Reducing Energy Bloat in Large Model Training

Jae-Won Chung[1] Yile Gu[1,2] Insu Jang[1] Luoxi Meng[1,3] Nikhil Bansal[1] Mosharaf Chowdhury[1]
[1]University of Michigan [2]University of Washington [3]University of California, San Diego

Abstract

Training large AI models on numerous GPUs consumes a massive amount of energy, making power delivery one of the largest limiting factors in building and operating datacenters for AI workloads. However, we observe that not all energy consumed during training directly contributes to end-to-end throughput; a significant portion can be removed without slowing down training. We call this portion *energy bloat*.

In this work, we identify two independent sources of energy bloat in large model training and propose Perseus, a training system that mitigates both. To do this, Perseus obtains the time–energy tradeoff frontier of a large model training job using an efficient graph cut-based algorithm, and schedules computation energy consumption across time to reduce both types of energy bloat. Evaluation on large models, including GPT-3 and Bloom, shows that Perseus reduces the energy consumption of large model training by up to 30% without any throughput loss or hardware modification.[1]

CCS Concepts: • **Computer systems organization**; • **Computing methodologies → Machine learning**; • **Software and its engineering → Power management**;

Keywords: Energy-efficiency, datacenter power management, straggler, distributed training, large model training

ACM Reference Format:
Jae-Won Chung, Yile Gu, Insu Jang, Luoxi Meng, Nikhil Bansal, and Mosharaf Chowdhury. 2024. Reducing Energy Bloat in Large Model Training. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3694715.3695970

1 Introduction

As deep neural networks (DNNs) continue to grow in model and dataset size [32, 40], the energy consumption of large model training is increasing as well. For instance, training GPT-3 [8] reportedly consumed 1.3 GWh [63]. Then, this was dwarfed by Amazon's training of a 200B model, which consumed about 11.9 GWh [29]—enough to power more than 1,000 average US households for a year [2]. Such energy-intensive large model training not only inflates datacenter operational expenses, but also made power delivery a primary challenge in building datacenters today [9, 11, 12, 52, 53].

Despite recent works on accelerating large model training [45, 57, 90], energy optimization remains an open challenge [63, 68]. While energy optimization is well-studied in the hardware community [7, 14, 75, 84], the power bottleneck of recent datacenters [9, 11, 12, 53, 65] shows that efficiency gains from hardware advancement alone are not sufficient to sustain the growing demand for AI compute. In light of this, recent works show that software can play a significant role in energy optimization by capturing application characteristics that general-purpose hardware cannot (e.g., no need to finish computation before the deadline), bringing hardware-agnostic energy-efficiency gains [15, 17, 78, 85, 86].

In this paper, we seek a *software* method that reduces the energy consumption of large model training *without* slowdown, thereby also reducing average power draw. To that end, we identify *energy bloat*, the portion of energy consumption that can be removed without slowdown in software systems for large model training. We find two independent sources of energy bloat—*intrinsic* and *extrinsic*—and propose a single optimization framework that minimizes both.

Intrinsic energy bloat comes from computation imbalance when a large model is distributed across multiple GPUs with pipeline parallelism (§2.2). Balancing the amount of computation in each pipeline stage is an important problem for distributed execution planning [23, 30, 55, 90], but perfectly balancing every stage is not always possible because layers in a DNN are coarse-grained tensor operations with varying amounts of computation. When stages have unequal computation times, those not on the *critical path* of computation run needlessly fast—that is, they consume energy that does not contribute to the overall training throughput. Such intrinsic energy bloat opens up the opportunity to *precisely* slow down each non-critical computation in the pipeline such that the length of the critical path does not change.

Extrinsic energy bloat, in contrast, arises when multiple pipelines run in parallel in a synchronous fashion, and one or more pipelines run slower than the rest (§2.3). Root causes behind such slowdowns are varied, including power/thermal throttling [47, 61, 62, 67, 93], I/O bottlenecks in the storage/network [54, 83, 89], and hardware/software failures [25,

[1]Perseus is open-source as part of Zeus [86] at https://ml.energy/zeus.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP '24, November 4–6, 2024, Austin, TX, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695970

37, 76], and the likelihood of their presence increases with the scale and duration of training [28, 38, 79]. All pipelines running faster than the *straggler pipeline* are needlessly fast, wasting energy that does not affect the overall training throughput. Thus, we can slow down entire pipelines without delaying gradient synchronization.

In this work, we propose *Perseus*, which formulates a unified optimization framework to remove both intrinsic and extrinsic energy bloat from large model training (§3). At its core, Perseus efficiently pre-characterizes the entire time–energy tradeoff frontier of a training iteration, allowing it to minimize intrinsic bloat under normal operation and to mitigate extrinsic bloat arising from stragglers. Existing works fall short on both fronts. EnvPipe [15] is limited to intrinsic bloat reduction with a point solution that leads to suboptimal energy reduction. Zeus [86], in contrast, ignores intrinsic bloat as it only considers single-GPU training, which also renders its time–energy frontier suboptimal for large models.

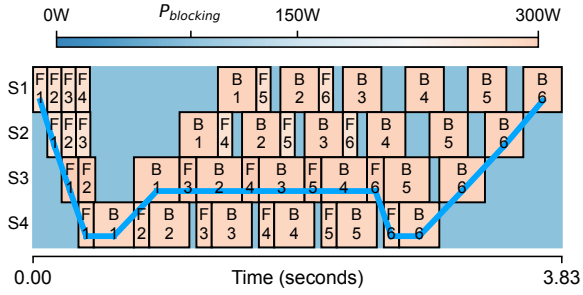
We show that characterizing the optimal time–energy Pareto frontier is NP-hard not only to solve, but also to approximate within any constant factor. Given this impasse, we propose an efficient algorithm that optimally solves a relaxed problem instead (§4). To do so, Perseus represents one training iteration as a directed acyclic graph (DAG) of forward and backward computations in each pipeline stage. Then, Perseus efficiently generates all *energy schedules*, defined as the planned time and energy consumption of each computation, that are on the time–energy frontier using a graph cut-based algorithm that iteratively *crawls up* the frontier from the bottom. Minimizing intrinsic and/or extrinsic energy bloat is then as simple as choosing the appropriate energy schedule from the pre-characterized time–energy frontier.

Perseus consists of a client library and a server (§5). The client library integrates with a large model training framework and accelerator to measure computation energy consumption and control accelerator speed. The server produces optimized energy schedules, using the abstract computation DAG and time/energy measurements provided by the client.

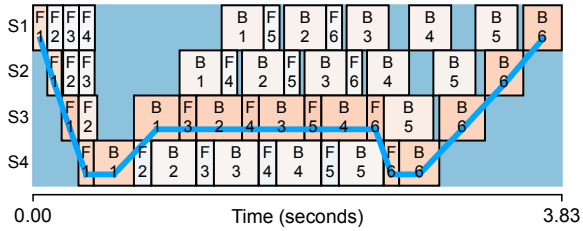
Evaluation on large models (GPT-3 [8], BERT [19], T5 [66], Bloom [82], and a scaled-up version of Wide-ResNet [87]), shows that Perseus is able to reduce per-iteration energy consumption by up to 30% with negligible or no slowdown, reducing energy consumption and average power draw (§6).

Overall, we make the following contributions in this paper:

- We identify intrinsic and extrinsic energy bloat in large model training, fundamentally caused by computation time imbalance at different levels.
- We propose Perseus, a software-only energy optimization system that reduces energy bloat through a unified optimization framework and a graph cut-based algorithm.
- We evaluate Perseus on a diverse set of large model workloads and show that it significantly reduces energy bloat, bringing hardware-agnostic energy savings.



(a) Execution timeline of one training iteration



(b) Execution timeline with reduced intrinsic energy bloat

Figure 1. One training iteration of GPT-3 1.3B with 4 pipeline stages and 6 microbatches on NVIDIA A100 GPUs, drawn to scale. For example, F5 and B5 in the S2 row denote forward and backward for the fifth microbatch on Stage 2. The critical path is traced with a blue line. Colors show power consumption. Other models are visualized in Appendix A.

2 Motivation

First, we provide necessary background regarding large model training (§2.1). Then, we introduce intrinsic (§2.2) and extrinsic (§2.3) energy bloat present in large model training, and discuss opportunities for energy reduction (§2.4).

2.1 Large Model Training

Large model training is mostly dominated by 3D (data, tensor, and pipeline) parallelism [5, 45, 57, 72, 82]. Especially, pipeline parallelism partitions a large model into multiple *stages* and its training batch into *microbatches*, and pipelines forward and backward computations through the stages. Then, such pipelines are replicated to perform data parallel training. Pipelines can only move on to the next iteration after every pipeline has finished and synchronized gradients.

2.2 Intrinsic Energy Bloat

We profile GPT-3 1.3B on NVIDIA A100 GPUs and visualize the timeline of one training iteration in Figure 1a. In addition to the familiar bubbles in the 1F1B schedule [57], we observe *gaps* between forward and backward computations, where the GPU is simply blocking on communication with an adjacent stage. Such gaps exist because the computation time of each pipeline stage is not perfectly balanced. Partitioning stages in a balanced manner is an important problem in

Model	# Parameters	Imbalance Ratio	
		4 stages	8 stages
GPT-3 [8]	3B	1.13	1.25
	7B	1.11	1.23
	13B	1.08	1.17
	175B	1.02	1.03
Bloom [82]	3B	1.13	1.25
	7B	1.13	1.25
	176B	1.05	1.10
BERT [19]	0.1B	1.33	2.00
	0.3B	1.17	1.33
T5 [66]	0.2B	1.19	1.50
	0.7B	1.05	1.11
	2.9B	1.06	1.16
Wide-ResNet50 [87]	0.8B	1.23	1.46
Wide-ResNet101 [87]	1.5B	1.09	1.25

Table 1. Forward latency ratio of the longest to the shortest stage on A100 GPUs. 1.00 would mean perfect balance.

distributed execution planning [23, 34, 55, 90], but *perfect* balancing is difficult because DNNs are essentially a sequence of coarse-grained tensor operations with varying sizes.

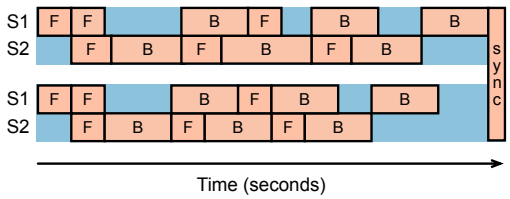
To understand the amount of possible pipeline stage imbalance, we exhaustively searched for the pipeline partition with the smallest imbalance ratio, defined as the ratio of the longest stage forward computation latency to the shortest.[2] Table 1 lists the minimum imbalance ratio for various models, which shows that perfect balance is difficult to achieve. See Appendix B for partitioning details and sources of imbalance.

Given stage imbalance, not all forward and backward computations are on the *critical path* of computation (Figure 1a). This means that non-critical computations running at their maximum speed are not contributing to faster iteration time, and thus simply wasting energy. We call this intrinsic energy bloat, which can be reduced by precisely slowing down each non-critical computation without lengthening the critical path (Figure 1b). Although seemingly simple, this problem is not only NP-hard to solve, but also NP-hard to even *approximate* to any constant factor [74].

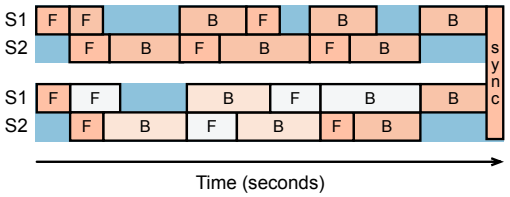
2.3 Extrinsic Energy Bloat

Numerous replicas of the same pipeline run in a data parallel fashion in large model training. Because every pipeline must synchronize gradients at the end, if one pipeline runs slower, *all other* pipelines must wait until the straggler pipeline finishes (Figure 2a). Since the straggler pipeline determines end-to-end iteration time, all other pipelines running at their fastest possible iteration time are wasteful. We call this extrinsic energy bloat, because unlike intrinsic energy bloat, its cause is *extrinsic* to the training pipeline. To reduce extrinsic

[2]For Transformer-based models, we partition at the granularity of Transformer layers. For Wide-ResNet, we partition at the granularity of bottleneck layers, which are three convolution layers wrapped with a skip connection.



(a) Extrinsic energy bloat caused by a straggler



(b) Reduced intrinsic and extrinsic energy bloat

Figure 2. Among two data parallel pipelines, the first one becomes a straggler. The non-straggler pipeline causes extrinsic energy bloat by running as fast as possible (a), which can be reduced by precisely slowing it down (b).

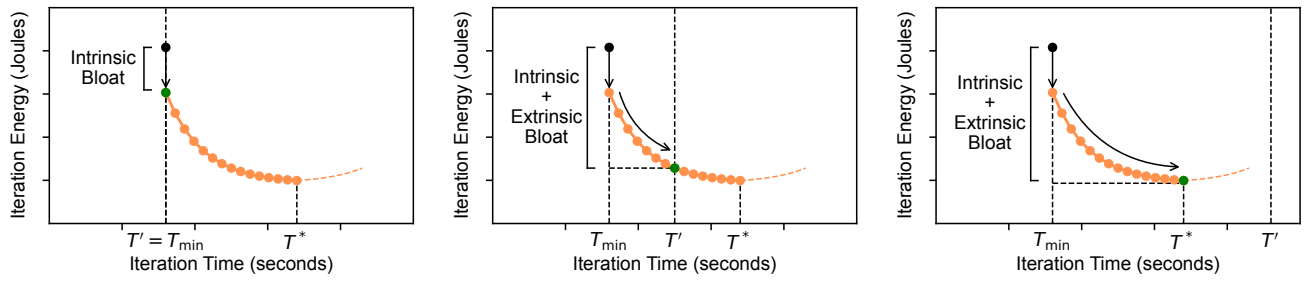
bloat while keeping intrinsic bloat low, one can determine the energy-optimal iteration time for non-straggler pipelines and precisely slow down computations so that non-straggler pipelines attain that iteration time (Figure 2b).

Stragglers arise from numerous sources. Thermal or power throttling in a datacenter can result in 10–50% slowdown [47, 61, 62, 67, 93], and I/O bottlenecks in the storage or network can be longer than GPU computation by up to 4× [54, 83, 89], acting like a persistent straggler pipeline. Recent failure-resilient training frameworks [25, 37, 76] deploy *heterogeneous* pipelines, introducing non-uniform iteration times. With increasing job and infrastructure scale, the probability of encountering stragglers increases [28, 38, 39, 79, 92].

In this work, we focus on stragglers that are known to and anticipated by the training infrastructure, generally because they were created by the infrastructure itself (e.g., power and thermal throttling, non-compute bottlenecks, fault-tolerant planning). Such stragglers also tend to persist beyond typical training iteration times. Therefore, Perseus focuses on planning time and energy consumption across time and allowing quick adaptation, assuming that information about stragglers is available.

2.4 Potential Benefits of Reducing Energy Bloat

To gauge potential energy savings, we measure the energy savings achieved by slowing down *every* computation in the pipeline to their minimum-energy frequencies. This will slow down iteration time, but can act as an upper bound for energy savings. For our workloads in Section 6.2, this gives on average 16% and 27% energy reduction on A100 and A40 GPUs, respectively. Section 6.2 shows that Perseus can realize most of the potential savings with negligible slowdown.



(a) $T' = T_{\min}$ **(b)** $T_{\min} < T' \leq T^*$ **(c)** $T^* < T'$

Figure 3. Three cases showing where the straggler pipeline's iteration time T' can be. $T_{\min}$ and T^* are the shortest and longest iteration times on the time–energy frontier. The black dot is when all computations run at the maximum speed, which wastes energy. The green dot is the energy-optimal iteration time of the non-straggler pipeline. Solid orange dots make up the frontier, and the orange dotted line shows that iteration energy increases beyond T^*.

3 Perseus Overview

We first present Perseus's unified optimization framework that aims to remove both types of energy bloat (§3.1), and then walk through the workflow of Perseus (§3.2).

3.1 Unified Optimization Framework

Intuitively, slowing down computations selectively in a training pipeline without affecting its critical path will keep the same iteration time while reducing its energy consumption (§2.2). Furthermore, when stragglers emerge, slowing down computations in a non-straggler pipeline without making it a straggler itself will reduce energy consumption even more (§2.3). We formalize these two intuitions into a unified optimization framework and derive a universal prescription for a non-straggler pipeline's *energy-optimal* iteration time.

Our goal is to minimize a pipeline's energy consumption by controlling the execution speed of each computation in the pipeline. In doing so, we can slow down a pipeline's iteration time up to the straggler's iteration time T':

$$\begin{aligned} \min_{F} \quad & \text{Energy}(F) \\ \text{s.t.} \quad & \text{Time}(F) \leq T' \end{aligned} \tag{1}$$

where F is an assignment of GPU frequencies[3] to each forward and backward computation in the pipeline, and $\text{Time}(F)$ and $\text{Energy}(F)$ are the iteration time and energy consumption of the pipeline when executed with F, respectively. Changing F will lead to different values of $\text{Time}(F)$ and $\text{Energy}(F)$, but we are only interested in $(\text{Time}(F), \text{Energy}(F))$ points that are on the time–energy tradeoff frontier.

Now, let us assume we have a fully characterized time–energy frontier, bookended by $T_{\min}$ and T^* (Section 4 is dedicated to describing how). $T_{\min}$ is the shortest iteration time on the frontier, which is the same as the iteration time of running every computation at the maximum speed, and T^* is the iteration time with minimum energy consumption, which is when each computation runs at the frequency that consumes the least amount of energy for that computation.[4] Figure 3 shows the three possible cases regarding where the straggler's iteration time T' can be:

1. Figure 3a: When there are no stragglers, we simply select the point on the frontier with iteration time $T_{\min}$, which reduces *only intrinsic energy bloat*.
2. Figure 3b: When a moderately slow straggler is detected, we *additionally reduce extrinsic energy bloat while keeping intrinsic bloat low* by slowing down all non-straggler pipelines until T', using up all the slack time.
3. Figure 3c: Finally, the straggler's iteration time may go beyond the minimum-energy point T^* on the frontier. In this case, we only slow down non-stragglers until T^*, because going past T^* will instead *increase* energy.

The three cases can be merged into one universal prescription for the pipeline's energy-optimal iteration time:

$$T_{\text{opt}} = \min(T^*, T'). \tag{2}$$

Therefore, when a straggler emerges (i.e., $T_{\min} < T'$), Perseus can compute T_{opt} using Equation 2 and quickly look up the frequency plan F_{opt} that leads to iteration time T_{opt} using the pre-characterized time–energy frontier.

Finally, we note that, unlike other problem settings that do not consider energy consumption, fully utilizing all the slack time created by the straggler *is not always energy-optimal*; being too fast or too slow can both waste energy.

3.2 Perseus Architecture

Energy Schedule. Perseus represents each iteration of the training pipeline as a static directed acyclic graph (DAG), where nodes are forward and backward computations in

[3]The SM frequency of NVIDIA GPUs can be set via NVML [4] in around 10 ms, which is much shorter than typical large model computation latencies. Locking the GPU's frequency provides deterministic computation latency [4, 27], making it suitable for tightly planning and packing execution over time.

[4]This is typically not the lowest frequency, because computations running with very low frequencies incur more latency increase than power reduction, resulting in *higher* energy consumption.

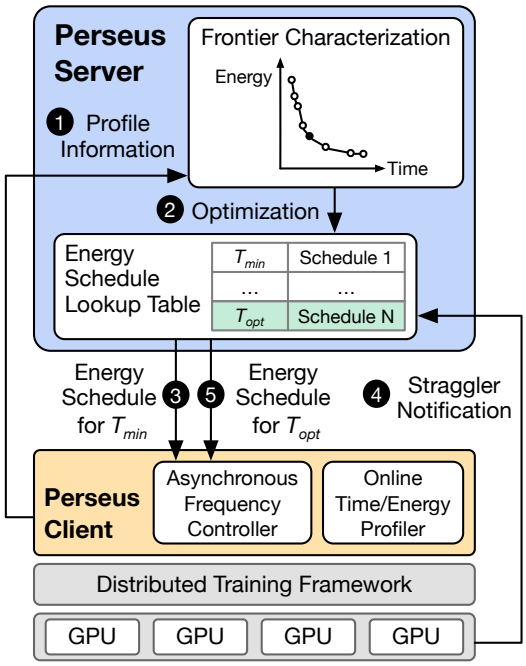


Figure 4. Perseus architecture and workflow.

each stage and edges are dependencies between computations. Each node on the computation DAG is annotated with its planned time and energy consumption, which we call the *energy schedule*. Perseus realizes an energy schedule by executing each computation with a specific GPU frequency.

System Components. Perseus's architecture is shown in Figure 4. Perseus consists of a framework- and accelerator-agnostic server and a framework-integrated and accelerator-specific client. The server is a cluster-wide singleton. For various training jobs, the server pre-characterizes the time–energy frontier of one iteration (§4) and caches energy schedules for fast lookup. The client profiles pipeline computations online during training and realizes energy schedules by setting the GPU's frequency during runtime (§5).

Training Lifecycle. For the Perseus server, a training job is primarily specified by its computation DAG for one training iteration. When the job begins execution, ❶ the Perseus client invokes its online time–energy profiler (§5) to measure the time and energy of each forward and backward computation on each supported frequency. Profiling is done *in vivo* during the initial tens of training iterations.

Upon receiving profiling results, ❷ the server begins *asynchronously* characterizing the time–energy frontier (§4) while training continues. When characterization finishes, energy schedules on the frontier are saved in a lookup table indexed by T'. Then, ❸ the energy schedule corresponding to $T_{\min}$ is deployed to the client. Energy schedules are realized by the client's asynchronous frequency controller, integrated into the training framework (§5).

During training, ❹ the training infrastructure (e.g., datacenter rack power/temperature manager) notifies the Perseus server of a straggler and its degree of slowdown. The server then ❺ quickly reacts to this by looking up the energy schedule corresponding to the anticipated straggler iteration time (the one with iteration time T_{opt}), and deploys it to the client.

4 Characterizing the Time–Energy Frontier

In this section, we describe our algorithm to efficiently obtain the time–energy tradeoff frontier for a training pipeline in detail. We first formulate the problem, show that it is NP-hard, and describe a relaxed version (§4.1). Then, we provide an overview of our algorithm (§4.2) and describe the core subroutine in our algorithm (§4.3). Finally, we extend our algorithm to support 3D/hybrid parallelism, constant-time operations, and diverse pipeline schedules (§4.4).

4.1 Problem Formulation

Expression for Energy Consumption. The energy consumption of a pipeline is not only from computation; it is the sum of three parts: (1) Computation; (2) Blocking on communication between computations; and (3) Blocking on communication until the straggler pipeline finishes:

$$\begin{aligned}&\sum_i e_i(f_i) + P_{\text{blocking}}(N \cdot T - \sum_i t_i(f_i)) + P_{\text{blocking}} \cdot N \cdot (T' - T) \\ &= \underbrace{\sum_i \big(e_i(f_i) - P_{\text{blocking}} \cdot t_i(f_i)\big)}_{①} + \underbrace{P_{\text{blocking}} \cdot N \cdot T'}_{②}\end{aligned} \tag{3}$$

where P_{blocking} is the power consumption of the GPU when it is blocking on communication, N is the number of pipeline stages, and $t_i(f_i)$ and $e_i(f_i)$ are the time and energy consumption of computation i with frequency f_i, respectively.[5]

As derived in Section 3.1, given straggler iteration time T', we draw a vertical line on the time–energy tradeoff frontier (Time(F) vs. ①+②) at T_{opt} and find F_{opt} where the two lines intersect. Equation 3 shows that the time–energy frontier of a pipeline depends on the straggler's iteration time T' *only* in the second term ②, which is merely an *upward shift* of the frontier. Therefore, if we characterize the tradeoff frontier of Time(F) vs. ①, that frontier can be used to find T_{opt} and F_{opt} for any straggler iteration time T'. Thus, we define

$$\text{Energy}(F) = \sum_i \big(e_i(f_i) - P_{\text{blocking}} \cdot t_i(f_i)\big) \tag{4}$$

and characterize the frontier of Time(F) vs. Energy(F).

Finding the Time–Energy Frontier. Finding *one point* on the Pareto-optimal tradeoff frontier with iteration time T is

[5] An assumption here is that P_{blocking} is constant, as a GPU blocking on communication is busy-looping inside a NCCL kernel without heavy computation utilization.

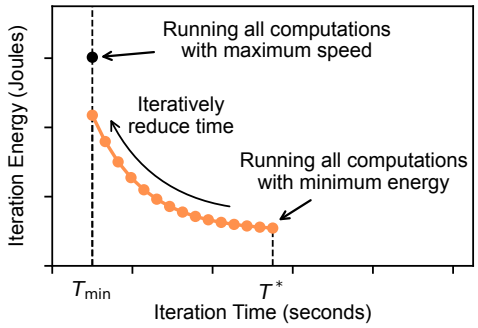


Figure 5. Starting from the energy schedule that consumes the minimum energy, we iteratively reduce its iteration time to trace up and iteratively discover the tradeoff frontier.

equivalent to solving the following optimization problem:

$$\begin{aligned} \min_{F} \quad & \text{Energy}(F) \\ \text{s.t.} \quad & \text{Time}(F) \leq T \end{aligned} \tag{5}$$

We call this problem *Pipeline Energy Minimization* (PEM).

Theorem 4.1. *Pipeline Energy Minimization is NP-hard.*

Proof. Reduction from Knapsack. Details in Appendix C. □

The complete Pareto-optimal tradeoff frontier can be obtained by solving PEM for all $T \in [T_{\min}, T^*]$, which is clearly intractable. Therefore, we seek an appropriate relaxation of the problem that will yield a *nearly* Pareto-optimal frontier.

One of the reasons PEM is NP-hard is because it is a *discrete* optimization problem where the possible choices of computation time and energy are discrete, which is in turn because GPUs only support discrete frequencies (e.g., in 15 MHz steps and nothing in the middle). However, if frequency choices were *continuous*, the problem is exactly and efficiently solvable [70]. This is akin to integer linear programs becoming tractable when relaxed to linear programs.

The transform from the original problem to the relaxed version is done by fitting a *continuous* exponential function $(a \cdot e^{bt} + c)$ to Pareto-optimal computation time and energy measurements for each forward and backward computation. We choose the exponential function due to its inherent flexibility and natural fit to data (more details in Appendix D). We show in Section 6.2 that this relaxation produces high-quality approximate solutions that realize most of the opportunity for savings. Solving the relaxed problem returns the time and energy consumption planned for each computation in the pipeline, or the energy schedule. Then, this is transformed back to a feasible solution of the original problem, which is the set of GPU frequencies F.

4.2 Iteratively Discovering the Frontier

Now, we first describe our iterative strategy of characterizing the frontier, and dive deeper into one iteration in Section 4.3.

Although our relaxed problem is no longer NP-hard, solving it for each $T' \in [T_{\min}, T^*]$ from scratch is inefficient.

Input: DAG $\mathcal{G}$ of computations $i \in \mathcal{G}$
Amount of time to reduce in one iteration τ
Iteration time with all max frequencies $T_{\min}$
Output: Set of optimized schedules $\mathcal{S}$

▷ Begin with the minimum energy schedule
1 $s \leftarrow$ Minimum energy for all computations
2 $\mathcal{S} \leftarrow \{s\}$
3 **while** IterationTime$(\mathcal{G}, s) > T_{\min}$ **do**
  ▷ Reduce time by τ with minimal energy increase (§4.3)
4 $s \leftarrow$ GetNextSchedule$(\mathcal{G}, s, \tau)$
5 $\mathcal{S} \leftarrow \mathcal{S} \cup \{s\}$
6 **return** $\mathcal{S}$

Algorithm 1: Iteratively discovering the frontier.

Instead, what if we can *tweak* an existing schedule already on the frontier to generate its *neighbor* energy schedule on the frontier? Then, we can start from one end of the frontier and trace along to the other end, discovering fine-grained optimized energy schedules.

Figure 5 visualizes our strategy. We start from the rightmost point T^* that consumes the *minimum energy*, which is simply running every computation with the minimum energy.[6] This energy schedule is in fact Pareto-optimal because there are no other schedules that achieve the same energy with faster time. Then, we iteratively reduce iteration time by unit time τ (e.g., 1 ms) while increasing total energy *minimally*, which gives us the neighbor energy schedule on the frontier.[7] This is repeated until iteration time reaches $T_{\min}$.

We note that tracing down from the energy schedule that consumes the maximum energy (i.e., Figure 5 black dot) would be incorrect. That schedule is far from optimized because, although it will execute with the least amount of time, stage imbalance leaves room for energy reduction (§2.2).

Algorithm 1 provides an overview of our optimization process. First, the energy schedule with the minimum energy consumption is constructed by planning every computation to run with minimum energy (line 1). Starting from there, the iteration time of the schedule is iteratively reduced by unit time τ *while incurring minimal energy increase* (line 4; Section 4.3). This is repeated until the total iteration time of the schedule can no longer be reduced, and every energy schedule encountered in the process forms our frontier.

4.3 Finding the Neighbor Energy Schedule

In this section, we describe our core subroutine GetNextSchedule (Algorithm 1, line 4). Figure 6 provides visualizations of the process. The entire procedure is given in Algorithm 2.

[6] The minimum energy consumption for each computation type can be queried from the computation time/energy profiling information (§5).
[7] τ is the unit time parameter that trades off the running time of Perseus's optimizer and the granularity of energy schedules discovered by Perseus.

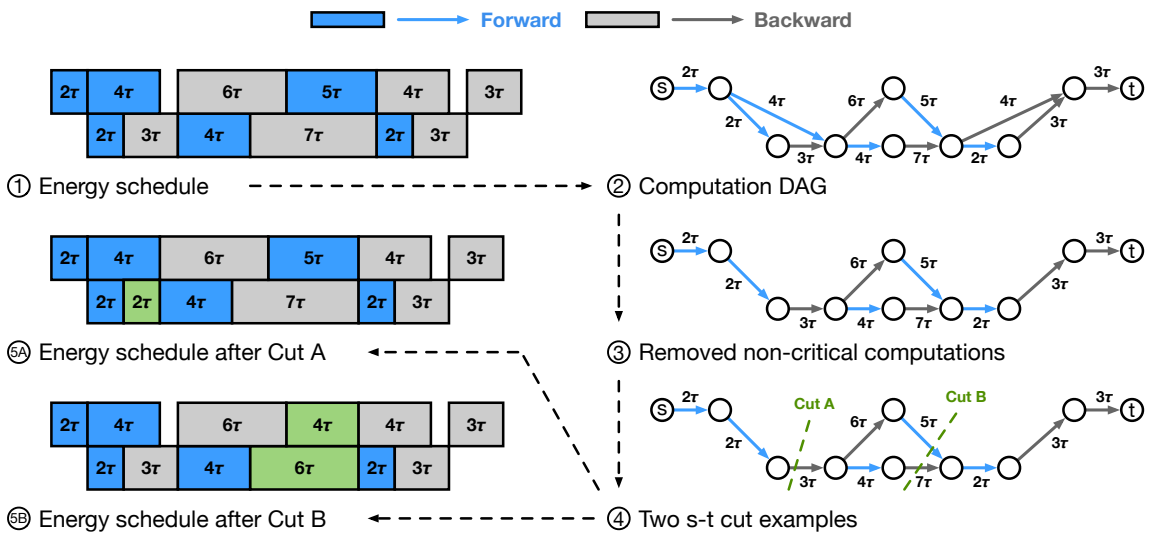


Figure 6. A simplified example of how to reduce iteration time by unit time τ. Given a 1F1B pipeline schedule with 2 stages and 3 microbatches (①), it is first transformed to an equivalent representation of computation DAG (②). Then the Critical DAG (③) is obtained by considering only the computations on the critical path. Our key observation is that any valid s-t cut on the Critical DAG will reduce the iteration time by unit time τ. Cut A and Cut B are two examples of valid s-t cuts (④). Either reducing the one computation associated with Cut A (⑤A) or reducing the two computations associated with Cut B (⑤B) reduces the iteration time by τ.

Node- and Edge-Centric Computation DAGs. Originally, Perseus's representation of the computation DAG is node-centric, which has forward and backward computations as nodes and their dependencies as edges. As a setup for subsequent steps, we convert this into an edge-centric computation DAG where computations are edges and dependencies are nodes (i.e., all incoming edges must complete before any outgoing edge can begin). This conversion can be done by splitting each node into two and connecting the two with an edge annotated with the computation on the original node.

Removing Non-Critical Computations. Our goal is to reduce the execution time of the computation DAG by τ, which is equivalent to reducing the length of *all critical paths* by τ.[8] Since computations that are not on any critical path (i.e., non-critical computations) do not affect the length of the critical path, we remove them from the computation DAG.

Finding Computations to Speed Up. Which computations on the DAG should we speed up in order to reduce the length of all critical paths by τ? The key observation is that *any* s-t cut on the computation DAG represents a way to reduce the execution time of the DAG by τ. Specifically, by speeding up the computations on all cut edges by τ, the entire computation DAG can be sped up exactly by τ.

Figure 6 shows two examples of this. ④ shows two valid s-t cuts: *Cut A* and *Cut B*. ⑤A speeds up the computation edge cut by *Cut A* from 3τ to 2τ, and the iteration time of the energy schedule was reduced by τ. Similarly, ⑤B speeds up the computation edges cut by *Cut B* from 5τ to 4τ and from 7τ to 6τ, and the iteration time of the energy schedule was also reduced by τ. Especially, in the second case, iteration time was only reduced because computations on two parallel critical paths were sped up *together*.

Solving with Minimum Cut. We have seen that any s-t cut represents a way to speed up the entire DAG by τ. But speeding up computations increases energy. Then, a natural question is, which cut brings the smallest energy increase?

We can precisely map the *flow capacity* of an s-t cut to the amount of *energy increase* from speeding up cut edges. That is, by finding the amount of energy increase each computation will incur with the slope of its exponential function (§4.1) and defining it to be the edge's flow capacity, we can reduce our problem to minimum cut, which we can solve with maximum flow. After finding the minimum cut, we modify the durations of the computations involved in the cut, obtaining the neighbor energy schedule. Appendix E provides details with mathematical expressions for flow capacities.

Converting Back to GPU Frequencies. Finally, we convert the energy schedule into GPU frequencies that can be realized by the Perseus client. For each computation, we convert its planned execution time t to the slowest GPU frequency that will execute *faster* than t. This is because when computations are tightly packed by our algorithm, while slightly speeding up a computation is acceptable, slowing down *any computation* on the critical path will directly slow down the entire DAG, increasing intrinsic energy bloat.

[8] Let's say there are two critical paths that run in parallel. They must be of equal length to both be critical paths. Here, if only one were shortened, the other will remain the sole critical path and the DAG will not execute faster.

API	Description
`profiler.begin(type)`	Begin time and energy profiling for computation `type`.
`profiler.end(type)`	Record time and energy profiling results for computation `type`.
`controller.set_speed(type)`	Set the hardware's execution speed as planned for computation `type`.
`server.set_straggler(id, delay, degree)`	Notify that a straggler is anticipated after `delay` seconds. A straggler returning to normal can be communicated by setting `degree` to 1.

Table 2. The minimal set of Perseus client and server APIs that require implementation. One client process manages each accelerator. The `type` parameter should be either `"forward"` or `"backward"`. On GPUs, the "speed" control knob is the SM frequency. `set_straggler` is invoked by the infrastructure with the `id` of an accelerator to notify the server via HTTP/RPC.

Input: DAG $\mathcal{G}$ of computations $i \in \mathcal{G}$
Current energy schedule s
Amount of iteration time to reduce τ
Output: Neighbor schedule with reduced time s'

▷ Construct edge-centric computation DAG (②)
$\mathcal{G} \leftarrow$ Split nodes into two and connect with edge
▷ Find and remove non-critical computations (③)
Annotate earliest & latest start times for $\forall i \in \mathcal{G}$
for $i \in \mathcal{G}$ **do**
  if i has different earliest and latest start **then**
    Remove i from $\mathcal{G}$
▷ Find set of computations to modify (④)
$S, T \leftarrow \text{FindMinCut}(\mathcal{G}, s)$
▷ Modify computation durations (⑤)
Modify duration of $\forall i$ in $S - T$ cut by τ
▷ Assign frequencies from planned computation times
$s' \leftarrow \min f_i$ that runs no slower than planned
return s'

Algorithm 2: GetNextSchedule: Reducing the execution time of the DAG by τ with minimal energy increase.

Time Complexity Analysis. Our optimization algorithm has polynomial runtime. Let N and M denote the number of stages and microbatches, respectively. Then, the computation DAG will have $O(NM)$ number of nodes and edges, and maximum flow with Edmonds-Karp runs in $O(N^3M^3)$. While for general DAGs the total number of steps is known to be exponential to the size of the DAG [71], we prove that for DAGs that represent pipeline computations, the number of steps is $O(N + M)$, yielding a final polynomial time complexity of $O((N + M)N^3M^3)$. See Appendix F for proof.

In reality, commonly used number of stages (N) is 4 to 8 (at most tens) to reduce pipeline bubble ratio [18, 57]. Number of microbatches (M) is typically around $4N$ [34, 72], but recently with high data parallel degree, far fewer have been reported even for high-performance settings [18]. As such, algorithm runtime is practically negligible (§6.5), especially given that large model training easily takes weeks or months [63].

4.4 Generalizations

In this section, we present generalizations to our optimization algorithm useful for planning large model training.

3D/Hybrid Parallelism. Operator parallelism techniques (e.g., data, tensor, or sequence parallelism) split operations in *equal sizes*, resulting in each GPU running the same computation. This allows Perseus to profile only one GPU per stage, decide the energy schedule for that GPU, and replicate it to all other GPUs in the same stage. We show that Perseus works well for 3D parallelism in Section 6.4.

Constant-Time Operations. There are operations in the training pipeline that may take non-trivial latency, other than computation and blocking on communication. For instance, loading and copying input data into VRAM or communication over slower links can take considerable latency. However, the time and energy consumption of these operations are not affected by the GPU's frequency. Perseus can take constant-time operations into account during planning by viewing them as a node with only one frequency choice.

Other Pipeline Schedules. There are various schedules for pipeline parallel training, including GPipe [34], 1F1B [56], interleaved 1F1B [69], and early recomputation 1F1B [45]. As long as the computations on the schedule can be expressed as a DAG, Perseus can optimize its energy consumption without modification. As long as there is stage imbalance, any pipeline schedule will have intrinsic energy bloat.

5 Implementation

The Perseus server and client are implemented in Python. Perseus can optimize any training infrastructure, framework, and accelerator as long as the APIs in Table 2 can be implemented, and the accelerator supports multiple execution speeds that trade off computation time and energy.

As a reference, we have integrated the Perseus client with Merak [45], which marries high-performance tensor parallelism of Megatron-LM [3] and the generic pipeline execution engine of DeepSpeed [1]. While training engine implementations differ widely, many have separate code blocks for forward and backward, allowing them to be wrapped with the `profiler` APIs. We provide an example of what is looks

like to integrate the client with a training engine in Appendix G. Activation recomputation [13] is enabled to allow large batch sizes to fit in GPUs.

Profiler. Accurate profiling is important to our optimization algorithm; inaccurate latency profiles (especially underestimations) may slow down the end-to-end latency of the DAG, whereas inaccurate energy profiles can lead the algorithm to incorrectly select computations to speed up.

Fortunately, the latency of a fixed set of GPU computations, especially with the GPU's frequency locked, is known to be very stable [4, 27]. Furthermore, to ensure that profiling results are representative of real training, the Perseus client profiles the time and energy of each forward and backward computation at the beginning of the training job *in vivo*. Each supported GPU frequency is profiled one by one from the highest to the lowest for about five iterations (more if one iteration has less microbatches). After a certain frequency, lower frequencies result in both more time *and* energy consumed, making them strictly suboptimal compared to higher frequencies. Profiling is terminated at that point.

Finally, we profile P_{blocking} using two GPUs. One GPU blocks on P2P communication and the other sleeps, and we measure the power consumption of the blocking GPU. It is sufficient to profile P_{blocking} *once per GPU model.*

Asynchronous Frequency Controller. The client-side controller spawns a separate process that asynchronously sets the GPU's frequency through NVML [4] without blocking the main training process. Training frameworks can call `set_speed` at the beginning of forward or backward to set the GPU's frequency as planned by the server.

6 Evaluation

We evaluate Perseus on five workloads and compare it against EnvPipe and Zeus. Our key findings are the following:

- Perseus can effectively reduce intrinsic and extrinsic energy bloat. Training on real GPUs shows up to 28.5% energy savings using Perseus (§6.2).
- In emulated large-scale training scenarios, Perseus significantly outperforms the baselines by consistently providing up to 30% energy savings (§6.3).
- Energy bloat reduction is possible because Perseus can enumerate efficient energy schedules on the time–energy frontier (§6.4).
- Perseus reduces energy bloat with low overhead (§6.5).

6.1 Experimental Setup

Testbed. We run our evaluation workloads in a GPU cluster, where each node is equipped with an AMD EPYC 7513 CPU, 512 GB DRAM, and four NVIDIA A40-48G GPUs. For A100 results, we use a node provided by Chameleon Cloud [41], equipped with two Intel Xeon Platinum 8380 CPUs, 512 GB DRAM, and four NVIDIA A100-80G PCIe GPUs.

Workloads and experiment parameters. We evaluate Perseus with various workloads spanning from GPT-3 [8], Bloom [82], BERT [19], T5 [66], to Wide-ResNet [87]. We use model variants with 1.3B to 6.7B parameters to run the models in our testbed, and scale them up to 176B parameters in large-scale emulation. We chose the microbatch size and number of microbatches that yield the highest throughput given the global batch size. We use the minimum imbalance stage partitioning method described in Section 2.2 for all workloads. Appendix B lists complete model configurations, parameters, and stage partitioning details.

Metrics. We report GPU energy reduction and slowdown of a training iteration (%) relative to using all maximum GPU frequencies. In most cases slowdown is close to zero, in which case energy and average power reductions coincide. Reducing only extrinsic bloat is not possible, because Perseus reduces extrinsic bloat *while keeping* intrinsic bloat low as it slows down non-straggler pipelines. Therefore, we report (1) intrinsic bloat reduction *without* stragglers and (2) intrinsic + extrinsic bloat reduction *with* stragglers.

Baselines. We mainly compare with two prior works:

- **EnvPipe** [15] reduces only intrinsic energy bloat while trying to minimize slowdown. We compare Perseus's energy bloat reduction with EnvPipe (§6.2, §6.3).
- **Zeus** [86] characterizes the time–energy tradeoff of single GPU training. We compare Perseus's time–energy frontier against that of Zeus (§6.4).

6.2 Reducing Energy Bloat

We start with overall energy bloat reduction—intrinsic bloat without stragglers (§6.2.1) and intrinsic + extrinsic bloat with stragglers (§6.2.2)—achieved by Perseus and EnvPipe. All numbers were obtained by running on testbed GPUs. All solutions use the same amount of GPU hardware resources.

6.2.1 Intrinsic Bloat Reduction Without Stragglers.

Table 3 compares the energy savings achieved by Perseus's minimum iteration time energy schedule (leftmost point of the time–energy frontier) and that by EnvPipe.

We make two observations regarding Perseus. First, models show varying amounts of energy savings because (1) their stage imbalances vary (Table 1), and (2) their forward and backward are composed of different computations, which affects time/energy sensitivity when changing the frequency. For instance, unlike other models, Wide-ResNet 1.5B on A100 after minimum imbalance stage partitioning has nearly perfect stage balance, resulting in low intrinsic energy bloat. However, as will be seen in Section 6.2.2, such models tend to achieve greater extrinsic bloat savings because most of their computations run at a high frequency, and slowing them down due to stragglers leads to higher energy reduction.

Second, A40 demonstrates more energy savings compared to A100. This is because the dynamic clock frequency range

Model	Energy Savings (%)		Slowdown (%)	
	Perseus	EnvPipe	Perseus	EnvPipe
GPT-3 1.3B	13.2	8.8	0.1	0.1
BERT 1.3B	12.9	8.0	0.5	0.0
T5 3B	10.6	7.4	1.3	3.4
Bloom 3B	11.7	8.9	0.2	0.2
Wide-ResNet 1.5B	3.2	3.7	2.3	4.1

(a) Four stage pipeline parallelism on A100 GPUs

Model	Energy Savings (%)		Slowdown (%)	
	Perseus	EnvPipe	Perseus	EnvPipe
GPT-3 2.7B	21.1	21.7	0.2	5.6
BERT 1.3B	15.7	16.5	0.0	9.7
T5 3B	28.5	19.3	0.0	0.0
Bloom 3B	22.4	19.9	0.0	0.0
Wide-ResNet 1.5B	20.4	16.5	0.2	0.5

(b) Eight stage pipeline parallelism on A40 GPUs

Table 3. [Experiment] Intrinsic energy bloat (without stragglers) reduction and iteration time slowdown.

of A100 (210–1410 MHz) is smaller than that of A40 (210–1740 MHz). Thus, tuning down the GPU's frequency yields a relatively smaller change in computation time and energy compared to those at the maximum frequency. However, we expect the more recent GPUs to have better percentage savings due to higher maximum frequency (e.g., 1980 MHz for H100 SXM [58]) and better absolute savings due to high TDP (e.g., 1,200 W for each GPU on GB200 [50]).

EnvPipe in general provides lower energy savings, primarily due to its assumption that the final stage of a pipeline is always the heaviest. This is only correct with a probability of $1/N$, where N is the number of pipeline stages. Additionally, it sometimes considerably degrades iteration time because it is not aware of *single-choice operations* inside the pipeline (§4.4) and can slow down some computations too much.

6.2.2 Intrinsic + Extrinsic Bloat Reduction With Stragglers. When stragglers create extrinsic energy bloat, the amount of energy savings depends on how much energy reduction the time–energy frontier yields for longer iteration times. Table 4 shows the amount of energy savings of a non-straggler pipeline given varying straggler slowdowns. For a given slowdown factor (T'/T), a non-straggler pipeline's iteration time is set to be $T_{\text{opt}} = \min(T^*, T')$ (§3.1), and energy reduction comes from (1) slowing down the pipeline itself and (2) reducing the time (and energy) blocking on communication, waiting for the straggler.

The percentage of savings initially increases with the straggler's iteration time, but then gradually decreases as it slows down beyond T^*. This is expected. The absolute amount of energy reduction in Joules is the largest when the straggler's iteration time is T^* and constant afterward, because Perseus does not slow down non-straggler pipelines beyond

Model # Params	Method	Energy Savings (%) given T'/T					
		1.05	1.1	1.2	1.3	1.4	1.5
GPT-3 1.3B	Perseus	14.7	15.9	15.5	15.0	14.6	14.3
	EnvPipe	8.7	8.5	8.3	8.1	7.9	7.7
Bloom 3B	Perseus	13.6	15.6	15.2	14.7	14.3	14.0
	EnvPipe	8.8	8.7	8.4	8.2	8.0	7.8
BERT 1.3B	Perseus	14.9	16.9	16.4	15.9	15.5	15.0
	EnvPipe	7.9	7.8	7.5	7.3	7.1	6.9
T5 3B	Perseus	15.3	18.0	17.9	17.4	16.9	16.5
	EnvPipe	8.4	8.2	8.0	7.8	7.6	7.4
Wide-ResNet 1.5B	Perseus	9.4	12.7	12.6	12.3	12.0	11.6
	EnvPipe	4.9	4.8	4.7	4.5	4.4	4.3

(a) Four stage pipeline parallelism on A100 GPUs

Model # Params	Method	Energy Savings (%) given T'/T					
		1.05	1.1	1.2	1.3	1.4	1.5
GPT-3 2.7B	Perseus	24.5	26.0	25.9	25.2	24.6	24.0
	EnvPipe	22.9	22.6	22.0	21.4	20.9	20.4
Bloom 3B	Perseus	25.5	26.4	25.9	25.2	24.6	24.0
	EnvPipe	19.6	19.3	18.8	18.3	17.8	17.4
BERT 1.3B	Perseus	20.0	22.6	24.1	23.4	22.8	22.2
	EnvPipe	19.2	18.9	18.3	17.8	17.4	16.9
T5 3B	Perseus	27.9	27.3	26.2	25.2	24.3	23.4
	EnvPipe	18.4	18.0	17.3	16.6	16.0	15.4
Wide-ResNet 1.5B	Perseus	24.3	26.2	26.3	25.7	25.0	24.4
	EnvPipe	16.4	16.2	15.8	15.4	15.0	14.6

(b) Eight stage pipeline parallelism on A40 GPUs

Table 4. [Experiment] Energy savings given varying straggler slowdown (T'/T). Perseus can reduce extrinsic bloat while keeping intrinsic bloat low, whereas EnvPipe cannot.

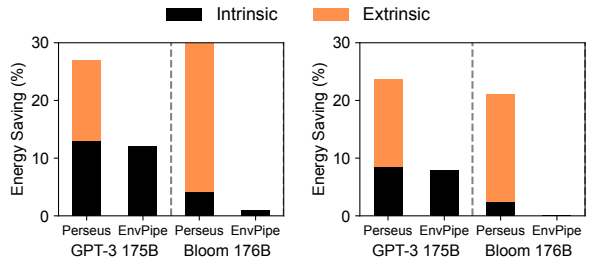
T^* (§3.1). Thus, as the straggler slows down beyond T^*, additional time and energy is consumed while waiting for the straggler, lowering the *percentage* of energy savings.

Finally, the point of maximum energy savings is different for each model. This is because each model has a different T^* value, which is determined by how much each stage's computation slows down on the minimum-energy frequency.

6.2.3 How Much Potential Saving Was Realized? The largest possible savings under our problem setting occurs when running every computation at their minimum-energy frequencies (i.e., the T^* point on the time–energy frontier). For intrinsic bloat without stragglers, Perseus realizes on average 74% and 89% of the potential savings on A100 and A40, respectively, with negligible slowdown. This is possible because there are much more non-critical computations in the DAG that can be slowed down than critical ones. With stragglers, Perseus fully realizes potential savings when the straggler's slowdown degrees are on average 1.1 and 1.15 on A100 and A40 respectively, which is not unrealistic considering slowdowns reported in literature (§2.3).

# GPUs	# Pipelines	# Microbatches Per Pipeline	Global Batch Size
1024	16	96	1536
2048	32	48	
4096	64	24	
8192	128	12	

Table 5. Strong scaling parameters for large-scale emulation. A pipeline has tensor parallel degree 8 and 8 pipeline stages.



(a) A100 **(b)** A40

Figure 7. [Emulation] Energy savings breakdown with straggler slowdown 1.2 and 1,024 GPUs.

6.3 Large-Scale Emulation

Because we do not have access to a GPU cluster required to run huge models like GPT-3 175B, we use *emulation* grounded on fine-grained profiling for large-scale evaluation. In general, trends in our emulation result match those obtained from real training in Section 6.2.

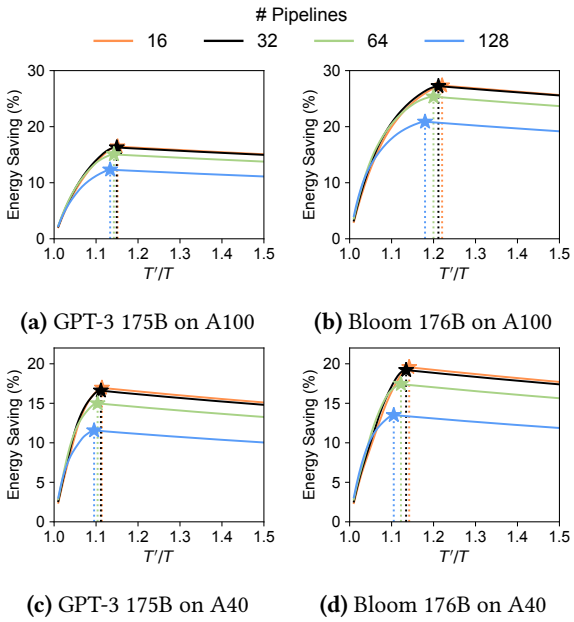
Emulation Methodology. We profile the time and energy consumption of each layer (e.g., Transformer decoder) in GPT-3 175B and Bloom 176B and run our optimization algorithm to obtain the time–energy frontier. We perform *strong scaling* when varying the number of GPUs (Table 5) in order to keep the global batch size constant [26, 42]. We used A100 SXM GPUs for emulation, which we believe are more representative of large-scale training infrastructure.

Emulator Fidelity. We compare the percentage of energy savings vs. running all maximum frequencies given by our emulator and real experiments for our A100 workloads and find that the emulator *always underestimates* energy savings. Specifically, savings on the leftmost and rightmost point of the frontier are underestimated by 18.6% and 21.7% on average, respectively. We believe this is due to our simplifying assumption that P_{blocking} is constant regardless of the GPU's frequency. This means the savings given by the emulator can be considered a lower bound for actual savings.

Result Summary. Figure 7 breaks down the amount of energy bloat reduction for GPT-3 175B and Bloom 176B when slowdown degree is 1.2 on 1,024 GPUs. EnvPipe can only reduce intrinsic bloat as it does not provide a time–energy frontier; even for intrinsic bloat, it is suboptimal. In contrast, Perseus reduces energy consumption by up to 30% by reducing both intrinsic and extrinsic energy bloat.

Model	GPU Type	Energy Savings (%) Given # Microbatches 12	24	48	96
GPT-3 175B	A100	15.20	14.19	13.62	13.32
	A40	11.81	10.22	9.34	8.88
Bloom 176B	A100	10.47	7.06	5.23	4.28
	A40	6.97	4.49	3.12	2.41

Table 6. [Emulation] Perseus's intrinsic energy bloat reduction without stragglers for GPT-3 175B and Bloom 176B. Number of microbatches is varied following Table 5.

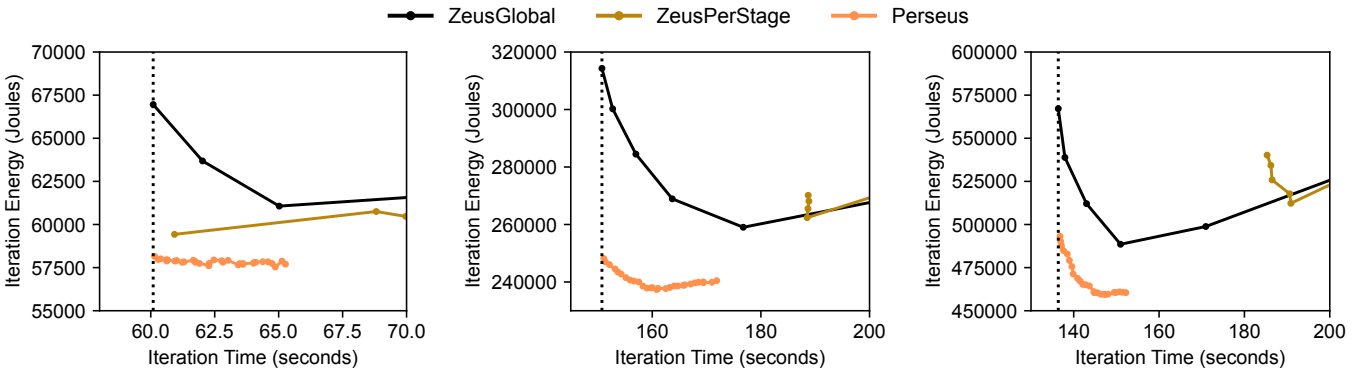


(a) GPT-3 175B on A100 **(b)** Bloom 176B on A100

(c) GPT-3 175B on A40 **(d)** Bloom 176B on A40

Figure 8. [Emulation] Perseus's intrinsic + extrinsic energy bloat reduction with varying straggler slowdown (T'/T). Number of pipelines is varied following Table 5. ★ denotes T^*/T for each pipeline. Please note the different Y-axes.

Intrinsic Bloat Reduction Without Stragglers. Table 6 shows Perseus's intrinsic energy bloat reduction without stragglers for GPT-3 175B and Bloom 176B. The number of microbatches is varied based on Table 5. For all models, as more microbatches are added to the pipeline, the amount of intrinsic bloat decreases. This is fundamentally due to the ratio of microbatches in the 1F1B's warm-up and flush phase (beginning and end) versus the steady state phase (middle). Most microbatches in the warm-up and flush phases can slow down until their minimum energy frequency, yielding large energy savings. However, microbatches in the pipeline's steady state cannot slow down to their full potential when the amount of stage imbalance is not large, thereby yielding modest savings. When the number of microbatches in the pipeline increases, only the number of steady state microbatches increases, and energy reduction converges to the average energy savings of steady state microbatches.



(a) PP=4 on A100, GPT-3 1.3B **(b)** PP=8 on A40, GPT-3 2.7B **(c)** DP=2, TP=2, PP=4 on A40, GPT-3 6.7B

Figure 9. [Experiment] Iteration time–energy frontiers for GPT-3, achieved by Perseus and the two baselines derived from Zeus [86]. Perseus Pareto-dominates all other approaches. The dotted vertical line is the iteration time of running all GPUs at their maximum power limit, which is the default mode of operation. Please note the different X- and Y-axes.

Intrinsic + Extrinsic Bloat Reduction With Stragglers. We introduce stragglers of varying slowdowns in large-scale emulation. Figure 8 reports the amount of intrinsic + extrinsic energy bloat reduction achieved by Perseus. The trend where energy saving increases until $T' < T^*$ and wanes afterward is consistent with Section 6.2.2.

An interesting observation here is that there is a trade-off between scale and energy savings: *configurations with more pipelines have less percentage of energy savings or less amount of energy savings per pipeline.* It may seem intuitive to assume that more pipelines bring more energy savings, as there is only one straggler pipeline and all the other pipelines can reduce their energy consumption. However, this holds only in weak scaling scenarios, where the per-pipeline batch size is held constant (i.e., increasing the global batch size proportionally with the number of pipelines). Instead, in the more realistic strong scaling configuration (i.e., the global batch size is constant and per-pipeline batch size is decreased as more pipelines deployed), each pipeline's number of microbatches changes. With fewer microbatches, the ratio of pipeline bubble (time that GPUs are idle) at the beginning and end of each pipeline iteration increases [57]. These bubbles cannot be perfectly eliminated by intrinsic energy bloat reduction, resulting in a smaller energy savings percentages. However, as the absolute number of GPUs increases, even a small savings percentage is expected to yield *huge absolute energy savings.*

6.4 Iteration Time–Energy Frontier Comparison

The energy bloat reductions in Sections 6.2 and 6.3 were made possible by the time–energy frontier obtained using Perseus's optimization algorithm (§4). Here, we further examine the frontier with different parallelization configurations and models and compare against Zeus [86]. Since Zeus only produces the training time–energy frontier for *single-GPU* training jobs, we implemented two Zeus-based baselines for large model training scenarios:

- **ZeusGlobal**: Scans one global power limit for all stages.
- **ZeusPerStage**: Finds a set of per-stage power limits that balances forward computation time.

Figure 9 shows the frontiers of all solutions for different sizes of GPT-3 under three parallelization configurations: (a) four stage pipeline parallelism on A100; (b) eight stage pipeline parallelism on A40; and (c) 3D parallelism (data parallelism 2, tensor parallelism 2, pipeline parallelism 4) on A40. All results were obtained from running on testbed GPUs. Appendix H has frontiers for other models.

Perseus Pareto-dominates both baselines derived from Zeus. ZeusGlobal is unaware of pipeline stage imbalances and slows down every stage, and therefore is unable to intrinsic energy bloat. ZeusPerStage can balance the forward computation time of each stage, but is unaware of the *critical path* of the DAG, slowing down critical computations. In contrast, Perseus can precisely slow down non-critical computations, effectively reducing energy bloat.

6.5 Overhead of Perseus

Profiling. At the beginning of training, the client-side profiler (§5) profiles forward and backward computations in each stage. For our A100 workloads, the initial profiling phase increased end-to-end training time by 13 minutes on average, which is negligible overhead for large model training.

Algorithm Runtime. The average runtime of the optimization algorithm (§4) across our A100 workloads was 6.5 minutes, with the longest being Bloom 3B (15.7 minutes). For our largest-scale emulation workload (GPT-3 175B on A100 with 8,192 GPUs), the algorithm took 87 seconds, which is short because it is sufficient to optimize just one data parallel pipeline (§4.4). While runtime will increase with larger DAGs for larger models, we believe the overhead is justified because training time is likely to also increase with the scale of the training job. Looking up the optimal energy schedule given the straggler's iteration time T' is instantaneous.

7 Related Work

Large Model Training. Many recent works focus on enabling and accelerating large model training using 3D parallelism. GPipe [34] and PipeDream [55] first introduced pipeline parallelism. 3D parallelism, especially popularized for Transformer-based models by Megatron-LM [57, 69], is considered to be the go-to solution for modern large model training due to strong open-source projects [1, 3, 6] and relatively low implementation complexity. Extending this, Alpa [90], GSPMD [88], and nnScaler [48] provide automatic parallelization for general DNNs. However, energy consumption is not an optimization metric for any of the major large model training frameworks.

Several works utilize computation idle times within large model training pipelines to insert additional useful work [33, 59, 76]. In contrast, Perseus's approach is to slow down preceding computations to fill the idle time and reduce power and energy consumption. The rationale is twofold. First, the amount of idle time–especially in the steady state of the pipeline–is typically short. Inserting extra work may slow down the entire pipeline or require pipeline re-partitioning to make it feasible. Second, GPUs, especially those in the earlier stage of the pipeline, are likely already constrained by memory capacity, leaving little headroom for extra computation. We note that even with extra computation inserted, some idle time is likely to remain, leaving room for Perseus to provide energy savings.

Deep Learning and Energy Consumption. A line of work measures or estimates the large amount of energy consumption and carbon emission of deep learning workloads [20, 44, 49, 63, 73]. In terms of optimization, some works determine the GPU's execution speed for a single fixed sequence of GPU computations [16, 36, 43, 78, 85, 91], falling short when the time–energy frontier of a complex large model computation DAG needs to be characterized. Zeus [86] is a recent work that observes the tradeoff between GPU computation time and energy consumption, but still focuses on single-GPU training. EnvPipe [15], on the other hand, aims to find a point solution that reduces the energy consumption of large model training with minimum slowdown. However, EnvPipe's heuristic assumes that the last pipeline stage is always the bottleneck, leading to suboptimal savings. In terms of optimizing a single training job, Perseus is a superset of both Zeus and EnvPipe, achieved by viewing large model training as a computation DAG and introducing a principled optimization algorithm.

Big Data and Energy Consumption. There are existing works that change the frequency of CPUs to improve the energy efficiency of big data workloads, including those that only execute computation-intensive phases (e.g., map and reduce) with the maximum frequency [35, 80], those that choose the lowest frequency that can meet a given deadline [10, 51], or those that choose frequencies that balance the completion time of parallel tasks (yet simply minimizing idle time for smaller tasks within one stage) [46]. Perseus provides a principled approach to setting the job's end-to-end completion time (§3.1) and task execution speed (§4), and can be directly applied to big data workloads by generating a DAG of CPU computations and optimizing the CPU's frequencies.

8 Conclusion

We presented Perseus, a software-based energy optimization system for large model training. Perseus builds on the observation that there are computation imbalances at different levels in large model training that cause *intrinsic and extrinsic energy bloat*, and simultaneously reduces both with a principled graph cut-based algorithm. As a result, Perseus advances the state-of-the-art of DNN training energy optimization by delivering energy savings with negligible slowdown, thereby also reducing average power draw and enhancing the sustainability of large model training.

Acknowledgements

We would like to thank the SOSP reviewers, our shepherd Eddie Kohler, and SymbioticLab members for their insightful feedback. This work is in part supported by NSF grants CNS-2104243, CNS-2106184, and CCF-2327011, NWO VICI grant 639.023.812, grants from VMware and the Mozilla Foundation, and a gift from Salesforce. We also thank Chameleon Cloud for providing A100 nodes as well as CloudLab. Jae-Won Chung is additionally supported by the Kwanjeong Educational Foundation.

References

[1] DeepSpeed. https://github.com/microsoft/DeepSpeed.

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

[3] Megatron-LM. https://github.com/NVIDIA/Megatron-LM.

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

[5] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, et al. The falcon series of open language models. *arXiv preprint arXiv:2311.16867*, 2023.

[6] Alex Andonian, Quentin Anthony, Stella Biderman, Sid Black, Preetham Gali, Leo Gao, Eric Hallahan, Josh Levy-Kramer, Connor Leahy, Lucas Nestler, Kip Parker, Michael Pieler, Jason Phang, Shivanshu Purohit, Hailey Schoelkopf, Dashiell Stander, Tri Songz, Curt Tigges, Benjamin Thérien, Phil Wang, and Samuel Weinbach. GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch, 9 2023.

[7] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *ASPLOS*, 2019.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *NeurIPS*, 2020.

[9] Trevor Cai. Predictable scaling and infrastructure (HotChips 2024 keynote talk). OpenAI.

[10] Xiaojun Cai, Feng Li, Ping Li, Lei Ju, and Zhiping Jia. Sla-aware energy-efficient scheduling scheme for hadoop yarn. *The Journal of Supercomputing*, 73(8):3526–3546, Aug 2017.

[11] CBRE. Global data center trends 2023. https://www.cbre.com/insights/reports/global-data-center-trends-2023, 2023.

[12] CBRE. Global data center trends 2024. https://www.cbre.com/insights/reports/global-data-center-trends-2024, 2024.

[13] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. 2016.

[14] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

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

[16] Marcus Chow and Daniel Wong. CoFRIS: Coordinated frequency and resource scaling for GPU inference servers. In *IGSC*, 2024.

[17] Jae-Won Chung, Nishil Talati, and Mosharaf Chowdhury. Toward cross-layer energy optimizations in AI systems. In *Energy-Efficient Computing for Science Workshop*, 2024.

[18] ML COMMONS. MLPerf training v3.1 benchmark results. https://github.com/mlcommons/training_results_v3.1.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019.

[20] Jesse Dodge, Taylor Prewitt, Remi Tachet des Combes, Erika Odmark, Roy Schwartz, Emma Strubell, Alexandra Sasha Luccioni, Noah A. Smith, Nicole DeCario, and Will Buchanan. Measuring the carbon intensity of ai in cloud instances. In *2022 ACM Conference on Fairness, Accountability, and Transparency*, 2022.

[21] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[22] Jeff Erickson. Extensions of maximum flow. https://courses.engr.illinois.edu/cs498dl1/sp2015/notes/25-maxflowext.pdf. [Online; accessed 05-April-2023].

[23] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. DAPPLE: A pipelined data parallel approach for training large models. In *ACM PPoPP*, 2021.

[24] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[25] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. ReCycle: Pipeline adaptation for the resilient distributed training of large DNNs. In *SOSP*, 2024.

[26] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *arXiv preprint arXiv:1811.12941*, 2018.

[27] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *OSDI*, 2020.

[28] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: Long-term measurement, analysis, and implications. In *SC*, 2017.

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

[30] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. PipeTransformer: Automated elastic pipelining for distributed training of large-scale models. In *ICML*, 2021.

[31] Dorit S. Hochbaum. A polynomial time repeated cuts algorithm for the time cost tradeoff problem: The linear and convex crashing cost deadline problem. *Computers & Industrial Engineering*, 95:64–71, 2016.

[32] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Thomas Hennigan, Eric Noland, Katherine Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karén Simonyan, Erich Elsen, Oriol Vinyals, Jack Rae, and Laurent Sifre. An empirical analysis of compute-optimal large language model training. In *NeurIPS*, 2022.

[33] Qinghao Hu, Zhisheng Ye, Meng Zhang, Qiaoling Chen, Peng Sun, Yonggang Wen, and Tianwei Zhang. Hydro: Surrogate-Based hyperparameter tuning service in datacenters. In *OSDI*, 2023.

[34] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *NeurIPS*, 2019.

[35] Shadi Ibrahim, Tien-Dat Phan, Alexandra Carpen-Amarie, Houssem-Eddine Chihoub, Diana Moise, and Gabriel Antoniu. Governing energy consumption in hadoop through cpu frequency scaling: An analysis. *Future Generation Computer Systems*, 54:219–232, 2016.

[36] Shashikant Ilager, Rajeev Muralidhar, Kotagiri Rammohanrao, and Rajkumar Buyya. A data-driven frequency scaling approach for deadline-aware energy efficient scheduling on graphics processing units (GPUs). In *CCGRID*, 2020.

[37] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *SOSP*, 2023.

[38] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *ATC*, 2019.

[39] 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.

[40] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[41] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S Gunawi, Cody Hammock, et al. Lessons learned from the chameleon testbed. In *ATC*, 2020.

[42] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *ICLR*, 2017.

[43] Adam Krzywaniak, Paweł Czarnul, and Jerzy Proficz. Dynamic GPU power capping with online performance tracing for energy efficient GPU computing using DEPO tool. *Future Generation Computer Systems*,

145:396–414, 2023.

[44] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*, 2019.

[45] Zhiquan Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. Merak: An efficient distributed DNN training framework with automated 3d parallelism for giant foundation models. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1466–1478, 2023.

[46] Hongjian Li, Yaojun Wei, Yu Xiong, Enjie Ma, and Wenhong Tian. A frequency-aware and energy-saving strategy based on dvfs for spark. *The Journal of Supercomputing*, 77(10):11575–11596, Oct 2021.

[47] Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sreekumar Kodakara, David Lo, and Parthasarathy Ranganathan. Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware power capping at scale. In *OSDI*, 2020.

[48] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, Mao Yang, Lintao Zhang, and Lidong Zhou. nnScaler: Constraint-Guided parallelization plan generation for deep learning training. In *OSDI*, 2024.

[49] Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. Estimating the carbon footprint of BLOOM, a 176b parameter language model. 2022.

[50] Tobias Mann. NVIDIA turns up the AI heat with 1,200w blackwell gpus. https://www.theregister.com/2024/03/18/nvidia_turns_up_the_ai.

[51] Stathis Maroulis, Nikos Zacheilas, and Vana Kalogeraki. ExpREsS: Energy efficient scheduling of mixed stream and batch processing workloads. In *ICAC*, 2017.

[52] Eric Masanet, Nuoa Lei, and Jonathan Koomey. To better understand ai's growing energy use, analysts need a data revolution. *Joule*, 2024.

[53] McKinsey & Company. Investing in the rising data center economy. https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/investing-in-the-rising-data-center-economy#/, 2023.

[54] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. 14(5):771–784, jan 2021.

[55] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *SOSP*, 2019.

[56] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel DNN training. In *ICML*, 2021.

[57] 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.

[58] NVIDIA. Nvidia H100 tensor core GPU architecture overview. https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

[59] Kazuki Osawa, Shigang Li, and Torsten Hoefler. PipeFisher: Efficient training of large language models using pipelining and fisher information matrices. In *MLSys*, 2023.

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *NeurIPS*, 2019.

[61] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. Characterizing power management opportunities for llms in the cloud. 2024.

[62] Pratyush Patel, Zibo Gong, Syeda Rizvi, Esha Choukse, Pulkit Misra, Thomas Anderson, and Akshitha Sriraman. Towards improved power management in cloud gpus. *IEEE Computer Architecture Letters*, 22(2):141–144, 2023.

[63] 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.

[64] Steve Phillips and Mohamed I. Dessouky. Solving the project time/cost tradeoff problem using the minimal cut concept. *Management Science*, 24(4):393–400, 1977.

[65] Leonardo Piga, Iyswarya Narayanan, Aditya Sundarrajan, Matt Skach, Qingyuan Deng, Biswadip Maity, Manoj Chakkaravarthy, Alison Huang, Abhishek Dhanotia, and Parth Malani. Expanding datacenter capacity with dvfs boosting: A safe and scalable deployment experience. In *ASPLOS*, 2024.

[66] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[67] Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, and Parthasarathy Ranganathan. Data center power oversubscription with a medium voltage power plane and Priority-Aware capping. In *ASPLOS*, 2020.

[68] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *Commun. ACM*, 63(12):54–63, 2020.

[69] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[70] Martin Skutella. Approximation algorithms for the discrete time-cost tradeoff problem. *Mathematics of Operations Research*, 23(4):909–929, 1998.

[71] Martin Skutella. *Approximation and randomization in scheduling*. PhD thesis, 1998.

[72] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using deepspeed and megatron to train Megatron-Turing NLG 530b, a Large-Scale generative language model. 2022.

[73] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[74] Ola Svensson. Hardness of vertex deletion and project scheduling. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 301–312. Springer, 2012.

[75] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[76] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *NSDI*, 2023.

[77] 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.

[78] Farui Wang, Weizhe Zhang, Shichao Lai, Meng Hao, and Zheng Wang. Dynamic GPU energy optimization for machine learning training workloads. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[79] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the

wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters. In *NSDI*, 2022.

[80] Thomas Wirtz and Rong Ge. Improving MapReduce energy efficiency for computation intensive workloads. In *International Green Computing Conference and Workshops*, 2011.

[81] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *EMNLP*, 2020.

[82] BigScience Workshop. BLOOM: A 176b-parameter open-access multilingual language model. 2023.

[83] Y. Xiao, L. Ju, Z. Zhou, S. Li, Z. Huan, D. Zhang, R. Jiang, L. Wang, X. Zhang, L. Liang, and J. Zhou. Antdt: A self-adaptive distributed training framework for leader and straggler nodes. In *ICDE*, 2024.

[84] Heejin Yang, Ji-Hwan Seol, Rohit Rothe, Zichen Fan, Qirui Zhang, Hun-Seok Kim, David T. Blaauw, and Dennis Sylvester. A 1.5-μW fully-integrated keyword spotting soc in 28-nm CMOS with skip-rnn and fast-settling analog frontend for adaptive frame skipping. *IEEE J. Solid State Circuits*, 59(1):29–39, 2024.

[85] Zhenning Yang, Luoxi Meng, Jae-Won Chung, and Mosharaf Chowdhury. Chasing low-carbon electricity for practical and sustainable dnn training. 2023.

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

[87] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *Proceedings of the British Machine Vision Conference (BMVC)*, 2016.

[88] Shiwei Zhang, Lansong Diao, Chuan Wu, Siyu Wang, and Wei Lin. Accelerating large-scale distributed neural network training with spmd parallelism. In *SoCC*, 2022.

[89] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *ISCA*, 2022.

[90] 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 *USENIX OSDI*, 2022.

[91] Pengfei Zou, Ang Li, Kevin Barker, and Rong Ge. Indicator-directed dynamic power management for iterative workloads on GPU-accelerated systems. In *CCGRID*, 2020.

[92] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, and Henri Bahini. Resiliency at scale: Managing Google's TPUv4 machine learning supercomputer. In *NSDI*, 2024.

[93] Matej Špeťko, Ondřej Vysocký, Branislav Jansík, and Lubomír Říha. DGX-A100 face to face DGX-2 – performance, power and thermal behavior evaluation. *Energies*, 14(2), 2021.

The Appendix has not been peer-reviewed and is provided for supplementary information.

A Visualizations for Intrinsic Energy Bloat

Figure 10 shows the timeline of running one training iteration of BERT 1.3B, T5 3B, Bloom 3B, and Wide-ResNet101 1.5B for maximum frequency and Perseus-optimized energy schedule, respectively. For visualization purposes, we set the number of microbatches to 6. Real evaluation workloads have more microbatches. Energy schedule found by Perseus successfully tunes down frequency for all models without slowing down the iteration time, tightly packing computations over time and reducing intrinsic energy bloat.

B Workload Details

B.1 Minimum Imbalance Pipeline Partitioning

We partition layers of a model into N stages such that the imbalance ratio, defined as the ratio of the longest stage forward latency to the shortest, is minimized. We only consider forward computation time as backward computations are typically proportional to forward computation latency. For Transformer-based models, we define layer as one Transformer layer. For Wide-ResNet, we define layer as one Bottleneck layer, which is three convolution layers wrapped with a skip connection. Due to P2P communication overhead and implementation challenges, many planners and frameworks do not support partitioning in the middle of skip connections. We call this *minimum imbalance pipeline partitioning*, and throughout the paper, every workload we use is partitioned as such.

Table 7 shows the computation time ratios of the heaviest stage to the lightest stage for 4 and 8 pipeline stages. More pipeline stages generally increases imbalance due to the coarse-grained nature of tensor operations. That is, the relative size of each layer becomes smaller and smaller compared to the total amount of computation allocated to each stage, and imbalance increases.

GPT-3, Bloom, and BERT. Arguably, these are one of the most homogeneous large models because they are a stack of identical Transformer [77] encoder or decoder layers. However, the very last layer is the language modeling head, which maps the final features to probabilities over the entire vocabulary. The vocabulary size of GPT-3 is 50k, Bloom 251k, and BERT 31k, which results in a very large linear layer for the last stage. This leads to varying amounts of imbalance and different minimum imbalance partitioning results for each model.

T5. This is also based on Transformer layers, but the first half of the layers are encoders, while the later half are decoders (which corresponds to the original Transformer [77] model's architecture). However, the decoder layers as an extra cross attention layer in the middle, making it computationally heavier. Finally, T5 also ends with a language model head with 32k vocabulary size. However, minimum imbalance partitioning still balances T5 to a reasonable degree, although it cannot be perfectly balanced.

Wide-ResNet. For Wide-ResNet, in order to make it suitable for large model training evaluation, we used the variant with width factor 8. Wide-ResNet is a collection of Bottleneck layers with three convolutions wrapped with a skip connection, and there are four different sizes of Bottleneck layers laid out sequentially. Therefore, even with minimum imbalance partitioning, it is difficult to perfectly balance stages.

B.2 Does Imbalance Decrease with Larger Models?

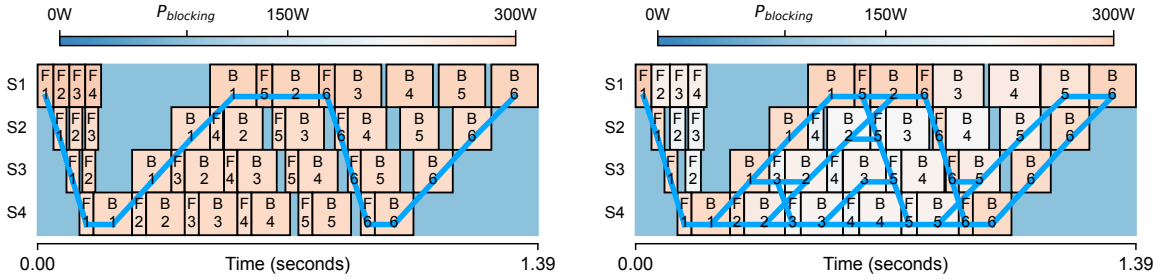
Specifically for Transformer [77]-based models with homogeneous intermediate Transformer layers, the degree of imbalance will decrease *if the number of pipeline stages is held constant*. This is because the relative size of the embedding layer and language model head will decrease as intermediate Transformer layers increase in number and size. However, as the model grow larger, the number of pipeline stages will have to be increased simultaneously. This increases imbalance, because the relative amount of computation in one layer with respect to the amount in one pipeline stage becomes larger. As such, there is no simple relationship between model size and the amount of imbalance.

B.3 Alternative Planning Methods

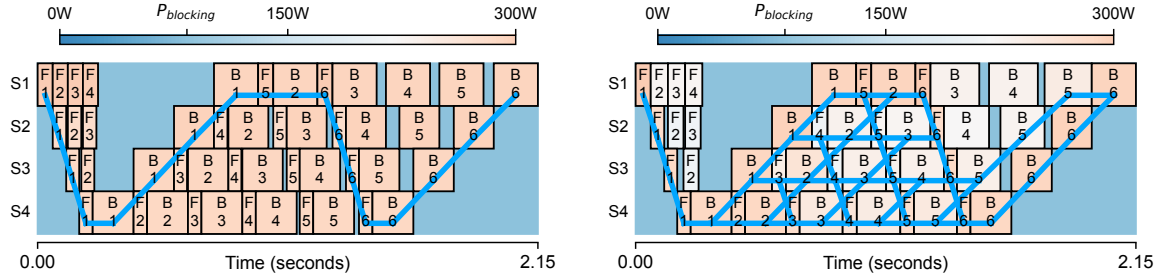
3D parallelism is the go-to solution for large model training, and our minimum imbalance pipeline partitioning method is optimal for 3D parallelism because we implemented a brute force search. More advanced planning methods such as Alpa [90] exist, but for repetitive language model architectures like GPT-3, Alpa equally allocates Transformer layers to each stage [90], resulting in the same stage imbalance caused by the language modeling head. Furthermore, it is hard for tensor parallelism to divide operations infinitely; practically at most degree 8 (within one node) due to collective communication overhead. Thus, in the vast majority of models, there will always be some degree of imbalance between stages due to the minimum granularity of computation time.

B.4 Experiment Parameters

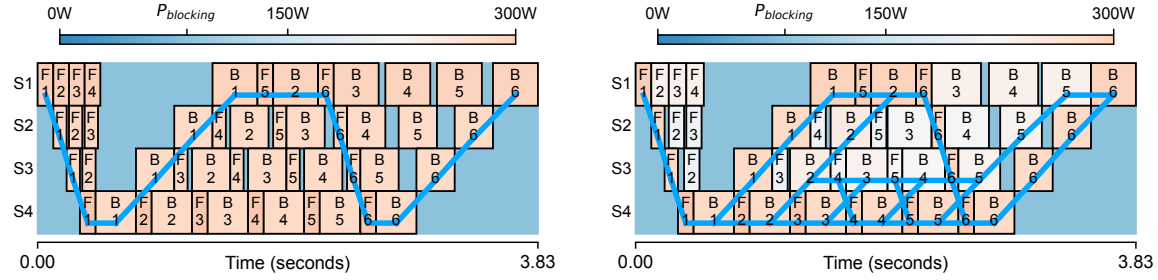
Tables 8, 9, and 10 list model variant names and experiment parameters for our experiments. Model names and configurations for GPT-3 were taken as is from the original model publication [8]. Especially, model names and configurations for BERT and T5 were directly taken from the Huggingface Hub pretrained model zoo, except for the huge variant of BERT, which we created to have hidden dimension 2048. Wide-ResNet was based on Torch Vision [60] but scaled up following the model's original publication [87] using its



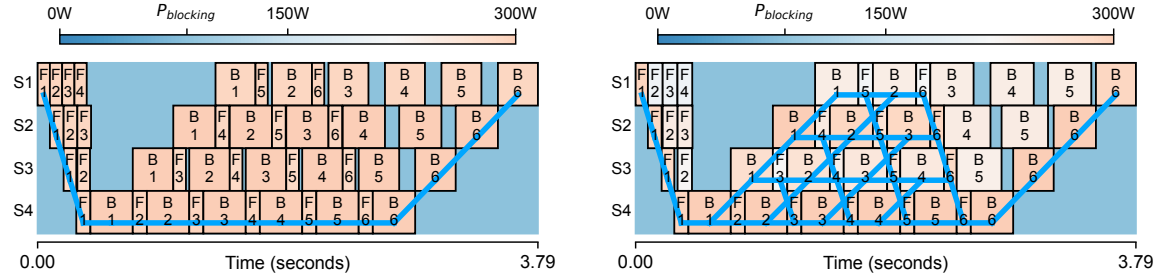
(a) BERT 1.3B



(b) T5 3B



(c) Bloom 3B



(d) Wide-ResNet101 1.5B

Figure 10. Visualization of Perseus's $T_{\min}$ solution for four stage pipeline workloads on NVIDIA A100 PCIe GPUs. For each workload, the left is running every computation at maximum frequency, and the right is Perseus's energy schedule that reduces only intrinsic energy bloat without inflating iteration time. Note that these are *not* real workloads we run in Section 6; real workloads have far more microbatches.

width factor parameter. The unit time parameter τ was set to 1 ms for all experiments.

C Pipeline Energy Minimization is NP-hard

The Pipeline Energy Minimization problem is stated again for convenience:

$$\begin{aligned} \min_{F} \quad & \text{Energy}(F) \\ \text{s.t.} \quad & \text{Time}(F) \leq T' \end{aligned} \tag{6}$$

Model	Size	Imbalance Ratio		Minimum Imbalance Ratio Partition	
		4 stages	8 stages	4 stages	8 stages
GPT-3 [8]	1B	1.17	1.33	[0, 6, 12, 19, 25]	[0, 4, 7, 10, 13, 16, 19, 22, 25]
	3B	1.13	1.25	[0, 8, 16, 25, 33]	[0, 5, 9, 13, 17, 21, 25, 29, 33]
	7B	1.11	1.23	[0, 8, 16, 24, 33]	[0, 4, 8, 12, 16, 20, 24, 28, 33]
	13B	1.08	1.17	[0, 10, 20, 30, 41]	[0, 5, 10, 15, 20, 25, 30, 35, 41]
	175B	1.02	1.03	[0, 24, 48, 72, 97]	[0, 12, 24, 36, 48, 60, 72, 84, 97]
Bloom [82]	3B	1.13	1.25	[0, 9, 17, 25, 31]	[0, 6, 11, 16, 22, 28, 31]
	7B	1.13	1.25	[0, 9, 17, 25, 31]	[0, 6, 11, 16, 22, 28, 31]
	176B	1.05	1.10	[0, 18, 36, 54, 71]	[0, 9, 18, 27, 36, 45, 54, 63, 71]
BERT [19]	0.1B	1.33	2.00	[0, 4, 7, 10, 13]	[0, 2, 3, 4, 6, 8, 10, 12, 13]
	0.3B	1.17	1.33	[0, 7, 13, 19, 25]	[0, 3, 6, 9, 12, 15, 18, 22, 25]
	1.3B	1.17	1.33	[0, 7, 13, 19, 25]	[0, 4, 7, 10, 13, 16, 19, 22, 25]
T5 [66]	0.2B	1.19	1.50	[0, 9, 15, 20, 25]	[0, 5, 9, 13, 15, 17, 19, 22, 25]
	0.7B	1.05	1.11	[0, 16, 29, 39, 49]	[0, 8, 16, 24, 29, 34, 39, 44, 49]
	2.9B	1.06	1.16	[0, 15, 28, 38, 49]	[0, 7, 15, 23, 28, 33, 38, 43, 49]
Wide-ResNet50 [87]	0.8B	1.23	1.46	[0, 5, 9, 14, 18]	[0, 3, 5, 7, 9, 11, 13, 15, 18]
Wide-ResNet101 [87]	1.5B	1.09	1.25	[0, 8, 17, 26, 35]	[0, 4, 8, 12, 16, 21, 26, 31, 35]

(a) NVIDIA A100 PCIe GPUs.

Model	Size	Imbalance Ratio		Minimum Imbalance Ratio Partition	
		4 stages	8 stages	4 stages	8 stages
GPT-3 [8]	1B	1.15	1.31	[0, 6, 12, 18, 25]	[0, 3, 6, 9, 12, 15, 18, 21, 25]
	3B	1.11	1.21	[0, 8, 16, 24, 33]	[0, 4, 8, 12, 16, 20, 24, 28, 33]
	7B	1.08	1.17	[0, 8, 16, 24, 33]	[0, 4, 8, 12, 16, 20, 24, 28, 33]
	13B	1.07	1.14	[0, 10, 20, 30, 41]	[0, 5, 10, 15, 20, 25, 30, 35, 41]
	175B	1.01	1.02	[0, 24, 48, 72, 97]	[0, 12, 24, 36, 48, 60, 72, 84, 97]
Bloom [82]	3B	1.13	1.25	[0, 9, 17, 25, 31]	[0, 5, 9, 13, 17, 21, 25, 29, 31]
	7B	1.13	1.25	[0, 9, 17, 25, 31]	[0, 5, 9, 13, 17, 21, 25, 29, 31]
	176B	1.03	1.06	[0, 18, 36, 54, 71]	[0, 9, 18, 27, 36, 45, 54, 63, 71]
BERT [19]	0.1B	1.33	2.00	[0, 4, 7, 10, 13]	[0, 1, 2, 4, 6, 8, 10, 12, 13]
	0.3B	1.17	1.33	[0, 7, 13, 19, 25]	[0, 4, 7, 10, 13, 16, 19, 22, 25]
	1.3B	1.17	1.33	[0, 7, 13, 19, 25]	[0, 3, 6, 9, 12, 15, 18, 22, 25]
T5 [66]	0.2B	1.20	1.50	[0, 9, 15, 20, 25]	[0, 5, 9, 13, 15, 17, 19, 22, 25]
	0.7B	1.06	1.12	[0, 16, 29, 39, 49]	[0, 8, 16, 24, 29, 34, 39, 44, 49]
	2.9B	1.07	1.17	[0, 15, 28, 38, 49]	[0, 8, 15, 23, 28, 33, 38, 43, 49]
Wide-ResNet50 [87]	0.8B	1.13	1.72	[0, 5, 9, 14, 18]	[0, 3, 5, 7, 9, 11, 13, 15, 18]
Wide-ResNet101 [87]	1.5B	1.08	1.25	[0, 8, 17, 26, 35]	[0, 4, 8, 12, 16, 21, 26, 31, 35]

(b) NVIDIA A40 GPUs.

Table 7. Imbalance ratio between the longest and the shortest stages for various models. 1.00 would mean perfect balance. Partitions for N stages is a list of $N + 1$ numbers, where the numbers represent layer indices. For instance, [0, 6, 12, 19, 25] for GPT-3 1.3B means there are 6, 6, 7, and 5 Transformer layers in each stage, and the final stage also has the language model head.

Model	# Parameters	Global Batch Size	Microbatch Size	# Microbatches
gpt3-6.7b	6.7 B	1024	4	128

Table 8. Experiment Parameters for 3D parallelism experiments on A40 GPUs. Microbatch size is per-pipeline, and there are two data parallel copies of the same pipeline. Thus, global batch size should be calculated as the product of microbatch size and the number of microbatches times two.

where F is the frequency assignment for each pipeline computation $i \in \mathcal{G}$, T' is the straggler pipeline's iteration time.

The decision problem corresponding to Equation 6 asks whether it is possible to find frequency assignment F such

Model	# Parameters	Global Batch Size	Microbatch Size	# Microbatches
gpt3-2.7b	2.7 B	1024	4	256
bert-huge-uncased	1.3 B	256	8	32
t5-3b	3.0 B	128	4	32
bloom-3b	3.0 B	512	4	128
wide-resnet101 (width factor 8)	1.5 B	1536	32	48

Table 9. Experiment Parameters for eight-stage pipeline parallelism experiments on A40 GPUs. Model variant names are as described in Torch Vision [60] or Huggingface Hub [81].

Model	# Parameters	Global Batch Size	Microbatch Size	# Microbatches
gpt3-xl	1.3 B	512	4	128
bert-huge-uncased	1.3 B	256	8	32
t5-3b	3.0 B	128	4	32
bloom-3b	3.0 B	512	4	128
wide-resnet101 (width factor 8)	1.5 B	1536	64	24

Table 10. Experiment Parameters for Pipeline Parallelism Experiments on A100 PCIe GPUs. Model variant names are as described in Torch Vision [60] or Huggingface Hub [81].

that the total energy consumption is minimized while the iteration time of pipeline $\mathcal{G}$ is no longer than the straggler's iteration time. We denote this problem PEM.

In the following, we show that a simplification of PEM is NP-hard by reduction from the 0/1 Knapsack problem, which makes PEM NP-hard.

C.1 One Stage Two Frequencies Simplification

A simplification of PEM is considering the case where there is only one pipeline stage and two frequencies to choose from.

For each pipeline computation $i \in \{1, 2, \ldots, n\}$, we can set the GPU frequency to either the lowest value or the highest, denoted as $[L, H]$ respectively. Choosing different frequencies will lead to different execution time and energy consumption. That is, if i is chosen to execute at frequency L, it will take $t_i(L)$ time and $e_i(L)$ energy. On the other hand, if i executes at frequency H, it takes $t_i(H)$ time and $e_i(H)$ energy. The time and energy consumption of i are rational numbers, as they are rounded up to τ.

Our goal is to minimize the energy consumption of executing all computations while satisfying the time constraint. Specifically, given a time deadline T', we want to pick a subset of operations $J \subseteq \{1, 2, \ldots, n\}$ and assign them to execute at the lowest frequency L and execute the rest of the operations with the highest frequency H, such that the total time needed to execute all operations is smaller than or equal to the deadline:

$$\sum_{i=1}^{n} (X_i t_i(L) + (1 - X_i) t_i(H)) \leq T'$$

where X_i is a 0/1 indicator variable where $X_i = 1$ if $i \in J$ and $X_i = 0$ otherwise. Under this time constraint, the goal is to minimize the total energy consumption of executing all computations:

$$\sum_{i=1}^{n} (X_i e_i(L) + (1 - X_i) e_i(H)).$$

Formally, we denote this problem as

$$\textsc{pem-1d}(T = (T_L, T_H), E = (E_L, E_H), T', EC)$$

where $T_L = [t_1(L), \ldots, t_n(L)]$, $T_H = [t_1(H), \ldots, t_n(H)]$ are execution time vectors for low and high frequency respectively, $E_L = [e_1(L), \ldots, e_n(L)]$, $E_H = [e_1(H), \ldots, e_n(H)]$ are energy consumption vectors for low and high frequency respectively, and EC the target energy consumption.

PEM-1D returns true if and only if there exists a subset of operations $J \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i=1}^{n} (X_i t_i(L) + (1 - X_i) t_i(H)) \leq T'$ and $\sum_{i=1}^{n} (X_i e_i(L) + (1 - X_i) e_i(H)) \leq EC$.

C.2 0/1 Knapsack Problem

Consider two length n arrays containing positive integer weights $W = (w_1, w_2, \ldots, w_n)$ and values $V = (v_1, v_2, \ldots, v_n)$ where the ith item has weight $w_i \in \mathbb{Q}^+$ and value $v_i \in \mathbb{Q}^+$, and a knapsack with weight capacity $C \in \mathbb{Q}^+$.

The goal is to pick a subset of items $S \subseteq \{1, 2, \ldots, n\}$, such that the total weight of the chosen items is less than or equal to the weight capacity: $\sum_{i \in S} w_i \leq C$. Under this constraint, the goal is to maximize the total value of items in the knapsack: $\sum_{i \in S} v_i$.

Formally, we denote the decision problem of 0/1 Knapsack as

$$\textsc{knapsack}(W[1, \ldots, n], V[1, \ldots, n], C, P)$$

where $P \in \mathbb{Q}$ is the target value.

KNAPSACK returns true if and only if there exists a subset of items $S \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in S} w_i \leq C$ and $\sum_{i \in S} v_i \geq P$.

It is well known that KNAPSACK is NP-hard.

C.3 NP-hardness Proof

Theorem C.1. *PEM-1D is NP-hard.*

Proof. We will show that KNAPSACK $\leq_p$ PEM-1D, i.e., the 0/1 Knapsack problem is polynomial-time reducible to the simplified pipeline energy minimization problem. Reduction function f takes $(W[1, \ldots, n], V[1, \ldots, n], C, P)$ as input and does the following:

1. Construct n computations and empty vectors T_L, T_H, E_L, E_H.
2. For $\forall i$, set $t_i(L) = w_i$ and append to T_L.
3. For $\forall i$, set $t_i(H) = 0$ and append to T_H.
4. For $\forall i$, set $e_i(L) = -v_i$ and append to E_L.
5. For $\forall i$, set $e_i(H) = 0$ and append to E_H.
6. Set $T' = C$ and $EC = -P$.
7. Output $(T = (T_L, T_H), E = (E_L, E_H), T', EC)$

Correctness Analysis If $(W[1, \ldots, n], V[1, \ldots, n], C, P) \in$ KNAPSACK, there exists a subset S such that $\sum_{i \in S} w_i \leq C$ and $\sum_{i \in S} v_i \geq P$. Now for PEM-1D$(T = (T_L, T_H), E = (E_L, E_H), T', EC)$, select computations that have the same indices as items in S to execute at low frequency L, while executing others at high frequency H. Then, for the time constraint, $\sum_{i=1}^{n}(X_i t_i(L) + (1 - X_i)t_i(H)) = \sum_{i=1}^{n} X_i t_i(L) = \sum_{i \in S} w_i \leq C = T'$, and for target energy, $\sum_{i=1}^{n}(X_i e_i(L) + (1 - X_i)e_i(H)) = \sum_{i=1}^{n} X_i e_i(L) = \sum_{i \in S} -v_i \leq -P = EC$.

If $(W[1, \ldots, n], V[1, \ldots, n], C, P) \notin$ KNAPSACK, there does not exist a subset S such that $\sum_{i \in S} w_i \leq C$ and $\sum_{i \in S} v_i \geq P$. There are two possibilities: either a subset S that satisfies the weight constraint does not exist at all $(w_i > C, \forall i)$ or none of the subsets S that satisfy the weight constraint satisfy $\sum_{i \in S} v_i \geq P$. For the first possibility, this means all the computations must select the high frequency as the low frequency does not satisfy the time constraint. Then total energy consumption is 0, which is larger than $EC = -P$ since $P \in \mathbb{Q}^+$. For the second possibility, for all subsets S, $\sum_{i \in S} v_i < P$, which means that for all subsets of computations $\sum_{i=1}^{n}(X_i e_i(L) + (1 - X_i)e_i(H)) = \sum_{i \in S} -v_i > -P = EC$, so none of them satisfy the energy constraint.

Efficiency Analysis Step 1–5 each takes $O(n)$ time. Step 6 takes $O(1)$ time. Finally, step 7 takes $O(n)$ time.

Therefore, the function f takes $O(n)$ time, which is polynomial time w.r.t the input size.

□

D Continuous Relaxation

The Pipeline Energy Minimization problem is NP-Hard, as we have proved in Appendix C. Thus, we perform continuous relaxation for the problem by fitting an exponential function to each Pareto-optimal (time, energy) points for each forward and backward computation in each stage. Furthermore, the exponential function captures the nature of *diminishing returns* well, in that the amount of energy consumption needed to reduce computation time by a constant amount increases multiplicatively.

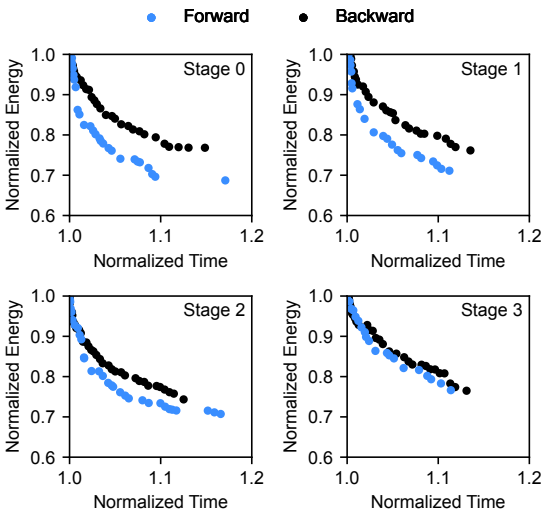


Figure 11. Pareto-optimal (time, energy) choices for forward and backward computations in each stage of GPT-3 0.3B on NVIDIA A40 GPUs.

Figure 11 visualizes Pareto-optimal (time, energy) measurements for GPT-3 0.3B on A40 GPUs, showing that the exponential function is a natural fit for data. This was our consistent observation across different GPUs and models in our evaluation.

E Full Details of GetNextSchedule

For the sake of presentation, we made a simplifying statement in the body of the paper that computations only speed up by τ. However, since we aim to speed up all critical paths by precisely τ, speeding up more than one computation from a critical path allows other computations on that critical path to be slowed down. We always take such slowdown opportunity because it will decrease energy consumption.

In the following, we describe the procedure of annotating edges with flow capacities and then solving the problem with maximum flow.

E.1 Generating Capacity DAG

On the computation DAG, we first remove all the computations that are not on any of the critical paths and construct the *Critical DAG*. We would like to find a set of edges I^+ to speed up by τ and I^- to slow down by τ on the Critical DAG, such that the total energy consumption increases minimally.

This can be described as the following problem:

$$\min_{I^+,\, I^-} \quad \sum_{i \in I^+} e_i^+ - \sum_{i \in I^-} e_i^-, \tag{7}$$

where $e_i(t_i)$ is the exponential function fit to Pareto-optimal (time, energy) measurements for computation i, $e_i^+ = e_i(t_i - \tau) - e_i(t_i)$ is the extra amount of energy needed to speed up i by τ and $e_i^- = e_i(t_i) - e_i(t_i + \tau)$ is the energy saved from slowing down i by τ.

An important fact that leads us to the solution is that (i) the problem of finding the set of edges to modify such that energy increase is minimized (i.e., solving Equation 7) coincides with (ii) finding the *minimum cut* of a DAG whose lower and upper bound flow capacities are defined as

$$(l_i, u_i) = \begin{cases} (0, e_i^+) & \text{if } t_i \text{ is longest possible (slowest)} \\ (e_i^-, \infty) & \text{if } t_i \text{ is shortest possible (fastest)} \\ (e_i^-, e_i^+) & \text{otherwise.} \end{cases} \tag{8}$$

This equivalence was given by the theoretical works of Phillips and Dessouky [31, 64].

Thus, we construct the *Capacity DAG* from the Critical DAG by annotating its edges with flow capacities given by Equation 8. The capacity of an $S - T$ cut (S is the set of nodes on the source side and T the sink side) on the Capacity DAG with $S \to T$ edges I^+ and $T \to S$ edges I^- is identical to the objective in Equation 7. Then, we use the Edmonds-Karp maximum flow algorithm [21] to find the minimum cut of the Capacity DAG. Finally, after the minimum cut has been identified from the Capacity DAG, edges in I^+ are sped up by τ and those in I^- are slowed down by τ, ultimately reducing the length of every critical path exactly by τ with the smallest possible energy increase.

E.2 Max Flow Algorithm on the Capacity DAG

A characteristic of our Capacity DAG that precludes the direct application of well-known max flow algorithms is that some edges also have flow *lower bounds*, asserting that at least a certain amount of flow must pass through in the edge. However, the Max Flow Min Cut theorem by Ford and Fulkerson holds for the case of non-zero flow lower bounds (See Chapter 1, Section 9 of [24]), allowing us to find the minimum cut (which is equivalent to the minimum energy modification set) using any maximum flow algorithm. We adopt an approach that adds dummy source/sink nodes to create a DAG that has zero flow lower bounds, finds the maximum flow on the new DAG, and extracts the flow so that it corresponds to a flow in the original DAG [22]. The algorithm is given in Algorithm 3.

Input: Directed Acyclic Graph $G = (V, E)$
Source node $s \in V$ and sink node $t \in V$
Lower and upper bounds $l(e)$, $u(e)$ for $\forall e \in E$
Output: A maximum feasible flow of G if it exists

Construct a new graph $G' = (V', E')$ by adding new source and sink nodes s' and t', edges from s' to each node in V, edges from each node in V to t', and an edge from t to s
▷ Define the capacity $c'(e)$ of each edge $e \in E'$
for $v \in V$ **do**
  $c'(s'v) \leftarrow \sum_{u \in V} l(uv)$
  $c'(vt') \leftarrow \sum_{w \in V} l(vw)$
for $uv \in E$ **do**
  $c'(uv) \leftarrow u(uv) - l(uv)$
$c'(ts) \leftarrow \infty$
▷ Find the max flow on G'
$f' \leftarrow$ `EdmondsKarp` (G', c')
▷ G has a feasible flow if and only if G' has a saturating flow
if `FlowValue` $(f') \neq \sum_{v \in V} c'(s'v)$ **then**
  return *nil*
▷ Convert f' to a feasible flow f in G
for $uv \in E$ **do**
  $f(uv) \leftarrow f'(uv) + l(uv)$
▷ Construct residual graph and improve f to max flow
for $uv \in E$ **do**
  $c(uv) \leftarrow u(uv) - f(uv)$
  $c(vu) \leftarrow f(vu) - l(vu)$
return `EdmondsKarp` (G, c)

Algorithm 3: Maximum Flow with Lower Bounds

F Proof for Polynomial Runtime

Perseus enumerates the entire time–energy frontier one by one, and the runtime of one iteration is polynomial time with respect to the number of stages N and the number of microbatches M. Thus, determining whether the entire algorithm runs in polynomial time reduces to whether the worst case number of iterations is polynomial with respect to N and M. While for general DAGs the maximum number of points on the frontier can be exponential with respect to the size of the DAG [71], here we prove that under mild assumptions for DAGs that represent pipeline schedules, the number of iterations is $O(N+M)$. The assumptions are valid for all pipeline schedules known to the authors, including GPipe [34] and 1F1B [56].

Theorem F.1. *For DAGs that represent pipeline schedules, the number of iterations needed is* $O(N + M)$.

Proof. Since we always reduce iteration time by τ, the number of iterations is

$$\frac{t_{\max} - t_{\min}}{\tau}$$

where $t_{\max}$ and $t_{\min}$ are the maximum and minimum possible iteration time, respectively.

Assume that any pipeline schedule representing one iteration of training has a prologue, a steady state, and an

```python
# In the framework's pipeline execution engine:
from perseus.client import profiler, controller

def train_step(model, dataloader):
    ...

    for instuction in pipeline_schedule:
        if isinstance(instruction, Forward):
            controller.set_speed("forward")
            profiler.begin("forward")
            # Run forward on microbatch
            profiler.end("forward")
        elif isinstance(instruction, Backward):
            controller.set_speed("backward")
            profiler.begin("backward")
            # Run backward on microbatch
            profiler.end("backward")
    ...
```

Listing 1. Perseus client API integration example.

epilogue. The prologue is when the pipeline starts from an empty state and is gradually filled with pipeline computations, while the epilogue is when the pipeline is drained to reach an empty state. It is easy to see that the number of pipeline computations on the critical path of both the prologue and epilogue is $O(N)$, as deeper pipelines (larger N) take longer to fill. On the other hand, the steady state of the pipeline is when the pipeline is completely filled, and the number of pipeline computations in any simple path through the steady state of the DAG is $O(M)$. Therefore, the total number of pipeline computations in the critical path of the entire DAG is $O(N + M)$.

$t_{\max}$ and $t_{\min}$ can be constructed by multiplying the number of computations with the average execution time of a computation. Computations are executed with frequencies $f_{\min}$ and $f_{\max}$, respectively, and thus the multipliers of N and M do not cancel out when $t_{\max} - t_{\min}$ is evaluated. Therefore, $t_{\max} - t_{\min}$, and hence $(t_{\max} - t_{\min})/\tau$, is $O(N + M)$. □

G Perseus Client Integration Example

Listing 1 shows an example integration of Perseus's Client API (first three rows in Table 2) with a hypothetical (but typical) training framework's pipeline execution engine.

A typical structure of a pipeline execution engine is to have *instructions* for each distinct operations in the pipeline, including not only forward and backward executions, but also P2P and collective communications, and implement a handler for each instruction. Therefore, framework developers can wrap such handlers with the Perseus client APIs to mark their beginning and end.

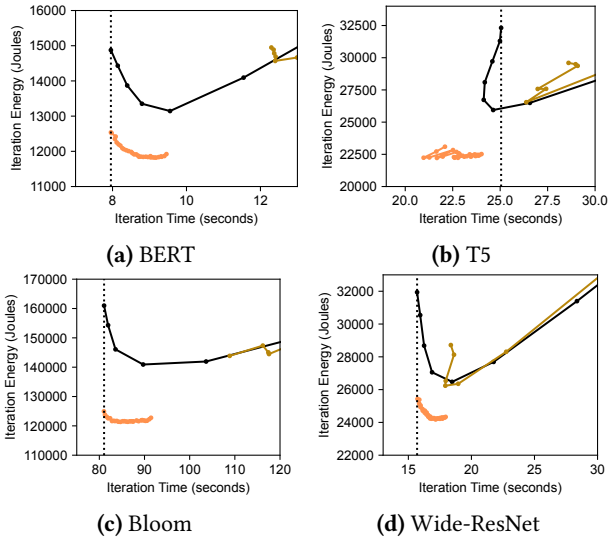


Figure 12. Eight stage pipeline parallelism on A40.

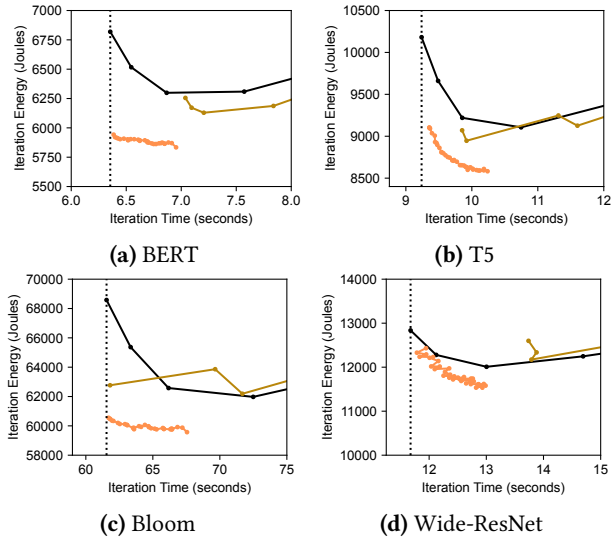
H Time–Energy Frontiers

Figure 12 shows the time–energy frontiers achieved by Perseus and the two baseline approaches for the rest of the workloads ran with eight stage pipeline parallelism, measured in NVIDIA A40 GPUs.

T5 shows an interesting frontier due to the hardware topology of our A40 machine setup: Each node has four GPUs and NVLink connects GPUs 0 and 1, and 2 and 3; GPUs 1 and 2 must communicate through the NUMA interconnect; Finally, different nodes are connected with Infiniband only adjacent to GPUs 0 and 1 (data to and from GPUs 2 and 3 must also go through the NUMA interconnect). The implication of this heterogeneous GPU interconnect is that if more than one P2P communications that need to go through the NUMA interconnect happen at the same time, contention happens and both of data transfers slow down significantly. However, Perseus's plan reduces this contention, overall decreasing iteration time noticeably. Yet, contention and noisy communication latencies still exist, leading to a noisy frontier.

For many ZeusPerStage lines, energy fluctuates significantly when iteration time increases due to ZeusPerStage being unaware of critical paths. Balancing the forward computation time between stages could even let the modified stages take over the critical path. As a result, the iteration time increases, which increases energy bloat, and more energy is spent on blocking on communication (§4.1). When the decreased energy on computation fails to cover the increased energy on P2P communication, total energy increases.

Figure 13 shows the time–energy frontiers achieved by Perseus and the two baseline approaches for the rest of the workloads, measured with four stage pipeline parallelism in NVIDIA A100 PCIe GPUs. Wide-ResNet has a noisy frontier because the variability in microbatch loading time introduces noise in the end-to-end iteration time when computations



(a) BERT **(b)** T5

(c) Bloom **(d)** Wide-ResNet

Figure 13. 4 stage pipeline parallelism on A100 PCIe.

are tightly packed by Perseus. This was not pronounced in A40 GPUs because compared to A100 PCIe, computation is slower, but data loading time is similar. Thus, the noise in data loading time becomes more noticeable in A100 PCIe.