TRAINING ON EDGE DEVICES WITH UNIFIED-MEMORY

Arnav Singhvi [1] **James DeLoye** [1] **Anish Shanbhag** [1] **Shishir Patil** [1] **Joseph Gonzalez** [1]

ABSTRACT

Training large-scale generative models directly on consumer devices like laptops and smartphones enhances privacy and conserves scarce network bandwidth. However, the limited memory and compute available on these edge devices has restricted on-device training to smaller models. Encouragingly, an emergent system design of these edge devices is the growing presence of unified memory - where device accelerators including the CPUs, GPUs, NPUs, etc all share a physically unified memory. We explore how to effectively utilize this emerging trend to enable the training of memory-intensive models on edge devices. We introduce Private Optimal Runtime Training (PORT), an accuracy-preserving algorithm to train state-of-the-art neural networks on memory-constrained edge devices. PORT frames training as an Integer Linear Programming (ILP) problem, where each layer of the network can be optimally scheduled on single or multi-accelerator platforms. Additionally, PORT jointly optimizes over the combined search space of rematerialization and paging while exploiting system optimizations such as in-place convolutions to reduce peak-memory consumption and runtime of model training. These optimizations ensure PORT is transparent, accuracy preserving, and can effectively make use of memory-constrained environments to enable neural network training. PORT achieves up to 1.25x runtime speedups over current state-of-the-art techniques.

1 INTRODUCTION

Generative models have seen unprecedented advancements in recent years, growing in complexity and size. While these larger models offer significant advantages in performance and capabilities, they remain largely constrained to the powerful computing environments found in cloud servers with edge devices primarily serving as platforms for inference rather than training. This limitation reflects an evident disparity between cloud-centric training and utilizing edge devices for training tasks. Additionally, systems deployed in deep-sea (Jang & Adib, 2019), space and farmlands (Vasisht et al., 2017) environments among others pose limitations that can prevent data transmission to these centralized cloud servers, and ensuring secure data privacy (Li et al., 2020) is paramount. Hence, there lies a need to develop mechanisms to enable training on edge devices within localized settings.

Interestingly, the landscape of edge devices is shifting, in particular towards the adoption of unified-memory architectures. A unified memory architecture is one where different accelerators such as the CPU and the GPU share the same RAM, denoted as main memory. This is unlike classical distributed systems within the cloud where each accelerator has its RAM, such as in a system where an Nvidia GPU (e.g, H100) is paired with a CPU (e.g., Intel Xeon). In contrast, increasingly, consumer platforms such as Jetson TX2 and Orin (Khoshsirat et al., 2023) (Spicer & Baidya, 2023) (SK et al., 2022) which share the same RAM across the CPU and the GPU, and Apple's M1/M2/A16 System-on-Chips (SoCs) for laptops and iPhones (Liao et al., 2022) which share the same RAM between three accelerators—CPU, GPU, and Apple Neural Engine (ANE) (Banerjee, 2018) (Kasperek et al., 2022) —showcase this emerging paradigm.

Within this changing space, generalized beliefs about the hardware capabilities of edge devices can be misleading. For instance, it was traditionally assumed that GPUs on iPhones are designed for performance-intensive tasks while the Apple Neural Engine (ANE) is optimized for low-power computing. We dispel such assumptions by demonstrating that, in reality, certain network layers execute faster on the GPU, while others are more efficient on the ANE. In this work Private Optimal Runtime Training (PORT) leverages this nuanced understanding to answer the following question - *which accelerator should each node of the Neural Network be mapped to for optimal performance?* Further, given that peak-memory consumption for training might overshoot the memory available on edge devices, PORT optimizes over the joint subspace to rematerialization (deleting the tensor, and recomputing it), and paging (paging the tensor to secondary storage such as SSD/HDD/SD card/Flash) to reduce the peak memory consumption. To do this, PORT first obtains

[1]Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, California, USA.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2024.

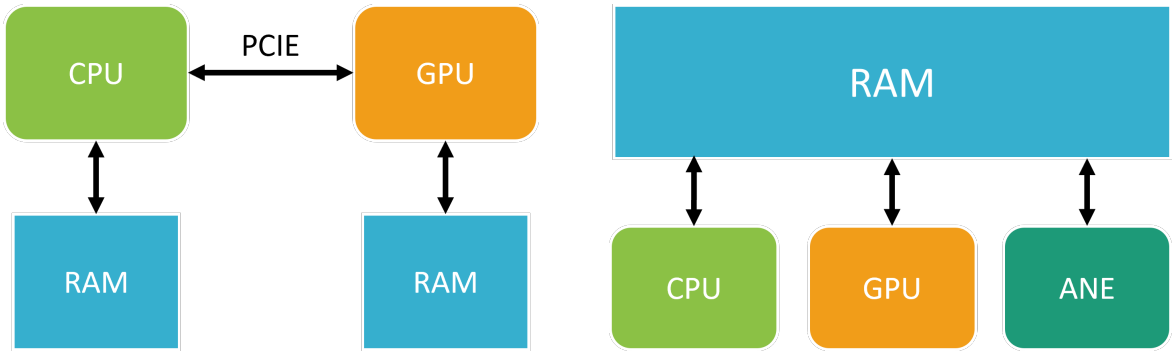

Figure 1. A comparison between memory architectures. The left shows a traditional multi-operator architecture while the right shows a unified memory architecture.

a fine-grained profile of each operator, and then formulates the node scheduling as an Integer Linear Program (ILP) to schedule each node of the training DAG (4.3).

With the hardware profiling of every operator on the target device, **PORT** generates a scheduler that is acutely aware of real-world hardware constraints and capabilities and optimizes training runtime and memory consumption within these environments.

Our methodology encompasses:

1. A novel Integer Linear Program formulation that enables **runtime- and memory-efficient** scheduling of deep learning models for unified memory architectures (Section 4.3).
2. A framework for fine-grained operator profiling to produce a **hybrid scheduling algorithm** that accounts for the unique characteristics of each accelerator (Section 4.2).
3. Real-world analysis of **PORT** on consumer edge devices, to understand the effect of different optimizations (Section 5).

To facilitate streamlined use of PORT we also introduce `Torch2PORT` (Section 4.4), a conversion algorithm to convert PyTorch model into graphical representations for processing by the PORT ILP. Once open-sourced, this will ensure general applicability for model training within PORT without manual conversion to maintain model compatibility.

2 Background

The digital landscape is rapidly shifting towards decentralization, highlighting the importance of edge devices in computational tasks (Sun et al., 2023) (Wang et al., 2022) (Sun et al., 2021) (Gheorghe et al., 2019) (Zhao et al., 2021). The ability to execute machine learning tasks closer to the data source is becoming crucial for several reasons. While prior works have showcased the potential of edge machine learning for inference (Boroumand et al., 2021b), (Boroumand et al., 2021a), there remains a critical gap in understanding how to train within memory and runtime confines and the importance of such training on edge devices.

Adapting to Data- and Concept-drifts (Tsymbal, 2004; Gama et al., 2014)**:** Real-world scenarios often demand rapid adaptability. Edge devices, which are naturally closer to the data source, can offer real-time updates to machine learning models and present quicker response times in these dynamic environments (Dou et al., 2023), (Khani et al., 2021). Such devices also operate in offline settings (Sudharsan et al., 2020) where internet connectivity is not often maintained, making it paramount that training on the edge can serve these devices to adapt and refine their operations.

Privacy: Training on edge devices ensures that user data remains localized, reducing potential data breaches or misuse (Zhao et al., 2018) (Das & Brunschwiler, 2019). By maintaining this decentralized approach, data transit is minimized, reducing exposure points where data could be potentially intercepted or compromised, promoting a more user-centric digital ecosystem with more trust in machine learning applications.

Low-Resource Utilization and Optimization: Edge devices are intrinsically constrained and often characterized by limited memory (Nasir et al., 2022). Many are complemented by off-chip secondary storage such as flash drives or SD cards, providing an avenue for on-device data retention and thereby reducing overheads associated with uploading to cloud storage.

However, such constrained environments require an optimized framework of resource allocation and data handling to ensure computational efficiency and feasibility. It's this very demand of constraint-driven innovation that we delve into, highlighting optimizations to meet the demands of

growing LLMs (Large Language Models) for the next wave of edge-based applications.

3 Related Work

Given the growing number of Edge deployments (Vasisht et al., 2017; Jang & Adib, 2019; Patil et al., 2019), there has been significant work in enabling ML on the edge devices.

3.1 Model Computation on Edge Devices

Paging is a memory management technique that allows data to be retrieved from secondary storage to main memory on an as-needed basis. DaCapo (Khan et al., 2023) extends paging techniques to address memory constraints in embedded systems by recomputing layer outputs during backpropagation to save memory space. Yet it lacks consideration on minimizing runtime overhead from such paging, which is reflected in our PORT ILP that efficiently distributes computations across multiple specialized compute units to find optimal solutions for edge devices (4.3).

While (Li et al., 2023) emphasizes hardware memory optimization, PORT goes a step further by addressing on-device training runtime constraints. (4.2) This addition is crucial as, beyond memory efficiency, achieving optimal processing speed is paramount for timely data processing and preventing latency issues that can drastically affect user experience or system output.

POET (Patil et al., 2022) optimized model training for edge devices by representing neural networks through a DAG (Directed Acyclic Graph) of tensor computations and managing memory via paging or rematerializing techniques. PORT goes beyond POET's foundations to ensure efficient training runtimes and memory consumption reduction by producing an ILP formulation that optimizes for these goals specifically (4.2). PORT does this by introducing network optimizations that optimize performance on single-accelerator configurations (4.1) while also providing node-scheduling compatibility across multi-accelerator platforms (4.3).

Rockmate (Zhao et al., 2023) provides memory utilization optimizations to POET's rematerialization techniques to minimize computational overhead. Similarly, MOCCASIN (Bartan et al., 2023) advances compute graph execution within memory limits using $O(n)$ integer variables for optimized rematerialization. PORT extends past these applications' single GPU/TPU accelerator setups by combining the power of multiple accelerators in parallel to offer competitive runtimes with ensured computational correctness and memory efficiency (4.3).

3.2 Network Optimizations

MIT MCUNet (Lin et al., 2020) introduces an algorithm-system framework that optimizes on-device training on memory-constrained microcontroller units (MCUs). MCUNet features TinyNas, a two-stage neural architecture search, optimizing for limited resource constraints. Their TinyEngine employs in-place depth-wise convolution to reduce peak memory overhead and employs model-adaptive memory scheduling to overcome layer-wise inefficiencies. While we take inspiration from the in-place convolution technique within our convolutional optimizations to reduce memory consumption (4.1.1), we do not adopt pruning or quantization methods as these do not mitigate the size of activations in finding feasible solutions. By introducing our refined computation layer optimizations, we eliminate the need for a search space overhead to demonstrate our significant reductions in memory usage.

(Lin et al., 2021) also presents a methodology that uses patch-based inference to address memory limitations in deep learning with CNNs on edge devices. Our frameworks also integrate patch-based inference within Conv2d layers, where we execute consecutive patch convolutions and assign them to the specific output path. However, we do not utilize different optimizers to avoid changing the model architecture while performing inference on edge devices, ensuring our optimizations are generalizable when finding feasible solutions.

3.3 Multi-Operator Algorithms

(Schuler et al., 2022) introduce XEngine, which utilizes a quadratic programming algorithm to efficiently schedule multiple network operators in heterogeneous systems with low memory. However, XEngine only considers rematerialization without paging when building schedules from its MIQP. PORT's formulation offers enhanced flexibility by enabling the same computation distribution across operators sharing a unified memory architecture while ensuring distributed rematerialization and paging (4.3). PORT-M presents a robust, scalable solution for emerging SOC designs with shared high-bandwidth memories.

3.4 Unified Memory

(Wu et al., 2023) highlights TGS (Transparent GPU Sharing) as an OS layer-level heuristic for deep learning training in container clouds that employs techniques like memory swapping between GPU and host memory across unified memory for maximal GPU utilization to ensure optimal performance. (Jung et al., 2023) also promotes GPU memory oversubscription, minimizing GPU fault handling time through prefetching techniques. PORT aligns with this approach to optimize hardware usage within real-world constraints and memory

environments from the memory layer level. PORT emphasizes optimal memory flow and accelerator and operator optimization during network training, providing balanced GPU usage within its scheduling heuristic while maintaining feasible training solutions within constrained environments.

(Bai et al., 2020) showcases PipeSwitch, an innovative system that enhances GPU utilization by enabling real-time GPU sharing on deep learning tasks through pipelined context switching, leveraging DNN layers for optimized task execution. (Kim et al., 2020) similarly applies context-switching techniques for reducing batches and handling page faults efficiently. PORT differs with its emphasis on streamlining individual task training through its memory optimization strategies tailored to the unified memory environments of edge devices, relying on dedicated hardware accelerator usage over context switching between multiple tasks in parallel.

(Choi et al., 2022) presents hierarchical unified virtual memory (HUVM) which utilizes the dormant memory of neighboring GPUs as an extension of virtual memory space. To manage this memory, (Choi et al., 2022) utilize memHarvester, a parallel memory manager that allows efficient delegation of open memory. On the other hand, PORT optimizes neural network training on edge devices by targeting reduced peak memory consumption, utilizing layer optimizations, scheduling techniques, and the unified memory architecture native to contemporary SOCs to facilitate effective training in reduced memory environments.

4 PORT

PORT employs layer optimizations within single-accelerator device configurations to reduce model training runtime and peak-memory consumption. Additionally, PORT approaches reducing training time as an ILP, scheduling each layer optimally while considering the combined search space of rematerialization and paging. PORT extends this for both single and multi-accelerator platforms, providing a comprehensive heuristic to make network training more efficient on edge devices.

4.1 Layer Optimizations

PORT aims to integrate network optimizations within layers to ensure faster training times. These optimizations not only enhance computational efficiency but also can reduce peak RAM consumption by avoiding excessive memory allocations through optimized memory storage patterns combined with efficient computation techniques.

4.1.1 In-Place Convolution

In-place convolution, introduced in TinyEngine (Lin et al., 2021), is an optimization algorithm that primarily targets the memory overhead associated with the forward pass for the convolution operation. The naive convolution forward pass creates an output that requires distinct memory allocation, leading to significant memory consumption, especially in deeper networks with multiple convolutional layers. Employing the in-place convolution optimization overcomes this by updating the output tensor directly in its original memory location, reducing the aforementioned memory consumption of new allocations for each input activation. Additionally, this provides significant runtime benefits by decreasing the time required for memory tensor allocation and deallocation. Improved cache utilization with the temporal locality of the same memory tensor reduces the need to fetch data within memory, speeding up computations. In-place optimizations can even lead to simplified computation graphs which are utilized by the PORT ILP to execute solutions (when feasible) even quicker. Moreover, hardware platforms often times are optimized for in-place operations.

4.1.2 Patch-Based Inference

Patch-based inference, a technique inspired by the MCUNetv2 model (Lin et al., 2021), reduces memory usage by processing data in manageable segments that fit device memory constraints. Instead of processing large input tensor batches, the input is divided into smaller, non-overlapping patches. Each patch is then processed and operated on separately, reducing peak memory consumption of the network. We integrate this optimization within the `Conv2d` layer of our model frameworks by executing consecutive patch convolution and respectively assigning to the specific output path. As each patch yields a smaller `Conv2d` output shape, operations become faster and require fewer computational resources. This ensures that the model training can run on devices with limited RAM without concerns of memory overflows or comprehensive extensive paging operations to account for such high memory requirements by larger models. Employing patch-based inference also maintains the mathematical integrity and feasibility of the training process, ensuring the correctness of gradient computations and model weight updates of model training are maintained on edge devices.

4.1.3 Im2Col

The `im2col` transformation for convolutional layers reshapes the input into a more suitable format for matrix multiplication (Jia, 2014). The transformation pads the input, iteratively slices the padded input into small segments determined by kernel size and stride, and then stretches them into columns. This ensures the matrix has been reshaped for matrix multiplication and multiplying the original matrix with the reshaped weight matrix effectively converts convolution operations into a series of efficient matrix multiplications. Employing the `im2col` optimization within

our model framework's convolutional layers allows us to harness the native advantages of hardware platforms which are optimized for these efficient matrix multiplications. This improves runtime and aligns with the overall goal of making on-device training feasible and efficient.

4.1.4 Loop Reordering

Loop reordering cache optimizations (Sarkar & Thekkath, 1992) (Kalamationos & Kaeli, 1998) transform the forward pass of convolutional layers from the traditional height-width-channel (HWC) format to the channel-height-width (CHW) format, improving memory locality by reducing the number of cache misses and speeding up computations. By padding the input systematically and iterating over the kernel dimensions and strides to align with the convolutional kernel, this ensures correctness for subsequent operations. Furthermore, by employing `einsum` techniques (Imambi et al., 2021), (Klaus et al., 2023) with the reshaped input tensor suited for efficient matrix multiplication operations, this method internally optimizes for memory access patterns to provide computational speedup.

4.1.5 Operator Fusion

Operator fusion (Lin et al., 2021) (Niu et al., 2021), (Chen et al., 2018) is another technique popular in ML frameworks that provides efficient memory utilization and reduced dispatch latency for GPUs. Applying this technique within our CNN frameworks finds a natural application in combining independent operations of consecutive layers, leading to fewer intermediate results, quicker computation, and reduced memory footprint by discarding unneeded tensor outputs. We use operator fusion for training by creating a forward with `Conv2d` + `BatchNorm` and `Conv2d` + `ReLU` combinations, ensuring correctness as we trace the dependency of all tensors (weights, gradients, activation) and reorder operators to fuse them.

4.2 PORT: Private Optimal Runtime Training

POET, the state-of-the-art framework for edge device training, was designed to minimize the power consumption of training models on edge devices within provided runtime and memory constraints by modeling neural networks as a DAG (Directed Acyclic Graph) of their tensor computations. POET then achieved significant memory savings by either paging tensors in and out of secondary storage or rematerializing tensors when needed. While provably optimal for this goal, when working with larger models or platforms that offer unified memory, minimal training and inference time is a more critical goal over power savings. To this end, we introduce PORT's ILP to ensure reduced runtime and peak memory consumption.

Below is POET's objective function and runtime constraint. It aims to minimize the combined power of tensor computations and paging while setting an upper bound on the processor's computational duration for the model.

$$\text{minimize} \quad \sum_{t=1}^{T} \left\{ \boldsymbol{R}_{t,:} \cdot \Phi^{\text{compute}} + \boldsymbol{M}^{\text{in}}_{t,:} \cdot \Phi^{\text{pagein}} + \boldsymbol{M}^{\text{out}}_{t,:} \cdot \Phi^{\text{pageout}} \right\} \tag{1}$$

$$\text{subject to:} \quad \sum_{t=1}^{T} \left\{ \boldsymbol{R}_{t,:} \cdot \Psi^{\text{compute}} \right\} \leq \mu^{\text{deadline}} \tag{2}$$

PORT does away with the runtime constraint and converts the primary objective to the function in Equation 1. This sums the time taken to compute every tensor in each timestep to output the schedule's total CPU computation time. This is represented with the binary computation matrix $\boldsymbol{R}$ where the entry $\boldsymbol{R}_{t,i}$ encodes if tensor i is computed in timestep t and the compute cost vector where $\Psi_i^{compute}$ reflects the cost of computing tensor i on the device.

$$\alpha \text{ minimize} \quad \sum_{t=1}^{T} \left\{ \boldsymbol{R}_{t,:} \cdot \Psi^{compute} \right\} \tag{3}$$

We additionally refine our formulation to ensure optimal paging behavior with a secondary objective to ensure that tensors are paged only when necessary (2). Much like the $\boldsymbol{R}$ matrix, the matrices $\boldsymbol{M}^{in}$ and $\boldsymbol{M}^{out}$ are binary matrices with entries representing if tensor i is paged in or out of secondary storage, respectively, in timestep t.

$$\beta \text{ minimize} \quad \sum_{t=1}^{T} \sum_{i=1}^{T} \left\{ \boldsymbol{M}^{in}_{t,i} + \boldsymbol{M}^{out}_{t,i} \right\} \tag{4}$$

We label the primary objective the α objective and the secondary objective the β objective, only minimizing the former among optimal solutions to the primary objective. As a result, we can be assured that the paging schedule output by PORT will minimize extraneous disk usage while maintaining proper paging to ensure minimized runtime.PORT hence reduces POET's dependence on a secondary dynamic programming algorithm that accounts for the time taken to page tensors in and out within its ILP to avoid external complexity.

Additionally, PORT ensures bandwidth awareness and device consumption within the ILP. Here ξ^{size} is the vector of tensor sizes while $\lambda^{bandwidth}$ is a constant giving the device bandwidth. By accounting for bandwidth as a constraint Equation 3, the bandwidth of the bus between the RAM and secondary storage cannot be exceeded by the scheduler.

$$\boldsymbol{M}^{in}_{t,:} \cdot \xi^{size} + \boldsymbol{M}^{out}_{t,:} \cdot \xi^{size} \leq \left(\boldsymbol{R}_{t,:} \cdot \Psi^{compute} \right) \lambda^{bandwidth} \quad \forall t \tag{5}$$

4.3 PORT-M: Private Optimal Runtime Training for Multiple Operators

With the proliferation of unified SOCs which have multiple specialized compute units (CPU, GPU, TPU) all using the same high bandwidth memory, there is potential for even faster training of large models on the same device using parallelism. In addition, the highly specialized nature of the hardware means that computations can be distributed to the hardware with the best-suited architecture. To support this effort we introduce PORT-M, a version of PORT designed to distribute computation across any number of operators that share a unified memory architecture.

4.3.1 Designing the Variables

To start, the binary computation matrix $\boldsymbol{R}$ is extended from a $T \times T$ matrix to a $T \times T \times N$ tensor $\mathbf{R}$ where T is the number of timesteps and tensors in the network DAG and N is the number of independent operators working on the same unified memory. If tensor i is computed at timestep t on operator n then $\mathbf{R}_{t,i,n} = 1$. With the extension of the runtime matrix, we (re)introduce the remaining variable matrices. As mentioned before, $\boldsymbol{M}^{in}$ and $\boldsymbol{M}^{out}$ are $T \times T$ binary matrices encoding if a tensor is paged in or out of secondary storage. Likewise, S^{RAM} and S^{AUX} are $T \times T$ binary matrices encoding if a tensor is resident in memory or secondary storage, respectively. We also introduce $\boldsymbol{A}$ which is a one-dimensional matrix with the time that each timestep takes. $\boldsymbol{U}$ is the matrix of memory consumption during the computation (or noncomputation) of tensor i in timestep t. $\boldsymbol{U}$ is defined using the $FREE$ and num_hazards matrices where $FREE_{t,k,i}$ specifies if tensor k can be freed in timestep t once tensor i has been computed and likewise num_hazards calculates how many dependencies are remaining at the same point in the computation. We also continue using the runtime and size cost vectors, $\Psi^{compute}$ and ξ^{size}, and the bandwidth and available memory constants of $\lambda^{bandwidth}$ and μ^{RAM}.

4.3.2 Designing the Constraints

With additional operators, we modify the original constraints of PORT accordingly. The constraint $\mathbf{R}_{t,i,n} + \boldsymbol{S}^{RAM}_{t,i} \geq \mathbf{R}_{t,j,n}$ ensures that any tensor j dependent on another tensor i to be computed beforehand either in the same timestep on the same device (rematerialization), or available in RAM, where it could have been computed in a previous timestep by another operator. $\bigvee_n^N \{\mathbf{R}_{t-1,i,n}\} + \boldsymbol{S}^{RAM}_{t-1,i} + \boldsymbol{M}^{in}_{t-1,i} \geq \boldsymbol{S}^{RAM}_{t,i}$ maintains that for a tensor to be in RAM at time t it must be in ram in the previous timestep, paged in, or computed by any operator. $\boldsymbol{S}^{AUX}_{t-1,i} + \boldsymbol{M}^{out}_{t-1,i} \geq \boldsymbol{S}^{AUX}_{t,i}$ similarly requires tensors present in auxiliary memory to be there in the previous timestep or be paged out from RAM. $\boldsymbol{S}^{AUX}_{t,i} \geq \boldsymbol{M}^{in}_{t,i}$ and $\boldsymbol{S}^{RAM}_{t,i} \geq \boldsymbol{M}^{out}_{t,i}$ both ensure that a tensor cannot be paged in if it is not present in secondary storage and cannot be paged out if it is not present in memory, respectively. $\max_n \{\mathbf{R}_{t,:,n} \cdot \Psi^{compute}_n\} = \boldsymbol{A}_t$ defines the helper matrix $\boldsymbol{A}$ with $\boldsymbol{A}_t$ as the maximum runtime step t takes among all N operators. The presence of tensors in RAM and auxiliary storage are initialized to be empty to ensure that tensors are computed, to begin with. $\bigvee_n^N \{\mathbf{R}_{v,v,n}\} = 1$ requires that tensor v is always computed at time v, ensuring that the neural network computation occurs as intended. Finally, as we are working on memory-constrained devices, $\boldsymbol{U}^{RAM}_{t,i} \leq \mu^{RAM}$ ensures that the formulation is within the memory bounds.

4.3.3 Designing the Objectives

Our primary, or α objective in PORT-M simply minimizes the sum of time each timestep takes across all devices. To reflect this, we introduce another secondary objective (β_1) in which we seek to minimize the total number of times tensors are computed since much like with paging in and out, tensors may be frivolously computed on other devices if their combined runtime is under that of the maximum for that timestep. We also seek to minimize unnecessary paging (β_2 objective) just as in single-operator PORT.

4.4 Torch2PORT: Integration with PyTorch

One of the common limitations of existing ILP training optimizers is their usability and integration with the open-source model ecosystem. For example, POET requires the computation of a training graph execution plan for each network layer and parameters, limiting scalability. Additionally, introducing network optimizations requires manual tuning of integrating novel layers within the ILP formulation, constraining optimized testing.

To alleviate this, we introduce Torch2Port, a new tool for automatically converting any PyTorch model to the ILP formulation usable by PORT. Given any PyTorch model, Torch2Port traces the full PyTorch computational graph produced by `torch.autograd`, automatically identifies the characteristics and output tensor sizes for each layer, detects and applies existing PyTorch optimizations such as inplace operations, and produces an ILP formulation for the model compatible with PORT.

5 EVALUATION

In our evaluation of PORT, we aim to answer the following questions:

1. Can PORT improve runtime for training across devices and models?
2. Can PORT lower the peak-memory consumption for training?

Algorithm 1 PORT-M: The ILP formulation to find an optimal schedule of rematerialization and paging for the computation of a neural network on multiple operators sharing a unified memory architecture

$$\text{minimize:} \quad \sum_{t=1}^{T} \boldsymbol{A}_t \quad + \quad \sum_{t=1}^{T}\sum_{i=1}^{T}\sum_{n=1}^{N} \mathbf{R}_{t,i,n} \quad + \quad \sum_{t=1}^{T}\sum_{i=1}^{T} \left\{ \boldsymbol{M}^{in}_{t,i} + \boldsymbol{M}^{out}_{t,i} \right\}$$

$$\text{subject to:}$$

$$\mathbf{R}_{t,i,n} + \boldsymbol{S}^{RAM}_{t,i} \geq \mathbf{R}_{t,j,n} \qquad \forall t \in V, \forall (v_i, v_j) \in E, \forall n \in N$$

$$\bigvee_{n}^{N} \{\mathbf{R}_{t-1,i,n}\} + \boldsymbol{S}^{RAM}_{t-1,i} + \boldsymbol{M}^{in}_{t-1,i} \geq \boldsymbol{S}^{RAM}_{t,i} \qquad \forall t, i \geq 2$$

$$\boldsymbol{S}^{AUX}_{t-1,i} + \boldsymbol{M}^{out}_{t-1,i} \geq \boldsymbol{S}^{AUX}_{t,i} \qquad \forall t, i \geq 2$$

$$\boldsymbol{S}^{AUX}_{t,i} \geq \boldsymbol{M}^{in}_{t,i} \qquad \forall t, i \geq 2$$

$$\boldsymbol{S}^{RAM}_{t,i} \geq \boldsymbol{M}^{out}_{t,i} \qquad \forall t, i \geq 2$$

$$\max_{n} \left\{ \mathbf{R}_{t,:,n} \cdot \Psi^{compute}_{n} \right\} = \boldsymbol{A}_t \qquad \forall t \in V$$

$$\boldsymbol{S}^{RAM}_{1,i} = 0 \qquad \forall i \in V$$

$$\boldsymbol{S}^{SSD}_{1,i} = 0 \qquad \forall i \in V$$

$$\bigvee_{n}^{N} \{\mathbf{R}_{v,v,n}\} = 1 \qquad \forall v \in V$$

$$\boldsymbol{M}^{in}_{t,:} \cdot \xi^{size} + \boldsymbol{M}^{out}_{t,:} \cdot \xi^{size} \leq \boldsymbol{A}_t \cdot \lambda^{bandwidth} \qquad \forall t \in V$$

$$\boldsymbol{U}^{RAM}_{t,i} \leq \mu^{RAM} \qquad \forall t, i \in V$$

$$\xi^{input} + 2\xi^{param} + \boldsymbol{S}^{RAM}_{t,:} \cdot \xi^{size} = \boldsymbol{U}_{t,0} \qquad \forall t \in V$$

$$\boldsymbol{U}_{t,i} + \sum_{k \in \text{DEPS}(i)} \xi^{size}_{i} FREE_{t,k,i} + \bigvee_{n}^{N} \{\mathbf{R}_{i+1,i,n}\}\, \xi^{size}_{i+1} = \boldsymbol{U}_{t,i+1} \qquad \forall t, i \in V$$

$$\left\{ \begin{array}{ll} 1 & \text{if num\_hazards}(t,k,i) = 0 \\ 0 & \text{else} \end{array} \right\} = FREE_{t,k,i} \qquad \forall t, k, i \in V$$

$$\sum_{j \in \text{USERS}(k) | j > i} \bigvee_{n}^{N} \{\mathbf{R}_{t,j,n}\} + (1 - \bigvee_{n}^{N} \{\mathbf{R}_{t,i,n}\}) + \boldsymbol{S}^{RAM}_{t+1,k} = \text{num\_hazards}(t,k,i) \qquad \forall t, k, i \in V$$

$$\boldsymbol{A} \in \mathbb{R}^{V}$$

$$\mathbf{R} \in \{0,1\}^{V \times V \times N}$$

$$\boldsymbol{S}^{AUX}, \boldsymbol{S}^{RAM}, \boldsymbol{M}^{in}, \boldsymbol{M}^{out} \in \{0,1\}^{V \times V}$$

$$\boldsymbol{U} \in \mathbb{R}^{V \times V}$$

$$FREE \in \{0,1\}^{V \times V \times V}$$

$$\text{num\_hazards} \in \mathbb{N}^{V \times V \times V}$$

5.1 Experimental Setup

To evaluate these questions, we test PORT on the following hardware devices: 1) embedded edge devices: ARM Cortex M0 class MKR1000, ARM Cortex M4F class nrf52840, and the A72 class Raspberry Pi 4B+, 2) consumer-available edge devices: Jetson Orin, iPhone 13 Pro Max, and iPhone 15 Pro Max.

For Nvidia Jetson Orin, we profiled the two accelerators CPU and GPU. For the iPhone 13 Pro Max and iPhone 15 Pro Max, we profiled the three accelerators.

Benchmarking on Apple devices is dependent on converting the Pytorch layers to an iPhone-compatible CoreML model format and measuring layer execution time with Apple Instruments profiling software. As Apple's devices have heavily shifted towards a system-on-chip model with unified memory architecture in recent years starting with the M1 MacBook and latest iPhone devices (Liao et al., 2022), we take advantage of this unique positioning to utilize the multi-operator execution scheduling generated by PORT-M. For instance, profiling on the iPhone 13 Pro Max which employs the Apple A15 Bionic system-on-chip and integrates a 6-core CPU, 5-core GPU, and 16-core Apple Neural Engine

(ANE) to interface with one set of 6GB LPDRR4X unified memory presents a readily-viable memory-constrained environment with multiple acceletors.

Our benchmark models include VGG16 and ResNet-18, and the BERT transformer model. VGG16 and ResNet-18 were trained with CIFAR-10 image configuration.

To evaluate PORT's ability to lower runtimes and memory consumption for training, our primary objective is a comparative analysis against the existing SotA POET baseline, with this two-pronged focus in mind:

1. Comparing PORT's efficiency across multi-platform configurations to the POET baseline. The PORT-M solver relies on the profiling data of the baseline networks. This solver was run with the following configuration combinations: Jetson Orin GPU/CPU and Apple CPU/GPU, GPU/ANE, CPU/ANE and CPU/GPU/ANE (respectively for the individual platforms).
2. Evaluating PORT's network optimizations in reducing runtime and RAM consumption without affecting solution feasibility.

To evaluate the former, we accurately measured execution times for network layers to collect data on the accelerators of the consumer-available edge devices. To do so, we utilized the PyTorch conversion within Torch2Port to isolate each layer in the model and benchmark execution time. Each measurement was repeated 100 times for accuracy, and then the median execution time for each layer was accounted for within the solver constraints.

To evaluate the network optimizations, we adopt a flop-based model. First, we gather FLOP, FLOP per Watt, Memory read-write throughput, etc. We then use these metrics to assess the runtime and RAM consumption benefits presented by the PORT network optimizations.

A unique feature of PORT is its approach to scheduling training graph nodes without altering training parameters. This ensures resilience against hyper-parameter changes while respecting the memory constraints of devices. Hence, we don't report any accuracy numbers. However, we note that since ILP solutions commonly yield lengthy solve times, we ensure that both PORT and the baseline POET's solutions adhere to universal 10-minute solve-time constraint on all platforms.

Furthermore, since PORT intends to minimize runtime and does not rely on setting a runtime constraint within the ILP, we impose this condition during our POET experiments. Since POET can sacrifice runtime for energy savings, this constraint acts as a tight bound for POET, yielding a constant network runtime for the POET ILP solution on all networks.

5.2 Multi-Accelerator Configurations Results

5.2.1 Jetson Orin CPU and GPU

On the Jetson Orin platform, when evaluating the ResNet18 model, the standalone PORT-M configuration interleaving the Jetson Orin CPU and GPU achieved a speedup of approximately 1.090x, compared to the POET baseline.

5.2.2 Apple CPUs, GPUs, and Apple Neural Engine

iPhone 13 Pro Max For the ResNet18 model A.2 on the iPhone CPU, the POET configuration registered at 41.02 ms while PORT using the ANE alongside the CPU presented runtime benefit with a speedup of 1.122x, registering at 36.56 ms. Turning to the iPhone GPU, POET was recorded at 119.18 ms while PORT taking advantage of all three accelerators CPU, GPU, and ANE yielded a runtime of 36.63 ms, translating to a 3.257x speedup on the GPU runs. On the iPhone ANE, POET registered a runtime of 74.27 ms while the aforementioned PORT configuration using CPU alongside the ANE at 36.56 ms produced a speedup of 2.030x over the POET ANE configuration. Overall, when juxtaposing the best from POET, which was the iPhone CPU at 41.02 ms, against the top performer from all PORT configurations, the CPU-ANE combination at 36.56 ms by PORT-M, we observe a speedup of 1.122x.

For the BERT model A.2 on the iPhone CPU, the POET configuration had a runtime of 54.79 ms, while the leading PORT configuration which utilized the ANE alongside the CPU demonstrated benefit with a 1.071x speedup with its 51.16 ms runtime. For the iPhone GPU, the POET configuration was at 117.16 ms, while PORT using the ANE alongside the GPU showcased a 1.260x speedup at 92.97 ms. On the iPhone ANE, POET registered a runtime of 113.63 ms while the mentioned interleaving of CPU alongside ANE allowed PORT-M to provide a notable 2.221x speedup with a runtime of 51.16 ms. Broadly, comparing the best of POET which was on the iPhone CPU at 54.79 ms, with the overall best from PORT, the CPU-ANE's 51.16 ms resulted in a speedup of 1.071x.

iPhone 15 Pro Max For the VGG16 model A.2 on the iPhone CPU, the POET configuration clocked in at 151.56 ms. In contrast, PORT using the ANE alongside the CPU presented a runtime of 101.86 ms, achieving a speedup of 1.487x. Considering the iPhone GPU, POET had a runtime of 238.51 ms, while PORT utilizing the CPU, GPU, and ANE - the PORT-M (CPU-GPU-ANE) configuration - displayed its best performance at 102.63 ms, resulting in a 2.324x speedup over the GPU runs. On the iPhone ANE, POET showed a runtime of 126.79 ms with the aforementioned PORT (CPU-GPU-ANE) configuration providing the best benefit on ANE as well at 102.63 ms, resulting in a 1.235x speedup. Overall, when contrasting the top perfor-

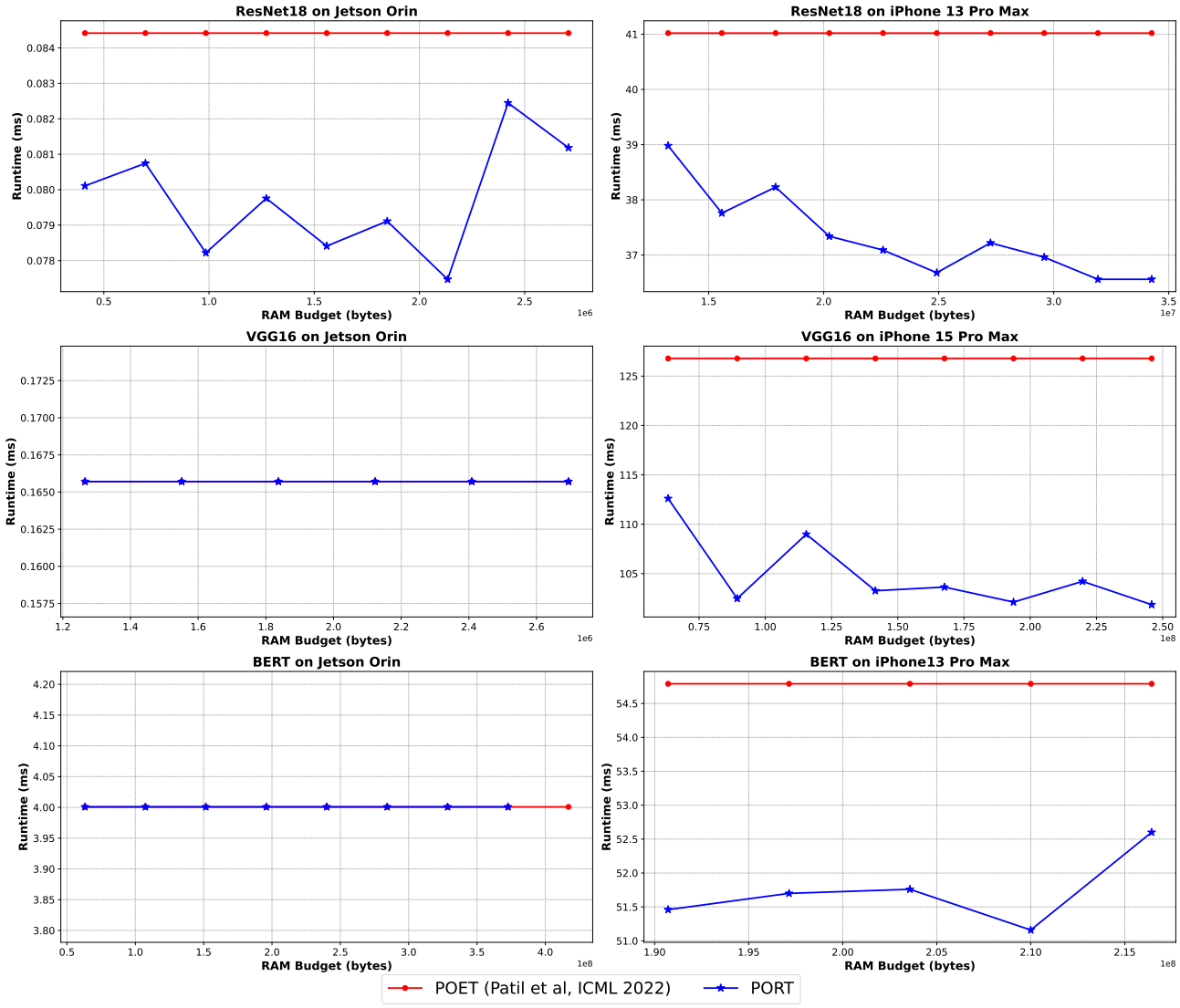

Figure 2. PORT provides runtime speedup across diverse models and devices. When comparing even just the best performing SoTA POET single-operator configuration to PORT's best configurations, PORT demonstrates runtime speedup on all networks for the iPhones with the most significant improvement observed on the iPhone 15 Pro Max for the VGG16 model at 1.245x, alongside notable speedup on JetsorinTX2. for some configurations like JetsorinTX2's VGG16 and BERT, there is no runtime enhancement, reflecting the complexity of profiling data on these devices. Even when leveraging their multi-operator settings, achieving runtime speedup isn't guaranteed due to inherent intricacies in task distribution and memory management, laying the path for further exploration. Breakdowns of these graphs can be found in Appendix A.

mance from POET, which was the iPhone CPU at 151.56 ms, with the best from all PORT configurations, the CPU-ANE combination at 101.86 ms, PORT provides a speedup of 1.245x is observed.

PORT's integration of unified memory to optimally leverage multiple accelerators reflects enhanced model execution times over single accelerator configurations run in POET. Even when comparing against the top-performing SoTA POET single-operator configuration, PORT demonstrates superior overall runtime gains on its best multi-accelerator configurations. Individual accelerators may offer speedup for specific layers in networks, and PORT takes advantage of this fact by effectively distributing tasks among the CPU, GPU, and ANE without the overhead of data transfer between isolated memory spaces. This benefit from unified memory ensures that the optimal accelerator for each layer is utilized and leads to runtime speedups in model training.

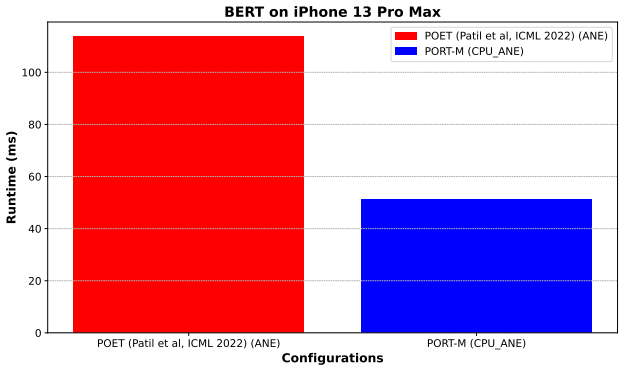

Figure 3. PORT-M profiled on iPhone 13 Pro Max (CPU and ANE) outperforms POET (ANE) with a 2.221x runtime speedup.

Table 1. **PORT performance for ResNet18 compared to POET**

	RAM Diff. ↓	Speed ↑
PORT patch	-84.81%	3.52x slower
PORT inplace	-67.95%	2.25x faster
PORT fused	-31.60%	1.03x faster
PORT cache	-0.99%	1.12x faster
PORT im2col	-0.99%	1.42x faster

5.3 Network Optimizations Results

For ResNet18, across all embedded device platforms, the PORT patch configuration significantly reduced RAM usage by 84.81% compared to the POET baseline configuration but led to the model performing 3.52× slower. The PORT inplace configuration demonstrated a runtime speedup of 2.25× while reducing RAM usage by 67.95%, reflecting improvements for both runtime and memory consumption. The PORT fused configuration showed more moderate gains with 1.03x speedup and a RAM reduction of 31.60%. Both PORT cache and PORT im2col configurations showed similar performance with no reduction in RAM consumption but runtime benefits of 1.12× and 1.42× speedup, respectively.

For VGG16, on all embedded device platforms, the PORT inplace configuration displayed a significant runtime speedup of 11.51x and RAM reduction of 94.24%. While the PORT patch configuration displayed a substantial reduction in RAM by 80.57%, it reflected a 3.25x slower runtime. The PORT fused configuration offered moderate memory reduction at 20.38% with no runtime improvements. The PORT im2col configuration also reflected no runtime improvements but didn't produce any memory consumption benefits either. Notably, the PORT cache configuration posed as an outlier in providing no reduction of RAM usage while considerably slowing down the model at 23.15x slower runtime.

Table 2. **PORT performance for VGG16 compared to POET**

	RAM Diff. ↓	Speed ↑
PORT patch	-94.24%	11.51x faster
PORT inplace	-80.57%	3.25x slower
PORT fused	-20.38%	1.00x faster
PORT cache	0.00%	23.15x slower
PORT im2col	0.00%	1.00x faster

6 Conclusion

As model architectures advance and grow, training these networks on edge devices presents unique challenges within memory-constrained environments, resulting in memory-intensive processes with increasingly large runtimes. However, modern edge devices often feature a unified memory architecture where device accelerators like CPUs, GPUs, and NPUs share a single physical memory. Our Private Optimal Runtime Training (PORT) algorithm capitalizes on this architecture, enabling neural network training under memory-limited constraints and optimal training times. Through an Integer Linear Programming (ILP) framework, PORT optimizes the scheduling of each network layer on both single and multi-accelerator platforms while balancing rematerialization and paging to maintain feasible training runtimes and constrained peak-memory usage. Additionally, by integrating layer-wise optimizations alongside the ILP layer scheduling, PORT enhances the efficiency of neural network training on edge devices.

We discover these benefits in runtime speedup and memory consumption reduction on a variety of device platforms and networks. With its application on both single and multi-accelerator platforms, PORT provides a robust solution for neural network training in memory-constrained environments. Future work could explore refining the scheduling algorithms and broadening PORT's compatibility with different device architectures.

References

Bai, Z., Zhang, Z., Zhu, Y., and Jin, X. {PipeSwitch}: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 499–514, 2020.

Banerjee, D. A microarchitectural study on apple's a11 bionic processor. *Arkansas State University: Jonesboro, AR, USA*, 2018.

Bartan, B., Li, H., Teague, H., Lott, C., and Dilkina, B. Moccasin: Efficient tensor rematerialization for neural networks. *arXiv preprint arXiv:2304.14463*, 2023.

Boroumand, A., Ghose, S., Akin, B., Narayanaswami, R., Oliveira, G. F., Ma, X., Shiu, E., and Mutlu, O. Google

neural network models for edge devices: Analyzing and mitigating machine learning inference bottlenecks. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 159–172. IEEE, 2021a.

Boroumand, A., Ghose, S., Akin, B., Narayanaswami, R., Oliveira, G. F., Ma, X., Shiu, E., and Mutlu, O. Mitigating edge machine learning inference bottlenecks: An empirical study on accelerating google edge models. *arXiv preprint arXiv:2103.00768*, 2021b.

Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E. Q., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. Tvm: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 11(20), 2018.

Choi, S., Kim, T., Jeong, J., Ausavarungnirun, R., Jeon, M., Kwon, Y., and Ahn, J. Memory harvesting in {Multi-GPU} systems with hierarchical unified virtual memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 625–638, 2022.

Das, A. and Brunschwiler, T. Privacy is what we care about: Experimental investigation of federated learning on edge devices. In *Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, pp. 39–42, 2019.

Dou, Z., Ye, D., and Wang, B. Autosegedge: Searching for the edge device real-time semantic segmentation based on multi-task learning. *Image and Vision Computing*, pp. 104719, 2023.

Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., and Bouchachia, A. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):1–37, 2014.

Gheorghe, A.-G., Crecana, C.-C., Negru, C., Pop, F., and Dobre, C. Decentralized storage system for edge computing. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 41–49. IEEE, 2019.

Imambi, S., Prakash, K. B., and Kanagachidambaresan, G. Pytorch. *Programming with TensorFlow: Solution for Edge Computing Applications*, pp. 87–104, 2021.

Jang, J. and Adib, F. Underwater backscatter networking. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pp. 187–199, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359566. doi: 10.1145/3341302.3342091. URL https://doi.org/10.1145/3341302.3342091.

Jia, Y. Learning semantic image representations at a large scale. 2014.

Jung, J., Kim, J., and Lee, J. Deepum: Tensor migration and prefetching in unified memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 207–221, 2023.

Kalamationos, J. and Kaeli, D. R. Temporal-based procedure reordering for improved instruction cache performance. In *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*, pp. 244–253. IEEE, 1998.

Kasperek, D., Podpora, M., and Kawala-Sterniuk, A. Comparison of the usability of apple m1 processors for various machine learning tasks. *Sensors*, 22(20):8005, 2022.

Khan, O., Park, G., and Seo, E. Dacapo: An on-device learning scheme for memory-constrained embedded systems. *ACM Transactions on Embedded Computing Systems*, 22 (5s):1–23, 2023.

Khani, M., Hamadanian, P., Nasr-Esfahany, A., and Alizadeh, M. Real-time video inference on edge devices via adaptive model streaming. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 4572–4582, 2021.

Khoshsirat, A., Perin, G., and Rossi, M. Divide and save: Splitting workload among containers in an edge device to save energy and time. *arXiv preprint arXiv:2302.06478*, 2023.

Kim, H., Sim, J., Gera, P., Hadidi, R., and Kim, H. Batch-aware unified memory management in gpus for irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1357–1370, 2020.

Klaus, J., Blacher, M., and Giesen, J. Compiling tensor expressions into einsum. In *International Conference on Computational Science*, pp. 129–136. Springer, 2023.

Li, S., Tian, C., Tam, K., Ma, R., and Li, L. Breaking on-device training memory wall: A systematic survey. *arXiv preprint arXiv:2306.10388*, 2023.

Li, T., Sahu, A. K., Talwalkar, A., and Smith, V. Federated learning: Challenges, methods, and future directions. *IEEE signal processing magazine*, 37(3):50–60, 2020.

Liao, X., Li, B., and Li, J. Impacts of apple's m1 soc on the technology industry. In *2022 7th International Conference on Financial Innovation and Economic Development (ICFIED 2022)*, pp. 355–360. Atlantis Press, 2022.

Lin, J., Chen, W.-M., Lin, Y., Gan, C., Han, S., et al. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33:11711–11722, 2020.

Lin, J., Chen, W.-M., Cai, H., Gan, C., and Han, S. Mcunetv2: Memory-efficient patch-based inference for tiny deep learning. *arXiv preprint arXiv:2110.15352*, 2021.

Nasir, M., Muhammad, K., Ullah, A., Ahmad, J., Baik, S. W., and Sajjad, M. Enabling automation and edge intelligence over resource constraint iot devices for smart home. *Neurocomputing*, 491:494–506, 2022.

Niu, W., Guan, J., Wang, Y., Agrawal, G., and Ren, B. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 883–898, 2021.

Patil, S. G., Dennis, D. K., Pabbaraju, C., Shaheer, N., Simhadri, H. V., Seshadri, V., Varma, M., and Jain, P. Gesturepod: Enabling on-device gesture-based interaction for white cane users. In *Proceedings of the 32Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, pp. 403–415, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6816-2. doi: 10.1145/3332165.3347881. URL http://doi.acm.org/10.1145/3332165.3347881.

Patil, S. G., Jain, P., Dutta, P., Stoica, I., and Gonzalez, J. Poet: Training neural networks on tiny devices with integrated rematerialization and paging. In *International Conference on Machine Learning*, pp. 17573–17583. PMLR, 2022.

Sarkar, V. and Thekkath, R. A general framework for iteration-reordering loop transformations. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pp. 175–187, 1992.

Schuler, M., Membarth, R., and Slusallek, P. Xengine: Optimal tensor rematerialization for neural networks in heterogeneous environments. *ACM Transactions on Architecture and Code Optimization*, 20(1):1–25, 2022.

SK, P., Kesanapalli, S. A., and Simmhan, Y. Characterizing the performance of accelerated jetson edge devices for training deep learning models. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6 (3):1–26, 2022.

Spicer, E. and Baidya, S. Performance tradeoff in dnn-based coexisting applications in resource-constrained cyber-physical systems. In *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 219–221. IEEE, 2023.

Sudharsan, B., Breslin, J. G., and Ali, M. I. Edge2train: A framework to train machine learning models (svms) on resource-constrained iot edge devices. In *Proceedings of the 10th International Conference on the Internet of Things*, pp. 1–8, 2020.

Sun, Y., Ochiai, H., and Esaki, H. Decentralized deep learning for multi-access edge computing: A survey on communication efficiency and trustworthiness. *IEEE Transactions on Artificial Intelligence*, 3(6):963–972, 2021.

Sun, Y., Shao, J., Mao, Y., Wang, J. H., and Zhang, J. Semi-decentralized federated edge learning with data and device heterogeneity. *IEEE Transactions on Network and Service Management*, 2023.

Tsymbal, A. The problem of concept drift: Definitions and related work. 05 2004.

Vasisht, D., Kapetanovic, Z., Won, J., Jin, X., Chandra, R., Sinha, S., Kapoor, A., Sudarshan, M., and Stratman, S. Farmbeats: An iot platform for data-driven agriculture. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pp. 515–529, 2017.

Wang, L., Xu, Y., Xu, H., Chen, M., and Huang, L. Accelerating decentralized federated learning in heterogeneous edge computing. *IEEE Transactions on Mobile Computing*, 2022.

Wu, B., Zhang, Z., Bai, Z., Liu, X., and Jin, X. Transparent {GPU} sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 69–85, 2023.

Zhao, J., Mortier, R., Crowcroft, J., and Wang, L. Privacy-preserving machine learning based data analytics on edge devices. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, pp. 341–346, 2018.

Zhao, X., Le Hellard, T., Eyraud-Dubois, L., Gusak, J., and Beaumont, O. Rockmate: an efficient, fast, automatic and generic tool for re-materialization in pytorch. 2023.

Zhao, Z., Wang, K., Ling, N., and Xing, G. Edgeml: An automl framework for real-time deep learning on the edge. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, pp. 133–144, 2021.

A APPENDIX

We include comprehensive results for the Profiling and FLOP-based experiments within this appendix. Graphs extend beyond just the best-performing configurations and provide detailed comparisons to the SoTA POET baseline.

A.1 FLOP-Based Results for PORT Optimizations

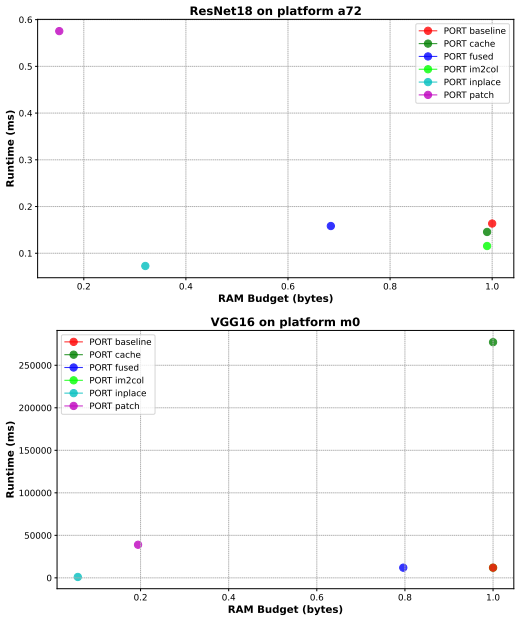

Figure 4. PORT inplace significantly outpaces baseline in reducing RAM budget and runtime for feasible solutions with PORT and patch and im2col also providing moderate improvements to reducing RAM consumption

We include FLOP-Based results for PORT optimizations against each other. We only include a single platform for each model because results are replicated across each platform regardless of the network. This accompanies the numbers discussed in section 5.3.

A.2 Profiled Results for PORT Multi-Accelerator Configurations

We provide a comprehensive overview of the benefits afforded by PORT over POET in profiled runs on both the iPhone13 Pro Max and iPhone15 Pro Max. Here it can be seen how PORT benefits network runtimes across differing combinations of operators, rather than just the best of all combinations as shown in Figure 2. This accompanies the explanation and breakdown provided in 5.2.

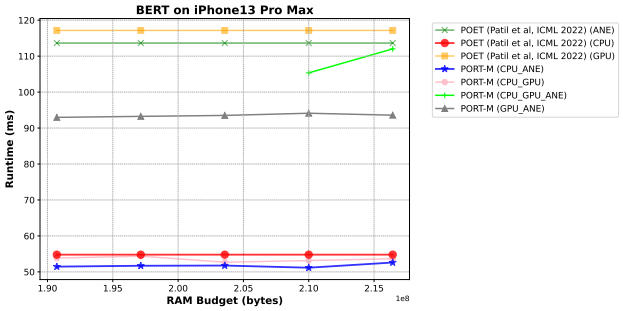

Figure 5. BERT on the iPhone13 Pro Max across all possible combinations of operators on PORT (vs) POET.

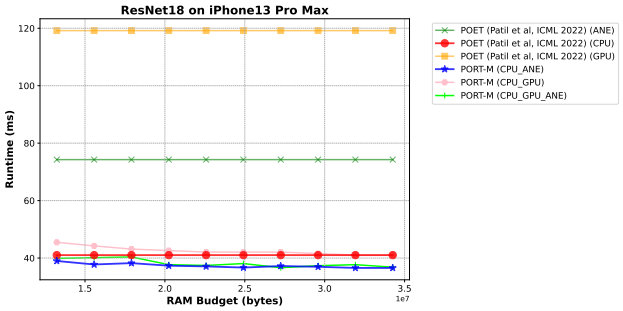

Figure 6. A breakdown of ResNet18 runs on the iPhone13 Pro Max across all possible combinations of operators on PORT versus all POET baselines

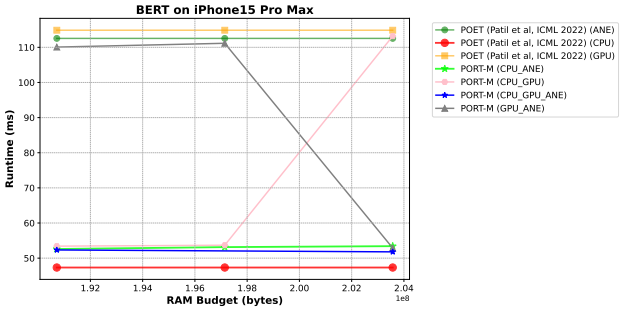

Figure 7. BERT on iPhone15 Pro Max across all possible combinations of operators on PORT (VS) POET.

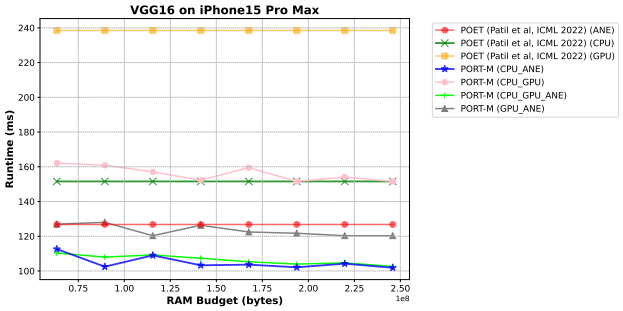

Figure 8. VGG16 on iPhone15 Pro Max across all possible combinations of operators on PORT (VS) POET