Kareus: Joint Reduction of Dynamic and Static Energy in Large Model Training

Ruofan Wu Jae-Won Chung Mosharaf Chowdhury
University of Michigan

Abstract

The computing demand of AI is growing at an unprecedented rate, but energy supply is not keeping pace. As a result, energy has become an expensive and contended resource that requires explicit management and optimization. Although recent works have made significant progress in large model training optimization, they focus on optimizing either dynamic or static energy consumption.

We find that fine-grained kernel scheduling and frequency scaling *jointly* and *interdependently* impact both dynamic and static energy consumption. Based on this finding, we design Kareus, a training system that pushes the time–energy tradeoff frontier by optimizing both aspects. Kareus decomposes the intractable joint optimization problem into local, partition-based subproblems. It then uses a multi-pass multi-objective optimization algorithm to find execution schedules that push the time–energy tradeoff frontier. Compared to the state of the art, Kareus reduces training energy by up to 28.3% at the same training time, or reduces training time by up to 27.5% at the same energy consumption.[1]

1 Introduction

Today, energy is the ultimate bottleneck for scaling AI [7, 31, 42]. The energy demand of training large models and serving them to millions is growing at an unprecedented rate [21, 33, 37], with projections indicating that by 2035, nearly 10% of US electricity demand could be from datacenters [31]. However, procuring energy at scale is slow, e.g., three years for natural gas and five to ten years for nuclear [46]. This mismatch makes energy an expensive and contended resource that must be explicitly budgeted, managed, and allocated with efficient systems and optimization methods.

In this context, large model training is a key target for optimization; a single training run can consume enough energy to power more than 24,000 average US households for a month [1, 33]. To understand the current state of large model training optimizations, we analyze them through the lens of *dynamic* energy (consumed by actual work) and *static* energy (consumed at all times regardless of work) consumption of GPUs (§2). Practically, dynamic energy can be reduced by lowering GPU frequency or activity, while static energy can be reduced by shortening iteration time. Perseus [15] reduces dynamic energy by scaling the frequency of computations off the critical path, but does not reschedule kernels. Recent works in fine-grained kernel scheduling [18, 20, 48, 49, 57] reduce static energy by overlapping computation and communication, but they ignore dynamic energy and frequency scaling. Therefore, we ask: can we combine both approaches to *jointly* reduce dynamic and static energy?

To answer this question, we study how *execution schedules* affect the time and energy consumption of large model training (§3). Any large model training iteration boils down to computation and communication kernels launched on the GPU. We define an *execution schedule* as the combination of three factors: (1) the *timing* of when communication kernels are launched within a sequence of computation kernels, (2) the *number of GPU Streaming Multiprocessors (SMs)* allocated to communication kernels, and (3) the *GPU frequency*.

We demonstrate that these factors *jointly* and *interdependently* determine time and energy consumption. Different execution schedules lead to different time and energy consumption—varying by as much as 3.29×—even when the total amount of work remains the same. The optimal schedule is achieved by carefully balancing the resource consumption of computation and communication kernels, which depends on the interplay of all three factors. Crucially, *the best SM allocation and kernel launch timing depend on GPU frequency*; even for the same sequence of kernels, we cannot use the same SM allocation and launch timing at different frequencies. Intuitively, frequency changes how overlapping kernels contend for resources: lowering frequency slows computation while leaving memory and communication bandwidth largely unchanged, thereby reshaping the optimal SM allocation and kernel launch timing. Existing solutions for kernel scheduling [18, 20, 48, 49, 57] and frequency scaling [15] optimize disjoint subsets of these factors, and combining them naively is suboptimal for pushing the time–energy tradeoff frontier.

Based on these observations, we present *Kareus*, an energy-efficient large model training system that automatically finds the best execution schedule by jointly optimizing SM allocation, launch timing, and GPU frequency (§4). For all kernels in the training iteration, it identifies (1) the right number of SMs for communication kernels, and (2) the right amount of computation–communication overlap, both in a frequency-specific manner. A collateral benefit is that executions on the critical path also speeds up due to improved overlap, reducing training time and thus shifting the entire time–energy tradeoff frontier toward the origin.

[1] Kareus is open-source at https://github.com/ml-energy/kareus.

Unfortunately, jointly optimizing all three factors via exhaustive search is impractical. The search space is prohibitively large to profile each configuration. Worse, profiling heats up the GPU, and without sufficient cooling intervals, one configuration's measurement affects subsequent ones.

Kareus addresses these challenges by introducing the *partitioned overlap* execution model. The execution graphs of forward and backward passes are divided into fixed partitions, decomposing the global optimization problem into local subproblems. Partitioned overlap generalizes recent *nanobatching* techniques [18,20,48,49,57] and adds fine-grained control over all three execution schedule factors, enabling precise control over the overlap between each communication kernel and its surrounding computation. For each partition, Kareus uses a multi-pass multi-objective optimization algorithm to identify candidates on the time–energy tradeoff frontier. It then hierarchically composes local (partition-level) frontiers into a global (iteration-level) one for the entire training iteration.

We implement Kareus by integrating with Perseus [15] and Megatron-LM [35] (§5). Kareus's optimizer is informed by time and energy profiling results from our *thermally stable* profiler, which minimizes thermal interference between different configurations during profiling. Once the best execution schedule is identified, Kareus interacts with Perseus and Megatron-LM to realize the plan throughout the entire training execution.

We evaluate Kareus across 14 representative workloads, including real testbed training on Llama 3.2 3B and Qwen 3 1.7B, and large-scale emulation on Llama 3.3 70B (§6). Compared to the state-of-the-art Perseus, Kareus reduces energy by up to 28.3% under the same time budget, or reduces time by up to 27.5% under the same energy budget.

In summary, we make the following contributions:

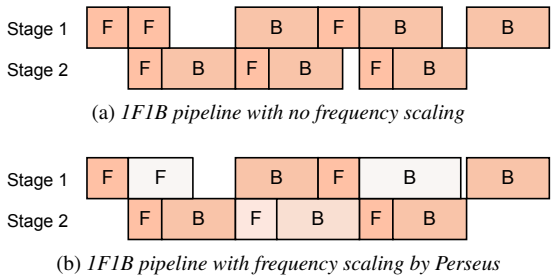
- We show that SM allocation, launch timing, and GPU frequency jointly determine time and energy consumption, and that existing solutions optimizing subsets of these factors and naive combinations thereof are suboptimal.
- We design Kareus around the *partitioned overlap* execution model, decomposing the global optimization problem into tractable local subproblems.
- We evaluate Kareus on large model training workloads, demonstrating significant time and energy reduction compared to the state-of-the-art.

2 Background

We begin by providing a brief background on GPU power consumption (§2.1). Then, we compare existing training systems in terms of their goals and techniques (§2.2), and discuss implications for training iteration time and energy (§2.3).

2.1 GPU Power Consumption

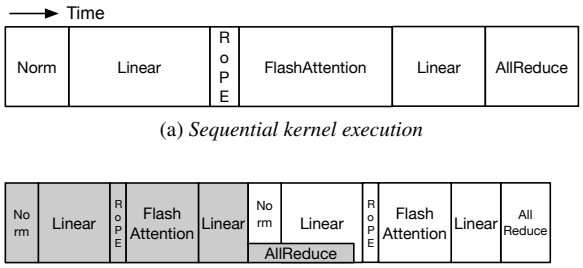
A GPU's power consumption can be divided into two components: *dynamic power and static power* [15, 32, 41]. Dynamic power is the power consumed by the chip's *compute and memory activity*; it is proportional to core frequency and the square of voltage. In contrast, static power is consumed by *all parts of the chip* at all times regardless of utilization or activity.



(a) *1F1B pipeline with no frequency scaling*

(b) *1F1B pipeline with frequency scaling by Perseus*

Figure 1: ***Existing training systems running the 1F1B pipeline schedule [35]. Redder colors indicate higher GPU frequency and power draw. (a) Megatron-LM [35] does not perform any frequency scaling, whereas (b) Perseus [15] scales GPU frequencies of non-critical forward and backward computations to reduce energy consumption while maintaining the same latency.***



(a) *Sequential kernel execution*

(b) *Nanobatching*

Figure 2: ***Transformer [47] Attention layer with tensor parallelism run with different execution models. (a) The sequential kernel execution model only runs one kernel at a time strictly following data dependencies. (b) Nanobatching [18, 20, 48, 49, 57] splits a pipeline microbatch into two nanobatches (illustrated with different colors) and staggers their execution, creating opportunities to overlap communication and computation.***

2.2 Existing Training Systems

Large model training relies on multi-GPU parallelism, which distributes model weights and computation while introducing communication. On each GPU, kernel scheduling determines how computation and communication are executed. Figure 1a illustrates Megatron-LM's [35] implementation of the 1F1B pipeline schedule. Each pipeline microbatch (boxes labeled F and B) contains multiple Transformer [47] blocks, each with an Attention layer and a feed-forward (MLP) layer. Figure 2a shows the sequential kernel execution model adopted by Megatron-LM [35], where kernels execute one after another following data dependencies.

Beyond parallelizing work across devices [5, 14, 25, 28, 35, 54], recent works have optimized latency with *nanobatching* [18, 20, 48, 49, 57], where a single pipeline microbatch is split into two equal-sized *nanobatches* with no data dependencies between them. This creates opportunities to over-

Table 1: ***Training iteration time (seconds) and energy breakdown (Joules) of Megatron-LM, Nanobatching, and each combined with Perseus, training Qwen 3 1.7B on 16 NVIDIA A100 GPUs. Megatron-LM achieves 99.0 TFLOP/s/GPU.***

	Iteration time	Static energy	Dynamic energy	Total energy
Megatron-LM	5.60	5,372	21,374	26,745
Megatron-LM + Perseus	5.60	5,374	19,531	24,905
Nanobatching	5.31	5,096	21,445	26,541
Nanobatching + Perseus	5.37	5,160	19,729	24,889

lap communication and computation kernels from different nanobatches,[2] as shown in Figure 2b.

A separate line of work has optimized the energy consumption of large model training by scaling GPU frequencies [13, 15, 51]. Notably, Perseus [15] dynamically reduces the GPU frequencies of microbatches off the critical path of computation, as shown in Figure 1b, to reduce energy consumption while keeping overall training iteration time the same. However, it follows the sequential kernel execution model (Figure 2a), missing opportunities from fine-grained kernel scheduling like nanobatching.

2.3 Breaking Down Training Energy Consumption

We can better understand the energy consumption of existing training systems and techniques by breaking down energy into static and dynamic components. Table 1 presents such a breakdown along with iteration time for Qwen 3 1.7B [44] training on 16 NVIDIA A100 GPUs.[3] Static energy is static power[4] multiplied by iteration time, and dynamic energy is static energy subtracted from total energy.

Compared to baseline Megatron-LM (first row), Nanobatching (third row) reduces iteration time, which directly lowers static energy. Dynamic energy is slightly higher because nanobatching incurs additional GPU activity from more memory accesses and extra gradient accumulations per nanobatch. Perseus applied to Megatron-LM (second row) reduces dynamic energy through explicit frequency scaling while keeping iteration time nearly the same, so static energy remains unchanged. Applying Perseus on top of Nanobatching (fourth row) combines both benefits: reduced static energy from shorter time and dynamic energy from frequency scaling.

However, we find that there is further opportunity to reduce energy beyond what is achieved by simply combining Nanobatching and Perseus. To understand this gap, we analyze how communication and kernel scheduling affect energy consumption (§3). The opportunities we find motivate the design of Kareus, which jointly optimizes kernel scheduling and frequency scaling (§4).

[2]Computation and communication exert different GPU resources, so overlapping them increases overall GPU utilization and can lead to speedup.

[3]Trained with 8 microbatches, each with size 16 and sequence length 4K, using pipeline parallelism 2, context parallelism 2, and tensor parallelism 4. The same experiment also appears in Table 3, Table 4, and Figure 11.

[4]Static power is set to be the GPU's power draw when it is in *ready* state (power state P0) without running any significant computations.

3 Energy Impact of Execution Schedules

We investigate the relationship between energy consumption and execution schedule. We first discuss how different execution schedules for the same *work* can lead to different energy consumption (§3.1), and then analyze the impact of key factors through case studies (§3.2). This reveals the necessity of joint control and the opportunity for significant energy savings (§3.3).

3.1 Execution Schedule and Energy Consumption

Regardless of *how* work (computation, memory access, and communication) is performed, the total amount of work done is the same. However, different execution schedules can still lead to different energy consumption.

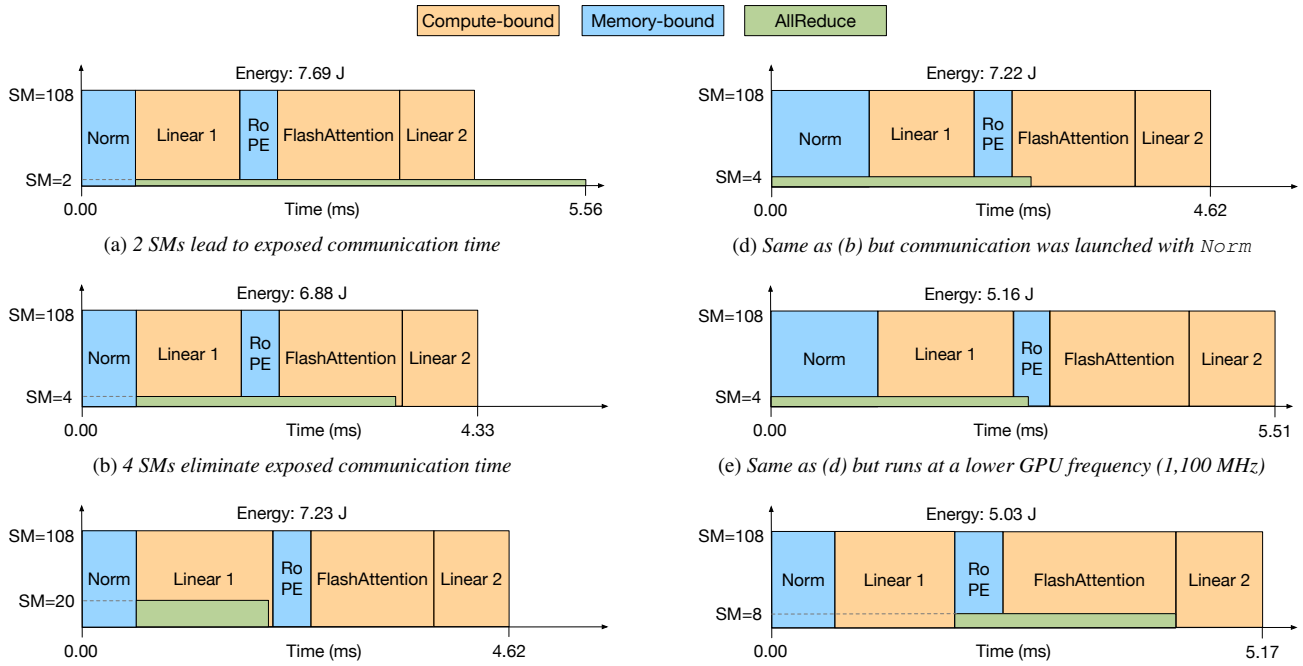
The distinction between dynamic and static power is key to understanding this. Dynamic energy reflects the total work done on the hardware (i.e., transistors switching from computation, memory access, and communication). At *the same GPU frequency*, it remains largely constant across execution schedules. In contrast, all parts of the hardware consume static power as long as they are powered on. Compute components (e.g., SMs) account for a significant portion of chip area and thus static power. When GPU resources are underutilized, static power is still dissipated without as much useful work being done, increasing total energy consumption.

In sum, GPU frequency primarily influences dynamic energy, whereas execution schedules largely determine static energy by changing end-to-end execution time. However, these factors are interdependent and cannot be decoupled and optimized separately, as we show next.

3.2 Interdependent Factors in Execution Schedules

Execution schedules differ in which kernels run concurrently on the GPU, how they split GPU resources, and at what GPU frequency they run. We study the impact of execution schedules on time and energy consumption by varying three key factors that correspond to each: (1) communication kernel launch timing, (2) number of SMs allocated to the communication kernel, and (3) GPU frequency. This definition of execution schedule generalizes nanobatching. The original nanobatching model (1) launches communication kernels as soon as possible, (2) uses default communication kernels like NCCL optimized for sequential execution (which may use excessive SMs assuming no concurrent kernels), and (3) does not perform GPU frequency scaling.

Figure 3 shows the timelines and total energy consumption of different execution schedules for a single Transformer Attention layer of Llama 3.2 3B [33], with tensor parallelism degree 4 on four fully-connected NVIDIA A100 GPUs. For simplicity of exposition, we focus on a single repeating segment: the computation kernels of one nanobatch and the communication kernel of the *previous* nanobatch. Thus, the communication kernel has no data dependencies with any



(a) *2 SMs lead to exposed communication time*

(b) *4 SMs eliminate exposed communication time*

(c) *20 SMs reduce communication latency but lead to compute interference*

(d) *Same as (b) but communication was launched with* `Norm`

(e) *Same as (d) but runs at a lower GPU frequency (1,100 MHz)*

(f) *Lower frequency (1,100 MHz) changes the energy-optimal schedule*

Figure 3: ***The time and total energy consumption of execution schedules for one Transformer Attention layer forward pass with varying SM allocation, communication launch timing, and GPU frequency. (a)–(c) show the effect of allocating different numbers of SMs to the communication kernel at 1,410 MHz, with (b) being the energy-optimal schedule. (d) is the same as (b), except that communication was launched earlier together with*** `Norm`***. Finally, (e) and (f) run at a lower GPU frequency of 1,100 MHz. (e) is the same as (d) other than frequency, whereas (f) is the energy-optimal schedule at 1,100 MHz, which is different from the schedule in (b).***

of the computation kernels in the same segment. This is the common-case execution pattern throughout training.

3.2.1 SM Allocation

First, we study the impact of GPU resource sharing via SM allocation. Allocating more SMs to communication kernels leaves fewer SMs for overlapped computation kernels given a fixed number of SMs. Figures 3a, 3b, and 3c show execution schedules that vary only in the number of SMs allocated to the communication kernel (2, 4, and 20 SMs, respectively) while launching the communication kernel together with `Linear 1`. With two SMs allocated to the communication kernel (Figure 3a), the communication does not complete before the computations finish, leaving an exposed communication period during which 106 SMs are idle. Static power is wasted during this exposed communication time, increasing total energy consumption. Increasing the number of SMs to four (Figure 3b) lets the communication kernel complete before the computation kernels, minimizing static power wastage and total energy consumption. However, further increasing the number of SMs to 20 (Figure 3c) speeds up communication only at the cost of taking SMs away from `Linear 1`, which ultimately increases total time and energy consumption. This is because the excess SMs allocated to the communication kernel slow down computation while remaining nearly idle themselves, again wasting static power.

3.2.2 Communication Launch Timing

Next, we study the impact of kernels that run together by changing when to launch the communication kernel. Figure 3d shows an execution schedule where the communication kernel runs with four SMs (the same as Figure 3b) but starts with `Norm` instead of `Linear 1`. As shown in colors, `Norm` and `RoPE` kernels are memory-bound, and with sufficient SMs, the communication kernel also requires high memory bandwidth to write the communicated data to memory. When such a communication kernel starts together with another memory-bound kernel like `Norm`, the two compete for memory bandwidth, prolonging both. This extends the period where compute resources are underutilized, wasting static power. Furthermore, compared to Figure 3b, memory bandwidth is underutilized during `FlashAttention`, wasting static power in memory components.

3.2.3 Varying GPU Frequency

Finally, we study the impact of changing the GPU frequency. Reducing GPU frequency lowers dynamic energy [15, 51]. More importantly, the energy-optimal execution schedule changes at lower frequencies. At lower frequencies, all kernels become relatively more compute-bound because reducing frequency only affects computation throughput and not memory

Table 2: ***Summary of factors and observations from case studies on the energy impact of execution schedules.***

Factor	Observation	Explanation
SM allocation	A middle-ground sweet spot exists	Too few SMs for a kernel can lead to periods where only one small kernel is running, wasting static power; too many SMs slow down other kernels without significantly speeding up the target kernel.
Launch timing	Resource demands of kernels running together matter	Communication, which can be memory-bound, running together with memory-bound operations (e.g., `Norm`, `RoPE`) causes bandwidth contention and slowdown for everyone.
GPU frequency	Changes relative kernel resource demands	Lower frequency makes kernels relatively more compute-bound, changing which kernels benefit from overlapping; it also amplifies static power's relative impact, making exposed communication more harmful.
All three	Time & energy impacts of all factors are interdependent	The energy-optimal SM allocation and launch timing depend on the GPU frequency; changing one factor shifts the energy-optimal configurations of the others.

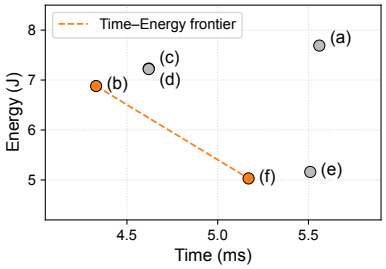


Figure 4: ***Time and energy consumption of the execution schedules in Figure 3 (a)–(f).***

throughput [38].[5] Thus, running the communication kernel with a memory-bound kernel causes less interference since memory bandwidth is less contended. Conversely, running it with a compute-bound kernel and taking SMs away from it causes more severe interference, with larger slowdowns and more static power wastage.

Figure 3e is the same as Figure 3d, but runs at a lower frequency (1,100 MHz). Reduced dynamic energy contributes substantially to lower total energy. However, execution time increases significantly because the communication kernel takes SMs away from `Linear 1`, which is now even more compute-bound at the lower frequency. The energy-optimal schedule at this frequency, shown in Figure 3f, launches communication together with `RoPE`. This lets the compute-bound `Linear` kernels run without interference, reducing total execution time and static power wastage. Figure 4 plots the time–energy frontier of the six execution schedules.

We also note that exposed communication time is relatively more harmful at lower frequencies. Dynamic power decreases significantly with frequency while static power does not—this increases static power's proportion in total power draw and thus its relative impact when wasted.

3.3 Joint Control and Opportunities

Table 2 summarizes observations from the case studies. The three factors that determine the execution schedule—SM allocation, communication launch timing, and GPU frequency—are (1) interdependent: changing one shifts the energy-optimal configuration of the others, and (2) highly impactful, with up to a 3.29× gap in time and energy consumption across the observed schedules. Therefore, optimizing time and energy requires *joint control* of all three. This is the key insight motivating the design of Kareus's optimization algorithm.

In fact, joint control of GPU frequency and kernel scheduling is provably more energy-efficient than leaving GPU frequency control to the hardware's power controller (Appendix A). The intuition is that since dynamic power grows roughly with the cube of frequency,[6] high-frequency periods cost more energy than what low-frequency periods save, so removing frequency variation reduces total dynamic energy.

Our analysis also shows that opportunities for energy savings with execution schedule control are significant. For simplicity of analysis, we have studied a single Attention layer with modest communication volume (tensor parallelism degree 4). Even in this setting, without any GPU frequency scaling, the energy-optimal execution schedule (Figure 3b) reduces energy consumption by 24.3% and time by 12.3% compared to Megatron-LM, and 7.1% energy and 5.4% time compared to Nanobatching. Larger models, more GPUs, and additional parallelism dimensions (e.g., context parallelism) provide even greater opportunities, as we show in Section 6.

4 Algorithm Design

In this section, we present the optimization algorithm of Kareus, whose goal is to find the time–energy tradeoff frontier of executing computation and communication kernels of large model training. We begin by formulating the optimization problem (§4.1), and then introduce the partitioned overlap execution model, which decomposes the problem into tractable subproblems (§4.2). Building on this model, Kareus efficiently explores the execution schedule space to derive the time–energy frontier for each subproblem (§4.3), and composes these local frontiers into a global frontier that characterizes the training iteration (§4.4). Finally, we discuss how Kareus generalizes to broader computation and communication patterns (§4.5).

[5] Essentially, the hardware roofline model's horizontal compute-bound ceiling is lowered, making it easier for kernels to be compute-bound.

[6] Dynamic power is proportional to $V^2 f$. In NVIDIA GPUs, voltage scales roughly linearly with frequency (up to a certain point), so $V^2 f$ becomes f^3.

4.1 Optimization Formulation

Objective. The goal of Kareus is to find an efficient time–energy frontier for the training iteration. Finding the whole frontier, instead of just the minimum-time point, is valuable because it allows users to select tradeoff points that meet job-level requirements (e.g., time deadlines, energy budgets) or perform dynamic adaptation to changing environments (e.g., stragglers) [15, 51]. Since the iteration frontier can be constructed by combining *microbatch* (forward and backward) frontiers (§4.4), Kareus's objective reduces to finding microbatch time–energy frontiers that Pareto-dominate those of prior works.

Decision variables. We consider three decision variables corresponding to the three key factors that influence time and energy (§3): (1) the number of SMs allocated to communication kernels; (2) kernel launch timing, which determines the execution order and overlap between kernels while respecting data dependencies; and (3) GPU frequency.

Solution space. Unfortunately, the combined solution space of these three factors is extremely large. For a typical Transformer-based LLM on an A100 GPU, this space comprises 85K candidate configurations (details in Appendix B). Each candidate in this large search space requires lengthy profiling for accurate energy measurements—on average, 13 seconds per candidate—for thermal stability (§5). An exhaustive search over the entire solution space can take up to 4,912 GPU-hours!

4.2 Partitioned Overlap Execution Model

To shrink the search space, we identify *repeating patterns* in computation and communication sequences, optimize them independently, and compose them into a global schedule.

Partition. Kareus groups kernels executing in repeating patterns into *partitions*. A partition consists of one communication kernel from one nanobatch and the longest contiguous sequence of computation kernels from the other nanobatch. There are no data dependencies between the communication kernel and the computation kernels in the same partition. This allows the communication kernel to overlap with any contiguous subsequence of computation kernels in the partition.

The second row of Figure 5 illustrates the repeating pattern of partitions. The two orange boxes (Attention–AllReduce partition) are identical, as are the two blue boxes (MLP–AllReduce partition), and both partition types repeat across all Transformer blocks.

Optimization overview. After automatically detecting the partitions, Kareus's optimizer first characterizes the time–energy frontier of each partition (§4.3), as shown in the fourth row of Figure 5. Each point on the frontier represents an efficient execution schedule for that partition, like the one depicted in the third row of Figure 5. The time–energy frontiers of all partitions are then combined to form the time–energy frontier of each forward and backward microbatch, and then combined again to form the global, iteration-level time–energy frontier (§4.4).

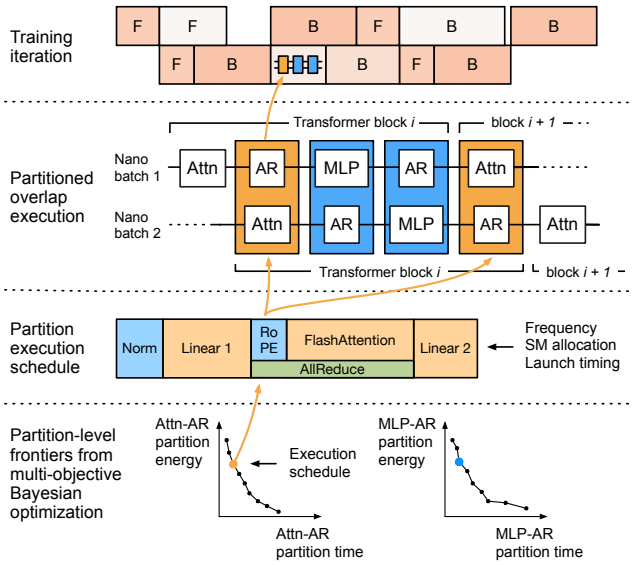


Figure 5: ***Kareus execution and optimization overview. The first row shows one training iteration with the 1F1B pipeline schedule. Inside each microbatch, partitions are sequentially executed; the second row illustrates this with the Transformer forward pass with tensor parallelism. The third row shows the execution schedule of the Attention–AllReduce partition. Each partition's execution schedule is chosen from the partition-level time–energy frontiers (fourth row) characterized by Kareus's optimization algorithm.***

4.3 Multi-Objective Bayesian Optimization

4.3.1 Bayesian Optimization Primer

For one partition, our optimization problem still involves a complex search space with mixed discrete and categorical variables. Importantly, the time and energy of each configuration have no known closed-form expression and can only be obtained through profiling. Bayesian Optimization (BO) is well-suited to this setting; it optimizes objectives over complex search spaces where the objective is unknown in form and costly to evaluate [19].

A BO algorithm has two key components. First, a *surrogate model* is trained on evaluated candidates to approximate the objective function. It is initialized with random candidates and updated as new candidates are evaluated. Second, an *acquisition function* uses the surrogate model to rank candidates and select the next one(s) to evaluate. BO then proceeds iteratively: it selects a batch of candidates using the acquisition function, evaluates them, updates the surrogate model, and repeats until convergence.

4.3.2 Kareus's MBO Algorithm

Although BO has been widely applied to systems problems [6, 11, 22, 52], it does not fit our setting as is. Standard BO targets a single objective, whereas we seek the Pareto frontier of time

and energy. We therefore design a Multi-objective Bayesian Optimization (MBO) algorithm with two tailored components: separate surrogate models for time and energy prediction, and a multi-pass candidate selection procedure that explores the frontier from multiple directions.

Surrogate model. We train two surrogate models: $\widehat{T}(x)$ for time and $\widehat{E}(x)$ for dynamic energy,[7] where x is a candidate configuration. Since time is primarily influenced by SM allocation and launch timing and dynamic energy by GPU frequency, the two models are largely orthogonal. Total energy consumption can then be computed as $\widehat{T}(x) \cdot P_{\text{static}} + \widehat{E}(x)$, where P_{static} is the GPU's static power consumption.

We adopt gradient-boosted decision trees (XGBoost) [10] as the surrogate model for two reasons. First, XGBoost training scales linearly with data points (versus cubic for Gaussian Processes), allowing fast retraining. Second, its tree-based structure handles discrete (GPU frequencies and SM allocations) and categorical (launch timing) parameters well.

Multi-pass candidate selection. Since our MBO aims to find solutions on the time–energy Pareto frontier rather than a single optimum, we design a *multi-pass* candidate selection procedure. At each iteration of MBO, we select a batch of k candidates to evaluate next. The batch consists of candidates selected based on multiple different acquisition functions that are designed to (1) cover complementary regions of the time–energy tradeoff space and (2) perform both exploration and exploitation to avoid over- or under-exploring certain regions.

Exploitation with hypervolume improvement. The acquisition functions for exploitation are based on *hypervolume improvement* (HVI), which quantifies how much a candidate can expand the current time–energy frontier toward lower time and energy (Figure 6). It is defined as

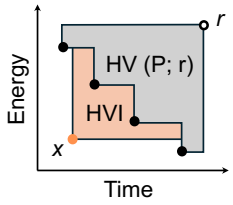


Figure 6: ***HVI.***

$$\text{HVI}(x) = \text{HV}\left(\mathcal{P} \cup \left\{\left(\widehat{T}(x), \text{Energy}(x)\right)\right\}; r\right) - \text{HV}(\mathcal{P}; r)$$

where $\mathcal{P}$ is the current set of candidates on the time–energy frontier, and r is a reference point set slightly worse (outward) than the worst observed points so that the hypervolume (HV) is computable. By switching the definition of $\text{Energy}(x)$ to total energy, dynamic energy, or static energy, each derived or taken as is from the surrogate models $\widehat{T}(x)$ and $\widehat{E}(x)$, we obtain three HVI acquisition functions. Each function guides frontier expansion in a distinct direction: dynamic energy favors lower-frequency candidates, static energy favors faster candidates, and total energy lies in the middle.

Exploration with uncertainty. To reduce the risk of under-exploring certain regions of the search space, we add an uncertainty-based acquisition function. We quantify uncertainty with *bootstrap ensembles*: multiple surrogate models trained on resampled datasets. The degree of *disagreement* between the surrogate models serves as a proxy for predictive uncertainty (i.e., how uncertain we are about the prediction). To promote exploration in either objective, we use the sum of per-objective standard deviations (rather than variances) as the acquisition score to prioritize candidates with high uncertainty.

[7] We use the hat notation to denote predictions, not true measurements.

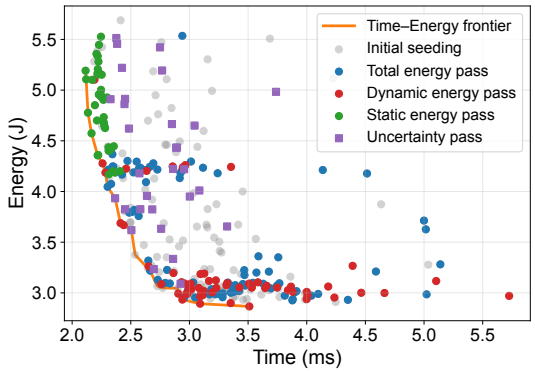


Figure 7: ***Multi-pass MBO in action with the Llama 3.2 3B model's MLP–AllReduce partition. The three exploitation passes (total energy, dynamic energy, static energy) and the exploration pass expand the efficient frontier in complementary directions, as shown by the colors of the points that land on the time–energy frontier.***

The full picture. Figure 7 illustrates frontier expansion for the Llama 3.2 3B model's MLP–AllReduce partition.[8]

Each pass expands the frontier in a complementary direction aligned with its acquisition function: dynamic energy pushes the frontier downward (lower energy), static energy leftward (lower time), and total energy toward the origin. In contrast, a single acquisition function as used in standard BO is fundamentally insufficient for MBO, as it only targets one direction. Algorithm 1 summarizes the overall Kareus MBO procedure for one partition, and this is repeated for each type of partition independently. Appendix C provides detailed settings of hyperparameters including initial random sample size, batch size, ensemble size, and stopping criteria.

4.4 Composing Frontiers

With each partition's time–energy frontier in hand, Kareus composes them into microbatch frontiers, and then into the iteration frontier.

Partition frontiers to microbatch frontier. A microbatch consists of multiple partitions executed sequentially. To construct the microbatch frontier, Kareus enumerates combinations of partition execution schedules (frequency, SM allocation, launch timing), sums their time and energy, and prunes suboptimal combinations. Algorithm 2 illustrates this process.

Two design decisions keep enumeration tractable. First, Kareus enforces a uniform GPU frequency across all partitions within a microbatch, since frequency switching takes several milliseconds and is non-negligible compared to the

[8] Microbatch size 8, sequence length 4K, and tensor parallelism degree 8.

Algorithm 1: Multi-pass multi-objective Bayesian Optimization for one partition

// Construct initial dataset for surrogate models
$\mathcal{D} \leftarrow N_{\text{init}}$ random candidates sampled and evaluated
for $b = 1, 2, \ldots, B_{\max}$ **do**
  Train surrogate models $\widehat{T}(x)$ and $\widehat{E}(x)$ on $\mathcal{D}$.
  // Exploitation: Hypervolume improvement
  foreach $x \in \mathcal{X} \setminus \mathcal{D}$ **do**
    Compute $\text{HVI}_{\text{tot}}(x)$, $\text{HVI}_{\text{dyn}}(x)$, $\text{HVI}_{\text{stat}}(x)$.
  // Exploration: Uncertainty via bootstrap ensemble
  for $m = 1$ **to** M **do**
    Train $\widehat{T}^m(x)$ and $\widehat{E}^m(x)$ on resampled $\mathcal{D}$
  foreach $x \in \mathcal{X} \setminus \mathcal{D}$ **do**
    $\text{Unc}(x) \leftarrow \sigma(\{\widehat{T}^m(x)\}) + \sigma(\{\widehat{E}^m(x)\})$
  // Multi-pass candidate selection
  $\mathcal{C} \leftarrow \text{TopK}(\{\text{HVI}_{\text{tot}}(x) : \forall x \in \mathcal{X} \setminus \mathcal{D}\}, k_1)$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \text{TopK}(\{\text{HVI}_{\text{dyn}}(x) : \forall x \in \mathcal{X} \setminus \mathcal{D}\}, k_2)$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \text{TopK}(\{\text{HVI}_{\text{stat}}(x) : \forall x \in \mathcal{X} \setminus \mathcal{D}\}, k_3)$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \text{TopK}(\{\text{Unc}(x) : \forall x \in \mathcal{X} \setminus \mathcal{D} \setminus \mathcal{C}\}, k - |\mathcal{C}|)$
  // Evaluate candidates and update dataset
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(x, T(x), E(x)) : x \in \mathcal{C}\}$
  // Stopping condition
  $\Delta \leftarrow$ Average HV improvement over R batches
  if $\Delta < \varepsilon$ **then**
    break
return GetFrontier($\mathcal{D}$)

latency of one partition. Second, partitions of the same type share the same SM allocation and launch timing. For instance, all Attention–AllReduce partitions in Figure 5 use identical configurations. Allowing per-partition configurations would grow the search space exponentially in the number of partitions with little benefit.

Microbatch frontiers to iteration frontier. The first row of Figure 5 shows an example of how forward (F) and backward (B) microbatches are scheduled across pipeline stages. Iteration time is the sum of microbatch latencies along the critical path, while iteration energy combines the energy of all microbatches and the static energy consumed during idle times. Kareus adopts Perseus's iterative algorithm [15] to construct the training iteration frontier from microbatch frontiers.

4.5 Generalizations

Multiple communication kernels. Section 4.2 illustrated partitions using tensor parallelism as a running example. Real workloads, however, present more complex patterns. When consecutive communication kernels appear (e.g., multiple All-Gather operations under context parallelism to aggregate key and value tensors [14, 33]), Kareus fuses them into a single kernel that shares an SM allocation. This reduces kernel launch overhead and keeps scheduling tractable. With this, any model and communication scheme reduces to alternating sequences of computations and a single (possibly fused) communication, fitting the partitioned overlap model.

Algorithm 2: Microbatch frontier construction

$\mathcal{C} \leftarrow \emptyset$ // All feasible (time, energy) pairs for microbatch
foreach $f \in$ GPU frequencies **do**
  // Cartesian product of configs across all partitions
  $\Theta \leftarrow \prod_{p \in \mathcal{P}} (\text{SM allocations} \times \text{launch timings})_p$
  foreach $\theta \in \Theta$ **do**
    // Accumulate time and energy for microbatch
    $T_m \leftarrow 0, E_m \leftarrow 0$
    foreach *partition* $p \in \mathcal{P}$ **do**
      $T_m \leftarrow T_m + T_p(f, \theta[p])$
      $E_m \leftarrow E_m + E_p(f, \theta[p])$
    foreach *non-partition component* c **do**
      $T_m \leftarrow T_m + T_c(f)$
      $E_m \leftarrow E_m + E_c(f)$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{(T_m, E_m)\}$
return GetFrontier($\mathcal{C}$)

Short consecutive memory-bound computations. When multiple short, memory-bound operations appear consecutively (e.g., `BiasDropoutAdd` followed by `Norm`), Kareus groups them into one logical operation. Treating them separately would expand the launch-timing search space with only marginal gains in solution quality.

Execution model switching. Partitioned overlap is not universally superior. When the amount of work within a microbatch is small (e.g., small models or small microbatch sizes), splitting the microbatch can further reduce arithmetic intensity, leading to compute underutilization and higher static power wastage. In these cases, sequential execution can be more energy-efficient. To capture this, Kareus also profiles each sequentially executed microbatch at each frontier and includes them as candidates when constructing the microbatch time–energy frontier. Thus, the final microbatch frontier will automatically be constructed with the better execution model.

GPU power model. Kareus adopts a simplified two-component GPU power model that decomposes GPU power into dynamic power and constant static power [15, 32, 41], which suffices for reasoning about GPU energy consumption in large model training. There are finer-grained models that further divide static power into chip leakage and constant power [29, 45], where chip leakage is sensitive to voltage. We leave extending Kareus to such models as future work.

5 Implementation

5.1 System Overview

Figure 8 shows an overview of the Kareus system. Given a workload, Kareus identifies partitions (❶; §4.2) and runs multi-objective Bayesian optimization per partition (❷; §4.3), evaluating candidates with the thermally stable profiler (§5.3).

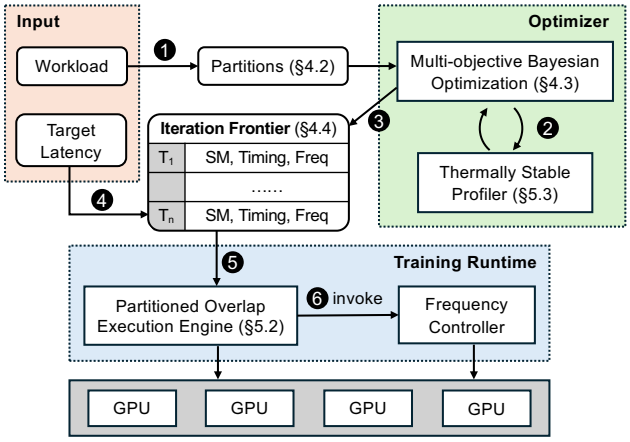


Figure 8: ***Kareus system overview.***

The per-partition frontiers are then composed into the iteration time–energy frontier (❸; §4.4). At runtime, given a target iteration latency, Kareus selects an execution schedule from the frontier (❹) and deploys it to the partitioned overlap execution engine (❺; §5.2), which invokes the GPU frequency controller to switch each GPU's frequency asynchronously as planned (❻).

5.2 Partitioned Overlap Execution Engine

Kareus's training execution engine builds on Megatron-LM [35], executing Transformer blocks as a sequence of partitions. Before each microbatch execution, Kareus loads the corresponding microbatch configuration, switches between Megatron-LM's default sequential execution and our partitioned overlap execution mode, and configures the SM allocation and launch timing for partitioned overlap execution. A custom `torch.autograd.Function` wraps all partitions of a Transformer block, allowing different partitions and partition schedules for forward and backward passes.

Communication kernels in Kareus are implemented with MSCCL++ [2,23], which provides fine-grained SM allocation control with grid size. Computation and communication run on separate CUDA streams to enable overlap, and CUDA events control launch timing. The GPU frequency controller is adapted from Perseus's open-source implementation [4,15].

5.3 Thermally Stable Profiling

Kareus measures the time and energy of a partition during MBO using Zeus [4,51], which internally uses NVML [3]. Accurate energy measurements require care; we present a detailed experimental analysis in Section 6.7.

Measurement window. NVML's sampling interval on NVIDIA GPUs is approximately 100 ms, so millisecond-scale measurements incur large errors. Kareus executes each partition repeatedly over a 5-second window, after which energy measurements stabilize.

Thermal cooldown. The power consumption of any hardware is temperature-dependent. Without cooldown between candidates, profiles of earlier partitions may heat up the GPU and bias subsequent measurements. Kareus inserts a 5-second cooldown period, which reliably brings the GPU below 32°C in our environment; the required duration depends on the server's cooling capability. In sum, profiling each candidate takes approximately 13 seconds in our setup, including initialization, warm-up, measurement, and cooldown.

6 Evaluation

We evaluate Kareus on 14 workloads and compare it against state-of-the-art baselines. Our key findings are as follows:

- Kareus achieves a superior time–energy frontier compared to prior approaches. In end-to-end training on real GPUs, it delivers up to 28.3% energy reduction under the same time budget and up to 27.5% time reduction under the same energy budget compared to the baselines (§6.2).
- In emulated large-scale training, Kareus consistently outperforms the baselines, achieving time and energy reductions comparable to real-world training results (§6.3).
- Ablation and sensitivity analyses show that jointly optimizing dynamic and static energy is necessary (§6.4) and is effective across a range of microbatch sizes (§6.5).
- Kareus's MBO algorithm constructs the time–energy frontier with reasonable overhead, and each pass in multi-pass candidate selection is indispensable (§6.6); the thermally stable profiler delivers stable energy measurements (§6.7).

6.1 Experimental Setup

Testbed. Experiments are conducted on 16 NVIDIA A100 40GB GPUs deployed across two AWS p4d.24xlarge instances. GPUs are fully connected intra-node via NVSwitch, and cross-node bandwidth is 400 Gbps.

Workloads. We evaluate Kareus on Llama 3.2 3B [33] and Qwen 3 1.7B [44] on the physical testbed, and perform large-scale emulation for Llama 3.3 70B [33]. For pipeline parallelism, we manually partition stages such that stages are as balanced as possible, following Perseus [15]. For context parallelism, we follow the scheme used in Llama 3 [33], where key–value tensors are collected across GPUs via AllGather. We use activation checkpointing to reduce memory pressure.

Baselines. We mainly compare against three baselines:

- **Megatron-LM ("M").** Baseline Megatron-LM [35] with the sequential execution model and maximum GPU frequency. Produces a single point on the time–energy plane.
- **Megatron-LM + Perseus ("M+P").** Perseus [15] applied to the above. This produces a time–energy frontier.
- **Nanobatching + Perseus ("N+P").** A training engine based on Megatron-LM that implements nanobatching, a special case of Kareus's partitioned overlap execution model (§3.2), integrated with Perseus like the above. This also produces a time–energy frontier.

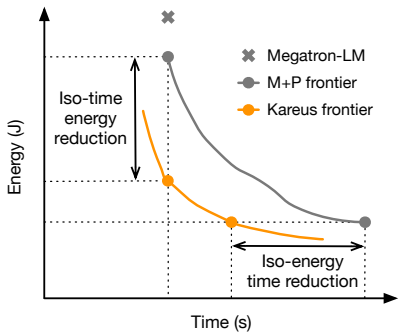


Figure 9: ***Illustrative example showing iteration time–energy frontiers. For the max-throughput comparison, we compare the leftmost points of each frontier with Megatron-LM. For frontier improvement, we report the iso-time energy reduction and iso-energy time reduction compared to Megatron-LM + Perseus.***

Metrics. Kareus's gain is fundamentally a superior time–energy tradeoff frontier, which cannot be fully captured by a single number. Therefore, we define two modes of comparison based on two viable use cases.

- **Max-throughput comparison.** When iteration time constraints (e.g., deadlines, stragglers) are not present, the training pipeline operates in maximum-throughput mode. For methods that produce a time–energy frontier, this means operating at the leftmost (lowest time) point. We set Megatron-LM as the baseline, and report iteration time and energy reduction (%) of M+P, N+P, and Kareus.
- **Frontier improvement.** We set M+P as the baseline, and define two metrics that quantify frontier improvements for N+P and Kareus (Figure 9): (1) *Iso-time energy reduction* (%): energy reduction with iteration time deadline set as M+P's minimum iteration time; and (2) *Iso-energy time reduction* (%): time reduction with iteration energy budget set as M+P's minimum iteration energy.

6.2 End-to-End Results

In this section, we evaluate the iteration time and energy performance on the testbed GPUs for the max-throughput (§6.2.1) and frontier improvement (§6.2.2) comparisons. The workload configurations (parallelism, microbatch size, and sequence length) can be found in Table 3.

6.2.1 Max-Throughput Comparison

Table 3 reports the iteration time and energy reductions achieved by all methods under the max-throughput regime. Overall, Kareus achieves up to 14.9% reduction in iteration time and 22.1% reduction in energy consumption compared to Megatron-LM, strictly outperforming the baselines on time and energy.

Time reduction. The time reduction mainly comes from reducing SM idle time caused by resource underutilization due to exposed communication or resource contention by poorly scheduled kernel overlaps. As a result, the time reduction trend largely follows the communication overhead induced by different model configurations. Compared to Megatron-LM, both overlap execution models achieve larger gains under tensor parallelism (TP) than under context parallelism with tensor parallelism (CP+TP). Moreover, increasing the microbatch size yields greater time reduction than increasing the sequence length, since the former introduces relatively higher communication overhead.

When compared to Nanobatching, Kareus provides larger additional gains under CP+TP than under TP alone, because fine-grained control of communication scheduling becomes more critical for complex communication patterns. Moreover, Nanobatching can significantly slow down computation (e.g., on Qwen 1.7B with CP2+TP4, microbatch size 8, and sequence length 4K), as it leads to GPU underutilization when the workload is small. In contrast, Kareus can automatically switch to the sequential execution model (§4.5).

Energy reduction. Energy reflects the combined effects of time and power, which is why energy reduction trends do not always track time reduction trends. In many cases, Nanobatching + Perseus reduces time but yields smaller energy reduction than Megatron-LM + Perseus, because overlap raises GPU utilization and thus also power, and the power increase offsets time reduction. In contrast, Kareus achieves larger energy reductions by simultaneously optimizing overlap and GPU frequency, enabling greater time reduction while also identifying opportunities to reduce power, as the following case studies show.

Case studies. We find an interesting case in Qwen 3 1.7B with TP8, microbatch size 8, and sequence length 4K. Both baselines operate the critical-path microbatch at 1,410 MHz (the maximum GPU frequency). However, Kareus discovers an overlap configuration that runs at 1,350 MHz yet attains the fastest latency, resulting in an additional 10.7% energy reduction over Nanobatching + Perseus. This happens because nanobatching overlap at 1,410 MHz improves resource utilization but raises instantaneous power, which triggers GPU frequency throttling. Thus, the time-averaged GPU frequency (which determines execution time) is close to 1,350 MHz, while the time-averaged dynamic power (which determines dynamic energy) remains closer to that of 1,410 MHz. Operating at a constant frequency provably consumes less energy than operating at a fluctuating frequency with the same average, which highlights the importance of jointly optimizing kernel scheduling and GPU frequency scaling (§3.3).

Figure 10 further illustrates representative partition execution schedules across microbatches of the aforementioned configuration. In the forward and backward Attention–AllReduce partitions, Kareus decides not to overlap `AllReduce` with memory-bound kernels (e.g., `Norm`) at a higher frequency (1,350 MHz); instead, it shifts the overlap to more memory-bound kernels at a lower frequency (1,290 MHz), consistent with the analysis in Section 3. For the MLP–AllReduce partitions, the selected configurations differ due to variations

Table 3: ***[Experiment] Iteration time and energy reductions (%) relative to Megatron-LM (higher is better) for all methods under the max-throughput configuration. Pipeline parallelism degree is fixed at 2, and number of microbatches is fixed at 8. `OOM` indicates that the GPU runs out of memory at the corresponding settings. Negative values indicate increased time or energy relative to the baseline.***

Model	Parallelism	ubatch Size	Sequence Length	Time Reduction (%)			Energy Reduction (%)			TFLOP/s /GPU
				Megatron-LM +Perseus	Nanobatching +Perseus	Kareus	Megatron-LM +Perseus	Nanobatching +Perseus	Kareus	
Llama 3.2 3B	TP8	8	4K	−0.3	8.5	12.3	10.0	15.5	19.6	104.8
		8	8K			OOM				
		16	4K			OOM				
Llama 3.2 3B	CP2TP4	8	4K	0.2	0.0	5.2	7.4	7.3	14.4	117.3
		8	8K	−1.4	1.5	6.2	11.5	7.3	14.9	138.0
		16	4K	−0.7	2.4	8.0	8.9	7.0	16.2	131.0
Qwen 3 1.7B	TP8	8	4K	−0.5	5.6	12.2	7.7	12.8	22.1	88.5
		8	8K	0.1	9.0	14.9	7.9	12.2	15.1	107.2
		16	4K	−0.1	9.4	14.8	7.6	13.4	17.3	97.0
Qwen 3 1.7B	CP2TP4	8	4K	−0.5	−20.4	−0.5	7.1	3.1	7.1	88.9
		8	8K	0.5	1.7	8.0	7.4	6.7	8.7	119.4
		16	4K	0.0	3.9	10.5	6.9	6.9	11.2	110.6

in kernel lengths and types. Kareus starts overlap from the beginning and allocates only 6 SMs to `AllReduce` at 1,350 MHz, which provides enough time to hide communication while leaving most SMs available for the long compute-bound `Linear 1` kernels.

6.2.2 Frontier Improvement

Table 4 reports the iso-time energy reduction and iso-energy time reduction for Nanobatching + Perseus and Kareus, which demonstrate the expansion of the time–energy frontier. Overall, Kareus achieves up to 28.3% iso-time energy reduction and 27.5% iso-energy time reduction compared to Megatron-LM + Perseus, establishing a strictly better time–energy tradeoff over Nanobatching + Perseus. Figure 11 shows a representative frontier comparison for Qwen 3 1.7B with CP2, TP4, microbatch size 16, and sequence length 4K. We present the complete frontier comparison plots for all model configurations in Appendix D.

Under the same time budget, where static energy is fixed, Kareus achieves higher GPU utilization and can therefore execute at lower GPU frequencies to complete the workload, reducing dynamic energy consumption. Conversely, under the same energy budget, Kareus minimizes static energy wastage, allowing more energy to be allocated to dynamic execution at higher GPU frequencies to complete the workload faster.

6.3 Large-Scale Emulation

In this section, we conduct large-scale emulation of Llama 3.3 70B based on smaller-scale profiling on our testbed hardware and compare against Megatron-LM + Perseus. We use the emulator from Perseus [15]. Results show that trends in time and energy reduction are consistent with those in Section 6.2 measured on testbed GPUs.

Methodology and parameters. For Kareus, we obtain the iteration-level time–energy frontier using the MBO algorithm described in Section 4 by composing partition-level frontiers.

Table 4: ***[Experiment] Iso-time energy reduction (%) and iso-energy time reduction (%) relative to Megatron-LM + Perseus for Nanobatching + Perseus and Kareus. "−" indicates that no configuration satisfies that constraint. For example, if Nanobatching + Perseus has a longer iteration time than Megatron-LM + Perseus at the leftmost point, then no iso-time point exists.***

				Iso-Time Energy Reduction (%)		Iso-Energy Time Reduction (%)	
				N+P	Kareus	N+P	Kareus
3B	TP8	8	4K	21.0	24.3	18.9	24.0
		8	8K		OOM		
		16	4K		OOM		
3B	CP2TP4	8	4K	–	17.4	7.0	13.9
		8	8K	4.0	12.4	4.4	16.4
		16	4K	6.6	20.4	5.1	12.5
1.7B	TP8	8	4K	16.8	26.8	17.8	27.5
		8	8K	20.0	23.1	17.7	23.1
		16	4K	20.4	28.3	19.9	26.7
1.7B	CP2TP4	8	4K	–	0.0	–	5.1
		8	8K	−0.7	15.0	−0.4	12.1
		16	4K	8.9	20.1	8.4	19.7

For Megatron-LM + Perseus, we profile the time and energy of each Transformer block using the same profiling methodology described in Section 5.3, and construct the iteration-level frontier. We perform strong scaling as we vary the number of GPUs (Table 5), while fixing the global batch size to 2048, which is adopted in the Llama 3 training [33]. We use pipeline parallelism degree 10 and tensor parallelism degree 8, with microbatch size 4 and sequence length 4K [33].

Max-throughput comparison. We report the max-throughput comparison results for Megatron-LM + Perseus and Kareus across different numbers of microbatches in Table 6, and plot the corresponding iteration-level time–energy frontiers in Appendix E. The energy reductions achieved by both methods are generally higher than those observed in the

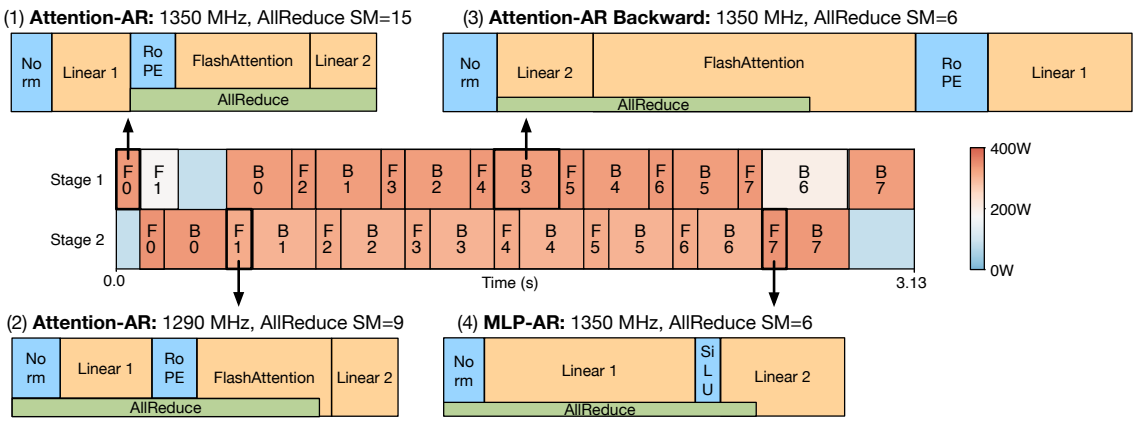


Figure 10: ***Case study of Qwen 3 1.7B with TP8, microbatch size 8, and sequence length 4K. The pipeline schedule is colored by average GPU power draw, and the partition execution schedules are drawn to scale for time. (1) and (2) compare the forward Attention–AllReduce partition at 1,350 MHz and 1,290 MHz, respectively. (3) and (4) show the backward Attention–AllReduce and forward MLP–AllReduce partitions at 1,350 MHz. Backward microbatch is partitioned following the definition in Section 4.2; `Norm` is treated as the first kernel because it follows the AllReduce kernel in the transformer block, and other kernels are ordered in reverse.***

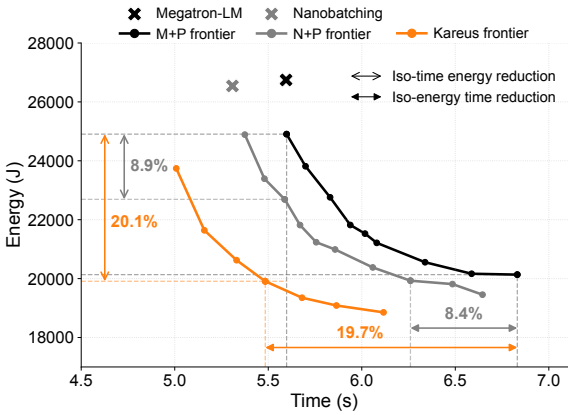


Figure 11: ***[Experiment] Representative iteration time–energy frontier comparison for Qwen 3 1.7B with CP2, TP4, microbatch size 16, and sequence length 4K.***

Table 5: ***Strong scaling configurations for large-scale emulation.***

# GPUs	# Pipelines	# Microbatches Per Pipeline	Global Batch Size
10240	128	16	2048
5120	64	32	
2560	32	64	
1280	16	128	

testbed experiments, because the larger number of pipeline stages and microbatches on non-critical paths can be slowed down to save dynamic energy. As the number of microbatches increases, the relative portion of pipeline bubbles during the warm-up and cooldown phases—which are normally reduced down to the lowest frequency—decreases, leading to a slight decrease in energy reduction. The time reduction achieved by Kareus is slightly lower than that of the TP=8 case in the testbed experiments, because a smaller microbatch size is used in this setting due to the GPU memory constraints.

Table 6: ***[Emulation] Iteration time and energy reduction (%) relative to Megatron-LM of Megatron-LM + Perseus and Kareus under the max-throughput configuration for Llama 3.3 70B.***

# Microbatches	Time Reduction (%)		Energy Reduction (%)	
	M+P	Kareus	M+P	Kareus
16	0.0	9.3	15.0	20.2
32	0.0	9.2	14.3	20.0
64	0.0	9.1	13.8	19.8
128	0.0	9.1	13.5	19.7

Table 7: ***[Emulation] Iso-time energy reduction (%) and iso-energy time reduction (%) relative to Megatron-LM + Perseus of Kareus for Llama 3.3 70B.***

	# Microbatches			
	16	32	64	128
Iso-Time Energy Reduction (%)	11.6	14.1	15.3	15.1
Iso-Energy Time Reduction (%)	19.1	16.8	16.4	16.0

Frontier improvement. We report the frontier improvement results for Kareus across different numbers of microbatches in Table 7. As the number of microbatches increases, the iso-time energy reduction improves, because the absolute iteration time grows, providing more slack for dynamic energy savings. In contrast, the iso-energy time reduction decreases, since static energy consumes a larger fraction of the fixed energy budget.

6.4 Ablation Study on Search Space

In this section, we conduct an ablation study to demonstrate the necessity of jointly optimizing dynamic and static energy in Kareus. We compare Kareus against three ablated variants: removing dynamic energy optimization (i.e., frequency scaling), removing static energy optimization (i.e., kernel scheduling with SM allocation and launch timing), and removing both (i.e., Nanobatching). We evaluate all systems on Qwen 3 1.7B with PP=2, TP=8, 8 microbatches of size

Table 8: ***[Experiment] Iteration time and energy increase (%) of ablated variants relative to Kareus under the max-throughput configuration.***

System	Time Inc. (%)	Energy Inc. (%)
Kareus w/o frequency	1.0	12.9
Kareus w/o kernel schedule	8.2	10.8
Nanobatching	7.8	20.6

8, and sequence length 4K, the same configuration used in end-to-end experiments (Tables 3 and 4).

Table 8 reports the iteration time and energy increases of each ablated variant relative to Kareus under the max-throughput configuration. Removing frequency scaling results in a 12.9% increase in energy, and removing kernel scheduling leads to a 10.8% increase in energy. As expected, removing either of the optimization dimensions fails to deliver the full optimization potential of Kareus, demonstrating the necessity of joint optimization of both dynamic and static energy.

6.5 Sensitivity to Microbatch Size

In this section, we further study how the microbatch size, an important training parameter that directly affects computation arithmetic intensity, the ratio of communication to computation, and pipeline imbalance, affects the effectiveness of Kareus. We choose Qwen 3 1.7B with TP=8 and sequence length 4K as in Section 6.4 and vary the microbatch size from 8 to 20.[9] Tables 9 and 10 report the max-throughput comparison and frontier improvement results, respectively; Appendix F shows the corresponding time–energy frontiers.

Overall, Kareus demonstrates consistent effectiveness across varying microbatch sizes. As microbatch size increases, Kareus achieves greater time reduction (Table 9), as communication–computation overlap more effectively utilizes SMs when decomposing microbatches into nanobatches. This is also why microbatch size 20 achieves the best iso-time and iso-energy reductions (Table 10). Max-throughput energy reduction (Table 9) shows a more complex trend for two reasons. First, microbatch size 8 has a pronounced energy reduction due to the same reason as the case study in Section 6.2.1, where Kareus discovers a more energy-efficient critical path execution schedule that runs at a lower frequency. Second, changing the microbatch size impacts pipeline stage imbalance, which affects the potential for energy savings by slowing down non-critical microbatches—this worked in favor of microbatch size 12 and 20 by increasing imbalance and allowing more energy savings.

6.6 MBO Analysis

Overhead and breakdown. Kareus's MBO (§4.3) obtains the time–energy frontier for each partition. In our testbed experiments, MBO for different partitions is conducted in parallel and takes 2 hours (32 GPU hours) on average. This overhead is substantially lower than that of exhaustive search (307 hours, as discussed in Section 4.1), and is negligible compared to typical end-to-end training times (e.g., 54 days for Llama 3 [33]).

The overhead of MBO is dominated by thermally stable profiling of candidates (§5.3), accounting for 97% of the total overhead. For a typical partition in our setting, which evaluates a batch of 32 candidates per MBO iteration, thermally stable profiling takes 6.9 minutes on average, while surrogate model training and acquisition function evaluation takes 11 seconds on average.

Multi-pass candidate selection. After random candidate initialization, MBO selects candidates across four passes: total energy pass, dynamic energy pass, static energy pass, and uncertainty pass (§4.3). On average, random initialization contributes 35% of the selected candidates—they happen to be on the frontier. The other 65% of the selected candidates are discovered by the total energy pass, dynamic energy pass, static energy pass, and uncertainty pass, each 25%, 22%, 12%, and 6% of the selected candidates, respectively. All passes contribute a non-negligible portion of the final candidates.

Table 9: ***[Experiment] Iteration time and energy reduction (%) relative to Megatron-LM of Megatron-LM + Perseus and Kareus across different microbatch sizes.***

ubatch Size	Time Reduction (%)		Energy Reduction (%)	
	M+P	Kareus	M+P	Kareus
8	-0.5	12.2	7.7	22.1
12	-0.4	14.7	9.4	17.6
16	-0.1	14.8	7.6	17.3
20	-0.2	18.1	9.4	18.8

Table 10: ***[Experiment] Iso-time energy reduction (%) and iso-energy time reduction (%) relative to Megatron-LM + Perseus of Kareus across different microbatch sizes.***

	Microbatch Size			
	8	12	16	20
Iso-Time Energy Reduction (%)	26.8	28.6	28.3	29.8
Iso-Energy Time Reduction (%)	27.5	27.3	26.7	28.8

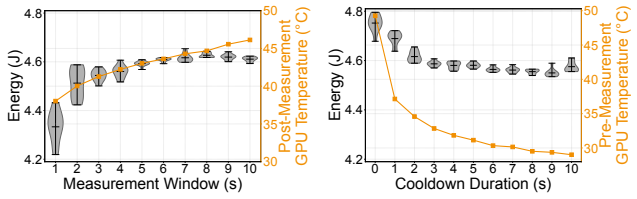
6.7 Thermally Stable Profiler

During MBO, Kareus adopts thermally stable profiling to measure the time and energy of each candidate (§5.3). We conduct an experimental study using the Attention–AllReduce partition of Llama 3.2 3B on 8 NVIDIA A100 GPUs[10] to analyze the impact of measurement window and cooldown duration on the energy measurement accuracy.

Measurement duration. We fix the cooldown duration to 5 seconds and vary the measurement window from 1 second to 10 seconds, each with 10 repeated profiling trials. Figure 12a shows the distribution of measured energy consumption under different measurement windows, together with the average GPU temperature *after* each measurement. With very short

[9] Larger microbatch sizes are not evaluated due to GPU memory capacity.

[10] Batch size is 4 and sequence length is 4K, with tensor parallelism degree 8, running at 1,410 MHz.



(a) *Measurement duration* (b) *Cooldown duration*

Figure 12: ***Impact of changing (a) measurement duration and (b) cooldown duration for the Thermally Stable Profiler. We report the distribution of energy across 10 repeated trials, along with the average GPU temperature before and after measurement.***

measurement windows (e.g., below 2 seconds), energy measurements exhibit large variability due to the 100 ms NVML counter update interval, and have a lower mean because the GPU has not fully warmed up. Energy measurement stabilizes from 5 seconds onward, which is why we chose 5 seconds as our measurement duration.

Cooldown duration. Cooldown duration is the waiting time between profiling consecutive partitions. We fix the measurement duration to 5 seconds and vary the cooldown duration from 0 second (no cooldown) to 10 seconds, with 10 repeated profiling trials. Figure 12b shows the distribution of measured energy consumption under different cooldown durations, along with the average GPU temperature *before* each measurement. We observe that mean energy consumption strongly correlates with the GPU temperature at the start of measurement, making sufficient cooldown essential for accurate measurement. Temperature and measurements stabilize from 5 seconds onward, so we cool down for 5 seconds between partitions. We note that the optimal cooldown duration would require tuning for different computing environments.

7 Related Work

7.1 ML Energy Optimization

The energy consumption of ML workloads has attracted extensive attention, prompting a growing body of work on energy measurement and optimization [13, 15–17, 24, 40, 43, 51, 53]. In particular, the time–energy tradeoff frontier has been a key tool for reasoning about performance and energy efficiency. For LLM serving, DynamoLLM [43] explores time–energy tradeoffs via model parallelism and GPU frequency scaling under latency constraints. The ML.ENERGY benchmark [16] characterizes the time–energy frontier across deployment configurations to guide latency-aware energy optimization. On the training side, Zeus [51] identifies job-level time–energy tradeoffs on recurring training workloads. Perseus [15] dynamically scales GPU frequency per microbatch to optimize energy under pipeline iteration time deadlines. However, these approaches often yield suboptimal frontiers because they overlook the impact of low-level execution scheduling, such as communication and computation overlap. Jayaweera et al. [27] propose energy-aware tile size selection for GPU kernels, but do not consider interactions with system-level latency targets, limiting the potential for energy optimization.

7.2 Communication Scheduling

As model scale grows, communication incurs increasingly significant overhead, motivating extensive work on overlapping communication with computation [8, 9, 12, 18, 20, 26, 28, 34, 36, 39, 48–50, 55–57]. Nanobatching is a key technique that removes data dependencies and enables overlap opportunities. In LLM serving, NanoFlow [57], TokenWeave [20], and vLLM Dual Batch Overlap [48] improve throughput via nanobatching with fine-grained kernel overlap. For training, DeepSpeed Domino [49] applies nanobatching to overlap tensor-parallel communication, while DeepSeek DualPipe [18] pipelines forward and backward nanobatches for communication overlap. Concerto [12] optimizes communication scheduling from a compiler perspective and adopts partial nanobatching when data dependencies are present. Other hand-crafted kernels fuse communication into computation [8, 36, 55, 56], but are tightly tied to specific parallelism strategies and operators. For example, Megatron-LM supports tensor-parallel communication overlap [36], which relies on its native sequence parallelism [30]. From the energy perspective, these approaches reduce static energy by reducing time, but do not explicitly optimize energy in their design. Kareus reveals the energy impact of communication scheduling and generalizes the nanobatching strategy to enable fine-grained SM allocation and launch-timing control, further minimizing static energy while jointly reducing dynamic energy by optimizing GPU frequency.

8 Conclusion

We present Kareus, an execution-aware energy optimizer for large model training. Kareus demonstrates that SM allocation, kernel launch timing, and GPU frequency jointly influence static and dynamic energy, and that the optimal time–energy frontier cannot be attained by optimizing them in isolation. Building on this insight, Kareus introduces a partitioned overlap execution model and a multi-objective Bayesian optimization framework that jointly searches over execution schedules, exposing a strictly better time–energy tradeoff frontier than prior systems.

Looking forward, we believe energy-aware execution scheduling should be a first-class concern in ML systems, not an afterthought. As AI scaling continues and energy constraints tighten for gigawatt-scale AI datacenters, systems that co-optimize computation, communication, and power will be essential, not just for cost savings but for enabling training runs that would otherwise be infeasible. Kareus takes a step in this direction, and we hope it spurs further work on energy as a core system design metric.

Acknowledgements

We would like to thank the OSDI reviewers, our shepherd, and SymbioticLab members for their insightful feedback. This work was supported in part by NSF grants CCF-2450085 and CNS-2106184, DARPA ML2P Award HR0011-26-9-E190, and by grants from Cisco, Ford, Mozilla Foundation, and Laude Institute. Jae-Won Chung is additionally supported by the Kwanjeong Educational Foundation.

References

[1] How much electricity does an American home use? https://www.eia.gov/tools/faqs/faq.php?id=97&t=3.

[2] MSCCL++. https://github.com/microsoft/mscclpp.git.

[3] NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml.

[4] Zeus. https://github.com/ml-energy/zeus.

[5] Sami Alabed, Daniel Belov, Bart Chrzaszcz, Juliana Franco, Dominik Grewe, Dougal Maclaurin, James Molloy, Tom Natan, Tamara Norman, Xiaoyue Pan, Adam Paszke, Norman A. Rink, Michael Schaarschmidt, Timur Sitdikov, Agnieszka Swietlik, Dimitrios Vytiniotis, and Joel Wee. PartIR: Composing SPMD partitioning strategies for machine learning. In *ASPLOS*, 2025.

[6] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, 2017.

[7] CBRE. Global data center trends 2025. https://www.cbre.com/insights/reports/global-data-center-trends-2025, 2025.

[8] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, Haibin Lin, Xin Jin, and Xin Liu. FLUX: Fast software-based communication overlap on GPUs through kernel fusion. *arXiv preprint arXiv:2406.06858*, 2024.

[9] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *ASPLOS*, 2024.

[10] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, 2016.

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *NeurIPS*, 2018.

[12] Shenggan Cheng, Shengjie Lin, Lansong Diao, Hao Wu, Siyu Wang, Chang Si, Ziming Liu, Xuanlei Zhao, Jiangsu Du, Wei Lin, and Yang You. Concerto: Automatic communication optimization and scheduling for large-scale deep learning. In *ASPLOS*, 2025.

[13] Sangjin Choi, Inhoe Koo, Jeongseob Ahn, Myeongjae Jeon, and Youngjin Kwon. EnvPipe: Performance-preserving DNN training framework for saving energy. In *ATC*, 2023.

[14] Weiwei Chu, Xinfeng Xie, Jiecao Yu, Jie Wang, Amar Phanishayee, Chunqiang Tang, Yuchen Hao, Jianyu Huang, Mustafa Ozdal, Jun Wang, Vedanuj Goswami, Naman Goyal, Abhishek Kadian, Andrew Gu, Chris Cai, Feng Tian, Xiaodong Wang, Min Si, Pavan Balaji, Ching-Hsiang Chu, and Jongsoo Park. Scaling Llama 3 training with efficient parallelism strategies. In *ISCA*, 2025.

[15] Jae-Won Chung, Yile Gu, Insu Jang, Luoxi Meng, Nikhil Bansal, and Mosharaf Chowdhury. Reducing energy bloat in large model training. In *SOSP*, 2024.

[16] Jae-Won Chung, Jeff J. Ma, Ruofan Wu, Jiachen Liu, Oh Jun Kweon, Yuxuan Xia, Zhiyu Wu, and Mosharaf Chowdhury. The ML.ENERGY benchmark: Toward automated inference energy measurement and optimization. In *NeurIPS Datasets and Benchmarks*, 2025.

[17] Jae-Won Chung, Ruofan Wu, Jeff J. Ma, and Mosharaf Chowdhury. Where do the Joules go? diagnosing inference energy consumption. *arXiv preprint arXiv:2601.22076*, 2026.

[18] DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[19] Peter I. Frazier. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[20] Raja Gond, Nipun Kwatra, and Ramachandran Ramjee. TokenWeave: Efficient compute-communication overlap for distributed LLM inference. In *MLSys*, 2026.

[21] James Hamilton. Constraint-driven innovation (CIDR 2024 keynote talk). https://mvdirona.com/jrh/talksandpapers/JamesHamiltonCIDR2024.pdf, 2024.

[22] Erik Orm Hellsten, Artur L. F. Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. BaCO: A fast and portable Bayesian compiler optimization framework. In *ASPLOS*, 2023.

[23] Changho Hwang, Peng Cheng, Roshan Dathathri, Abhinav Jangda, Saeed Maleki, Madan Musuvathi, Olli Saarikivi, Aashaka Shah, Ziyue Yang, Binyang Li, Caio Rocha, Qinghua Zhou, Mahdieh Ghazimirsaeed, Sreevatsa Anantharamu, and Jithin Jose. MSCCL++: Rethinking GPU communication abstractions for AI inference. In *ASPLOS*, 2026.

[24] Md. Monzurul Amin Ifath and Israat Haque. Characterizing performance–energy trade-offs of large language models in multi-request workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2026.

[25] Insu Jang, Runyu Lu, Nikhil Bansal, Ang Chen, and Mosharaf Chowdhury. Efficient distributed MLLM training with Cornstarch. In *ICML*, 2026.

[26] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *ASPLOS*, 2022.

[27] Malith Jayaweera, Martin Kong, Yanzhi Wang, and David Kaeli. Energy-aware tile size selection for affine programs on GPUs. In *CGO*, 2024.

[28] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *NSDI*, 2024.

[29] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G Rogers, Tor M Aamodt, and Nikos Hardavellas. AccelWattch: A power modeling framework for modern GPUs. In *MICRO*, 2021.

[30] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *MLSys*, 2023.

[31] Helen Kou and Nathalie Limandibhratha. Power for AI: Easier said than built. https://about.bnef.com/insights/commodities/power-for-ai-easier-said-than-built/, 2025.

[32] Jingwen Leng, Tayler H. Hetherington, Ahmed ElTantawy, Syed Zohaib Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling energy optimizations in GPGPUs. In *ISCA*, 2013.

[33] AI Meta Llama Team. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[34] Runyu Lu, Shiqi He, Wenxuan Tan, Shenggui Li, Ruofan Wu, Jeff J. Ma, Ang Chen, and Mosharaf Chowdhury. TetriServe: Efficiently serving mixed DiT workloads. In *ASPLOS*, 2026.

[35] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *SC*, 2021.

[36] NVIDIA. Megatron Core tp_comm_overlap. https://docs.nvidia.com/megatron-core/developer-guide/0.17.0/apidocs/core/core.model_parallel_config.html#core.model_parallel_config.ModelParallelConfig.tp_comm_overlap, 2026.

[37] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[38] Ryan Prescott. Advanced API performance: SetStablePowerState. https://developer.nvidia.com/blog/advanced-api-performance-setstablepowerstate/, 2022.

[39] Xinwei Qiang, Yue Guan, Zhengding Hu, Keren Zhou, Yufei Ding, and Adnan Aziz. Syncopate: Efficient multi-GPU AI kernels via automatic chunk-centric compute-communication overlap. In *OSDI*, 2026.

[40] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. Power-aware deep learning model serving with μ-Serve. In *ATC*, 2024.

[41] Saurabhsingh Rajput, Alexander Brandt, Vadim Elisseev, and Tushar Sharma. FlipFlop: A static analysis-based energy optimization framework for GPU kernels. In *ICSE*, 2026.

[42] SemiAnalysis. InferenceMAX: Open source inference benchmarking. https://newsletter.semianalysis.com/p/inferencemax-open-source-inference, 2025.

[43] Jovan Stojkovic, Chaojie Zhang, Inigo Goiri, Josep Torrellas, and Esha Choukse. DynamoLLM: Designing LLM inference clusters for performance and energy efficiency. In *HPCA*, 2025.

[44] Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[45] Brandon Tran, Matthias Maiterth, Woong Shin, Matthew D. Sinclair, and Shivaram Venkataraman. Wattchmen: Watching the wattchers – high fidelity, flexible GPU energy modeling. In *ICS*, 2026.

[46] U.S. Energy Information Administration (EIA). Capital cost and performance characteristics for utility-scale electric power generating technologies. https://www.eia.gov/analysis/studies/powerplants/capitalcost/pdf/capital_cost_AEO2025.pdf, 2024.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

[48] vLLM Team. Dual Batch Overlap. https://docs.vllm.ai/en/v0.20.0/design/dbo/, 2026.

[49] Guanhua Wang, Chengming Zhang, Zheyu Shen, Ang Li, and Olatunji Ruwase. Domino: Eliminating communication in LLM training via generic tensor slicing and overlapping. *arXiv preprint arXiv:2409.15241*, 2024.

[50] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *ASPLOS*, 2023.

[51] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. Zeus: Understanding and optimizing GPU energy consumption of DNN training. In *NSDI*, 2023.

[52] Yiwen Zhang, Xumiao Zhang, Ganesh Ananthanarayanan, Anand Iyer, Yuanchao Shu, Victor Bahl, Z. Morley Mao, and Mosharaf Chowdhury. Vulcan: Automatic query planning for live ML analytics. In *NSDI*, 2024.

[53] Zhendong Zhang, Foteini Strati, and Ana Klimovic. Characterizing energy and performance for distributed training of large language models. In *EuroMLSys*, 2026.

[54] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *OSDI*, 2022.

[55] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, Yifan Guo, Ningxin Zheng, Ziheng Jiang, Xinyi Di, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, Liqiang Lu, Yun Liang, Jidong Zhai, and Xin Liu. Triton-distributed: Programming overlapping kernels on distributed AI systems with the Triton compiler. *arXiv preprint arXiv:2504.19442*, 2025.

[56] Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, and Xin Liu. TileLink: Generating efficient compute-communication overlapping kernels using tile-centric primitives. In *MLSys*, 2025.

[57] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. NanoFlow: Towards optimal large language model serving throughput. In *OSDI*, 2025.

A Energy Efficiency of Constant Frequency

Below, we formalize why steady GPU frequency minimizes energy at a fixed average frequency.

Theorem 1 (Energy Efficiency of Constant Frequency)**.** *Let $f(t)$ denote the GPU frequency over a time interval $[0,T]$, and let $\bar{f} = \frac{1}{T}\int_0^T f(t)\,dt$ be its time-average. Under the following assumptions:*

1. *Dynamic power scales cubically with frequency: $P_{dyn}(t) = k \cdot f(t)^3$ for some constant $k > 0$.*
2. *Static power is constant: $P_{static}(t) = P_s$ for some constant $P_s \geq 0$.*
3. *Execution time depends only on average frequency: workloads with the same average frequency $\bar{f}$ complete in the same time T. This is supported by empirical observations from Kareus optimization results.*

Then the total energy consumption is minimized when frequency is held constant at $\bar{f}$.

Proof. Total energy is the sum of dynamic and static energy:

$$E_{\text{total}} = E_{\text{dyn}} + E_{\text{static}} = \int_0^T k \cdot f(t)^3\,dt + P_s \cdot T.$$

By Assumption 3, T is identical for both the fluctuating and constant frequency cases. Thus, static energy $E_{\text{static}} = P_s \cdot T$ is equal in both cases.

For dynamic energy, consider the function $g(x) = x^3$, which is convex for $x > 0$ since $g''(x) = 6x > 0$. By Jensen's Inequality:

$$g(\bar{f}) = \bar{f}^3 \leq \frac{1}{T}\int_0^T f(t)^3\,dt.$$

Multiplying both sides by kT:

$$E_{\text{dyn}}^{\text{constant}} = kT \cdot \bar{f}^3 \leq k\int_0^T f(t)^3\,dt = E_{\text{dyn}}^{\text{fluctuating}}.$$

Equality holds if and only if $f(t) = \bar{f}$ almost everywhere, i.e., when frequency is constant. Therefore:

$$E_{\text{total}}^{\text{constant}} \leq E_{\text{total}}^{\text{fluctuating}},$$

with strict inequality when $f(t)$ varies over time. □

Implications for GPU frequency scaling. This theorem explains why frequency fluctuation due to power limits can be energy-inefficient. When a GPU operates at a high frequency and triggers power throttling, the instantaneous frequency fluctuates, while the time-averaged frequency remains close to what could be achieved by operating steadily at a lower frequency. Consequently, the execution time—and thus static energy consumption—are the same as those under steady lower-frequency operation. However, this fluctuation incurs higher dynamic energy consumption because dynamic power is a strictly convex function of frequency. By Jensen's Inequality, any variance in frequency increases the expected dynamic energy consumption.

B Global Solution Space

The global solution space of combining GPU frequency, SM allocation, and kernel launch timing is extremely large. In this section, we illustrate the space using the NVIDIA A100 GPU as an example.

GPU frequencies. Supported GPU frequencies by A100 are from 210 MHz to 1,410 MHz at a stride of 15 MHz. We restrict the search space to 900 MHz–1,410 MHz (35 choices) because lowering the frequency below 900 MHz no longer reduces energy.[11]

SM allocations. A100 has 108 SMs, and we restrict the search space to up to 30 SMs because allocating additional SMs beyond this empirically no longer improved communication latency. This leaves 30 choices for SM allocation.

Kernel launch timing. Finally, given a sequence of computation and communication kernels, possible execution orders can be expressed as a recurrence relation, which enumerates the number of subproblems. We elaborate on this formulation in the following paragraphs. In summary, for a typical LLM composed of Transformer blocks, this can result in 81 possible groupings, leading to a *global solution space* with in total 85,050 candidates.

Formulation of launch timing. Kareus generalizes the nanobatching scheme to eliminate dependencies between computation and communication, which enables overlap and results in two operation sequences:

$$\mathcal{S}_1 = \{O_0, O_1, \ldots, O_i\}, \qquad \mathcal{S}_2 = \{O'_0, O'_1, \ldots, O'_j\}.$$

where operations within the same sequence must be executed in order, while operations across sequences are independent and may overlap. We regard the consecutive communication operations as a single operation. The time–energy frontier of the launch timing schedule can be formulated as a dynamic programming (DP) recurrence:

$$\mathcal{P}(i,j) = \min_{\text{Pareto}} \begin{cases} (E(O_i) + E(\mathcal{P}(i+1,j)),\ T(O_i) + T(\mathcal{P}(i+1,j))) \\ (E(O'_j) + E(\mathcal{P}(i,j+1)),\ T(O'_j) + T(\mathcal{P}(i,j+1))) \\ (E(O_i \parallel O'_{j..j+k}),\ T(O_i \parallel O'_{j..j+k})) + \mathcal{P}(i+1,j+k) \\ (E(O'_j \parallel O_{i..i+k}),\ T(O'_j \parallel O_{i..i+k})) + \mathcal{P}(i+k,j+1) \end{cases}$$

Here, $O_i \parallel O'_{j..j+k}$ means overlapping operation O_i with a subsequence $\{O'_j, \ldots, O'_{j+k}\}$; the notation is symmetric for the opposite case, $O'_j \parallel O_{i..i+k}$. The cost functions $E(\cdot)$ and $T(\cdot)$ account for the interference when overlapping.

This recurrence enumerates the time and energy of all single operations and all feasible overlap patterns between the

[11] Power reduction slows down but time increases, leading to higher energy.

two sequences. For a typical Transformer block with 9 computation operations and 1 AllReduce, if we restrict overlap to occur only between communication and computation and cap the maximum overlap length at 9 (assuming the communication is no longer than a full Transformer block), there are 81 possible overlap patterns. Including the non-overlapped cases, this yields a total of 91 subproblems.

C Adaptation of MBO Hyperparameters

In this section, we describe how Kareus configures MBO hyperparameters. Some are adapted across different partitions based on partition complexity, while others remain fixed.

Search space. For GPU frequency, we restrict the search space to 900–1,410 MHz with a stride of 30 MHz, since the time and energy differences between adjacent 15 MHz settings are marginal. For SM allocation, the search space is determined by the communication group size. If the number of GPUs in a communication group is fewer than 4, we search SM allocations from 1 to 20 with a stride of 1. If the group size is 4 or larger (e.g., the common cases of 4 and 8), we search SM allocations from 3 to 30 with a stride of 3. For launch timing, we enumerate all computation operators within a partition and exclude options that always lead to exposed communication, such as launching AllReduce from Linear2 in Figure 3a.

Sample size. Our goal is to control the total profiling time while preserving the quality of the time–energy frontier. We adopt a relatively large initialization set for informative exploration and scale the batch search budget based on partition complexity. Specifically, we classify partitions into three categories: *small* partitions containing only one computation, *medium* partitions containing 2–3 computations, and *large* partitions containing more than three computations. We then configure the sample sizes as follows: initial sample size $N_{\text{init}} = 36$, maximum number of batches $B_{\max} = 3$, batch size $k = 16$ for small partition; $N_{\text{init}} = 48$, $B_{\max} = 4$, $k = 16$ for medium partition; $N_{\text{init}} = 96$, $B_{\max} = 4$, $k = 32$ for large partition. We set the proportions of the total energy pass, dynamic energy pass, static energy pass, and uncertainty pass to 0.4, 0.2, 0.2, and 0.2, respectively. As shown in Section 6.6, the MBO overhead of Kareus is controlled within two hours on average.

XGBoost hyperparameters. Since the configuration space is three-dimensional, we adopt XGBoost hyperparameter settings commonly used for low-dimensional regression. To ensure fast and stable convergence while avoiding overfitting, we use relatively shallow trees with $max\_depth = 6$, a high learning rate $\eta = 0.3$, and a modest model capacity with $num\_boost\_round = 100$. For the bootstrap ensemble, we set the ensemble size to 5, use a bootstrap sampling fraction of 0.8, and vary the random seed across bootstrap resamples.

HV reference point. At each batch iteration, we compute the HV reference point using values slightly worse than the worst observed measurements to ensure boundedness. Specifically, we set $r = \big(1.1 \times \max T(x),\ 1.1 \times \max E(x)\big)$.

Stopping conditions. The MBO algorithm is terminated when either a sufficient number of samples has been collected or the objective has converged, according to the following criteria: (1) Stop after a fixed number of batches $B_{\max}$. (2) After each batch, compute the dominated hypervolume of the measured frontier. Stop if the moving average of the relative HV improvement over the last R batches falls below ε.

We set the window size in the stopping criterion to $R = 2$ and the convergence threshold to $\varepsilon = 10^{-3}$. Energy and time are normalized throughout the MBO process, ensuring stable stopping behavior across partitions.

D End-to-End Results

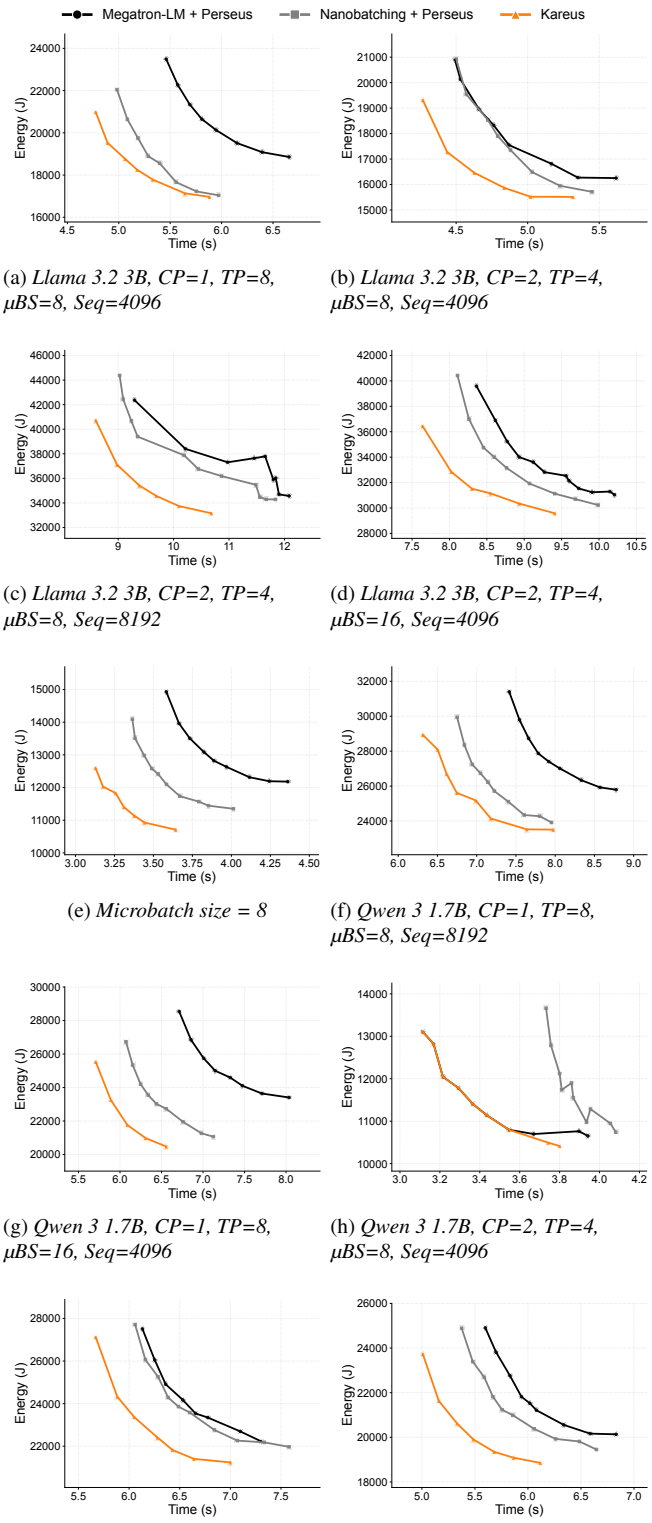
Figure 13 shows the iteration time–energy frontiers measured on the testbed GPUs for all model configurations, complementing the evaluation in Section 6.2.2.

E Large-Scale Emulation

Figure 14 shows the iteration time–energy frontiers in large-scale emulation for Llama 3.3 70B across different numbers of microbatches, complementing the evaluation in Section 6.3.

F Sensitivity to Microbatch Size

Figure 15 shows the iteration time–energy frontiers measured on the testbed GPUs for Qwen 3 1.7B with TP=8 and sequence length 4K across different microbatch sizes, complementing the sensitivity study in Section 6.5.



(a) *Llama 3.2 3B, CP=1, TP=8, μBS=8, Seq=4096*

(b) *Llama 3.2 3B, CP=2, TP=4, μBS=8, Seq=4096*

(c) *Llama 3.2 3B, CP=2, TP=4, μBS=8, Seq=8192*

(d) *Llama 3.2 3B, CP=2, TP=4, μBS=16, Seq=4096*

(e) *Microbatch size = 8*

(f) *Qwen 3 1.7B, CP=1, TP=8, μBS=8, Seq=8192*

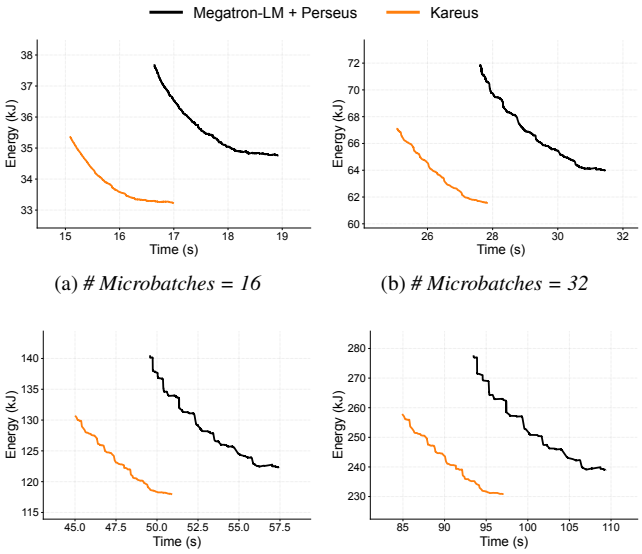
(g) *Qwen 3 1.7B, CP=1, TP=8, μBS=16, Seq=4096*

(h) *Qwen 3 1.7B, CP=2, TP=4, μBS=8, Seq=4096*

(i) *Qwen 3 1.7B, CP=2, TP=4, μBS=8, Seq=8192*

(j) *Qwen 3 1.7B, CP=2, TP=4, μBS=16, Seq=4096*

Figure 13: ***[Experiment] Itertaion time–energy frontiers of Megatron-LM + Perseus, Nanobatching + Perseus, and Kareus for all model configurations. CP: Context Parallelism, TP: Tensor Parallelism, μBS: Microbatch Size, Seq: Sequence Length.***



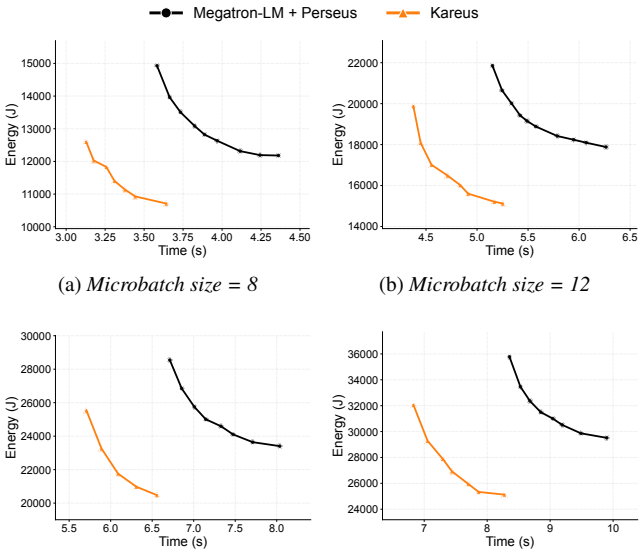
(a) *# Microbatches = 16*

(b) *# Microbatches = 32*

(c) *# Microbatches = 64*

(d) *# Microbatches = 128*

Figure 14: ***[Emulation] Iteration time–energy frontiers of Megatron-LM + Perseus and Kareus for Llama 3.3 70B across different numbers of microbatches.***



(a) *Microbatch size = 8*

(b) *Microbatch size = 12*

(c) *Microbatch size = 16*

(d) *Microbatch size = 20*

Figure 15: ***[Experiment] Iteration time–energy frontiers of Megatron-LM + Perseus and Kareus for Qwen 3 1.7B across different microbatch sizes.***