PolyThrottle: Energy-efficient Neural Network Inference on Edge Devices

Minghao Yan
Department of Computer Sciences
University of Wisconsin-Madison
myan@cs.wisc.edu

Hongyi Wang
School of Computer Science
Carnegie Mellon University
hongyiwa@andrew.cmu.edu

Shivaram Venkataraman
Department of Computer Sciences
University of Wisconsin-Madison
shivaram@cs.wisc.edu

Abstract

As neural networks (NN) are deployed across diverse sectors, their energy demand correspondingly grows. While several prior works have focused on reducing energy consumption during training, the continuous operation of ML-powered systems leads to significant energy use during inference. This paper investigates how the configuration of on-device hardware—elements such as GPU, memory, and CPU frequency, often neglected in prior studies, affects energy consumption for NN inference with regular fine-tuning. We propose PolyThrottle, a solution that optimizes configurations across individual hardware components using Constrained Bayesian Optimization in an energy-conserving manner. Our empirical evaluation uncovers novel facets of the energy-performance equilibrium showing that we can save up to 36 percent of energy for popular models. We also validate that PolyThrottle can quickly converge towards near-optimal settings while satisfying application constraints.

1 Introduction

The rapid advancements in neural networks and their deployment across various industries have revolutionized multiple aspects of our lives. However, this sophisticated technology carries a drawback: high energy consumption which poses serious sustainability and environmental challenges [1, 2, 3, 4]. Emerging applications such as autonomous driving systems and smart home assistants require real-time decision-making capabilities [5], and as we integrate NNs into an ever-growing number of devices, their collective energy footprint poses a considerable burden to our environment [6, 7, 8]. Moreover, considering that many devices operate on battery power, curbing energy consumption not only alleviates environmental concerns but also prolongs battery life, making low-energy NN models highly desirable for numerous use cases.

In prior literature, strategies for reducing energy consumption revolve around designing more efficient neural network architectures [9, 10], quantization [11, 12, 13, 14, 15], or optimizing maximum GPU frequency [16, 17]. From our experiments, we make new observations about the tradeoffs between energy consumption, inference latency, and various other hardware configurations. Memory frequency, for example, emerges as a significant contributor to energy consumption (as shown in Figure 3), beyond the commonly investigated relationship between maximum GPU compute frequency and energy consumption. Table 2 shows that even with optimal maximum GPU frequency,

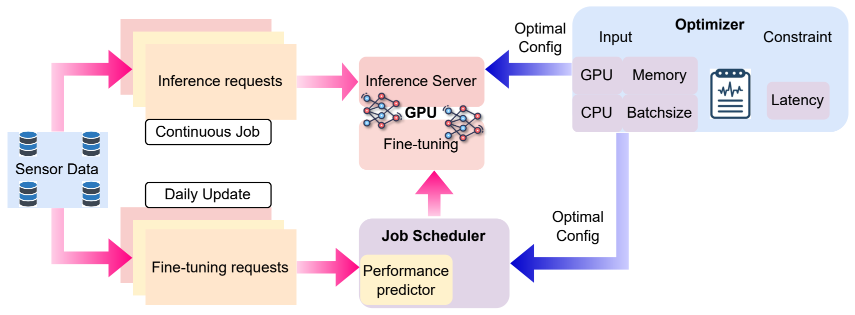


Figure 1: Figure illustrating the overall workflow of PolyThrottle. The optimizer first identifies the optimal hardware configuration for a given model. When new data arrives, the inference server handles the inference requests. Upon receiving a fine-tuning request, our performance predictor estimates whether time-sharing inference and fine-tuning workloads would result in SLO violations. Then the predictor searches for feasible adjustments to meet the SLO constraints. If such adjustments are identified, the system implements the changes and schedules fine-tuning requests until completion.

we can save up to 25% energy by further tuning memory frequency. In addition, minimum GPU frequency also proves to be of importance in certain cases, as shown in Figure 3 and Table 3.

We also observe that a simple linear relationship falls short of capturing the tradeoff between energy consumption, neural network inference latency, and hardware configurations. The complexity of this tradeoff is illustrated by the Pareto Frontier in Figure 2. This nuanced interplay between energy consumption and latency poses a challenging question: *How can we find a near-optimal configuration that closely aligns with this boundary?*

Designing an efficient framework to answer the above question is challenging due to the large configuration space, the need to re-tune each model and hardware, and frequent fine-tuning operations. A naive approach, such as grid search, is inefficient and can take hours to find the optimal solution for a given model and desired batch size on a given hardware. The uncertainty in inference latency, especially at smaller batch sizes [18], further exacerbates the challenge. Furthermore, given that distinct hardware platforms and NN models display unique energy consumption patterns (Section 3), relying on a universally applicable pre-computed optimal configuration is not feasible. Every deployed device must be equipped to quickly identify its best configuration tailored to its specific workload. Finally, in production environments, daily fine-tuning is often necessary to adapt to a dynamic external environment and integrate new data [19, 20]. This demands a mechanism that can quickly adjust configurations to complete fine-tuning requests in time while ensuring the online inference workloads meet Service Level Objectives (SLOs).

In this paper, we explore the interplay between inference latency, energy consumption, and hardware frequency and propose PolyThrottle as our solution. PolyThrottle takes a holistic approach, optimizing various hardware components and batch sizes concurrently to identify near-optimal hardware configurations under a predefined latency SLO. PolyThrottle complements existing efforts to reduce inference latency, including pruning, quantization, and knowledge distillation. We use Constrained Bayes Optimization with GPU, memory, CPU frequencies, and batch size as features, and latency SLO as a constraint to design an efficient framework that automatically adjusts configurations, enabling convergence towards near-optimal settings. Furthermore, PolyThrottle uses a performance prediction model to schedule fine-tuning operations without disrupting ongoing online inference requests. We integrate PolyThrottle into Nvidia Triton on Jetson TX2 and Orin and evaluate on state-of-the-art CV and NLP models, including EfficientNet and Bert [10, 21].

To summarize, our key contributions include:

1. We examine the influence of hardware components beyond GPUs on energy consumption, delineate new tradeoffs between energy consumption and inference performance, and reveal new possibilities for optimization.

2. We construct an adaptive framework that efficiently finds energy-optimal hardware configurations. To accomplish this, we employ Constrained Bayesian Optimization.

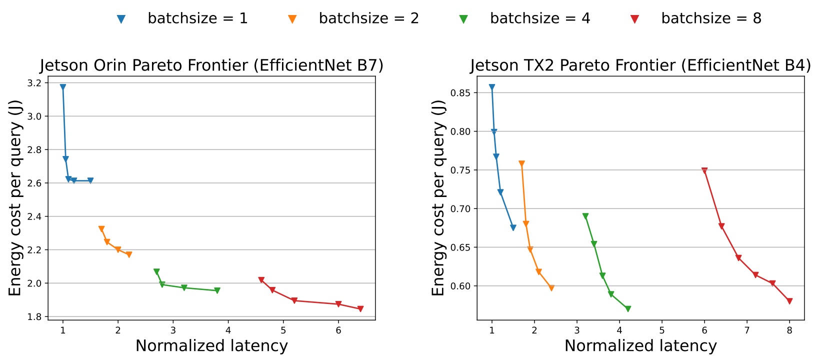


Figure 2: **Left** figure shows the Pareto Frontier of energy vs. latency tradeoff for various batch sizes on EfficientNet B7 on Jetson Orin. **Right** figure shows the Pareto Frontier of energy vs. latency tradeoff for various batch sizes on EfficientNet B4 on Jetson TX2. Each data point in this plot is representative of a unique hardware configuration, and each line corresponds to a batch size. The figure shows that the tradeoff does not always conform to the same pattern across varied hardware platforms and models.

3. We develop a performance model to capture the interaction between inference and fine-tuning processes. We use this model to schedule fine-tuning requests and carry out real-time modifications to meet inference SLOs.

4. We implement and evaluate PolyThrottle on a state-of-the-art inference server on Jetson TX2 and Orin. With minimal overheads, PolyThrottle reduces energy consumption per query by up to 36%.

2 Motivation

Many deep neural networks have been deployed on edge devices to perform tasks such as image classification, object detection, and dialogue systems. Scenarios including smart home assistants [22], inventory and supply chain monitoring [5], and autopilot [23] often use battery-based devices that contain GPUs to perform the aforementioned tasks. In these scenarios, pre-trained models are installed on the devices where the inference workload is deployed.

Prior works have focused on optimizing the energy consumption of GPUs [24, 25, 26, 27, 28] in cloud scenarios [29, 30, 31] and training settings [32, 33, 34]. On-device inference workloads exhibit different characteristics and warrant separate attention. In this section, we outline previous efforts in optimizing on-device neural network inference and discuss our approach to holistically optimize energy consumption.

2.1 On-device Neural Network Deployment

Prior work in optimizing on-device neural network inference focuses on quantization [11, 12, 13, 14, 15], designing hardware-friendly network architectures [35, 36, 37, 38, 39], and leveraging hardware components specific to mobile settings, such as DSPs [40]. Our work explores an orthogonal dimension and aims to answer a different question: **Given a neural network to deploy on a specific device, how can we tune the device to reduce energy consumption?**

In our work, we focus on edge devices that contain CPUs, memory, and GPUs. These devices are generally more powerful than DSPs often found on mobile devices. One such example is the Nvidia Jetson series, which is capable of handling a wide array of applications, ranging from AI to robotics and embedded IoT solutions [5]. The devices also come with dynamic voltage and frequency scaling (DVFS) capabilities that allow for the optimization of power consumption and thermal management during complex computational tasks. The Jetson series features a unified memory shared by both CPU and GPUs. We refer to the operating frequency of CPU, GPU, and shared memory as CPU frequency, GPU frequency, and memory frequency in this paper.

Case Study on Inventory Management: To understand the system requirements in edge NN inference, we next describe a case study of how NNs are deployed in an inventory management company. From our conversations, Company A works with Customer B to deploy neural networks on edge devices to optimize inventory management. To comply with regulations and protect privacy, data from each inventory site are required to be stored locally. The vast difference in the layout of the inventories makes it impossible to pre-train the model on data from every warehouse. Therefore, these devices come with a pre-trained model based on data from a small sample of inventories, which may have significantly different layouts and external environments compared to the actual deployment venue. Consequently, daily fine-tuning is required to enhance performance in the deployed sites, as the environment continually evolves. Similar arguments apply to smart home devices, where a model is pre-trained on selected properties, but the deployed households may be much more diverse. To address privacy concerns, on-device fine-tuning of neural networks is preferred, as it keeps sensitive data locally. Therefore, edge devices often need to run both inference and periodic fine-tuning. Combining multiple workloads on edge devices can lead to SLO violations due to interference and increased energy use.

2.2 Holistic Energy Consumption Optimization

Some recent works have explored reducing energy consumption by optimizing for batch size and GPU maximum frequency [16, 41, 42, 17] and developing power models for modern GPUs [43, 44, 45, 46]. In this work, we argue that other hardware components also cause energy inefficiency and require separate optimization. We perform a grid search over GPU, memory, and CPU frequencies and various batch sizes to examine the Pareto frontier of inference latency and energy consumption. Figure 2 shows the tradeoff between the per-query energy consumption and inference latency (normalized to the optimal latency) on Jetson TX2 and Jetson Orin. Each point in the figure represents the optimal configuration that we find through grid search under a given inference latency budget and batch size. As Figure 2 shows, the Pareto frontier is not smooth globally and is difficult to capture by a simple model, which warrants more sophisticated optimization techniques to quickly converge to a hardware configuration that lies on the Pareto Frontier [47].

Zeus [16] attempts to reduce the energy consumption of neural network training by changing the GPU power limit and tuning training batch size. PolyThrottle also includes these two factors. In Zeus [16], the focus is on training workloads in data center settings, where batch size tuning helps achieve an accuracy threshold in an energy-efficient way. We include batch size as part of PolyThrottle as it provides a trade-off between inference latency and throughput. Our empirical evaluation reveals new avenues available for optimization which complicates the search space, as we describe next.

3 Opportunities

In this section, we perform empirical experiments to uncover new opportunities for optimizing energy use in NN inference. As discussed in Section 2, prior work did not study how memory frequency, minimum GPU frequency, and CPU frequency play a role in energy consumption. This is partially limited by hardware constraints. Specialized power rails need to be built into the device during manufacturing to enable accurate measurement of energy consumption associated with each component. We leverage two Jetson developer kits, TX2 and Orin, which offer native support for component-wise energy consumption measurement and frequency tuning, to study how these frequencies impact inference latency and energy consumption in modern deep learning workloads. We find that the default frequencies are much larger than optimal and throttling all these frequency knobs offer energy consumption reduction with minimal impact on inference latency.

Figure 3 illustrates the energy optimization landscape when varying GPU and memory frequencies, without imposing any constraints on latency SLO. The plot reveals that, without any other constraints, the energy optimization landscape generally exhibits a bowl shape. However, this shape varies depending on the models, devices, and other hyperparameters, such as batch sizes (See Appendix B for more results). Next, we dive into how each hardware component affects inference energy consumption.

Setup: Experiments in this section are performed with 16-bit floating point number precision, as it has been demonstrated to have minimal impact on model accuracy in practice. We use Bert and EfficientNet models and vary the EfficientNet model size between B0, B4, B7 (Table 4).

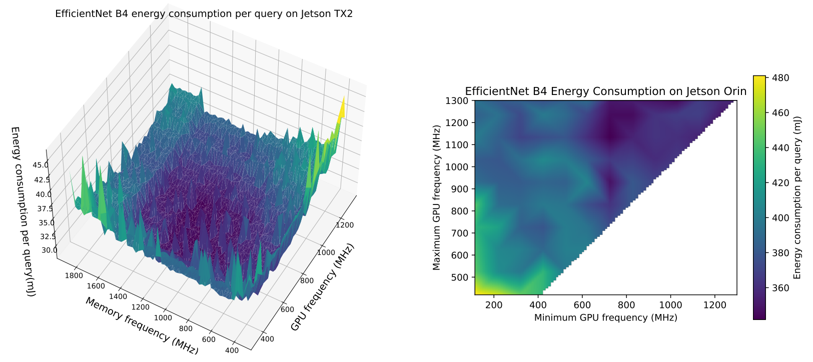


Figure 3: Left figure shows per query energy cost as we vary the GPU frequency and memory frequency for EfficientNet B4 on Jetson TX2 versus varying memory and GPU frequency with batch size fixed at 1. Right figure shows per query energy cost as we vary the minimum and maximum GPU frequency. As we increase the minimum GPU frequency, energy cost decreases.

Table 1: This table shows the energy frequency range for Jetson TX2 and Jetson Nano in MHz.

Device	Min GPU	Max GPU	Min Mem	Max Mem
TX2	114.75	1300.5	40.8	1866
Orin	114.75	1300.5	204	3199

Memory frequency experiment: For each model, we fix the GPU frequency at the optimal frequency determined by grid-search of all possible frequency configurations. We then examine the tradeoff between inference latency and energy consumption as we progressively throttle memory frequency. The range of available memory frequencies can be found in Table 1.

Results: Table 2 reveals that memory frequency plays a vital role in reducing energy consumption. The savings provided by memory frequency tuning are similar and consistent across models on both hardware platforms, ranging from approximately 12% to 25%. This indicates that the default memory frequency is higher than optimal for modern Deep Learning workloads. For heavy workloads such as Bert, memory tuning can account for the majority of the energy consumption reduction. This can be partially attributed to the memory-bound nature of Transformer-based models [48]. Our result demonstrates that systems that aim to optimize energy use in neural network inference need to take memory frequency into account.

CPU Frequency Experiment: CPUs are only used for data pre-processing. Thus, we first measure the time spent in the data processing part of the inference pipeline. Next, we measure the energy saved by throttling the CPU frequency and assess the inference latency slowdown caused by reducing CPU frequency. The data preprocessing we perform is standard in almost all image processing and object detection pipelines, where we read the raw image file, convert it to an RGB scale, resize it, and reorient it to the desired input resolution and data layout.

Results: The preprocessing time across different EfficientNet models remains constant since the operations performed are identical. As a result, the relative impact of CPU tuning on overall energy consumption depends on the ratio between preprocessing time and inference time. As the model size increases and inference duration increases, the influence of CPU tuning on overall energy consumption decreases. We observe that on both Jetson TX2 and Orin platforms, CPU tuning can decrease preprocessing energy consumption by approximately 30%. Depending on the model, quantization level, and batch size, this results in up to a 6% reduction in overall energy consumption.

Minimum GPU frequency experiment: We maintain the default hardware configuration and only adjust the minimum GPU frequency on Jetson Orin. Increasing the minimum GPU frequency forces the GPU DVFS mechanism to operate within a smaller range. We scale the model from EfficientNet B0 to EfficientNet B7 to illustrate the effect of the GPU minimum frequency on inference latency.

Results: Table 3 indicates that tuning the minimum GPU frequency can significantly reduce energy consumption when the workload cannot fully utilize the computational power of the hardware.

Table 2: This table shows the optimal memory frequency (in MHz) and the corresponding energy savings for various models on Jetson TX2. B0/B4/B7 represent different models in the Efficient Net series.

Model	Energy Reduction	Optimal Mem Freq
B0	14.9%	1331
B4	14.3%	1331
B7	12.0%	1331
Bert Base	25.4%	1062

Table 3: This table shows the optimal minimum GPU frequency (MHz) and the corresponding energy savings for various models with 16-bit floating-point precision on Jetson Orin. B0/B4/B7 represent different models in the Efficient Net series.

Model	Size	Energy Reduction	Optimal Min GPU Freq
B0	5.3M	47.6%	1236
B4	19M	29.1%	1033
B7	66M	8.8%	1134
Bert Base	110M	1.1%	217

Notably, energy consumption and inference latency are reduced by forcing the GPU to operate at a higher frequency. This differs from the tradeoff observed in other experiments, where we exchange inference latency for lower energy consumption. Tuning minimum GPU frequency can nearly halve the energy consumption for small models. As computational power becomes saturated with increasing model size, the return on tuning the minimum GPU frequency diminishes.

Figure 3 shows the per query energy cost as we vary the minimum and maximum GPU frequency. It shows that increasing the minimum GPU frequency from the default minimum leads to lower energy costs and inference latency.

4 Architecture Overview

To take advantage of the opportunities described in the previous section, we design PolyThrottle, a system that navigates the tradeoff between latency SLO, batch size, and energy. PolyThrottle optimizes for the most energy-efficient hardware configurations under performance constraints and handles scheduling of on-device fine-tuning.

Figure 1 shows a high-level overview of PolyThrottle's workflow. In a production environment, sensors on the edge devices continuously collect data and send the data to the deployed model for inference. In the meantime, to adapt to a changing environment and data patterns, these data are also saved for fine-tuning later. Due to the limited computation resources on these edge devices, fine-tuning workloads are often scheduled in conjunction with the continuously running inference requests. To address the challenges in model deployment on edge devices, PolyThrottle consists of two key components:

1. An optimization framework that finds optimal hardware configurations for a given model under predetermined SLOs using few samples.

2. A performance predictor and scheduler to dynamically schedule fine-tuning requests and adjust for the optimal hardware configuration while satisfying SLO.

PolyThrottle tackles these challenges separately. Offline, we automatically find the best CPU frequency, GPU frequency, memory frequency, and recommended batch size for inference requests that satisfy the latency constraints while minimizing per-query energy consumption. We discuss the details of the optimization procedure in Section 5. We also show that our formulation can find near-optimal energy configurations in a few minutes using just a handful of samples. Compared to the lifespan of long-running inference workloads, the overhead is negligible.

The optimal configuration is then installed on the inference server. At runtime, the client program processes the input and sends inference requests to the inference server. Meanwhile, if there are pending fine-tuning requests, the performance predictor predicts the inference latency when running concurrent fine-tuning, and decides whether it is possible to satisfy the latency SLO if fine-tuning is scheduled concurrently. A detailed discussion on performance prediction can be found in Section 6. The scheduler then decides what the new configuration that can satisfy the latency SLO while minimizing per-query energy consumption is. If such a configuration is attainable, it will schedule fine-tuning requests iteration-by-iteration until all pending requests are finished.

Online vs. Offline: Adjusting the frequency of each hardware component entails writing to one or multiple hardware configuration files, a process that takes approximately 17ms each. On Jetson TX2 and Orin, each CPU core, GPU, and memory has a separate configuration file that determines operating frequency. As a result, setting the operating frequencies for CPUs, GPU, and memory could require up to 150ms. This duration could exceed the latency SLO for many applications, and this is without accounting for the additional overhead of completing frequency changes. Since the latency SLO for a specific workload does not change frequently, PolyThrottle determines the optimal hardware configuration before deployment and only performs online adjustments to accommodate fine-tuning workloads.

5 Problem Formulation: Two-phase Tuning

Our objective is to automatically find the optimal hardware configurations that minimize energy consumption while satisfying latency SLOs. Formally, we are solving the optimization problem:

$$\begin{aligned} \min \quad & f(x_{CPU},\, x_{GPU_{min}},\, x_{GPU_{max}},\, x_{Mem},\, b) \\ \text{s.t.} \quad & t(x_{CPU},\, x_{GPU_{min}},\, x_{GPU_{max}},\, x_{Mem},\, b) \leq c \end{aligned}$$

where f and t represent the energy consumption and latency associated with a workload under the given hardware configurations and batch size. We use $x_{CPU},\ x_{GPU_{min}},\ x_{GPU_{max}}, x_{Mem}$ to denote the frequency limit of CPU, GPU, and memory on the device, and use b to denote the maximum batch size. We use c to denote the inference latency SLO limit, which is set by application users. This optimization problem is challenging on several fronts:

1. The search space is large, and performing a grid search will take hours depending on the model size. On TX2 and Orin, there are 5005 and 1820 points in the grid if we allow 5 different batch sizes. An exhaustive search would take 14 and 5 hours, respectively.

2. To satisfy latency constraints, it is hard to decouple each dimension and optimize them separately as they jointly affect inference latency in a non-trivial way.

3. The optimization landscape may vary across models and devices.

We observe that CPU frequency can be decoupled from GPU frequency, memory frequency, and batch size, as it mainly affects the preprocessing latency and energy consumption. We pipeline the requests so different requests can use CPU and GPU resources at the same time to increase inference throughput.

Based on this observation, we propose a two-phase hardware tuning framework, where CPU tuning is done separately from tuning other hardware components. The challenge that remains is to efficiently optimize for an unknown function with noise. As shown in Figure 3 and 3, the performance of neural network inference with changes in memory and GPU frequency is difficult to predict, therefore, a good solution must be able to handle the variance while converging to a near-optimal configuration in a sample-efficient fashion. This requires the method to adaptively balance the tradeoff between exploration and exploitation. To solve this, we formulate the optimization problem as a Bayesian Optimization problem and leverage recent advances in the field to incorporate the SLO constraints unique to our setting.

5.1 Constrained Bayesian Optimization

Bayesian Optimization is a prevalent method for hyperparameter tuning [49, 50], as it can optimize various black-box functions. This method is especially advantageous when evaluating the objective function is expensive and requires a substantial amount of time and resources.

However, some applications may involve constraints that must be satisfied in addition to optimizing the objective function. Constrained Bayesian Optimization (CBO) [51] is an extension of Bayesian Optimization that tackles this challenge by incorporating constraints into the optimization process.

In CBO, the objective function and constraints are treated as distinct functions. The optimization algorithm seeks to identify the set of input parameters that maximize the objective function while adhering to the constraints. These constraints are usually expressed as inequality constraints that must be satisfied during the optimization process. The expected constrained improvement acquisition function in CBO is defined as follows: $EI_C(\hat{x}) = PF(\hat{x}) \times EI(\hat{x})$.

Here $EI(\hat{x})$ represents the expected improvement (EI) [52] within an unconstrained Bayesian Optimization scenario, while $PF(\hat{x})$ is a univariate Gaussian cumulative distribution function, delineating the anticipated probability of whether $\hat{x}$ can fulfill the constraints. Intuitively, EI chooses the next configuration by optimizing the expected improvement relative to the best recently explored configuration. In PolyThrottle, we choose EI since our empirical findings and corroborations from additional studies [53] show that EI performs better than other widely-used acquisition functions [54].

CBO [51] also employs a joint prior distribution over the objective and constraint functions that captures their correlation structure. This joint prior is constructed by assuming that the objective and constraint functions are drawn from a multivariate Gaussian distribution with a parameterized mean vector and covariance matrix. These hyperparameters are learned from data using maximum likelihood estimation.

During the optimization process, the algorithm uses this joint prior to compute an acquisition function that balances exploration (sampling points with high uncertainty) and exploitation (sampling points where the objective function is expected to be low and subject to feasibility constraints). The algorithm then selects the next point to evaluate based on this acquisition function. During each iteration, the algorithm will test whether the selected configuration violates any of the given constraints and take the result into account for the next iteration. Encoding more system-specific hints as constraints can be of independent research interests, however, we show in Section 7 that the current formulation performs well under a variety of scenarios.

6 Modeling Workload Interference

Consider the case where we run an inference workload and aim to support fine-tuning without interfering with the online inference process. When a fine-tuning request arrives, we need to decide if it is possible to execute the fine-tuning request without violating inference SLOs. Time-sharing has been the default method for sharing GPU workloads. In time-sharing, shared workloads use different time slices and alternate GPU use between them. Recently, CUDA streams, Multiprocess Service (MPS) [55] and MIG [56] have been proposed to perform space-sharing on GPUs. However, these approaches are not supported on edge GPU devices [57, 58, 59, 60]. Given this setup, we propose building a performance model that can predict the inference latency in the presence of fine-tuning requests and only execute fine-tuning requests if the predicted latency can satisfy SLO.

Feature selection: To build the performance model, we leverage the following insights to select features: 1. In convolutional neural networks, the 2D convolution layers' performance largely determines the overall performance of the network. Its latency is correlated to the number of floating point operations (FLOPs) required during forward / backward propagation. 2. The ratio between the number of FLOPs and the number of memory accesses, also known as arithmetic intensity, together with total FLOPs, encapsulates whether a neural network is compute-bound or memory-bound. Using these insights, we add the following features to our model: Inference FLOPs, Inference Arithmetic Intensity, Fine-tuning FLOPs, Fine-tuning Arithmetic Intensity, and Batchsize.

Model selection: We propose using a linear model to predict inference latency when a fine-tuning workload is running concurrently on the same device. The model aims to capture how the proposed variables affect the resource contention between the inference workload and the fine-tuning workload, and therefore, affect the inference latency. The proposed model can be summarized as follows:

Table 4: This table shows the scaling pattern of the EfficientNet model family.

Model	Input dim / width	Width coef	Depth coef
B0	224×224	1.0	1.0
B4	380×380	1.4	1.8
B7	600×600	2.0	3.1

$$\begin{aligned}\text{Inference time} =&\theta_0 + \theta_1 \times \text{FLOPs}_{inf} + \theta_2 \times AI_{inf} \\ &+ \theta_3 \times \text{FLOPs}_{ft} + \theta_4 \times AI_{ft} \\ &+ \theta_5 \times \text{Batchsize}\end{aligned}$$

Given the above performance model, we use a Non-negative Least Squares (NNLS) solver to find the model that best fits the training data. An advantage of NNLS for linear models is that we can solve this with very few training data points [61]. We collect a few samples on the provided model by varying the inference and fine-tuning batch sizes and the output dimension, which captures various fine-tuning settings. This model is used as part of the workload scheduler during deployment to predict whether it is possible to schedule a fine-tuning request.

Fine-tuning scheduler: During inference, when there are outstanding fine-tuning requests, PolyThrottle uses the model to decide whether it is possible to schedule the request online without violating the SLO. When the model finds a feasible configuration, it adjusts accordingly until either all pending requests are finished or a new latency constraint is imposed.

7 Experiments

7.1 Setup

Hardware Platform: Our experiments are conducted on the Jetson TX2 Developer Kit and Jetson Orin Developer Kit. To assess the energy consumption of our program, we employ the built-in power monitors on the Jetson TX2 and Jetson Orin Developer Kits. We also cross-validate our measurements with an external digital multimeter (See Appendix A for more details on hardware and energy measurement).

Workload Selection: We base our experiments on the EfficientNet family and Bert models [10, 21]. EfficientNet is chosen not only for its status as a state-of-the-art convolutional network in on-device and mobile settings but also for its principled approach to scaling the width, depth, and resolution of convolution layers. Table 4 summarizes the scaling pattern of EfficientNet from the smallest B0 to the largest B7. We select Bert to investigate energy usage patterns in a Transformer-based model [62], where the workload is more memory-bounded compared to convolution-based neural networks. Bert and its variants [11, 21, 37, 63] are widely used for Question and Answering tasks [64], making it applicable for numerous edge devices, such as smart home assistants and smart speakers.

Dataset: We evaluate PolyThrottle on real-world traffic streams data [65] and sample frames uniformly to feed into EfficientNet. For Bert, we evaluate on SQuAD [64] for Question Answering. Note that datasets do not affect PolyThrottle's performance since inference latency would not change significantly across datasets once the model is chosen.

Implementation: PolyThrottle is built on the Nvidia Triton inference server. To maximize performance, we generate TensorRT kernels that profile various data layouts and tiling strategies to identify the fastest execution graph for a given hardware platform. Our modules include a Bayesian optimizer for determining the best configuration, an inference client responsible for preprocessing and submitting requests to the inference server, and a performance predictor module integrated into the inference client for scheduling fine-tuning requests. We maintain separate queues for inference and fine-tuning requests.

7.2 Efficiently Searching for Optimal Configuration

In this experiment, we carry out an extensive empirical analysis of tuning various models across different hardware configurations while also adjusting the quantization level. We perform a grid

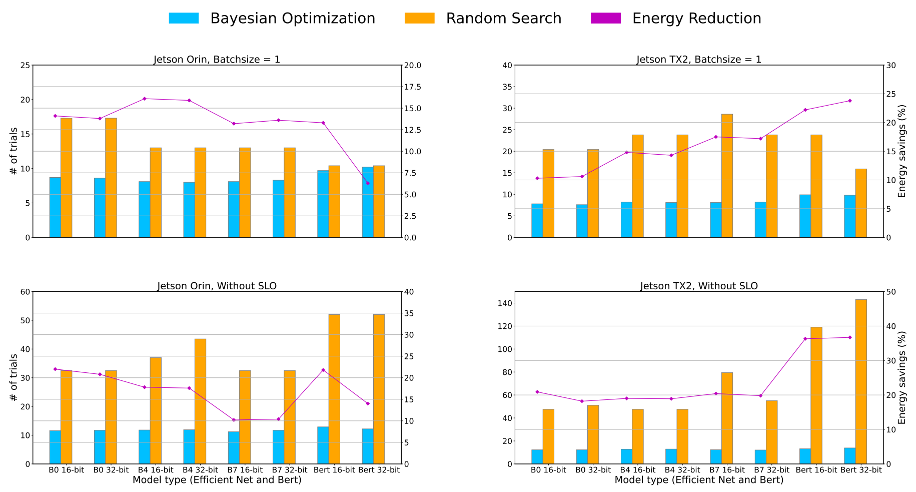


Figure 4: This figure compares search efficiency between Constrained Bayesian Optimization and Random Search. The y-axis represents the number of attempts it takes to find a near-optimal configuration and the x-axis represents the deployed and associated quantization level. The **first row** corresponds to the setting where we set a latency target but restrict the batch size to 1. The **second row** where we relax the latency constraint and allow batching inference requests.

search on EfficientNet B0, B4, B7, and Bert Base to examine the potential energy savings and identify the optimal GPU and memory frequencies for each model. We also adjust the quantization level for each tested model. We evaluate 16-bit and 32-bit floating point (FP16/FP32) precision. The optimal energy consumption and configuration referenced later in this section use the results obtained here as the baseline and optimal solution. Having obtained the optimal frequency using grid search, we next evaluate the average number of attempts it takes for PolyThrottle to find a solution within 5% of the optimal solution. We compare our Constrained Bayesian Optimization (CBO) formulation against Random Search (RS).

Experiment Settings: We measure the average number of attempts needed to find a near-optimal configuration. For Random Search, we calculate the expected number of trials needed to find a near-optimal configuration based on the grid size by computing the fraction of near-optimal configurations and taking the reciprocal. For CBO, we set the ξ parameter associated with the Expected Improvement function to 0.1 and initial random samples to be 5, which we find to work well across different models and hardware platforms. We conduct two experiments where we set different inference latency constraints, the results can be found in Figure 4:

1. We restrict inference latency to close to the optimal latency (20%). In this setting, the tight latency constraints make it impossible to batch the inference query, essentially reducing the search space for the optimal configuration.

2. In the second benchmark, we relax the inference latency constraint to include the configurations that provide the lowest energy-per-query in Figure 2. In this setting, we need to explore the batch size dimension to find the configuration that minimizes energy. We test on EfficientNet B0, B4, and B7, as well as Bert Base on both Jetson TX2 and Jetson Orin.

Results: Figure 4 shows that CBO outperforms RS in both scenarios. Since CBO models the relationship between hardware configuration and latency, it can find a near-optimal solution with only 5 to 15 samples. In the second scenario, the performance of RS deteriorates as it is unable to leverage the relationship between latency and batch size when dealing with a multiplicatively increasing search space. Overall CBO takes 3-10x fewer samples in the second setting. The overhead of performing CBO is also minimal. As shown in Figure 4, CBO only requires around 15 samples to find a near-optimal solution and the optimization procedure can be completed in a few minutes. In cases where a new model is deployed, only a few minutes of overhead are needed to find optimal configurations for the new model.

Table 5: This table shows the SLO violation rate and job completion time of various scheduling strategies for **Jetson Orin** on **EfficientNetB7**. The baseline shows the SLO violation rate without fine-tuning.

Method	Workload	SLO violation
Greedy	Uniform	37.91%
Adaptive	Uniform	**2.08%**
Greedy	Poisson	60.41%
Adaptive	Poisson	**5.42%**
Greedy	Twitter	22.0%
Adaptive	Twitter	**5.8%**
Baseline	Uniform	0.4%
Baseline	Poisson	1.67%
Baseline	Twitter	3.8%

Table 6: This table shows the SLO violation rate and job completion time of various scheduling strategies for **Jetson TX2** on **EfficientNetB4**. The baseline shows the SLO violation rate without fine-tuning.

Method	Workload	SLO violation
Greedy	Uniform	16%
Adaptive	Uniform	**5.5%**
Greedy	Poisson	35.4%
Adaptive	Poisson	**7.4%**
Greedy	Twitter	35.0%
Adaptive	Twitter	**7.5%**
Inference Only	Uniform	1%
Inference Only	Poisson	3.5%
Inference Only	Twitter	5.3%

It is important to note that though RS might achieve performance comparable to CBO under certain conditions, this result is merely the expected value and the variance of RS is large. For instance, if 10 out of 200 configurations are near-optimal, the expected number of trials needed to reach a near-optimal configuration is 20, with a standard deviation of 19.49. Consequently, it's plausible that even after 40 trials, RS might still fail to identify a near-optimal configuration. On the other hand, the standard deviation of CBO is smaller; in all experiments, CBO's standard deviations are less than 3.

7.3 Workload-aware Fine-tuning Scheduling

Next, we evaluate how well PolyThrottle handles fine-tuning requests alongside inference. The central question we aim to address is whether our performance predictor can effectively identify and adjust accordingly when the SLO requirement is at risk of being violated, and if reducing the inference batch size and trading throughput can satisfy the latency SLO. To simulate this scenario, we generate two distinct inference arrival patterns (Uniform and Poisson) and use the publicly available Twitter trace [66] and compare our adaptive scheduling approach to greedy scheduling, where a fine-tuning request is scheduled as soon as it arrives. The three arrival patterns represent scenarios that are highly controlled and bursty, respectively. In this context, we contrast PolyThrottle's adaptive scheduling mechanism with the greedy scheduling approach to assess the efficacy of PolyThrottle in meeting the desired SLO requirement.

Experiment Settings: We evaluate on both synthetic and real workloads. For synthetic workloads, we generate a stream of inference requests using both Uniform and Poisson distributions. For real-world workload, we first uniformly sample a day of Twitter streaming traces and then compute the variance of requests during each minute. We then picked the segment with the highest variance to test PolyThrottle's capability in handling request bursts [66, 67]. On Jetson Orin, we replay the stream for 30 seconds and measure the SLO violation rate during the replay using EfficientNet B7. Since each burst only lasts for a few seconds, this suffices to capture many bursts in the workload. We find that running the experiment for longer durations produces similar results. We set the fine-tuning batch size to 64, the number of fine-tuning iterations to 10, SLO to 0.7s, the output dimension to 1000, and an average of 8 inference requests per second. On Jetson TX2, we do the same experiment on EfficientNet B4. Due to memory constraints, we perform the fine-tuning batch size to 8, SLO to 1s, the output dimension to 100, and an average of 4 inference requests per second. We select a less performative model on TX2 to meet a reasonable SLO target (under 1s). The number of fine-tuning iterations is chosen based on the duration of the replay. We then measure the energy costs when deploying PolyThrottle at the default and optimal hardware frequency, respectively, to measure how much energy we save during this period. The optimal hardware frequency is obtained from results in Section 7.2.

For greedy scheduling, we employ a standard drop policy [68, 65], whereby a request is dropped if it has already exceeded its deadline. In the adaptive setting, we use the predictor to determine whether to drop an inference request. We also replay the inference request stream without fine-tuning requests to serve as a baseline.

Results: Table 5 and 6 show the SLO violation rates under various workloads and latency targets. The findings indicate that greedy scheduling may lead to significant SLO violations owing to the interference introduced by the fine-tuning workload. In contrast, PolyThrottle's adaptive scheduling mechanism demonstrates the ability to achieve low SLO violation rates by dynamically adjusting configurations. The baseline figures in the table represent SLO violation rates in the absence of interference from fine-tuning requests.

Inherent variance in neural network inference resulted in 1% of SLO violations in the case of Uniform distribution. However, bursts in the Poisson distribution and the Twitter workload generated more SLO violations. PolyThrottle's adaptive scheduling mechanism significantly reduces the SLO violation rate, meeting the SLO requirements while concurrently handling fine-tuning requests. Nevertheless, in several instances, we were unable to achieve near-zero SLO violation rates. This limitation can be attributed to the granularity of scheduling as we process the current batch of requests over an extended timespan due to interference from the fine-tuning workload.

We also **reduce energy consumption** by 14% on EfficientNet B7 on Jetson Orin and by 23% on EfficientNet B4 on Jetson TX2 across the workloads. We show in Appendix D how PolyThrottle reacts to changing SLOs when there are outstanding fine-tuning requests.

8 Conclusion

In this work, we examine the unique characteristics of energy consumption in neural network inference, especially for edge devices. We identified unique tradeoffs and dimensions between energy consumption and inference latency SLOs and empirically demonstrated hidden components in optimizing energy consumption. We then propose an optimization framework that automatically and holistically tunes various hardware components to find a configuration aligned with the Pareto Frontier. We empirically verify the effectiveness and efficiency of PolyThrottle. PolyThrottle also adapts to the need for fine-tuning and proposes a simple performance prediction model to adaptively schedule fine-tuning requests while keeping the online inference workload under the inference latency SLO whenever possible. We hope our study sheds more light on the hidden dimension of NN energy optimization.

References

[1] T. Anderson, A. Belay, M. Chowdhury, A. Cidon, and I. Zhang, "Treehouse: A case for carbon-aware datacenter software," in *HotCarbon*, 2022.

[2] U. Gupta, Y. G. Kim, S. Lee, J. Tse, H.-H. S. Lee, G.-Y. Wei, D. Brooks, and C.-J. Wu, "Chasing carbon: The elusive environmental footprint of computing," *IEEE Micro*, vol. 42, no. 4, pp. 37–47, 2022.

[3] Q. Cao, A. Balasubramanian, and N. Balasubramanian, "Towards accurate and reliable energy measurement of nlp models," in *Proceedings of SustaiNLP: Workshop on Simple and Efficient Natural Language Processing*, 2020.

[4] L. F. W. Anthony, B. Kanding, and R. Selvan, "Carbontracker: Tracking and predicting the carbon footprint of training deep learning models," in *ICML Workshop on Challenges in Deploying and monitoring Machine Learning Systems*, 2020.

[5] NVIDIA, "Jetson partner solutions ebook," 2023. https://resources.nvidia.com/en-us-jetson-success-stories/jetson-partner-solutions-ebook?lx=XRDs_y.

[6] C.-J. Wu, R. Raghavendra, U. Gupta, B. Acun, N. Ardalani, K. Maeng, G. Chang, F. Aga, J. Huang, C. Bai, M. Gschwind, A. Gupta, M. Ott, A. Melnikov, S. Candido, D. Brooks, G. Chauhan, B. Lee, H.-H. Lee, B. Akyildiz, M. Balandat, J. Spisak, R. Jain, M. Rabbat, and K. Hazelwood, "Sustainable ai: Environmental implications, challenges and opportunities," in *Proceedings of Machine Learning and Systems*, 2022.

[7] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green ai," *Commun. ACM*, vol. 63, no. 12, pp. 54–63, 2020.

[8] A. Lacoste, A. Luccioni, V. Schmidt, and T. Dandres, "Quantifying the carbon emissions of machine learning," *arXiv preprint arXiv:1910.09700*, 2019.

[9] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[10] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, pp. 6105–6114, PMLR, 2019.

[11] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-bert: Integer-only bert quantization," in *International conference on machine learning*, pp. 5506–5518, PMLR, 2021.

[12] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," *Advances in neural information processing systems*, vol. 31, 2018.

[13] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," *Advances in neural information processing systems*, vol. 28, 2015.

[14] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[15] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *arXiv preprint arXiv:2103.13630*, 2021.

[16] J. You, J.-W. Chung, and M. Chowdhury, "Zeus: Understanding and optimizing gpu energy consumption of dnn training," *arXiv preprint arXiv:2208.06102*, 2022.

[17] D. Gu, X. Xie, G. Huang, X. Jin, and X. Liu, "Energy-efficient gpu clusters scheduling for deep learning," *arXiv preprint arXiv:2304.06381*, 2023.

[18] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving {DNNs} like clockwork: Performance predictability from the bottom up," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 443–462, 2020.

[19] H. Cai, T. Wang, Z. Wu, K. Wang, J. Lin, and S. Han, "On-device image classification with proxyless neural architecture search and quantization-aware fine-tuning," in *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pp. 0–0, 2019.

[20] H. Cai, C. Gan, L. Zhu, and S. Han, "Tinytl: Reduce memory, not parameters for efficient on-device learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 11285–11297, 2020.

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[22] C. He, S. Li, J. So, X. Zeng, M. Zhang, H. Wang, X. Wang, P. Vepakomma, A. Singh, H. Qiu, *et al.*, "Fedml: A research library and benchmark for federated machine learning," *arXiv preprint arXiv:2007.13518*, 2020.

[23] I. Gog, S. Kalra, P. Schafhalter, J. E. Gonzalez, and I. Stoica, "D3: a dynamic deadline-driven approach for building autonomous vehicles," in *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 453–471, 2022.

[24] Y. Wang, Q. Wang, S. Shi, X. He, Z. Tang, K. Zhao, and X. Chu, "Benchmarking the performance and energy efficiency of ai accelerators for ai training," in *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020.

[25] F. Wang, W. Zhang, S. Lai, M. Hao, and Z. Wang, "Dynamic gpu energy optimization for machine learning training workloads," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[26] Z. Tang, Y. Wang, Q. Wang, and X. Chu, "The impact of gpu dvfs on the energy and performance of deep learning: An empirical study," in *Proceedings of the Tenth ACM International Conference on Future Energy Systems*, 2019.

[27] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in nlp," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[28] X. Mei, Q. Wang, and X. Chu, "A survey and measurement study of gpu dvfs on energy conservation," *Digital Communications and Networks*, vol. 3, no. 2, pp. 89–100, 2017.

[29] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Coadaptive cluster scheduling for goodput-optimized deep learning," in *OSDI*, 2021.

[30] C. Wan, M. Santriaji, E. Rogers, H. Hoffmann, M. Maire, and S. Lu, "Alert: Accurate learning for energy and timeliness," in *ATC*, 2020.

[31] M. Hodak, M. Gorkovenko, and A. Dholakia, "Towards power efficiency in deep learning on data center hardware," in *IEEE International Conference on Big Data*, 2019.

[32] G. Wang, S. Venkataraman, A. Phanishayee, N. Devanur, J. Thelin, and I. Stoica, "Blink: Fast and generic collectives for distributed ml," in *Proceedings of Machine Learning and Systems*, 2020.

[33] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *SOSP*, 2019.

[34] D.-K. Kang, K.-B. Lee, and Y.-C. Kim, "Cost efficient gpu cluster management for training and inference of deep learning," *Energies*, vol. 15, no. 2, p. 474, 2022.

[35] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A first look at deep learning apps on smartphones," in *The World Wide Web Conference*, WWW '19, p. 2125–2136, 2019.

[36] J. Lee, N. Chirkov, E. Ignasheva, Y. Pisarchyk, M. Shieh, F. Riccardi, R. Sarokin, A. Kulik, and M. Grundmann, "On-device neural net inference with mobile gpus," *arXiv preprint arXiv:1907.01989*, 2019.

[37] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[38] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers & distillation through attention," in *International Conference on Machine Learning*, pp. 10347–10357, PMLR, 2021.

[39] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 1314–1324, 2019.

[40] N. D. Lane and P. Georgiev, "Can deep learning revolutionize mobile sensing?," in *Proceedings of the 16th international workshop on mobile computing systems and applications*, pp. 117–122, 2015.

[41] S. M. Nabavinejad, S. Reda, and M. Ebrahimi, "Batchsizer: Power-performance tradeoff for dnn inference," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, 2021.

[42] T. Komoda, S. Hayashi, T. Nakada, S. Miwa, and H. Nakamura, "Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping," in *2013 IEEE 31st International Conference on computer design (ICCD)*, IEEE, 2013.

[43] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T. G. Rogers, T. M. Aamodt, and N. Hardavellas, "Accelwattch: A power modeling framework for modern gpus," in *MICRO*, 2021.

[44] S. Hong and H. Kim, “An integrated gpu power and performance model,” in *ISCA*, 2010.

[45] Y. Arafa, A. ElWazir, A. ElKanishy, Y. Aly, A. Elsayed, A. Badawy, G. Chennupati, S. Eidenbenz, and N. Santhi, “Verified instruction-level energy consumption measurement for nvidia gpus,” in *Proceedings of the 17th ACM International Conference on Computing Frontiers*, 2020.

[46] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, *et al.*, “The gem5 simulator: Version 20.0+,” *arXiv preprint arXiv:2007.03152*, 2020.

[47] Y. Censor, “Pareto optimality in multiobjective problems,” *Applied Mathematics and Optimization*, vol. 4, no. 1, pp. 41–59, 1977.

[48] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, “Data movement is all you need: A case study on optimizing transformers,” *Proceedings of Machine Learning and Systems*, vol. 3, pp. 711–732, 2021.

[49] K. Kandasamy, K. R. Vysyaraju, W. Neiswanger, B. Paria, C. R. Collins, J. Schneider, B. Poczos, and E. P. Xing, “Tuning hyperparameters without grad students: Scalable and robust bayesian optimisation with dragonfly,” *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 3098–3124, 2020.

[50] A. Klein, S. Bartels, S. Falkner, P. Hennig, and F. Hutter, “Towards efficient bayesian optimization for big data,” in *NIPS 2015 Bayesian Optimization Workshop*, 2015.

[51] J. R. Gardner, M. J. Kusner, Z. E. Xu, K. Q. Weinberger, and J. P. Cunningham, “Bayesian optimization with inequality constraints.,” in *ICML*, vol. 2014, pp. 937–945, 2014.

[52] E. Brochu, V. M. Cora, and N. De Freitas, “A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning,” *arXiv preprint arXiv:1012.2599*, 2010.

[53] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, “Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics.,” in *NSDI*, vol. 2, pp. 4–2, 2017.

[54] J. Snoek, H. Larochelle, and R. P. Adams, “Practical bayesian optimization of machine learning algorithms,” *Advances in neural information processing systems*, vol. 25, 2012.

[55] NVIDIA, “Stream management,” 2023. https://docs.nvidia.com/cuda/cuda-runtime-api.

[56] NVIDIA, “Nvidia multi-instance gpu,” 2023. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[57] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, “Pipeswitch: Fast pipelined context switching for deep learning applications,” in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pp. 499–514, 2020.

[58] Y. Zhao, X. Liu, S. Liu, X. Li, Y. Zhu, G. Huang, X. Liu, and X. Jin, “Muxflow: Efficient and safe gpu sharing in large-scale production deep learning clusters,” *arXiv preprint arXiv:2303.13803*, 2023.

[59] P. Yu and M. Chowdhury, “Fine-grained gpu sharing primitives for deep learning applications,” *Proceedings of Machine Learning and Systems*, vol. 2, pp. 98–111, 2020.

[60] X. Wu, J. Rao, W. Chen, H. Huang, C. Ding, and H. Huang, “Switchflow: preemptive multitasking for deep learning,” in *Proceedings of the 22nd International Middleware Conference*, pp. 146–158, 2021.

[61] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, “Ernest: Efficient performance prediction for {Large-Scale} advanced analytics,” in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pp. 363–378, 2016.

[62] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, “Transformers: State-of-the-art natural language processing,” in *EMNLP*, 2020.

[63] T. Tambe, C. Hooper, L. Pentecost, T. Jia, E.-Y. Yang, M. Donato, V. Sanh, P. Whatmough, A. M. Rush, D. Brooks, *et al.*, “Edgebert: Sentence-level energy optimizations for latency-aware multi-task nlp inference,” in *MICRO*, 2021.

[64] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," in *EMNLP*, 2016.

[65] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A gpu cluster engine for accelerating dnn-based video analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 322–337, 2019.

[66] Twitter, "Twitter streaming traces," 2018. https://archive.org/details/archiveteam-twitter-stream-2018-04.

[67] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "{INFaaS}: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 397–411, 2021.

[68] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system.," in *NSDI*, vol. 17, pp. 613–627, 2017.

A Hardware Details

A.1 Jetson platform details

The **Jetson TX2 Developer Kit** features a 256-core NVIDIA Pascal GPU, a Dual-Core NVIDIA Denver 2 64-bit CPU, a Quad-Core ARM Cortex-A57 MPCore CPU, and 8GB of 128-bit LPDDR4 memory with 59.7 GB/s bandwidth. The kit's maximum power consumption is 15W. The **Jetson Orin Developer Kit** includes a 2,048-core NVIDIA Ampere GPU with 64 Tensor Cores and a 12-core Arm CPU. This kit comes with 32GB of 256-bit LPDDR5 memory, featuring a 204.8GB/s bandwidth, and has a maximum power consumption of 60W.

A.2 Power consumption measurement

The Nvidia Jetson TX2 Developer Kit allows for separate measurements of GPU, CPU, DDR, and total energy consumption, while the Jetson Orin uses the built-in tegrastats module for measuring power usage across hardware components. Due to power rail design limitations, GPU power usage on the Jetson Orin can only be measured alongside SoC power usage.

On Jetson TX2, we measure power usage by querying the total power input. We then average the peak power consumption to obtain the power usage during inference. Then we compute the energy cost for each inference request by multiplying the power and the inference time. On Jetson Orin, we leverage the existing tegrastats tool and repeatedly query tegrastats at a fixed interval (50ms). We then sum up each component's power consumption to obtain the overall power consumption, before multiplying the power and the inference time to obtain the energy cost for each inference request. To obtain a steady reading, we send 1000 inference requests for each hardware configuration for every model that we test.

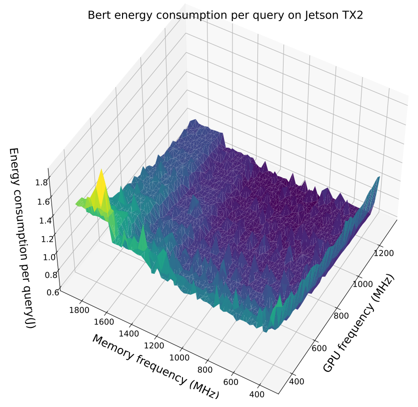
We cross-validate our measurements using a USB digital multimeter capable of transmitting data to computer software in real-time via Bluetooth. The measurements obtained from the multimeter generally align with those from the internal power rails on Jetson Kits, although external measurements are consistently around 10% higher than Jetson internal measurements. This discrepancy may be attributable to unaccounted factors in the power rail design. We opted to use internal measurements since they provide component-specific readings, whereas the multimeter can only measure overall energy consumption. Moreover, the multimeter supports one measurement per second, while Jetson tools allow for millisecond-scale measurements, which are better suited to inference workloads.

A.3 Measurement Overhead

Since we repeatedly query the power input or built-in power management tool, we want to understand whether these queries affect total energy consumption. We use a USB digital multimeter capable of transmitting data to computer software in real-time via Bluetooth. We then run our inference program with and without querying the power input or the power management tool. We find that the power consumption reported by the multimeter increases around $5 \to 10\%$, depending on the base power consumption. We observe that this increment is near constant across different models and runs and therefore we believe using internal measurement as described in the section above will not affect our findings. The multimeter cannot provide the precision and flexibility we need to measure the energy cost of inference, which often operates at a millisecond scale.

B Experimental Results

In this section, we further demonstrate the tradeoff between memory frequency and maximum GPU frequency by presenting an array of results. These results underline the interesting observation that the energy consumption patterns may vary for the same model operating on different devices. Furthermore, even for the same model-device pairing, the optimization landscape can be significantly influenced by the batch size. This underlines the complexities of energy optimization and the need for an adaptive framework that can take these factors into account. Figures $6 - 12$ show the energy consumption patterns of EfficientNet and Bert on Jetson TX2 and Orin under various batch sizes. Table 7 shows the optimal CPU frequency and corresponding energy consumption reduction in image preprocessing.



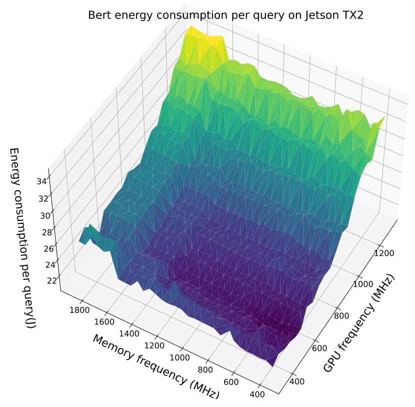
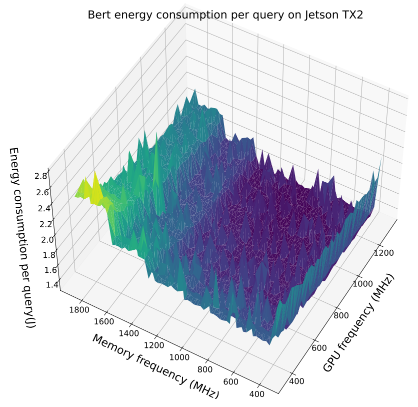


Figure 5: Left figure shows per query energy cost as we vary the GPU frequency and memory frequency for Bert at FP16 on Jetson TX2 versus varying Memory and GPU frequency with batch size fixed at 1. Right figure shows per query energy cost as we vary the GPU frequency and memory frequency for Bert at FP32 on Jetson TX2 versus varying Memory and GPU frequency with batch size fixed at 1.



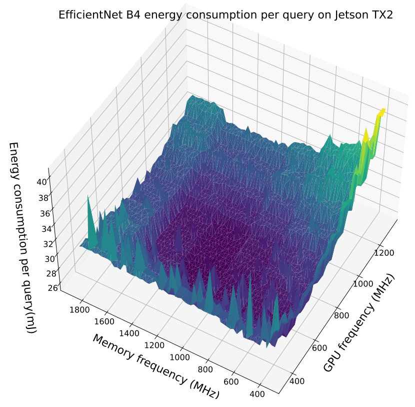
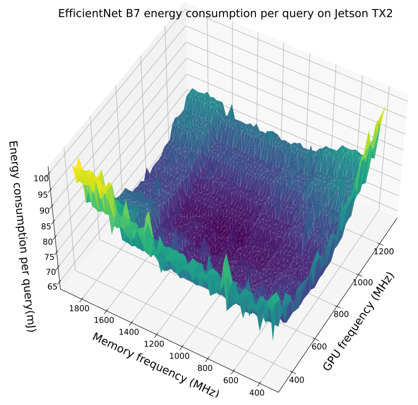


Figure 6: Left figure shows per query energy cost as we vary the GPU frequency and memory frequency for Bert at FP16 on Jetson TX2 versus varying Memory and GPU frequency with batch size fixed at 8. Right figure shows per query energy cost as we vary the GPU frequency and memory frequency for EfficientNet B4 at FP16 on Jetson TX2 versus varying Memory and GPU frequency with batch size fixed at 16.

Table 7: This table shows how CPU frequency (MHz) affects energy consumption for image processing on Jetson TX2 and Jetson Orin.

Device	Energy Reduction	Optimal Min CPU Freq
Orin	28.6%	900
TX2	29.4%	1000



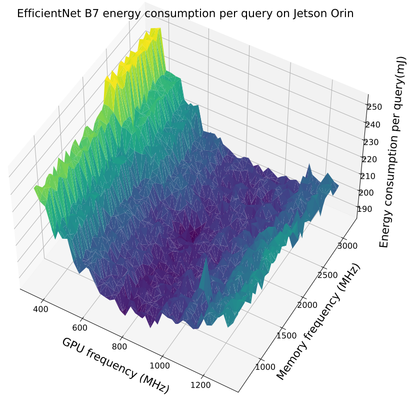
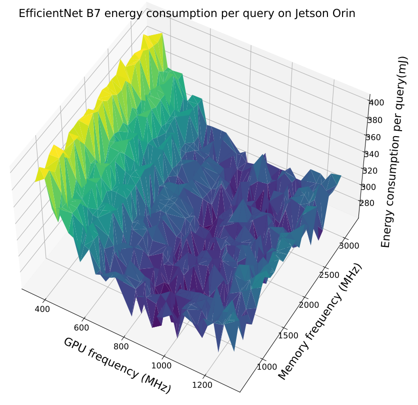


Figure 7: Left figure shows per query energy cost as we vary the GPU frequency and memory frequency for EfficientNet B7 at FP16 on Jetson TX2 versus varying Memory and GPU frequency with batch size fixed at 16. Right figure shows per query energy cost as we vary the GPU frequency and memory frequency for EfficientNet B7 at FP16 on Jetson Orin versus varying Memory and GPU frequency with batch size fixed at 8.



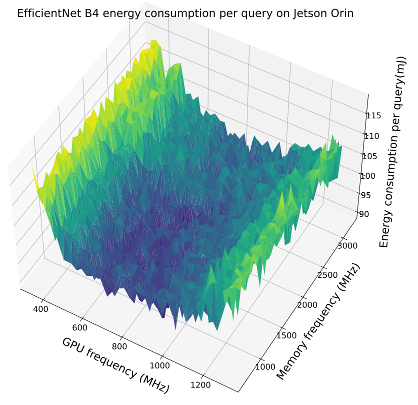


Figure 8: Left figure shows per query energy cost as we vary the GPU frequency and memory frequency for EfficientNet B7 at FP16 on Jetson Orin versus varying Memory and GPU frequency with batch size fixed at 1. Right figure shows per query energy cost as we vary the GPU frequency and memory frequency for EfficientNet B4 at FP16 on Jetson Orin versus varying Memory and GPU frequency with batch size fixed at 8.

C Arithmetic intensity

The arithmetic intensity of a 2D convolution layer can be computed as:

Table 8: The notation table defines the variable used in equation 1

Variable	Definition
N	Batch size
C	Input number of channels
H	Input tensor height
W	Input tensor width
K	Output number of channels
P	Output tensor height
Q	Output tensor width
R	Convolution filter height
S	Convolution filter width

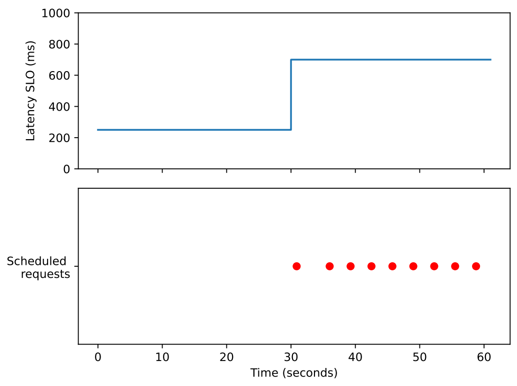


Figure 9: This figure shows the inference latency SLO and when fine-tuning requests are scheduled.

$$AI_{conv} = \frac{N \cdot K \cdot P \cdot Q \cdot C \cdot R \cdot S}{N \cdot C \cdot H \cdot W + K \cdot C \cdot R \cdot S + N \cdot K \cdot P \cdot Q} \tag{1}$$

The notations used in equation 1 can be found in table 8.

The FLOPs term captures the total computation of each workload, while the arithmetic intensity term captures how much computation power and memory bandwidth will affect the final performance. Combining the aforementioned features with an intercept term, which captures the fixed overhead in neural network inference, we can build a model that predicts inference latency if the hardware operating frequency is stable.

D Predictor Analysis

We vary the latency SLO to assess how the predictor schedules the fine-tuning requests. We replay a 60-second stream where we initially set the latency SLO to 250ms for the first half (30 seconds), and then increase it to 700ms for the remainder. As shown in Figure 9, under stringent latency conditions, the predictor deduces that it is impractical to schedule fine-tuning requests while adhering to the latency SLO, hence no fine-tuning requests are scheduled. Conversely, when the latency SLO is more relaxed, the predictor determines that it is feasible to schedule fine-tuning requests and sequentially schedules each request once the preceding one is completed and has issued a completion signal.