SMARTBCD: SMART BLOCK COORDINATE DESCENT VIA COST-AWARE SCHEDULING

Anonymous authors
Paper under double-blind review

ABSTRACT

Full-parameter fine-tuning (FFT) of large language models requires substantial GPU memory. Block coordinate descent (BCD)-based methods address this challenge through block-wise parameter updates rather than simultaneous updates to all parameters, enabling more memory-efficient FFT. However, existing BCD-based methods either follow predetermined or randomized block schedules, or prioritize blocks primarily based on optimization signals, without explicitly accounting for the joint variation in optimization gain and computation cost across training stages. This can lead to suboptimal use of limited training resources. We propose SmartBCD, a cost-aware block scheduling method that improves resource utilization. SmartBCD estimates each block's utility as the predicted loss reduction per unit of training time and uses this estimate to schedule updates as training progresses. A maximum-wait constraint ensures that every block continues to receive updates. We conduct experiments with TinyLlama-1.1B and Qwen2.5-7B on instruction-following and mathematical reasoning tasks. On our Qwen2.5-7B MathInstruct setting, under the same optimization-step budget, SmartBCD reduces end-to-end training time by 4.4% and final validation loss by 1.3% relative to BAdam-Random, without negligible additional GPU-memory overhead.

1 INTRODUCTION

Large language models (LLMs) have demonstrated strong capabilities across a wide range of downstream tasks, including instruction following, reasoning, and domain-specific applications. Adapting pretrained LLMs to such tasks typically involves fine-tuning, with full-parameter fine-tuning (FFT) offering the greatest flexibility by updating all model parameters on the target data. However, this flexibility comes at a substantial computational cost. In practice, widely used optimizers such as Adam (Kingma & Ba, 2015) require maintaining gradients and optimizer states for the entire model, leading to considerable GPU memory footprint and traning computation which limit the applicability of FFT in resource-constrained environments. These requirements have motivated extensive research on efficient fine-tuning, ranging from parameter-efficient methods such as LoRA (Hu et al., 2022) to memory-efficient full-parameter optimization methods such as Block Coordinate Descent (BCD) (Tseng, 2001). While parameter-efficient methods reduce resource requirements by restricting the trainable parameter space, memory-efficient FFT seeks to retain the ability to optimize the full model and reduce the resources required at each training step.

BCD provides an effective paradigm for memory-efficient full-parameter optimization: instead of updating all parameters simultaneously, it partitions the model into parameter blocks and updates only a subset of them at a time. For example, BAdam (Luo et al., 2024) applies Adam updates to one active block while keeping the remaining blocks fixed, substantially reducing the memory required for gradients and optimizer states. Meanwhile, Cyclic BCD (Li et al., 2016) and BCSC (Nakamura et al., 2021) update parameter blocks in a predetermined cyclic order, sequentially optimizing one block at a time. These BCD-based methods commonly adopt predetermined or randomized schedules to update blocks. More recent studies have demonstrated that the choice of active blocks can affect training efficiency. For example, Pan et al. (Pan et al., 2024) introduce a layer-wise importance sampling strategy and Ramesh et al. (Ramesh et al., 2024) employ optimization-related signals for block scheduling . These studies suggest that treating all blocks identically is unnecessary and that block selection can play an important role in memory-efficient LLM training. However, existing scheduling strategies mainly focus on which blocks are important to update, while

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

largely overlooking the computation cost required to achieve the corresponding optimization gain. This distinction is particularly relevant for Transformer models, where parameter blocks can exhibit heterogeneous optimization behavior and incur significant difference in computation costs during block-wise training.

This motivates us to reconsider block scheduling from the perspective of *training resource utilization*, which means the trade-offs between the optimization gain and computation cost. During BCD training, different blocks contribute differently to reducing the training objective, and updating them incur different computation costs. Consequently, a block that produces a large loss reduction may not be a resource-efficient choice. Fixed or random schedules cannot capture such differences, while importance-only selection does not explicitly account for associated computation costs. This raises an essential question: Can we dynamically schedule parameter blocks according to the optimization gain obtained per unit of training cost, thereby making more effective use of limited training resources throughout fine-tuning?

To address this question, we propose **SmartBCD**, a cost-aware block scheduling method for memory-efficient full-parameter fine-tuning. Our key observation is that trade-offs between the optimization gain and computation cost of each block can change over the course of training. SmartBCD therefore casts block scheduling in terms of a *utility* metric, defined as the predicted loss reduction from updating a block per unit of expected training time. During training, SmartBCD estimates the optimization gain and update cost of individual blocks, computes their time-normalized utilities, and prioritizes blocks expected to provide greater loss reduction per unit of training time. After a selected block is updated, the scheduling information is progressively refreshed to reflect the evolving optimization state. Since purely utility-driven scheduling may favor a subset of blocks, we additionally introduce a maximum-wait constraint that prevents any block from remaining inactive for an excessive period. In this way, SmartBCD combines adaptive resource allocation with continued coverage of the full parameter.

We evaluate SmartBCD on TinyLlama-1.1B and Qwen2.5-7B across instruction-following and mathematical reasoning tasks. On Qwen2.5-7B MathInstruct setting, under the same optimization-step budget, SmartBCD reduces end-to-end training time by **4.4%** and final validation loss by **1.3%** relative to BAdam-Random, demonstrating that cost-aware scheduling can improve both computational and optimization efficiency. Meanwhile, peak reserved GPU memory differs by only **0.07%**, indicating that the benefits of SmartBCD are not obtained at additional memory resources. Together, these results suggest that cost-aware block scheduling can further improve the computational efficiency of BCD-based LLM fine-tuning by allocating updates according to both optimization gain and training cost.

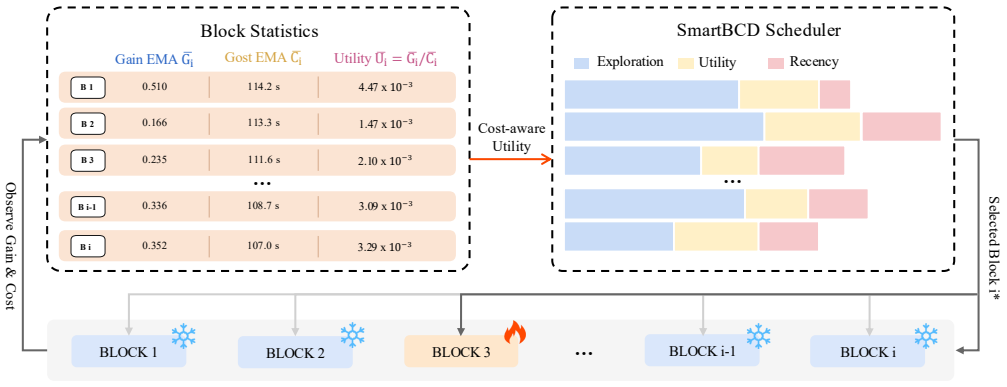

Figure 1: SmartBCD overview.

2 Related Work

Memory-efficient full-parameter fine-tuning. The substantial memory cost of full-parameter fine-tuning has motivated methods that reduce training memory while preserving updates to the orig-

inal model parameters. LOMO (Lv et al., 2024) reduces memory consumption by fusing gradient computation and parameter updates, avoiding the need to retain full gradients during optimization. GaLore (Zhao et al., 2024) instead projects gradients into a low-rank subspace, substantially reducing the memory required by optimizer states while retaining full-parameter learning. Another line of work reduces the number of parameters that are active at each step. BAdam (Luo et al., 2024) applies Adam within a block coordinate descent (BCD) framework, updating one parameter block at a time and thereby reducing the memory required for gradients and optimizer states. Our work builds on this block-wise formulation and focuses on improving how the limited computation available during BCD training is allocated across parameter blocks.

Block-wise and selective optimization. The choice of which parameters to update has long been recognized as an important component of coordinate optimization, where cyclic, randomized, and greedy selection rules can lead to different optimization behavior (Nutini et al., 2022). Recent LLM fine-tuning methods have further explored non-uniform parameter selection. LISA (Pan et al., 2024) exploits layer-wise heterogeneity through importance sampling and activates only a small subset of layers during training. More closely related to our work, BlockLLM (Ramesh et al., 2024) dynamically selects blocks using gradient-based importance and adjusts the selected set according to optimization progress, demonstrating that adaptive block selection can improve memory-efficient LLM training. However, these methods primarily determine block selection from optimization-related signals, without explicitly incorporating the block-specific computation required to obtain the corresponding training benefit. In contrast, SmartBCD considers optimization gain and computation cost jointly: it schedules blocks according to their predicted loss reduction per unit of training time and introduces a maximum-wait constraint to maintain updates across all blocks.

3 SMARTBCD: COST-AWARE BLOCK SCHEDULING

In this section, we present SmartBCD, a cost-aware scheduler for block-wise full-parameter optimization. The key idea is to use previously executed updates to estimate how much optimization progress each block provides relative to its training cost, and to allocate future block updates accordingly. Unlike a fixed block order, SmartBCD continuously adapts to changes in block utility during training while ensuring that no block is ignored for an excessively long period.

3.1 PRELIMINARIES: BLOCK-WISE OPTIMIZATION

We consider a model whose parameters θ are partitioned into D disjoint blocks $\{\theta_i\}_{i=1}^{D}$ and optimized with respect to a training objective $F(\theta)$. Following BAdam (Luo et al., 2024), training proceeds in *block epochs*. At block epoch t, a single block i_t is activated while all other blocks are frozen, and the selected block is optimized for K consecutive optimizer steps before a new block is chosen. The final block epoch may contain fewer than K steps.

A fixed or cyclic schedule implicitly allocates the same update opportunities to all blocks. However, blocks can differ in both the optimization progress they produce and the computation required to update them, and these differences may change over the course of training. SmartBCD therefore treats block selection as a dynamic resource-allocation problem: blocks that currently provide more optimization progress per unit training time should receive more update opportunities.

3.2 COST-AWARE NON-STATIONARY BLOCK SCHEDULING

Estimating block utility. Evaluating all blocks before every scheduling decision would introduce additional forward and backward passes. Instead, SmartBCD estimates utility only from updates that have already been executed.

For the k-th optimizer step of block i in block epoch t, let $g_{i,t,k}$ denote the gradient and $\Delta\theta_{i,t,k}$ the parameter change produced by the optimizer. A first-order approximation gives a predicted decrease of $-g_{i,t,k}^{\top}\Delta\theta_{i,t,k}$. We accumulate this quantity over the block epoch and define

$$G_{i,t} = -\sum_{k=1}^{K_t} g_{i,t,k}^{\top}\Delta\theta_{i,t,k}, \tag{1}$$

where $K_t \leq K$ for an incomplete final block epoch. Unlike the gradient norm alone, $G_{i,t}$ accounts for both the local gradient and the parameter displacement actually taken by the optimizer. It should be interpreted as a first-order proxy for training-objective improvement rather than an observed decrease in validation loss.

Let $C_{i,t}$ denote the measured training time of the same block epoch. Because both optimization gain and training cost may vary during training, we maintain exponential moving averages $\bar{G}_i$ and $\bar{C}_i$ for each block, updated only when that block is selected. The cost-aware utility is then

$$U_i = \frac{\bar{G}_i}{\max(\bar{C}_i, \epsilon)}. \tag{2}$$

Thus, U_i estimates the recent predicted loss reduction obtained per unit of training time. Maintaining separate moving averages for gain and cost makes the estimate less sensitive to fluctuations in a single block epoch than directly averaging their ratio. Before adaptive scheduling begins, SmartBCD visits every block once in a deterministic warm-up phase to initialize these statistics.

Why normalize optimization gain by computation cost? Using predicted gain alone implicitly treats all block updates as having the same computational price. This assumption is undesirable in block-wise fine-tuning, where two blocks with similar predicted optimization gain may require different amounts of training time. In this case, repeatedly selecting the higher-gain block does not necessarily yield the largest optimization progress under a fixed computation cost. SmartBCD therefore treats block scheduling as a resource-allocation problem rather than an importance-ranking problem: the scheduler favors blocks according to the optimization progress expected from each unit of computation, rather than according to gain or cost in isolation. Importantly, cost normalization does not imply selecting the cheapest blocks. A relatively expensive block can still receive high priority whenever its predicted gain compensates for the additional computation.

Adaptive scheduling under non-stationarity. After warm-up, SmartBCD favors high-utility blocks while retaining exploration of blocks whose estimates may have become stale. Since the scale of U_i can change during training, we first standardize utilities across the current set of blocks:

$$Z_i = \frac{U_i - \mu_U}{\max(\sigma_U, \epsilon)}, \tag{3}$$

where μ_U and σ_U are the mean and standard deviation of the current block utilities.

A purely greedy rule that repeatedly selects the block with the largest current utility is insufficient in this setting. The utility estimates are inherently asynchronous: only the statistics of an executed block can be refreshed, while estimates associated with inactive blocks become progressively stale. Consequently, repeatedly selecting the currently best block can create a self-reinforcing scheduling pattern. Frequently selected blocks receive fresh measurements and can continue to dominate the ranking, whereas temporarily underestimated blocks may receive few opportunities to reveal that their utility has increased. This issue is particularly relevant because block utility is non-stationary during fine-tuning: as different parts of the model adapt, both the attainable optimization gain and the relative value of updating individual blocks can change over time.

To balance exploitation with continued re-evaluation, SmartBCD maintains two additional scheduling states. First, we maintain a discounted selection count $\widetilde{N}_i$. After each block epoch, all historical counts are discounted by a factor $\rho \in (0, 1)$ and the count of the selected block is incremented by one. Thus, selections made far in the past contribute progressively less to the current estimate. Second, let $a_i(t)$ denote the number of block epochs since block i was last selected. We introduce a maximum-wait horizon H, which specifies the longest interval for which a block is allowed to remain inactive. The quantity $a_i(t)/H$ therefore provides a normalized measure of how close a block is to this coverage limit.

When no block has reached the maximum-wait horizon, SmartBCD assigns each block the score

$$S_i(t) = Z_i + c\sqrt{\frac{\log(\max(t+1, 2))}{\max(\widetilde{N}_i, \epsilon)}} + \lambda \frac{a_i(t)}{H}. \tag{4}$$

The three terms serve complementary purposes. The first exploits blocks with high recent optimization return per unit time. The second provides a discounted UCB-style exploration bonus to blocks

with fewer recent selections, while the third gradually increases the priority of blocks that have not been updated for longer. Together, these terms allow the scheduler to favor currently efficient blocks while periodically reconsidering blocks whose utility estimates may have become stale.

Maximum-wait coverage. The adaptive score encourages exploration but does not by itself guarantee that every block will be revisited within a finite interval. Therefore, before applying Eq. 4, SmartBCD checks whether any block satisfies $a_i(t) \geq H$. If so, the block with the largest age is selected directly. Otherwise, the scheduler selects the block with the largest $S_i(t)$.

This hard coverage constraint prevents block starvation without enforcing uniform update frequencies. As a result, SmartBCD allocates most of the training budget according to recent cost-aware utility while retaining continual coverage of the full parameter space.

Overall scheduling loop. SmartBCD therefore forms a closed loop between block optimization and resource allocation. After the initial one-pass warm-up, each block epoch begins with the statistics collected from previously executed updates. The scheduler first checks the maximum-wait constraint and, when no forced selection is required, ranks the blocks using the combination of standardized cost-aware utility, discounted exploration, and recency. The selected block is then activated for K optimizer steps while all remaining blocks stay frozen. Its resulting predicted gain and measured training cost are incorporated into the corresponding moving averages, after which the scheduler makes the next decision using the refreshed state. Crucially, SmartBCD never evaluates inactive blocks solely for scheduling purposes; all scheduling signals are obtained from optimization work that would already be performed during block-wise training.

4 Experiments

4.1 Experimental Setup

We evaluate SmartBCD on TinyLlama-1.1B and Qwen2.5-7B using MathInstruct and Alpaca-GPT4 for mathematical reasoning and instruction-following tasks. For Qwen2.5-7B, we partition the model into $D = 28$ blocks with a one-pass warm-up before adaptive scheduling. We compare against BAdam-Random, BAdam-Cyclic, Gain-only, Adam-mini, LoRA, and GaLore, with identical training configurations and optimization budgets for all block-wise methods. We report final validation loss and end-to-end wall-clock training time; the 896-step controlled experiments and ablations use three random seeds and $K = 2$, while full-scale experiments use $K = 100$. All experiments were conducted on a server with two NVIDIA A100-SXM4 80GB GPUs. Each run used a single GPU; the two GPUs were used only for independent concurrent runs, not distributed training. Additional implementation details and hyperparameters are provided in the Appendix D.

4.2 Does SmartBCD Select Better Blocks?

A central premise of SmartBCD is that the value of updating a parameter block is neither uniform across blocks nor stationary throughout training. We therefore examine whether block utilities exhibit meaningful temporal structure and whether adaptive selection translates into better realized optimization efficiency.

Dynamic Block Utilities. Figure 2-left shows the EMA-smoothed predicted optimization gain $\bar{G}_i$, measured training cost $\bar{C}_i$, and the resulting cost-aware utility U_i for all 28 parameter blocks throughout training. The dashed line marks the end of the one-pass warm-up. Both predicted gain and update cost vary substantially across blocks, and the resulting utility landscape continues to evolve after warm-up. In particular, the relative utility ordering is not fixed throughout training, supporting the need for an adaptive rather than predetermined scheduling strategy. SmartBCD accordingly reallocates updates as these estimates evolve while retaining full block coverage through its exploration and maximum-wait mechanisms.

Realized Optimization Efficiency. To determine whether these adaptive selections are beneficial beyond SmartBCD's internal utility estimates, we evaluate their realized optimization outcomes using the controlled 896-step experiments with three random seeds. For method m, seed s, and

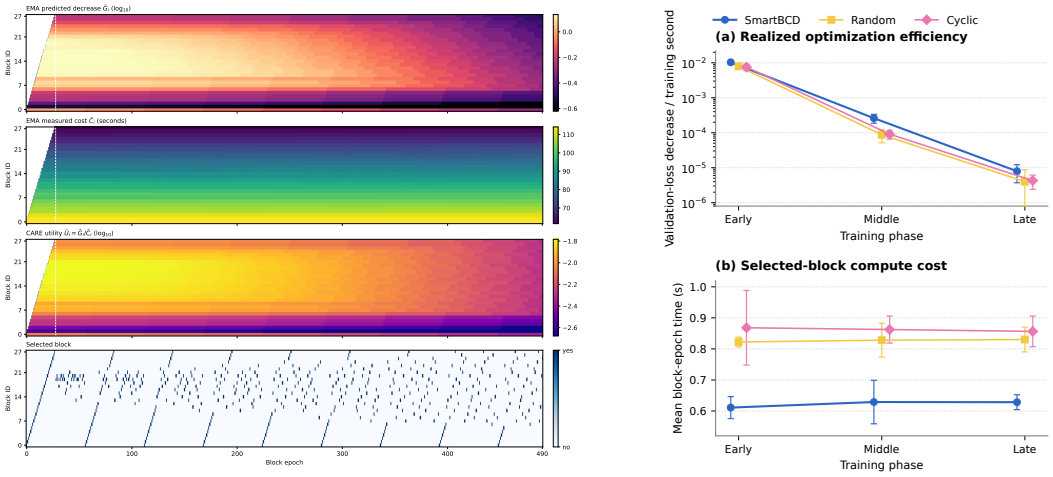

Figure 2: Block utility dynamics and selection outcomes. Left: Evolution of predicted gain, measured cost, cost-aware utility, and block selections. Right: (a) Realized optimization efficiency and (b) selected-block computation cost across training phases. Mean and 95% t-CI over three seeds.

phase $p \in \{\text{Early}, \text{Middle}, \text{Late}\}$, we define

$$E_{m,s,p} = \frac{L_{m,s}(a_p) - L_{m,s}(b_p)}{T_{m,s}(b_p) - T_{m,s}(a_p)}, \tag{5}$$

where a_p and b_p denote the validation points at the beginning and end of phase p, respectively, and T denotes cumulative CUDA-synchronized training time. Thus, $E_{m,s,p}$ measures the observed validation-loss decrease per second of training computation.

As shown in Figure 2-right(a), SmartBCD achieves the highest mean realized efficiency among SmartBCD, Random, and Cyclic in all three phases, with the clearest separation in the middle stage. Although optimization efficiency decreases for all methods as training progresses, SmartBCD maintains a consistent mean advantage.Figure 2-right(b) further shows that SmartBCD selects blocks with substantially lower average block-epoch cost than Random and Cyclic. Since selection is based on predicted gain relative to cost rather than cost alone, this indicates that SmartBCD reallocates computation toward blocks offering a more favorable optimization return per unit time.

Analysis and Takeaways. These results show that effective block scheduling requires adapting to the evolving value of different parameter blocks during training. SmartBCD addresses this by dynamically reallocating the training budget toward more valuable updates. We next examine whether explicitly accounting for block-specific computation cost is necessary for achieving this benefit.

4.3 Does Cost-Aware Scheduling Matter?

Section 4.2 shows that SmartBCD adapts its block selection to the evolving optimization state. We next isolate whether explicitly accounting for block-specific computation cost is necessary for this advantage. To this end, we compare the full SmartBCD scheduler with a *Gain-only* variant that removes cost normalization and uses $\bar{G}_i$ as the selection utility, while keeping the remaining scheduling mechanism unchanged.

Optimization under Step and Time Budgets. Figure 3-left compares the validation-loss trajectories of SmartBCD and Gain-only under both optimizer-step and wall-clock views. When measured against optimizer steps, the two methods follow similar overall optimization trajectories, with SmartBCD generally attaining lower validation loss at matched checkpoints. This indicates that incorporating computation cost does not improve efficiency merely by sacrificing optimization progress.

The difference becomes more meaningful when training progress is measured against wall-clock time. SmartBCD reaches comparable or lower validation loss with less elapsed training time

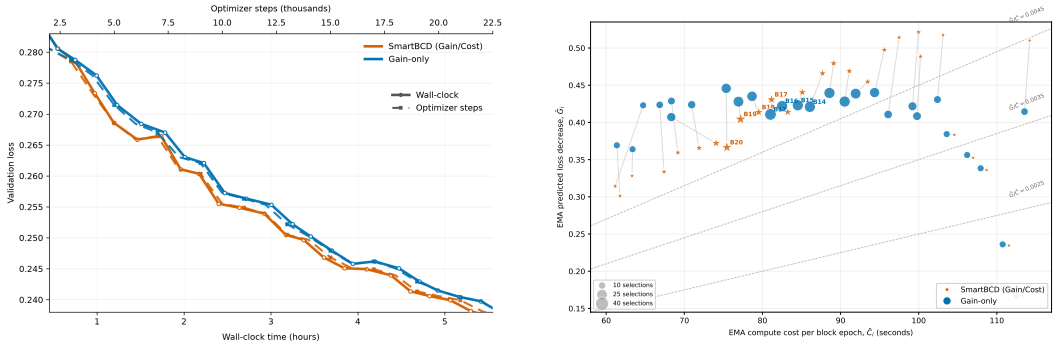

Figure 3: Effect of cost-aware block scheduling. Left: Validation-loss trajectories of SmartBCD and the Gain-only variant under wall-clock time and optimizer-step views. Right: Gain–cost distribution of selected blocks. Each block is characterized by its EMA-smoothed predicted gain $\bar{G}_i$ and computation cost $\bar{C}_i$; marker size indicates selection frequency, and dashed diagonal lines denote constant $\bar{G}_i/\bar{C}_i$ ratios.

Table 1: Final validation loss and end-to-end training time across models and datasets.

Model	**Dataset**	**Val. Loss** ↓			**Time (h)** ↓		
		Random	Cyclic	SmartBCD	Random	Cyclic	SmartBCD
Qwen2.5-7B	MathInstruct	0.2305	0.2289	**0.2275**	12.458	12.090	**11.908**
	Alpaca-GPT4	0.8781	0.8774	**0.8767**	2.265	2.292	**2.119**
TinyLlama-1.1B	MathInstruct	0.6108	0.6125	**0.6074**	3.63	3.713	**3.403**
	Alpaca-GPT4	1.0592	1.0596	**1.0584**	0.734	0.746	**0.671**

throughout most of the trajectory. Since the two variants differ only in whether block cost is incorporated into the selection utility, this result suggests that cost-aware scheduling improves how the available computation cost is converted into optimization progress. The plotted points correspond directly to observed validation checkpoints; no interpolation is applied.

How Does Cost Awareness Change Block Selection? Figure 3-right compares the two schedulers in the gain–cost plane, where marker size denotes selection frequency and diagonal lines indicate constant gain-to-cost ratios. The substantial overlap shows that SmartBCD does not simply favor cheaper blocks; instead, cost normalization induces a moderate redistribution toward more efficient updates. Relative to Gain-only, SmartBCD reduces mean block-epoch cost by 1.9% while increasing the average gain-to-cost ratio by 1.2%.

Analysis and Takeaways. Cost awareness therefore acts as a resource-allocation correction rather than a fundamentally different block-selection strategy. Gain remains the primary optimization signal, while block-specific cost helps distinguish between updates with similar optimization value but different computational demands. Even modest shifts in this allocation accumulate over training, leading to a more favorable optimization trajectory under a fixed wall-clock budget.

4.4 Main Results

We next evaluate whether the scheduling advantages identified in the controlled analyses translate into improved end-to-end fine-tuning efficiency. We consider three complementary perspectives. First, we compare final optimization quality and total wall-clock time across models and datasets under the same prescribed optimization budget. Second, because SmartBCD explicitly optimizes predicted progress per unit of training time, we evaluate all block-wise schedulers under a common wall-clock budget. Finally, we compare SmartBCD with broader efficient fine-tuning baselines and examine whether its efficiency gains come at the expense of downstream task performance.

End-to-End Optimization Efficiency. Table 1 shows that SmartBCD consistently achieves the lowest final validation loss and shortest training time across both model scales and datasets. On

Table 2: Fixed-budget validation loss.

Model	Dataset	Budget (h)	Random ↓	Cyclic ↓	SmartBCD ↓
Qwen2.5-7B	Math	11.908	0.2305	0.2288	**0.2274**
	Alpaca	2.119	0.8771	0.8765	**0.8753**
TinyLlama-1.1B	Math	3.403	0.6107	0.6125	**0.6074**
	Alpaca	0.671	1.0592	1.0596	**1.0585**

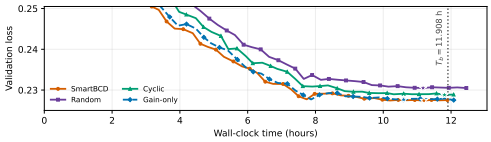

Figure 4: Validation loss versus wall-clock time on Qwen2.5-7B with MathInstruct.

Qwen2.5-7B with MathInstruct, SmartBCD reduces training time from 12.458/12.090 hours (Random/Cyclic) to 11.908 hours, while improving validation loss from 0.2305/0.2289 to 0.2275. The same trend holds on Alpaca-GPT4 and TinyLlama-1.1B. Across all four settings, SmartBCD reduces wall-clock time by 4.4–8.6% relative to Random and 1.5–10.1% relative to Cyclic, while consistently attaining lower validation loss. These results indicate that cost-aware scheduling improves computation allocation across models and tasks without requiring additional optimization steps. The wall-clock gains incur negligible GPU-memory overhead, with peak reserved memory differing by only 0.07% from BAdam baselines (Appendix E.1).

Optimization under a Fixed Wall-Clock Budget. The previous comparison evaluates end-to-end optimization efficiency by considering both final validation loss and total wall-clock time. We further examine the complementary question: given the same amount of wall-clock time, how much optimization progress can each scheduler achieve? For each model–dataset setting, we use the shortest complete training time among the compared block-wise schedulers as a shared wall-clock budget and report the best validation loss achieved within this budget.

As shown in Table 2, SmartBCD obtains the lowest validation loss in all four settings. On Qwen2.5-7B with MathInstruct, for example, SmartBCD reaches a validation loss of 0.2274 within 11.908 hours, compared with 0.2305 for Random and 0.2288 for Cyclic. The same trend is observed on Alpaca-GPT4 and TinyLlama-1.1B, indicating that the advantage persists across both model scales and task domains. These results show that SmartBCD's shorter training time is accompanied by greater optimization progress under a fixed wall-clock budget.

Figure 4 provides a finer-grained view on Qwen2.5-7B with MathInstruct. SmartBCD maintains a favorable loss–time trajectory relative to Random and Cyclic throughout training, while Gain-only remains relatively close. This is consistent with gain being the primary optimization signal, with cost normalization providing an additional resource-allocation correction. Together with the gain–cost analysis in Section 4.3, these results suggest that SmartBCD improves how training computation is allocated rather than merely accelerating an otherwise equivalent optimization trajectory.

Comparison with Efficient Fine-Tuning Baselines and Downstream Performance. Table 3 further compares SmartBCD with broader efficient fine-tuning baselines on Qwen2.5-7B with MathInstruct. SmartBCD completes training in 11.908 hours, substantially faster than Adam-mini, LoRA, and GaLore, which require 30.143, 28.434, and 46.200 hours, respectively. Adam-mini attains a lower final validation loss of 0.2217, but requires more than twice the wall-clock training time. This comparison highlights the objective of SmartBCD: rather than minimizing terminal loss in isolation, it improves optimization progress per unit of training time.

The wall-clock advantage also does not lead to a systematic degradation in downstream quality. Across seven mathematical reasoning benchmarks, SmartBCD achieves an average score of 41.40, compared with 40.12 for BAdam-Random and 41.16 for BAdam-Cyclic, while remaining close to Gain-only at 41.75. Together, these results show that the efficiency gains are not obtained at the expense of downstream performance.

Overall, the results across optimization efficiency, fixed-budget optimization, and downstream evaluation suggest that gain-cost scheduling improves how computation is allocated without being tied to a particular model, dataset, or training configuration.

4.5 Ablation of Adaptive Scheduling

We ablate the key components of SmartBCD on Qwen2.5-7B, keeping all other training configurations fixed and reporting results over three random seeds. Removing EMA yields slightly lower

Table 3: Main results on Qwen2.5-7B fine-tuned on the full MathInstruct dataset.

	Optimization Efficiency		Downstream Performance							
Method	**Val. Loss ↓**	**Time (h) ↓**	**GSM8K**	**AQuA**	**SAT**	**MMLU**	**MATH**	**MATH500**	**NumGLUE**	**Avg. ↑**
Adam-mini	**0.2217**	30.143	**71.09**	27.17	26.45	39.49	41.30	40.00	47.12	**41.80**
LoRA	0.2353	28.434	54.44	26.38	24.55	38.21	**42.46**	**41.00**	47.48	39.22
GaLore	0.2504	46.200	63.00	27.17	26.82	47.89	33.12	32.60	50.14	40.11
BAdam-Random	0.2305	12.458	61.56	27.56	23.64	46.71	36.86	38.80	45.72	40.12
BAdam-Cyclic	0.2289	12.090	66.10	**27.95**	**26.82**	42.31	37.86	39.40	**47.67**	41.16
Gain-only	0.2275	12.091	67.95	24.80	26.36	**51.05**	37.66	37.00	47.48	41.75
SmartBCD	0.2275	**11.908**	59.06	**27.95**	25.45	50.59	39.68	40.40	46.70	41.40

Table 4: Ablation of SmartBCD's adaptive scheduling components. Efficiency is validation-loss reduction per unit of CUDA-synchronized training time and is reported in units of 10^{-3}. Max Age is the maximum number of block epochs since any block was last selected, and Min Visits is the minimum number of block epochs assigned to any block.

	Val. Loss ↓			Efficiency ↑			Max Age ↓			Min Visits ↑		
Method	**Seed 1**	**Seed 2**	**Seed 3**	**Seed 1**	**Seed 2**	**Seed 3**	**Seed 1**	**Seed 2**	**Seed 3**	**Seed 1**	**Seed 2**	**Seed 3**
SmartBCD	0.3077	0.3314	0.3064	2.471	**2.733**	**2.701**	56.0	56.0	56.0	8.0	8.0	8.0
w/o EMA	**0.3060**	**0.3273**	**0.3042**	**2.474**	2.603	2.668	56.0	56.0	56.0	**9.0**	**9.0**	**9.0**
w/o Exploration	0.3079	0.3326	0.3069	2.489	2.667	2.605	56.0	56.0	56.0	8.0	8.0	8.0
w/o Recency	0.3091	0.3315	0.3078	2.639	2.562	2.654	56.0	56.0	56.0	8.0	8.0	8.0
w/o Max-wait	0.3142	0.3360	0.3114	2.546	2.348	2.761	83.0	82.0	83.0	6.0	6.0	6.0

terminal validation loss across all seeds, but reduces efficiency in two of the three runs and lowers the average efficiency from 2.635 to 2.582. Thus, EMA primarily stabilizes the gain–cost estimates used for online scheduling rather than directly improving the terminal objective.

Exploration and recency provide complementary signals for block allocation. Removing either component leaves the coverage statistics unchanged, while decreasing average efficiency from 2.635 to 2.587 and 2.618, respectively. Their modest per-run effects suggest that both mechanisms refine allocation among competitive blocks without altering the scheduler's dominant preference.

Maximum-wait, in contrast, is essential for coverage. With $D = 28$ blocks and threshold $H = 2D = 56$, every variant retaining this constraint limits the maximum block age to 56 across all seeds. Removing it increases the maximum age to 82–83 block epochs and reduces the minimum visit count to 6, indicating substantial under-allocation to low-ranked blocks.

Takeaway. EMA, exploration, and recency improve the efficiency and stability of adaptive allocation, whereas maximum-wait provides the explicit safeguard needed to ensure coverage.

5 Conclusion and Discussion

We introduced SmartBCD, a cost-aware scheduler for block-wise FFT of LLMs. SmartBCD treats block selection as a dynamic resource-allocation problem, prioritizing blocks according to their predicted optimization gain per unit of training time. On the Qwen2.5-7B MathInstruct setting, SmartBCD reduces end-to-end training time by 4.4% and final validation loss by 1.3% relative to BAdam-Random under the same optimization-step budget, without negligible additional GPU-memory cost. Our analysis further shows that these improvements arise not from preferentially selecting cheaper blocks, but from allocating computation to blocks that offer a more favorable trade-off between optimization gain and computational cost.

Future work could incorporate additional resource costs beyond training time, adapt block partitioning and update duration and study cost-aware scheduling at larger model scales and in multi-block update settings. These directions may further broaden cost-aware scheduling for efficient FFT.

REFERENCES

Zeyuan Allen-Zhu, Zheng Qu, Peter Richtarik, and Yang Yuan. Even faster accelerated coordinate descent using non-uniform sampling. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 1110–1119. PMLR, 2016.

Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. BitFit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 1–9. Association for Computational Linguistics, 2022. doi: 10.18653/v1/2022.acl-short.1.

Dominik Csiba and Peter Richtárik. Importance sampling for minibatches. *Journal of Machine Learning Research*, 19(27):1–21, 2018.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Olivier Fercoq and Peter Richtárik. Accelerated, parallel, and proximal coordinate descent. *SIAM Journal on Optimization*, 25(4):1997–2023, 2015. doi: 10.1137/130949993.

Demi Guo, Alexander Rush, and Yoon Kim. Parameter-efficient transfer learning with diff pruning. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 4884–4896. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.acl-long.378.

Filip Hanzely and Peter Richtárik. Accelerated coordinate descent with arbitrary sampling and best rates for minibatches. In *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, volume 89 of *Proceedings of Machine Learning Research*, pp. 304–312. PMLR, 2019.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Xingguo Li, Tuo Zhao, Raman Arora, Han Liu, and Mingyi Hong. An improved convergence analysis of cyclic block coordinate descent-type methods for strongly convex minimization. In *Artificial Intelligence and Statistics*, pp. 491–499. PMLR, 2016.

Baohao Liao, Yan Meng, and Christof Monz. Parameter-efficient fine-tuning without introducing new latency. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4242–4260. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.acl-long.233.

Zhaosong Lu and Lin Xiao. On the complexity analysis of randomized block-coordinate descent methods. *Mathematical Programming*, 152(1–2):615–642, 2015. doi: 10.1007/s10107-014-0800-2.

Qijun Luo, Hengxu Yu, and Xiao Li. BAdam: A memory efficient full parameter optimization method for large language models. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8187–8198, 2024.

Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D. Lee, Danqi Chen, and Sanjeev Arora. Fine-tuning language models with just forward passes. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Kensuke Nakamura, Stefano Soatto, and Byung-Woo Hong. Block-cyclic stochastic coordinate descent for deep neural networks. *Neural Networks*, 139:348–357, 2021.

Hongseok Namkoong, Aman Sinha, Steve Yadlowsky, and John C. Duchi. Adaptive sampling probabilities for non-smooth optimization. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 2574–2583. PMLR, 2017.

Yurii Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012. doi: 10.1137/100802001.

Julie Nutini, Mark Schmidt, Issam Laradji, Michael Friedlander, and Hoyt Koepke. Coordinate descent converges faster with the Gauss–Southwell rule than random selection. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 1632–1641. PMLR, 2015.

Julie Nutini, Issam Laradji, and Mark Schmidt. Let's make block coordinate descent converge faster: Faster greedy rules, message-passing, active-set complexity, and superlinear convergence. *Journal of Machine Learning Research*, 23(131):1–74, 2022.

Dayan Pan, Jingyuan Wang, Yilong Zhou, J. Cheng, Pengyue Jia, and Xiangyu Zhao. RoSA: Enhancing parameter-efficient fine-tuning via RoPE-aware selective adaptation in large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 15600–15608, 2026. doi: 10.1609/aaai.v40i18.38589.

Rui Pan, Xiang Liu, Shizhe Diao, Renjie Pi, Jipeng Zhang, Chi Han, and Tong Zhang. LISA: Layerwise importance sampling for memory-efficient large language model fine-tuning. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Dmytro Perekrestenko, Volkan Cevher, and Martin Jaggi. Faster coordinate descent via adaptive importance sampling. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pp. 869–877. PMLR, 2017.

Amrutha Varshini Ramesh, Vignesh Ganapathiraman, Issam H. Laradji, and Mark Schmidt. BlockLLM: Memory-efficient adaptation of LLMs by selecting and optimizing the right coordinate blocks. arXiv preprint arXiv:2406.17296, 2024. URL https://arxiv.org/abs/2406.17296.

Peter Richtárik and Martin Takáč. Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function. *Mathematical Programming*, 144(1–2):1–38, 2014. doi: 10.1007/s10107-012-0614-z.

Peter Richtárik and Martin Takáč. Distributed coordinate descent method for learning with big data. *Journal of Machine Learning Research*, 17(75):1–25, 2016a.

Peter Richtárik and Martin Takáč. On optimal probabilities in stochastic coordinate descent methods. *Optimization Letters*, 10(6):1233–1243, 2016b. doi: 10.1007/s11590-015-0916-1.

Peter Richtárik and Martin Takáč. Parallel coordinate descent methods for big data optimization. *Mathematical Programming*, 156:433–484, 2016c. doi: 10.1007/s10107-015-0901-6.

Sebastian U. Stich, Anant Raj, and Martin Jaggi. Safe adaptive importance sampling. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Yi-Lin Sung, Varun Nair, and Colin Raffel. Training neural networks with fixed sparse masks. In *Advances in Neural Information Processing Systems*, volume 34, 2021.

Paul Tseng. Convergence of a block coordinate descent method for nondifferentiable minimization. *Journal of optimization theory and applications*, 109(3):475–494, 2001.

Stephen J. Wright. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015. doi: 10.1007/s10107-015-0892-3.

Runxin Xu, Fuli Luo, Zhiyuan Zhang, Chuanqi Tan, Baobao Chang, Songfang Huang, and Fei Huang. Raise a child in large language model: Towards effective and generalizable fine-tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 9514–9528. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.emnlp-main.749.

Kai Yao, Penglei Gao, Lichun Li, Yuan Zhao, Xiaofeng Wang, Wei Wang, and Jianke Zhu. Layer-wise importance matters: Less memory for better performance in parameter-efficient fine-tuning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 1977–1992. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-emnlp.109.

Yushun Zhang, Congliang Chen, Ziniu Li, Tian Ding, Chenwei Wu, Diederik P. Kingma, Yinyu Ye, Zhi-Quan Luo, and Ruoyu Sun. Adam-mini: Use fewer learning rates to gain more. In *International Conference on Learning Representations*, 2025.

Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. GaLore: Memory-efficient llm training by gradient low-rank projection. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 61121–61143, 2024.

Qinghua Zhao, Xueling Gong, Xinyu Chen, Zhongfeng Kang, and Xinlu Li. A layer-wise analysis of supervised fine-tuning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 9981–9992. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.453.

APPENDIX CONTENTS

A More Related Work **13**

B Additional Analysis of SmartBCD **15**

B.1 Derivation of the First-Order Block Gain 16

B.2 Interpretation of Cost-Aware Utility 16

B.3 Bounded Revisit Interval under Maximum Wait 17

B.4 Computational Complexity 18

C SmartBCD Algorithm **18**

D Implementation Details **18**

D.1 Training Configuration 19

D.2 SmartBCD Configuration 19

D.3 Baseline Configuration 20

D.4 Hardware and Measurement Protocol 20

E Additional Experimental Results **20**

E.1 Complete System-Efficiency Results 20

E.2 Additional Analysis of Cost-Aware Selection 21

E.3 Cross-Model and Cross-Dataset Results 22

E.4 Hardware Portability on a Memory-Constrained GPU 22

A MORE RELATED WORK

Coordinate and block-coordinate optimization. Coordinate descent provides a classical framework for large-scale optimization by updating only a subset of variables at each iteration. Randomized coordinate descent has been extensively analyzed for smooth, composite, and block-structured objectives (Nesterov, 2012; Richtárik & Takáč, 2014; Wright, 2015; Lu & Xiao, 2015). These analyses establish that partial updates can substantially reduce the per-iteration cost while retaining favorable convergence properties. Related work has further considered parallel and distributed coordinate updates, showing how coordinate methods can scale to large optimization problems when updates are appropriately decomposed across variables or workers (Richtárik & Takáč, 2016c;a). Accelerated and proximal variants extend this framework to broader objective classes and improve convergence rates under suitable smoothness assumptions (Fercoq & Richtárik, 2015).

A key implication of this literature is that the coordinate or block selection rule is itself an important component of the optimization algorithm. Uniform random selection is simple and broadly applicable, but it ignores heterogeneity in the local optimization landscape. Gauss–Southwell-type rules instead prioritize coordinates exhibiting stronger local descent signals and can improve convergence relative to uniform random selection when the selection overhead remains moderate (Nutini et al., 2015). This observation motivates treating update allocation as an optimization decision rather than merely as an implementation detail.

Non-uniform and importance-based sampling. A large body of work studies whether more informative sampling distributions can improve coordinate optimization. Richtárik and Takáč (Richtárik & Takáč, 2016b) analyze how coordinate-dependent sampling probabilities influence

stochastic coordinate descent and derive probability choices adapted to problem structure. Allen-Zhu et al. (Allen-Zhu et al., 2016) show that non-uniform sampling based on coordinate smoothness can improve accelerated coordinate descent. More general analyses consider arbitrary sampling distributions and minibatch coordinate updates (Hanzely & Richtárik, 2019), while importance sampling has also been combined with minibatching to preferentially allocate computation to more informative updates (Csiba & Richtárik, 2018).

These results provide a useful perspective for block-wise language-model training: when different blocks exhibit heterogeneous optimization behavior, assigning identical update frequency to every block need not be the most efficient use of a fixed training budget. However, most classical importance-sampling rules are derived from static or analytically available quantities such as smoothness constants, coordinate Lipschitz parameters, or prescribed sampling probabilities. In large neural networks, the relative usefulness of parameter blocks can instead evolve substantially over the course of training, motivating selection rules that update their beliefs online.

Adaptive sampling under non-stationary optimization. Several methods therefore allow the sampling distribution itself to change during optimization. Perekrestenko et al. (Perekrestenko et al., 2017) construct adaptive coordinate-sampling rules from optimization-state information such as dual residuals and primal–dual gap estimates, allowing selection probabilities to change across iterations. Namkoong et al. (Namkoong et al., 2017) formulate adaptive sampling as an online learning problem, learning probabilities for coordinates or training examples while controlling the additional cost of adaptation. Stich et al. (Stich et al., 2017) similarly develop safe adaptive importance-sampling schemes that approximate desirable gradient-dependent sampling distributions without repeatedly computing all gradient information.

This line of work highlights a trade-off that is directly relevant to SmartBCD: a more informed selection policy may improve optimization, but obtaining sufficiently fresh selection statistics can itself be expensive. SmartBCD addresses this issue by collecting scheduling signals only from block epochs that are already executed for training. No inactive block is evaluated solely to refresh its scheduling score. Consequently, the scheduler operates under partial and increasingly stale information about blocks that have not been selected recently. The exploration and recency terms compensate for this uncertainty, while the maximum-wait mechanism imposes a deterministic coverage constraint when a block has remained inactive for too long.

SmartBCD further differs from classical adaptive coordinate sampling in its scheduling objective. Existing coordinate-selection analyses primarily optimize iteration complexity, expected descent, or sampling variance. In block-wise LLM fine-tuning, however, block updates can also differ in their actual execution cost. SmartBCD therefore evaluates a block through its estimated optimization return per unit of measured training time, rather than through optimization gain alone. This turns block selection into an online resource-allocation problem in which both optimization gain and execution cost evolve during training.

Layer-wise heterogeneity and selective LLM adaptation. Recent studies of language-model fine-tuning provide increasing evidence that adaptation is heterogeneous across model depth. Layer-wise importance analyses show that different layers can contribute unequally to downstream adaptation, motivating selective allocation of trainable parameters or update opportunities (Yao et al., 2024). More recent analyses characterize how supervised fine-tuning affects different depth regions and show that effective adaptation can be localized to selected subsets of layers (Zhao et al., 2026). RoSA further combines parameter-efficient adaptation with dynamic layer selection, using layer-dependent signals to identify components that should receive adaptation capacity (Pan et al., 2026).

These studies reinforce the broader premise that uniformly treating all layers or parameter groups may be unnecessary. Their focus, however, is primarily on identifying which layers or parameters should be adapted, often under a parameter-efficient fine-tuning objective. SmartBCD instead considers a recurrent scheduling problem: every block remains part of the optimization process, but the frequency with which blocks receive updates is dynamically reallocated according to their evolving gain–cost statistics. The maximum-wait constraint further distinguishes temporary deprioritization from permanent parameter exclusion.

Sparse and selective parameter updates. A related family of methods reduces fine-tuning cost by restricting optimization to a selected subset of pretrained parameters. Child-Tuning masks gradients so that only a dynamically or task-informed subset of parameters is updated during fine-tuning (Xu et al., 2021). Diff Pruning instead learns a task-specific parameter difference and encourages this update to become sparse through differentiable pruning (Guo et al., 2021). FISH Mask selects parameters according to approximate Fisher information and applies a fixed sparse update mask throughout training (Sung et al., 2021), whereas BitFit adopts the more restrictive strategy of updating only bias parameters (Ben Zaken et al., 2022). PaFi further shows that sparse update masks can be constructed from pretrained parameter magnitudes without introducing additional inference latency (Liao et al., 2023).

These approaches demonstrate that effective adaptation can often be obtained without updating the complete parameter set. Their selection, however, primarily determines *which parameters remain trainable* over substantial portions of fine-tuning. SmartBCD addresses a different temporal allocation problem: all parameter blocks remain eligible for optimization, while their update frequencies are continuously reallocated according to evolving optimization gain and measured execution cost. Thus, sparse parameter selection and cost-aware block scheduling represent complementary mechanisms for reducing the resources devoted to model adaptation.

Complementary approaches to efficient LLM fine-tuning. A separate line of work reduces training cost by changing the representation or computation associated with each update. QLoRA combines low-rank adaptation with quantized pretrained weights, substantially reducing the memory required for fine-tuning (Dettmers et al., 2023). MeZO approaches efficiency from a different direction by replacing backpropagation with an in-place zeroth-order optimization procedure based on forward evaluations (Malladi et al., 2023). Adam-mini reduces optimizer-state memory by exploiting structural groupings of neural-network parameters and sharing adaptive learning-rate statistics within suitable parameter blocks (Zhang et al., 2025).

These approaches primarily reduce the resource requirement of an individual optimization update. SmartBCD addresses an orthogonal question: given a training procedure in which only a subset of parameters is active at a time, how should the available sequence of updates be distributed across blocks? This distinction makes cost-aware block scheduling conceptually complementary to methods that reduce per-update memory, optimizer-state size, or gradient-computation cost.

Positioning SmartBCD. Taken together, prior work establishes several complementary principles. Coordinate-descent theory shows that partial parameter updates can make large-scale optimization substantially more tractable (Nesterov, 2012; Richtárik & Takáč, 2014; Wright, 2015; Lu & Xiao, 2015). Non-uniform and importance-based methods demonstrate that update frequency should reflect heterogeneity across coordinates (Nutini et al., 2015; Richtárik & Takáč, 2016b; Allen-Zhu et al., 2016; Hanzely & Richtárik, 2019; Csiba & Richtárik, 2018), while adaptive sampling further shows that such preferences can change with the optimization state (Perekrestenko et al., 2017; Namkoong et al., 2017; Stich et al., 2017). Recent selective fine-tuning studies extend the same intuition to the layer structure of large language models (Yao et al., 2024; Zhao et al., 2026; Pan et al., 2026).

SmartBCD builds on these ideas but targets a distinct objective: *online allocation of block-update computation under heterogeneous and time-varying optimization return and execution cost*. Rather than selecting blocks solely according to optimization importance, it combines predicted first-order gain with measured training cost, continuously updates this estimate from executed block epochs, and couples adaptive allocation with an explicit coverage safeguard. This perspective complements the memory-oriented block-wise methods discussed in the main text and places SmartBCD within the broader literature on adaptive, cost-aware partial optimization.

B Additional Analysis of SmartBCD

This section provides additional derivations and analysis for the SmartBCD scheduler introduced in Section 3. The purpose of these results is to clarify the motivation behind the predicted block gain, the cost-normalized utility, and the maximum-wait mechanism. We do not assume that the resulting scheduler solves the full non-stationary block-selection problem optimally; rather, the analysis

explains the local quantities used by SmartBCD and establishes the deterministic coverage property induced by the maximum-wait constraint.

B.1 Derivation of the First-Order Block Gain

Let $F(\theta)$ denote the training objective, and consider one optimizer step that updates only block i. Let $\Delta\theta_i$ denote the parameter displacement applied to this block. A first-order Taylor expansion around the current parameters gives

$$F(\theta + \Delta\theta_i) = F(\theta) + \nabla_{\theta_i} F(\theta)^\top \Delta\theta_i + O(\|\Delta\theta_i\|^2). \tag{6}$$

Writing $g_i = \nabla_{\theta_i} F(\theta)$, the corresponding decrease in the objective is

$$F(\theta) - F(\theta + \Delta\theta_i) = -g_i^\top \Delta\theta_i + O(\|\Delta\theta_i\|^2). \tag{7}$$

Therefore,

$$-g_i^\top \Delta\theta_i \tag{8}$$

provides a first-order approximation to the optimization progress produced by the actual parameter update.

For a block epoch containing K_t optimizer steps, SmartBCD accumulates the corresponding first-order quantities,

$$G_{i,t} = -\sum_{k=1}^{K_t} g_{i,t,k}^\top \Delta\theta_{i,t,k}, \tag{9}$$

which is Eq. 1 in the main text.

This quantity differs from a gradient-norm criterion such as $\|g_{i,t,k}\|$. The gradient norm describes the local sensitivity of the objective, whereas $-g^\top \Delta\theta$ additionally incorporates the parameter displacement actually produced by the optimizer. This distinction is relevant for adaptive optimizers, for which the magnitude and direction of $\Delta\theta$ need not be determined by the raw gradient norm alone.

The approximation in Eq. 7 is local. Consequently, $G_{i,t}$ should be interpreted as a proxy for training-objective improvement rather than an exact measurement of loss reduction, and it does not directly estimate validation-loss improvement. This derivation is intended to motivate the scheduling signal used by SmartBCD rather than to establish a new optimization guarantee.

B.2 Interpretation of Cost-Aware Utility

SmartBCD defines the utility of block i as

$$U_i = \frac{\bar{G}_i}{\max(\bar{C}_i, \epsilon)}, \tag{10}$$

where $\bar{G}_i$ and $\bar{C}_i$ denote the recent predicted optimization gain and measured training cost, respectively.

To motivate this ratio, consider a simplified local scheduling problem in which the gain and cost of each block remain approximately constant over a short scheduling horizon. Suppose block i is selected n_i times. The resulting predicted optimization gain is approximately

$$\mathcal{G} = \sum_{i=1}^{D} n_i \bar{G}_i, \tag{11}$$

while the associated computation cost is

$$\mathcal{C} = \sum_{i=1}^{D} n_i \bar{C}_i. \tag{12}$$

Under a computation budget B, the local resource-allocation problem can be written as

$$\max_{\{n_i\}} \sum_{i=1}^{D} n_i \bar{G}_i \qquad \text{s.t.} \qquad \sum_{i=1}^{D} n_i \bar{C}_i \le B. \tag{13}$$

For a single additional scheduling decision, selecting block i consumes approximately $\bar{C}_i$ units of computation and yields approximately $\bar{G}_i$ units of predicted progress. Its local return per unit computation is therefore

$$\frac{\bar{G}_i}{\bar{C}_i}. \tag{14}$$

This gives the cost-aware utility used by SmartBCD.

The ratio should not be interpreted as implying that inexpensive blocks are always preferable. A block with larger computation cost can still obtain higher utility whenever its expected optimization gain increases sufficiently. Hence, cost normalization compares the optimization return obtained from the consumed computation rather than minimizing computation cost alone.

The argument above is intentionally local. During actual fine-tuning, both $\bar{G}_i$ and $\bar{C}_i$ can evolve over time, so the scheduling problem is non-stationary. This motivates the adaptive statistics and exploration mechanisms in Section 3.2, rather than treating the ratio as a fixed global ranking of blocks.

B.3 Bounded Revisit Interval under Maximum Wait

The adaptive score in Eq. (4) intentionally permits non-uniform update frequencies. Exploration and recency encourage revisiting under-selected blocks, but neither mechanism alone provides a deterministic bound on how long a block may remain inactive. The maximum-wait rule provides such a bound.

Let $a_i(t)$ denote the number of block epochs since block i was last selected at scheduling decision t. Recall that whenever at least one block satisfies $a_i(t) \ge H$, SmartBCD temporarily overrides the adaptive score and selects a block with maximum age.

[Bounded revisit interval] Consider D parameter blocks under the maximum-wait rule of SmartBCD. Once the age of any block i reaches H, that block is selected within at most D scheduling decisions. Consequently, the age of every block is bounded by

$$a_i(t) \le H + D - 1,$$

and, up to boundary effects, every block is selected at least once within any $H + D - 1$ consecutive block epochs. Equivalently, if $N_i(T)$ denotes the number of selections of block i during the first T adaptive block epochs, then

$$\liminf_{T\to\infty} \frac{N_i(T)}{T} \ge \frac{1}{H + D - 1}.$$

Suppose block i first reaches age H at scheduling decision t. If it is not selected immediately, the maximum-wait rule selects another overdue block whose age is at least as large as $a_i(t)$. While block i remains inactive, its age increases by one after every block epoch.

Any block that is younger than i at decision t remains younger while both blocks are inactive, since their ages increase at the same rate. Therefore, only blocks that are at least as old as i at decision t can be selected ahead of it. There are at most $D - 1$ such blocks. Moreover, once one of these blocks is selected, its age is reset and it cannot overtake i before i is selected.

Hence, after at most $D-1$ other overdue blocks are selected, block i must be selected. Its age can therefore never exceed $H+D-1$. The bound on the long-run selection frequency follows from the resulting upper bound on the interval between consecutive selections of the same block.

This result is a coverage property rather than a convergence guarantee for the underlying non-convex optimization problem. It does not imply that SmartBCD is optimal for the full non-stationary scheduling problem. Instead, it shows that adaptive gain–cost allocation cannot indefinitely suppress any parameter block, while still allowing substantially non-uniform update frequencies.

B.4 Computational Complexity

SmartBCD maintains a constant number of scalar statistics for each of the D parameter blocks, including the EMA gain, EMA cost, discounted selection count, and block age. The scheduler therefore requires $O(D)$ additional state.

At each block epoch, computing the standardized utilities, exploration terms, recency terms, and the maximum-wait condition requires $O(D)$ scalar operations. Selecting the next block can likewise be implemented with a linear scan over the D blocks.

Importantly, SmartBCD does not evaluate inactive blocks when making a scheduling decision. Both the predicted gain and computation cost are collected from block epochs that are already executed as part of training. Therefore, the scheduler introduces no additional model forward or backward passes solely for block selection. Its additional computation consists only of block-level scalar statistics and is independent of the total number of model parameters.

C SmartBCD Algorithm

Algorithm 1 summarizes the complete SmartBCD procedure. The scheduler operates at the granularity of *block epochs*. At each scheduling decision, one parameter block is activated and optimized for K consecutive optimizer steps, while the remaining blocks are frozen. The resulting parameter update is then used to update the block-level gain and cost statistics.

SmartBCD consists of two stages. It first performs a single warm-up pass over all blocks to obtain an initial estimate of their optimization gain and execution cost. Subsequent block selections are adaptive. Unless the maximum-wait constraint is triggered, the scheduler combines the cost-normalized utility with the exploration and recency terms defined in Eq. 4. When one or more blocks exceed the maximum-wait horizon, the adaptive score is temporarily overridden and the oldest block is selected. This separates efficiency-oriented scheduling from the hard coverage safeguard analyzed in Appendix B.3.

Importantly, all gain and cost statistics are collected from block epochs that are already executed for training. SmartBCD therefore does not require additional forward or backward passes to evaluate candidate blocks.

The warm-up stage ensures that every block has an observed gain and cost before adaptive scheduling begins. Afterwards, only the statistics of the executed block are refreshed at each block epoch, while information for inactive blocks becomes progressively stale. The exploration and recency terms in Eq. 4 compensate for this partial observability, whereas the maximum-wait rule provides the deterministic coverage safeguard.

The resulting scheduler should therefore be interpreted as an online allocation rule rather than a fixed ordering of parameter blocks. Both the estimated optimization return and the measured execution cost are updated throughout training, allowing the relative preference among blocks to change as optimization progresses.

D Implementation Details

We provide additional implementation details for the experiments in Section 4. Unless otherwise specified, all methods share the same data split, optimization schedule, effective batch size, and evaluation protocol. Method-specific changes are introduced only when required by the corresponding optimization scheme.

Table 5: Main training configuration used for Qwen2.5-7B-Instruct on MathInstruct.

Configuration	**Value**
Training epochs	3
Optimizer steps	49,038
Validation examples	512
Sequence length	512
Effective batch size	16
Learning rate	1×10^{-5}
Learning-rate schedule	Cosine
Weight decay	0.01
Maximum gradient norm	1.0
Precision	BF16
Gradient checkpointing	Enabled
Evaluation interval	1,000 steps

Table 6: Default SmartBCD scheduler configuration.

Hyperparameter	**Value**
Number of blocks D	28
Block epoch length K	100 optimizer steps
Warm-up	One pass over all blocks
Gain EMA coefficient	0.9
Discounted-count decay ρ	0.97
Exploration coefficient c	0.1
Recency coefficient λ	0.25
Maximum-wait horizon H	56 block epochs
Utility stabilizer ϵ	10^{-12}

D.1 Training Configuration

Our primary experiments fine-tune Qwen2.5-7B-Instruct on the full MathInstruct dataset for three epochs, corresponding to 49,038 optimizer steps. We reserve 512 examples for validation and evaluate the model every 1,000 optimizer steps.

The maximum sequence length is 512. We use an effective batch size of 16, a learning rate of 1×10^{-5} with cosine decay, weight decay of 0.01, and gradient clipping with a maximum norm of 1.0. Training is performed in BF16 precision with gradient checkpointing enabled. The main training configuration is summarized in Table 5.

D.2 SmartBCD Configuration

We partition the trainable parameters into $D = 28$ blocks and keep each selected block active for $K = 100$ optimizer steps before making the next scheduling decision. SmartBCD begins with a single warm-up pass over all blocks to initialize the block-level gain and cost statistics.

The predicted gain is smoothed using an exponential moving average with coefficient 0.9. The discounted selection count uses decay factor $\rho = 0.97$. We set the exploration coefficient to $c = 0.1$ and the recency coefficient to $\lambda = 0.25$.

The maximum-wait horizon is defined as

$$H = 2D = 56, \tag{15}$$

so that a block that has remained inactive for sufficiently long is prioritized by the coverage rule described in Appendix B.3.

The default scheduler configuration is summarized in Table 6.

D.3 Baseline Configuration

For the block-coordinate baselines, we use the same 28-block partition and the same 100-step block epoch as SmartBCD. **BAdam-Random** samples the next block uniformly at random, while **BAdam-Cyclic** traverses blocks in a fixed cyclic order.

Gain-only is constructed as a controlled variant of SmartBCD. It retains the same warm-up procedure, adaptive statistics, exploration, recency, and maximum-wait mechanism, but removes cost normalization from the scheduling utility. Thus, the comparison between SmartBCD and Gain-only isolates the contribution of gain-per-cost scheduling.

For the non-block-coordinate baselines, we use LoRA with rank 8 and $\alpha = 32$, GaLore with rank 128, update interval 200, and scale 2.0, and full-parameter Adam-mini. The effective batch size is kept consistent across methods; Adam-mini uses a smaller per-device batch size with correspondingly larger gradient accumulation to satisfy its memory requirement.

D.4 Hardware and Measurement Protocol

All experiments are conducted on NVIDIA A100-SXM4-80GB GPUs, with one GPU used for each training run. Training is implemented in PyTorch 2.5.1 with CUDA 12.4 using LLaMA-Factory augmented with our block scheduler.

We report both trainer wall-clock time and GPU energy consumption. Whole-GPU power is sampled through NVIDIA Management Library (NVML) at 1 Hz, and cumulative energy is computed using trapezoidal integration,

$$E = \sum_{j=1}^{M-1} \frac{P_j + P_{j+1}}{2} \left(t_{j+1} - t_j\right), \tag{16}$$

where P_j denotes the measured GPU power at time t_j.

For the complete resource comparison, energy is integrated over the entire NVML monitoring trace. For loss–energy convergence curves, we instead integrate over the trainer window so that each validation checkpoint is associated with the energy consumed up to that point. These two measurement boundaries are kept separate throughout the experiments.

Peak allocated and peak reserved GPU memory are obtained from PyTorch CUDA memory statistics. For block-coordinate methods, CUDA synchronization is performed at block-epoch timing boundaries before recording elapsed time, ensuring that the measured block cost reflects completed GPU computation rather than asynchronous kernel dispatch.

E Additional Experimental Results

This section complements the main experiments with a more detailed analysis of system efficiency, cost-aware block selection, and generalization across models, datasets, and hardware environments. We focus on measurements that are difficult to accommodate in the main text but help clarify where the efficiency gains of SmartBCD originate and whether they persist beyond the primary experimental setting.

E.1 Complete System-Efficiency Results

Table 7 reports the complete system-level measurements for the primary Qwen2.5-7B experiment on MathInstruct. In addition to validation loss and training time, we include measured GPU energy and peak memory usage.

Among the block-coordinate methods, SmartBCD reaches the lowest validation loss while requiring the least wall-clock time and GPU energy. Relative to random block scheduling, SmartBCD reduces training time by approximately 4.4% and energy consumption by 6.2%, while also obtaining a lower terminal validation loss. Importantly, its peak allocated memory remains essentially unchanged from

Table 7: Complete system-efficiency results on Qwen2.5-7B fine-tuned on the full MathInstruct dataset. Energy is computed from the complete NVML power trace. Peak allocated and reserved memory are measured using PyTorch CUDA memory statistics.

Method	Val. Loss ↓	Trainer Time (h) ↓	NVML Time (h) ↓	Energy (kWh) ↓	Peak Alloc. (MiB) ↓	Peak Reserved (MiB) ↓
Adam-mini	**0.2217**	30.143	30.154	11.422	45,768	47,312
LoRA	0.2355	28.434	28.442	10.563	**18,835**	**19,612**
GaLore	0.2504	46.200	46.211	15.673	38,383	42,450
BAdam-Random	0.2305	12.458	12.472	4.719	30,339	31,072
BAdam-Cyclic	0.2289	12.090	12.104	4.564	30,354	41,340
Gain-only	0.2275	12.091	12.105	4.653	30,355	41,242
SmartBCD	0.2275	**11.908**	**11.921**	**4.425**	30,339	31,094

the other block-coordinate variants, indicating that the efficiency gain is not obtained by increasing the memory footprint.

Compared with full-parameter or parameter-efficient alternatives, the results expose a different trade-off. Adam-mini attains the lowest terminal validation loss, whereas LoRA substantially reduces memory consumption. SmartBCD instead targets the complementary objective of improving the efficiency of block-coordinate training: it preserves the memory profile of BAdam-based optimization while reducing the time and energy required by the block-scheduling process.

The reported trainer time and NVML wall time use slightly different measurement boundaries, as described in Appendix D.4. For BAdam-Random, a resumed segment contained duplicated training steps; the duplicated interval is removed when computing the reported time and energy so that all measurements correspond to unique optimizer steps.

E.2 Additional Analysis of Cost-Aware Selection

The comparison between SmartBCD and Gain-only isolates the effect of cost normalization: both methods share the same adaptive scheduling mechanism, while Gain-only ranks blocks using predicted optimization gain without accounting for execution cost.

Convergence under a fixed optimization target. Although the two methods reach nearly identical terminal validation losses, their trajectories differ substantially before convergence. Table 8 reports the first observed validation checkpoint at which each method reaches a given loss target. No interpolation between checkpoints is used.

Across the intermediate target range from 0.275 to 0.240, SmartBCD requires on average 9.2% fewer optimizer steps, 11.7% less wall-clock time, and 14.6% less GPU energy than Gain-only. The advantage is therefore not merely a reduction in the cost of individual block epochs; cost-aware scheduling also changes which blocks receive updates early enough to alter the optimization trajectory.

The advantage becomes smaller near terminal convergence. At the 0.235 and 0.230 targets, SmartBCD no longer reaches the target in fewer steps or less wall-clock time, although its cumulative energy remains slightly lower. We therefore interpret the primary benefit of cost-aware scheduling as reaching useful intermediate optimization levels more efficiently, rather than uniformly accelerating every point on the convergence curve.

What changes after cost normalization? To examine the mechanism behind this behavior, we compare the block epochs selected by SmartBCD and Gain-only during the adaptive phase. As shown in Table 9, SmartBCD reduces the average execution cost of a selected block epoch by approximately 1.9%, while increasing the average realized gain per unit cost by approximately 1.2%.

The magnitude of these shifts is deliberately modest. As shown in Figure 3-right in the main text, the selected-block cost and realized gain-per-cost distributions of SmartBCD and Gain-only overlap substantially. The additional statistics in Table 9 quantify this shift: cost normalization reduces the mean execution cost of selected block epochs by 1.9% while increasing their mean realized gain per unit cost by 1.2%.

Table 8: First-passage cost for different validation-loss targets. Targets are evaluated only at the observed 1,000-step validation checkpoints, without interpolation. Positive savings indicate an advantage for SmartBCD over Gain-only.

Target	SmartBCD Steps	Gain-only Steps	SmartBCD Time (h)	Gain-only Time (h)	Time Saving	Energy Saving
0.280	3,000	3,000	0.748	0.749	+0.2%	+3.9%
0.275	4,000	5,000	0.973	1.231	+21.0%	+23.6%
0.270	5,000	6,000	1.193	1.508	+20.9%	+23.2%
0.265	8,000	8,000	1.962	2.009	+2.3%	+5.3%
0.260	10,000	10,000	2.399	2.469	+2.9%	+5.9%
0.255	11,000	13,000	2.637	3.248	+18.8%	+21.4%
0.250	14,000	15,000	3.377	3.699	+8.7%	+11.7%
0.245	17,000	19,000	4.099	4.707	+12.9%	+15.9%
0.240	21,000	22,000	5.063	5.412	+6.5%	+9.8%
0.235	26,000	25,000	6.281	6.165	−1.9%	+1.7%
0.230	31,000	30,000	7.503	7.413	−1.2%	+2.3%

Table 9: Statistics of executed block epochs during adaptive scheduling.

Method	Selection Events	Mean Cost (s) ↓	Median Cost (s) ↓	Mean Gain/Cost ↑	Median Gain/Cost ↑
SmartBCD	462	**84.753**	**82.243**	**0.006346**	0.004816
Gain-only	462	86.371	85.820	0.006271	**0.004899**

These results clarify that SmartBCD does not obtain its efficiency gain by systematically selecting the cheapest blocks. Rather, cost normalization introduces a moderate bias toward blocks with a more favorable gain–cost trade-off while retaining substantial overlap with the selections made by Gain-only. We emphasize that these realized gain-per-cost values are descriptive statistics of executed block epochs; they do not estimate the counterfactual gain of blocks that were not selected.

E.3 Cross-Model and Cross-Dataset Results

To test whether the observed efficiency pattern is specific to a single model–dataset pair, we additionally evaluate Random, Cyclic, and SmartBCD scheduling across two model scales and two datasets. Table 11 reports the corresponding validation loss, wall-clock time, energy consumption, and peak memory.

Across all four settings, SmartBCD attains both the lowest validation loss and the shortest wall-clock time among the three block-scheduling strategies. Energy consumption follows a similar trend, while peak allocated memory remains nearly unchanged within each model family.

These results suggest that the benefit of adaptive gain-per-cost allocation is not confined to the primary Qwen2.5-7B MathInstruct setting. We nevertheless treat these experiments as evidence of cross-setting consistency rather than a statistical comparison, since each entry corresponds to a single training run.

E.4 Hardware Portability on a Memory-Constrained GPU

The main experiments are conducted on server-class A100 GPUs. To examine whether the scheduling behavior of SmartBCD depends strongly on this hardware environment, we additionally evaluate the block-scheduling methods on an NVIDIA GeForce RTX 5080 with 16 GB of GPU memory. We use TinyLlama-1.1B and retain the full-data, three-epoch training budget for both MathInstruct and Alpaca-GPT4.

Table 10 shows that the relative wall-clock advantage of SmartBCD persists on the substantially more memory-constrained GPU. On MathInstruct, SmartBCD reduces training time by approximately 5% relative to both Random and Cyclic scheduling, while also attaining the lowest terminal validation loss. On Alpaca-GPT4-English, the runtime reduction increases to approximately 7–9%, while the terminal losses of the three scheduling strategies remain close.

Table 10: Additional block-scheduling results on an RTX 5080 with 16 GB of GPU memory.

Dataset	Method	Val. Loss ↓	Time (h) ↓
MathInstruct	Random	0.61412	8.21
	Cyclic	0.61536	8.21
	SmartBCD	**0.61177**	**7.77**
Alpaca-GPT4	Random	**1.05038**	1.62
	Cyclic	1.05110	1.66
	SmartBCD	1.05137	**1.50**

Table 11: Cross-model and cross-dataset resource comparison. Each run uses one A100-80GB GPU.

Model	Dataset	Method	Val. Loss ↓	Time (h) ↓	Energy (kWh) ↓	Peak Alloc. ↓	Peak Reserved ↓
Qwen2.5-7B	MathInstruct	Random	0.2305	12.458	4.719	30,339	31,072
		Cyclic	0.2289	12.090	4.564	30,354	41,340
		SmartBCD	**0.2275**	**11.908**	**4.425**	30,339	31,094
Qwen2.5-7B	Alpaca-GPT4	Random	0.8782	2.264	0.862	30,352	37,922
		Cyclic	0.8773	2.292	0.862	30,352	38,280
		SmartBCD	**0.8766**	**2.119**	**0.793**	30,353	**36,896**
TinyLlama-1.1B	MathInstruct	Random	0.6108	3.630	1.222	16,829	18,138
		Cyclic	0.6126	3.713	1.268	16,829	18,084
		SmartBCD	**0.6074**	**3.403**	**1.201**	16,829	**18,084**
TinyLlama-1.1B	Alpaca-GPT4	Random	1.0593	0.734	0.242	16,831	**18,036**
		Cyclic	1.0596	0.746	0.251	**16,830**	18,058
		SmartBCD	**1.0585**	**0.671**	**0.233**	**16,830**	18,058

These results suggest that the observed efficiency improvement is not specific to the A100 execution environment. In particular, the gain–cost scheduler continues to provide a wall-clock advantage when the same block-wise training paradigm is deployed on a consumer GPU with a substantially smaller memory budget. We do not compare absolute runtime across GPU platforms, since hardware throughput differs substantially; the relevant comparison is between scheduling strategies within the same hardware environment.

AI USE STATEMENT

Generative AI tools, primarily OpenAI ChatGPT, were used as research and writing assistants during the preparation of this work. Their use included feedback on the presentation of the SmartBCD methodology and experimental design, assistance in organizing and refining mathematical derivations and arguments, discussion of experimental results, and support for manuscript structure, wording, and LaTeX formatting.

AI-assisted suggestions were treated as intermediate drafts rather than as authoritative research outputs. All methodological choices, mathematical statements, experimental protocols, and interpretations included in the paper were reviewed and revised by the authors. In particular, all reported training runs, measurements, tables, and numerical results were produced by the authors' experimental pipeline and were independently checked against the underlying experimental records.

Generative AI was not used to generate evaluation results or to replace the execution of any reported experiment. The authors take full responsibility for the final content of the paper, including all claims, derivations, experimental results, and AI-assisted text.

Algorithm 1 SmartBCD: Cost-Aware Adaptive Block Scheduling

Require: Model parameters partitioned into D blocks $\{\theta_i\}_{i=1}^{D}$; block-epoch length K; maximum-wait horizon H; scheduler hyperparameters.
Ensure: Trained parameters θ.

Initialize block statistics for all $i \in \{1, \dots, D\}$.

Warm-up
for $i = 1, \dots, D$ **do**
  Activate block θ_i and freeze all other blocks.
  Optimize θ_i for K steps.
  Measure the block-epoch cost C_i.
  Compute the realized first-order gain

$$G_i = -\sum_{k=1}^{K} g_{i,k}^{\top} \Delta\theta_{i,k}.$$

  Initialize the smoothed statistics $\bar{G}_i \leftarrow G_i$ and $\bar{C}_i \leftarrow C_i$.
  Update the selection statistic of block i.
  Update block ages and reset $a_i \leftarrow 0$.
end for

Adaptive scheduling
while training budget remains **do**
  Define the overdue set

$$\mathcal{O}_t = \{i \mid a_i(t) \geq H\}.$$

  if $\mathcal{O}_t \neq \emptyset$ **then**
    Select the oldest overdue block

$$i_t \in \arg\max_{i \in \mathcal{O}_t} a_i(t).$$

  else
    for $i = 1, \dots, D$ **do**
      Compute the cost-aware utility

$$U_i(t) = \frac{\bar{G}_i(t)}{\max\big(\bar{C}_i(t), \epsilon\big)}.$$

    end for
    Compute the adaptive scores $\{S_i(t)\}_{i=1}^{D}$ using Eq. 4.
    Select

$$i_t \in \arg\max_{i} S_i(t).$$

  end if
  Activate block θ_{i_t} and freeze all other blocks.
  Optimize θ_{i_t} for K steps.
  Measure the realized block-epoch cost $C_{i_t,t}$.
  Compute the realized first-order gain

$$G_{i_t,t} = -\sum_{k=1}^{K} g_{i_t,t,k}^{\top} \Delta\theta_{i_t,t,k}.$$

  Update the smoothed gain and cost statistics of block i_t.
  Update its selection statistic.
  Increment the ages of inactive blocks and reset $a_{i_t}(t+1) \leftarrow 0$.
end while
return θ.