger energy reduction, while Cholesky receives most of its benefit from placement and a smaller additional energy gain. MiniWeather remains the near-neutral latency control, but it still reduces energy and therefore lowers EDP.

4.9 Diagnostic Overheads and Claim Boundaries

The overhead analysis uses static replay as a diagnostic control. A dynamic Bandit run first records the final device assignment of each task. The static-replay scheduler then reuses that trace and bypasses the online Bandit device-selection logic. This comparison does not replace the dynamic scheduler, because the replay trace is produced in a separate run and cannot reproduce all runtime state. It is nevertheless useful for checking whether the online scheduling path introduces a visible cost at the scale of the reported benchmarks.

	Latency (ms)		Relative delta		
Benchmark	Dynamic	Static replay	Latency	Energy	EDP
FDTD	6283.0	6247.0	−0.57%	−0.51%	+3.68%
CG	12404.0	12582.5	+1.44%	+0.96%	+2.76%
MiniWeather	15101.0	15092.0	−0.06%	+0.13%	−0.08%
Cholesky	18701.5	18558.3	−0.77%	+0.05%	−0.13%
LU decomposition	15389.7	15516.2	+0.82%	+1.06%	+3.05%

Table 4.11: Static-replay overhead diagnostic for the Bandit scheduler. Deltas are computed for static replay relative to the corresponding dynamic Bandit run; values close to zero indicate that online scheduling cost is not visible at benchmark scale.

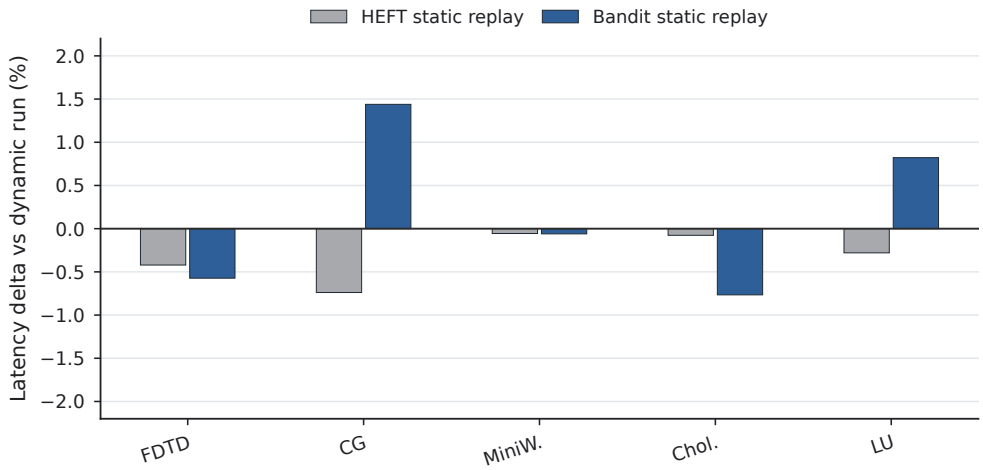

Figure 4.9: Static-replay overhead diagnostic. Bars report the latency delta of static replay relative to the corresponding dynamic run. The Bandit replay row is the main overhead diagnostic; the HEFT replay row is shown only as a trace-replay sanity check.

The Bandit replay rows are the cleanest diagnostic for scheduling overhead. Across the five benchmarks, the latency delta between dynamic Bandit and static replay ranges from −0.77% to +1.44%, with a geometric-mean static-replay delta of +0.17%. Energy deltas remain between −0.51% and +1.06%. These values are small relative to the placement gains reported in Section 4.4, which indicates that the online Bandit decision path is not the source of the measured latency improvements.

Figure 4.9 also shows a HEFT static-replay sanity check. This row is not used for the main overhead result; the interpretation is based on the Bandit replay rows in Table 4.11. Static replay is therefore used only as a diagnostic check, not as a

replacement for the main dynamic execution results.

The evaluation has four main limitations. First, all results are measured on a single eight-GPU node, so the magnitude of the results may change with GPU generation, interconnect topology, driver behavior, and benchmark scale. Second, static replay is a diagnostic approximation and does not reproduce all online scheduler state. Third, FDTD remains more sensitive to run-to-run variation than the linear-algebra workloads. Fourth, the DVFS conclusion is based on energy and latency measured over the benchmark execution window; it does not replace a full-system power study.

4.10 Results Summary

The experimental evidence supports both layers of the proposed design. The bandit scheduler improves geometric-mean latency by 9.99% relative to HEFT, with the largest gains appearing on copy-sensitive workloads. The data-movement results are consistent with a locality-driven explanation of these gains: the strongest placement cases reduce copy traffic, while the near-neutral workload remains close to HEFT.

WinDVFS reduces geometric-mean energy by 5.32% with a 0.48% geometric-mean latency overhead. The geometric-mean EDP ratio improves to 0.951, and all five reported benchmarks remain below an EDP ratio of 1.0. When the two layers are evaluated end to end against HEFT, the final Bandit+WinDVFS configuration reduces geometric-mean latency by 8.80%, energy by 11.87%, and EDP by 19.63%.

The boundary of the claim is therefore explicit. The placement result is a locality-sensitive latency improvement, not a claim that every task graph benefits equally. The DVFS result is measured over the benchmark execution window on one eight-GPU L4 node, not as a full-system energy study. The static-replay overhead diagnostic further indicates that online Bandit scheduling cost is small compared with the measured placement gains. Within these boundaries, this chapter shows three measured outcomes: the placement layer improves suite-level latency while preserving near-baseline behavior on low-opportunity workloads; the DVFS layer reduces suite-level energy while keeping the geometric-mean latency overhead below one percent; and the combined system improves the energy–delay product relative to HEFT.

5

Discussion

The results in Chapter 4 support a specific claim rather than a universal one. The placement layer improves runtime when the CUDASTF task graph exposes copy-saving alternatives to HEFT, and it falls back to HEFT-like behavior when such alternatives are absent. The DVFS layer then provides conservative energy control on top of the same placement framework. Its effect is measurable but workload dependent, which is consistent with the energy–performance trade-offs described in Chapter 2.

5.1 Interpreting the Placement Result

The placement results support the main design choice of this thesis: HEFT should remain the performance anchor, while locality-aware deviations should be admitted only when there is sufficient copy-saving evidence. In the placement performance experiment, the bandit scheduler improves geometric-mean latency by 9.99% across the five reported benchmarks. The improvement is not uniform. CG, Cholesky, and LU decomposition show the strongest gains, FDTD is a moderate positive case, and MiniWeather remains near neutral.

This non-uniformity is expected for a residual placement policy. The scheduler does not try to replace HEFT with a globally different placement strategy. Instead, it asks whether a non-HEFT GPU can reduce predicted copy cost while keeping the local finish-time penalty bounded. The locality-attribution analysis confirms that this behavior is visible in the measurements. The largest improvements coincide with large copy byte reductions: Cholesky reduces copy bytes by 38.22%, LU decomposition reduces copy bytes by 23.91%, and CG reduces copy bytes by 7.69%. MiniWeather keeps the same measured copy footprint and remains close to HEFT, which is consistent with the interpretation that residual placement is useful only when the task graph exposes reducible data movement.

The trace-based mechanism analysis in Section 4.6 adds an important nuance to this interpretation. Copy-byte reduction should not be read as a direct decomposition of latency. It is a locality-volume metric, not a measured trace of transfer time or critical-path delay. The no-deviation control shows why the locality explanation remains plausible: when the same scheduler path is used but non-HEFT residual placements are disabled, the result returns to the near-HEFT regime. The task trace then explains where the larger latency effect comes from: the accepted residual placements reduce idle and readiness gaps along the repeated execution chain, even

though the measured task work itself does not become shorter.

The placement result should therefore be read as a workload-dependent locality result rather than as a uniform scheduler replacement. This matters for a runtime scheduler because a method that improves copy-sensitive workloads while preserving near-neutral behavior on low-opportunity workloads is more useful than one that pursues aggressive deviations everywhere.

5.2 Interpreting the DVFS Result

The WinDVFS result should be interpreted as a conservative energy-control result rather than as a universal energy optimum. Across the five final rows, WinDVFS reduces geometric-mean energy by 5.32% with a geometric-mean latency overhead of 0.48%. The geometric-mean EDP ratio is 0.951. FDTD, CG, MiniWeather, LU decomposition, and Cholesky all reduce EDP relative to their no-DVFS counterparts, but the size of the benefit remains workload dependent.

This outcome is consistent with the proposal–commit design. The DVFS layer does not directly optimize each task in isolation. It accumulates task-level frequency intents into per-GPU windows and commits a lower frequency only when guardrails indicate that the performance risk is acceptable. The DVFS clock and power measurements are used as supporting diagnostics rather than as the main claim. The final DVFS measurements show that the larger energy reductions are consistent with lower average power in the selected configurations, but the primary evidence remains the measured energy, latency, and EDP trade-off. Cholesky is the clearest trade-off case: it reduces energy, but its latency increase limits the EDP improvement. MiniWeather provides a contrasting case where latency remains essentially unchanged and the energy reduction is smaller. These differences show that the controller should be interpreted as a guarded trade-off mechanism, not as a rule that every workload should be down-clocked in the same way.

The DVFS evidence is therefore best viewed as showing that task-graph runtime information can support safe, window-level frequency control. It does not show that the current heuristic is globally optimal, nor that every benchmark should be down-clocked. The measured benefit is modest, but it is obtained while preserving the separation between placement and DVFS and while keeping the aggregate latency cost small.

5.3 Answering the Research Questions

The first research question asks how a CUDASTF runtime scheduler can preserve the performance advantages of HEFT while exploiting data locality. The answer in this thesis is a HEFT-relative residual gate. HEFT provides the baseline GPU choice, and the scheduler deviates only when the predicted copy saving is large enough and the finish-time penalty is bounded. The placement performance and locality results show that this design improves latency on copy-sensitive workloads while remaining close to HEFT on neutral workloads.

The second research question asks whether observed latency changes can be attributed to concrete locality behavior. The locality-attribution experiment provides the strongest evidence for this question. The benchmarks with the largest latency improvements also show substantial copy-byte reductions. Conversely, the near-neutral benchmark shows no measured copy-byte reduction. This alignment does not prove that copy traffic explains every millisecond of runtime variation, but it gives a clear data-movement account of the main placement improvements.

The third research question asks how GPU DVFS decisions can be integrated into the task scheduler without unstable per-task control. The proposal–commit layer addresses this by separating task-level frequency intent from actual frequency commitment. Tasks vote for high, mid, or low frequency, but the committer changes the device-level committed level only at window boundaries and only after guardrail checks.

The fourth research question asks whether the two-level policy can improve energy while keeping runtime degradation bounded. The final DVFS result gives a qualified positive answer. Energy decreases by 5.32% geometric mean, while latency increases by only 0.48% geometric mean under the fixed Bandit placement layer. The EDP ratio improves to 0.951. The result is workload dependent, so the correct conclusion is that the runtime can extract energy savings on the reported benchmarks without large aggregate slowdown, not that DVFS is always beneficial.

The fifth research question concerns overhead. The overhead diagnostics show that the dynamic Bandit scheduler and the static-replay diagnostic differ by only -0.77% to $+1.44\%$ in latency across the five reported benchmarks, with a geometric-mean static-replay delta of $+0.17\%$. This supports the interpretation that online Bandit decision overhead is small at benchmark scale. The HEFT replay row is shown only as a trace-replay sanity check; the main overhead interpretation comes from the Bandit replay row. Static replay is therefore treated as diagnostic evidence rather than as an exact zero-overhead scheduler.

5.4 Practical Implications

The main practical implication is that placement and DVFS should be separated in an online multi-GPU runtime. Placement has an immediate effect on data movement and queueing; DVFS affects power, frequency transitions, and possible slowdown. Combining both into one large online action space would make reward attribution difficult and would increase the risk of unstable behavior. The layered design used here keeps each decision interpretable: placement can be explained relative to HEFT and copy savings, while DVFS can be explained through proposal shares, committed levels, clock exposure, and guardrail events.

The results also suggest that non-regression behavior is as important as peak improvement. A scheduler that improves only a subset of benchmarks is still useful if it reliably avoids harming low-opportunity workloads. In this thesis, MiniWeather is important precisely because it is a near-neutral control. It shows that the scheduler does not need to force placement changes where the measured locality opportunity

is small.

5.5 Limitations

The main limitation is experimental scope. The final measurements are taken on a single-node eight-GPU configuration. The results may change on other GPU models, interconnect topologies, CUDA driver versions, or problem sizes. In particular, data-locality benefits depend on the relative cost of computation, copy traffic, and queueing. A topology with different peer bandwidth or copy engine behavior may change the magnitude of the placement gains.

The workload set is also limited. The five reported benchmarks cover several useful cases, including strong locality cases, a near-neutral placement control, and DVFS-positive rows, but they do not exhaust the range of CUDASTF applications. The FDTD results also show that benchmark-level noise can affect the apparent magnitude of small effects. This is why the thesis treats FDTD as a moderate placement case rather than as the primary proof point.

The placement model uses a lightweight residency and copy-cost approximation. This is appropriate for an online scheduler, but it cannot perfectly model peer-copy overlap, copy-engine contention, topology-dependent bandwidth, or all interactions with launch overheads. Similarly, the criticality proxy used by placement and DVFS is local rather than a full dynamic critical path computation. These simplifications keep the method practical, but they also limit how precisely the scheduler can predict future global makespan.

Finally, the DVFS evaluation uses energy measured over the benchmark execution window as the primary energy metric. This is the most relevant metric for the frequency-control effect studied in this thesis, but it is not a full-system energy study. A complete deployment evaluation would need longer runs, more hardware platforms, full-system power measurements, and a more detailed analysis of transition costs and background runtime overhead.

6

Future Work

The results in this thesis show that task-graph information can support locality-aware placement and guarded DVFS control inside CUDASTF. At the same time, the current implementation is deliberately conservative, and the evaluation is limited to a single-node benchmark suite. The most important extensions concern broader evaluation, richer runtime models, stronger DVFS objectives, lower-overhead instrumentation, and more systematic deployment support.

6.1 Broader Hardware and Workload Evaluation

A first extension is broader evaluation. The current results are from one eight-GPU node and five reported benchmarks. Future experiments should repeat the placement and DVFS evaluation on different GPU generations, different interconnect topologies, and different problem sizes. This would test whether the locality improvements remain stable when peer-copy bandwidth, copy-engine contention, memory capacity, and computation time change.

The benchmark suite should also be expanded. The current workloads include strong locality winners and neutral controls, but a larger suite would make it easier to identify the boundary between useful residual placement and HEFT-like fallback behavior. In particular, applications with irregular data reuse, phase changes, and mixed compute/communication behavior would be useful stress tests for both the placement gate and the DVFS guardrails.

6.2 More Accurate Slack and Criticality Models

The current scheduler uses a local criticality proxy based on task cost, dependency count, and input-byte volume. This proxy is inexpensive and suitable for online use, but it is not a full dynamic critical-path analysis. A natural extension is to add a lightweight DAG-aware slack estimator that updates as the runtime observes task completions and queue states.

Such a model could improve both layers of the scheduler. For placement, more accurate slack estimates could allow the residual gate to distinguish between safe non-HEFT deviations and deviations that are likely to delay the critical path. For DVFS, better slack information could identify windows where lower frequency is safe because the extra runtime is hidden by dependency structure or communication.

The challenge is to gain this information without adding enough overhead to erase the benefit.

6.3 Topology-Aware Locality Modelling

The current locality model treats copy saving as a central scheduling signal, but the copy-cost estimate is intentionally lightweight. Future work should make this model topology aware. On modern multi-GPU systems, the cost of moving data depends on whether GPUs are connected by NVLink, PCIe, or another interconnect, and on whether copy engines are already busy.

A topology-aware model could assign different costs to different source–target GPU pairs and could account for asymmetric bandwidth or contention. It could also distinguish copy bytes that are likely to overlap with computation from copy bytes that are likely to appear on the critical path. This would make the locality certificate more precise while preserving the HEFT-relative structure of the scheduler.

6.4 Richer DVFS Objectives

The WinDVFS layer in this thesis focuses on guarded energy reduction over the benchmark execution window with bounded latency overhead. Future work could extend the committer to optimize different objectives, such as EDP or a user-defined energy budget. The same proposal–commit structure could support these objectives by changing the reward or eligibility rule used at window boundaries.

Another direction is to learn the DVFS commit policy online. The current committer is rule based: it uses proposal shares, criticality mass, backlog, slack budget, and slowdown guardrails. A learned committer could adapt these thresholds across workloads while keeping the existing guardrails as hard safety constraints. This would preserve the conservative behavior needed for runtime use while reducing manual parameter sensitivity.

The DVFS action space could also be expanded. The current design primarily controls graphics frequency and treats memory frequency conservatively. Some workloads may benefit from coordinated graphics- and memory-clock decisions, especially when memory bandwidth rather than core throughput dominates runtime. Adding memory-clock control would require stronger safeguards because incorrect memory-frequency decisions can increase latency substantially.

6.5 Transition Placement and Overhead Reduction

The proposal–commit design amortizes frequency decisions over windows, but it does not fully solve transition placement. Future work should measure and model where transitions become visible in the CUDASTF execution timeline. If the runtime can

schedule transitions during communication, synchronization, or other slack periods, the visible latency cost of DVFS could be reduced.

Instrumentation overhead should also be reduced. The static-replay diagnostic in this thesis suggests that online Bandit decision overhead is small at benchmark scale, but the replay experiment is still only an approximation of a zero-overhead scheduler. A production-oriented implementation should separate low-overhead counters used during normal execution from detailed trace logging used only for debugging and thesis-style attribution. This would keep the scheduler explainable without making detailed logs part of the critical path.

6.6 Reproducibility and Deployment

A final direction is reproducibility and deployment. Future versions should standardize the experiment records produced by each run, including benchmark summaries, scheduler configuration, device metadata, and power-sampling summaries. A stable record format would make it easier to compare results across hardware and to regenerate thesis figures without relying on ad hoc log parsing.

For deployment, the scheduler should expose a small set of user-facing modes: for example, placement-only, conservative DVFS, and diagnostic logging. These modes would let users choose between maximum stability, energy savings, and explainability depending on the application. The long-term goal is a CUDASTF scheduler that offers locality and energy-efficiency modes by default while remaining transparent enough for users to understand when and why it deviates from HEFT.

7

Conclusion

This thesis investigated runtime-level scheduling for single-node multi-GPU CUDASTF task graphs. The motivation was that GPU placement and GPU frequency control are coupled through data movement, slack, and critical-path behavior, but combining them into one monolithic online decision would make the scheduler difficult to learn, explain, and stabilize. The proposed solution is a two-layer runtime framework: a HEFT-relative residual placement scheduler followed by a guarded, windowed proposal–commit DVFS controller.

The placement layer keeps HEFT as the latency-oriented baseline. It evaluates candidate GPUs using data-ready time, copy time, queueing delay, and predicted finish time, and it accepts a non-HEFT placement only when the predicted copy saving is large enough and the normalized finish-time penalty is bounded. A window-level contextual bandit adapts the aggressiveness of this residual gate without directly choosing GPUs. The resulting policy remains explainable: every accepted deviation is expressed relative to the HEFT baseline and to a copy-saving condition.

The DVFS layer is deliberately separated from placement. Each task produces a frequency intent, but actual GPU frequency changes are committed only at per-device window boundaries. Criticality, backlog, heavy-task detection, slack budget, consecutive-window requirements, cooldowns, and slowdown guardrails prevent unstable per-task frequency switching and force recovery to high frequency when performance risk appears.

The experimental results support the value of this layered design. With DVFS disabled, the bandit scheduler improves geometric-mean latency by 9.99% across the five reported benchmarks. The behavior is visible in the locality-attribution experiment: the largest improvements coincide with large reductions in copy bytes, while the near-neutral workload keeps the same copy footprint and remains close to HEFT.

The final DVFS evaluation shows a smaller but meaningful energy result. WinDVFS reduces geometric-mean energy by 5.32% with a geometric-mean latency overhead of 0.48% under the fixed Bandit placement layer. The geometric-mean EDP ratio improves to 0.951, and all five reported benchmarks remain below an EDP ratio of 1.0. The observed energy reductions are consistent with lower average power in the selected DVFS configurations, but the result should be interpreted as an energy–performance measurement rather than as a full clock-residency or full-system power study.

The main conclusion is therefore that CUDASTF task-graph information can be used

to make online scheduling both more locality aware and more energy aware, provided that the scheduler remains anchored to a strong performance baseline. The placement contribution is the clearer and stronger result: bounded HEFT-relative deviations can reduce unnecessary data movement and improve runtime on copy-sensitive workloads. The DVFS contribution is more modest but still useful: guarded window-level frequency control can reduce energy over the benchmark execution window while keeping aggregate latency overhead small. The end-to-end result combines these effects: Bandit+WinDVFS reduces geometric-mean latency by 8.80%, energy by 11.87%, and EDP by 19.63% relative to HEFT.

The work also identifies the limits of the current implementation. The results are single-node measurements, the benchmark suite is limited, the criticality and copy-cost models are lightweight approximations, and the DVFS evaluation focuses on benchmark-window energy rather than full-system energy. The static-replay overhead diagnostic further shows that online Bandit scheduling cost is small at benchmark scale, but it is still a diagnostic approximation rather than a full production-overhead study. These limitations do not invalidate the main findings, but they define the next steps required before the scheduler can be treated as a general-purpose production policy. Overall, the thesis demonstrates that a carefully layered runtime scheduler can improve locality and expose energy-saving opportunities without abandoning the stability and interpretability of HEFT-based scheduling.

Bibliography

[1] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey, "Recalibrating global data center energy-use estimates," *Science*, vol. 367, no. 6481, pp. 984–986, 2020. DOI: 10.1126/science.aba3758.

[2] Xinxin Mei, Qiang Wang, and Xiaowen Chu, "A survey and measurement study of GPU DVFS on energy conservation," *Digital Communications and Networks*, vol. 3, no. 2, pp. 89–100, 2017. DOI: 10.1016/j.dcan.2016.10.001.

[3] Ghazanfar Ali, Sridutt Bhalachandra, Nicholas J. Wright, Mert Side, and Yong Chen, "Optimal GPU frequency selection using multi-objective approaches for HPC systems," in *Proceedings of the 2022 IEEE High Performance Extreme Computing Conference*, ser. HPEC, IEEE, 2022, pp. 1–7. DOI: 10.1109/HPEC55821.2022.9926317.

[4] Ghazanfar Ali, Mert Side, Sridutt Bhalachandra, Nicholas J. Wright, and Yong Chen, "An automated and portable method for selecting an optimal GPU frequency," *Future Generation Computer Systems*, vol. 149, pp. 71–88, 2023. DOI: 10.1016/j.future.2023.07.011.

[5] Qiang Wang, Laiyi Li, Weile Luo, Yijia Zhang, and Bingqiang Wang, "DSO: A GPU energy efficiency optimizer by fusing dynamic and static information," in *Proceedings of the 2024 IEEE/ACM 32nd International Symposium on Quality of Service*, ser. IWQoS, IEEE, 2024, pp. 1–6. DOI: 10.1109/IWQOS61813.2024.10682917.

[6] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás, "GPGPU power modeling for multi-domain voltage-frequency scaling," in *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA, IEEE, 2018, pp. 789–800. DOI: 10.1109/HPCA.2018.00072.

[7] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás, "Modeling and decoupling the GPU power consumption for cross-domain DVFS," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 11, pp. 2494–2506, 2019. DOI: 10.1109/TPDS.2019.2917181.

[8] Qiang Wang and Xiaowen Chu, "GPGPU performance estimation with core and memory frequency scaling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 12, pp. 2865–2881, 2020. DOI: 10.1109/TPDS.2020.3004623.

[9] Kaijie Fan, Biagio Cosenza, and Ben Juurlink, "Predictable GPUs frequency scaling for energy and performance," in *Proceedings of the 48th International*

Conference on Parallel Processing, ser. ICPP, ACM, 2019, pp. 1–10. DOI: 10.1145/3337821.3337833.

[10] Srikant Bharadwaj, Shomit Das, Kaushik Mazumdar, Bradford M. Beckmann, and Stephen Kosonocky, "Predict; do not 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*, ser. ASPLOS, ACM, 2023, pp. 253–267. DOI: 10.1145/3623278.3624756.

[11] Kaijie Fan, Marco D'Antonio, Lorenzo Carpentieri, Biagio Cosenza, Federico Ficarelli, and Daniele Cesarini, "SYnergy: Fine-grained energy-efficient heterogeneous computing for scalable energy saving," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC, ACM, 2023, pp. 1–13. DOI: 10.1145/3581784.3607055.

[12] Lorenzo Carpentieri, Antonio De Caro, Majid Salimi Beni, Kaijie Fan, and Biagio Cosenza, "Phase-based frequency scaling for energy-efficient heterogeneous computing," in *Proceedings of the 2025 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS, IEEE, 2025, pp. 824–836. DOI: 10.1109/IPDPS64566.2025.00078.

[13] Hadi Zamani, Laxmi Bhuyan, Jieyang Chen, and Zizhong Chen, "GreenMD: Energy-efficient matrix decomposition on heterogeneous multi-GPU systems," *ACM Transactions on Parallel Computing*, vol. 10, no. 2, pp. 1–23, 2023. DOI: 10.1145/3583590.

[14] Chao Chen, Chris Porter, and Santosh Pande, "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*, ser. PPoPP, ACM, 2022, pp. 17–31. DOI: 10.1145/3503221.3508423.

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

[16] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011. DOI: 10.1002/cpe.1631.

[17] Cédric Augonnet, Andrei Alexandrescu, Albert Sidelnik, and Michael Garland, "CUDASTF: Bridging the gap between CUDA and task parallelism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC, IEEE/ACM, 2024. DOI: 10.1109/SC41406.2024.00049.

[18] Haluk Topcuoglu, Salim Hariri, and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002. DOI: 10.1109/71.993206.

[19] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *Proceedings*

of the 19th International Conference on World Wide Web, ser. WWW, ACM, 2010, pp. 661–670. DOI: 10.1145/1772690.1772758.

[20] Seokwoo Song, Minseok Lee, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu, "Energy-efficient scheduling for memory-intensive GPGPU workloads," in *Proceedings of the 2014 Design, Automation and Test in Europe Conference and Exhibition*, ser. DATE, IEEE, 2014, pp. 1–6. DOI: 10.7873/DATE.2014.032.

[21] Yidi Wang, Mohsen Karimi, Yecheng Xiang, and Hyoseung Kim, "Balancing energy efficiency and real-time performance in GPU scheduling," in *Proceedings of the 2021 IEEE Real-Time Systems Symposium*, ser. RTSS, IEEE, 2021, pp. 110–122. DOI: 10.1109/RTSS52674.2021.00021.

[22] Qiang Wang, Xinxin Mei, Hai Liu, Yiu-Wing Leung, Zongpeng Li, and Xiaowen Chu, "Energy-aware non-preemptive task scheduling with deadline constraint in DVFS-enabled heterogeneous clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4083–4099, 2022. DOI: 10.1109/TPDS.2022.3181096.

[23] Zhishan Guo, Ashikahmed Bhuiyan, Di Liu, Aamir Khan, Abusayeed Saifullah, and Nan Guan, "Energy-efficient real-time scheduling of DAGs on clustered multi-core platforms," in *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS, IEEE, 2019, pp. 156–168. DOI: 10.1109/RTAS.2019.00021.

[24] Yanhui Huang, Bing Guo, and Yan Shen, "GPU energy optimization based on task balance scheduling," *Journal of Systems Architecture*, vol. 107, p. 101 808, 2020. DOI: 10.1016/j.sysarc.2020.101808.

[25] Albert d'Aviau de Piolant, Hayfa Tayeb, Bérenger Bramas, Mathieu Faverge, Abdou Guermouche, and Amina Guermouche, "Improving energy efficiency of HPC applications using unbalanced GPU power capping," in *Proceedings of the 2025 IEEE International Parallel and Distributed Processing Symposium Workshops*, ser. IPDPSW, IEEE, 2025, pp. 820–829. DOI: 10.1109/IPDPSW66978.2025.00132.

[26] Robert Chab, Fei Li, and Sanjeev Setia, "Algorithmic techniques for GPU scheduling: A comprehensive survey," *Algorithms*, vol. 18, no. 7, p. 385, 2025. DOI: 10.3390/a18070385.

[27] Jing Chen, Madhavan Manivannan, Bhavishya Goel, and Miquel Pericàs, "JOSS: Joint exploration of CPU-memory DVFS and task scheduling for energy efficiency," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP, ACM, 2023, pp. 828–838. DOI: 10.1145/3605573.3605586.

[28] Jing Chen, Madhavan Manivannan, Bhavishya Goel, and Miquel Pericàs, "SWEEP: Adaptive task scheduling for exploring energy performance tradeoffs," in *Proceedings of the 2024 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS, IEEE, 2024, pp. 325–336. DOI: 10.1109/IPDPS57955.2024.00036.

A

Supplementary Result Tables

This appendix reports the absolute measurements behind the compact delta tables in Chapter 4. Values are rounded to one decimal place in the same way as the corresponding Results tables.

	Latency (ms)		Energy (J)		EDP (J·s)	
Benchmark	Base	WinDVFS	Base	WinDVFS	Base	WinDVFS
FDTD	6398.7	6407.0	3182.1	2984.6	20361.3	19122.0
CG	12432.0	12373.5	3077.9	2766.9	38264.8	34235.7
MiniWeather	15126.5	15119.0	4957.8	4875.7	74994.3	73715.7
LU decomposition	15544.7	15777.5	4044.1	3842.6	62863.9	60626.8
Cholesky	18562.7	18804.6	5327.8	5145.5	98898.3	96759.7

Table A.1: Absolute measurements for the WinDVFS comparison in Table 4.8. Base denotes the no-DVFS baseline under the same placement layer.

	Latency (ms)		Energy (J)		EDP (J·s)	
Benchmark	HEFT	B+DVFS	HEFT	B+DVFS	HEFT	B+DVFS
FDTD	6784.0	6407.0	3256.7	2984.6	22093.3	19122.0
CG	14301.0	12373.5	3464.1	2766.9	49540.8	34235.7
MiniWeather	15111.6	15119.0	5080.7	4875.7	76778.0	73715.7
LU decomposition	18591.6	15777.5	4825.7	3842.6	89716.8	60626.8
Cholesky	20672.9	18804.6	5415.0	5145.5	111943.7	96759.7

Table A.2: Absolute measurements for the end-to-end comparison in Table 4.10. B+DVFS denotes the bandit scheduler with WinDVFS enabled.