μ-TWO: 3× FASTER MULTI-MODEL TRAINING WITH ORCHESTRATION AND MEMORY OPTIMIZATION

Sanket Purandare [1] **Abdul Wasay** [2] **Animesh Jain** [3] **Stratos Idreos** [1]

ABSTRACT

In this paper, we identify that modern GPUs - the key platform for developing neural networks - are being severely underutilized, with $\sim$ 50% utilization, that further drops as GPUs get faster. We show that state-of-the-art training techniques that employ operator fusion and larger mini-batch size to improve GPU utilization are limited by memory and do not scale with the size and number of models. Additionally, we show that using state-of-the art data swapping techniques (between GPU and host memory) to address GPU memory limitations lead to massive computation stalls as network sizes grow.

We introduce μ-two, a novel compiler that maximizes GPU utilization. At the core of μ-two is an approach that leverages selective data swapping from GPU to host memory only when absolutely necessary, and maximally overlaps data movement with independent computation operations such that GPUs never have to wait for data. By collecting accurate run-time statistics and data dependencies, μ-two automatically fuses operators across different models, and precisely schedules data movement and computation operations to enable concurrent training of multiple models with minimum stall time. We show how to generate μ-two schedules for diverse neural network and GPU architectures and integrate μ-two into the PyTorch framework. Our experiments show that μ-two can achieve up to a 3× speed-up across a range of network architectures and hardware, spanning vision, natural language processing, and recommendation applications.

1 INTRODUCTION

Deep learning: Ubiquitous but expensive. Widespread deep learning workflows have enabled groundbreaking results for many applications, including but not limited to image recognition (Szegedy et al., 2017), recommendation systems (Naumov et al., 2019), natural language translation (Devlin et al., 2019), and video games (Berner et al., 2019). However, training neural networks is expensive and has adverse environmental impact (Zhu et al., 2018; Strubell et al., 2019). For instance, training BERT – a natural language model (with 200 million parameters) – takes 79 hours on 64 high-end GPUs resulting in an expense of approximately 12,000 USD and a carbon footprint of 1438 lbs (Devlin et al., 2019). To put this in perspective, BERT's training phase (including architecture search) emits as much carbon as six typical US cars would over their lifetimes.

Deep learning workflows train multiple models. Various stages of deep learning workflows involve training more than one models which effectively multiplies the cost of training. For example, during the model design stage, neural architecture search and hyper-parameter tuning require training of several models to come up with a near-optimal hyperparameter set (e.g., learning rate, momentum, and regularization) and architecture (e.g., number and types of layers) (Bergstra et al., 2011; Bergstra & Bengio, 2012; Elsken et al., 2019). Similarly, during the training phase, ensemble learning trains multiple models to improve the accuracy (Wasay et al., 2020; Ganaie et al., 2021). Such repetitive training tasks are prevalent, making up 70.2% of hardware resource consumption in a single-GPU setting (46.2%) and multi-GPU setting (24%) combined (Wang et al., 2021).

Low compute utilization. Modern deep learning hardware (e.g. GPUs, TPUs and accelerators) has high compute power and data parallelism. However, existing neural network operators cannot fully utilize modern hardware (Coleman et al., 2017; Zhu et al., 2018; Narayanan et al., 2018; Liu et al., 2020; Wang et al., 2021). This low utilization is due to the prevalence of small memory-bound kernels and the inherent complexity of writing code that fully utilizes this compute-rich hardware (Wang et al., 2021).

[1]School of Engineering and Applied Sciences, Harvard University, Cambridge, USA [2]Intel Corporation, Santa Clara, California, USA [3]Meta Platforms Inc., Menlo Park, California, USA. Correspondence to: Sanket Purandare <sanketpurandare@g.harvard.edu>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023.

State-of-the-art research proposes two strategies to tackle the problem of low compute utilization:

1) Increasing the minibatch size within a single neural network helps increase data parallelism and maximizes the number of floating point operations per second, thereby improving utilization on powerful accelerators (Rhu et al., 2016; Jain et al., 2018; Peng et al., 2020; Wahib et al., 2020).

2) Concurrently training multiple models on a single accelerator improves compute utilization by taking advantage of structural similarity across the various models. This concurrent training is achieved by fusing identical operators across the different models into a single operator (known as horizontal fusion) (Narayanan et al., 2018; Liu et al., 2020; Wang et al., 2021).

Large memory footprint limits scaling. While these strategies can improve compute utilization, the size and number of models that we can concurrently train on a single GPU are drastically limited by the large (and increasing) memory requirements of the training process as well as the limited memory capacity of modern GPUs.

Feature maps lead to memory over-subscription. For widely used models, the source of these large memory footprints are the feature maps (Rhu et al., 2016; Jain et al., 2018; Peng et al., 2020). For instance, feature maps occupy 83% of the memory when training VGG-16, whereas it is 97% for Inception (Jain et al., 2018).

A key observation is that feature maps have high inactive time in memory, which we define as the time between when they are produced in the forward pass and utilized in the backward pass. This is the key issue that leads to inefficient use of the limited GPU memory. We refer to this problem as memory over-subscription.

Swap and/or recompute tensors. Two techniques have been proposed to address memory over-subscription.
(1) Tensor recomputation: a subset of feature maps are discarded after their use in the forward pass and recomputed when needed during the backward pass (Chen et al., 2016; Jain et al., 2020).
(2) Tensor swapping: a subset of feature maps are offloaded to the larger host memory (i.e. CPU DRAM) during the forward pass and are fetched back into the GPU memory during the backward pass (Rhu et al., 2016; Peng et al., 2020; Wahib et al., 2020; Ren et al., 2021).

Tensor swapping and recomputation do not scale. As the size and number of models grow, directly applying the cutting-edge tensor swapping and recomputation techniques to concurrent multi-model training, leads to significant slowdowns. This is due to non-trivial overheads from tensor fetching delays during the backward pass and performance overheads from superfluous tensor recomputation. We show in our experiments that this slowdown can be as high as 50%.

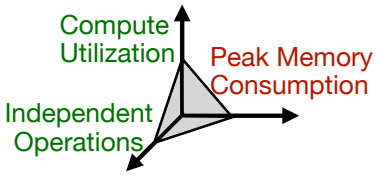


Figure 1. Multi-model training performance (latency), is a function of the trade-off space defined by compute utilization, peak memory consumption and degree of independence between operations.

Problem. This paper tackles the problem of scaling concurrent multi-model training as models grow and peak memory requirement surpasses the available GPU memory capacity.

Challenge. To efficiently scale concurrent training, we need to answer several questions: How many operations should we fuse across models to saturate compute? How many, and which intermediate tensors should be swapped or recomputed (if any)? When to discard/offload them and when to recompute/prefetch them back? How to maximally overlap the data movement with useful compute to minimize stalls? Many of these questions are NP-Hard problems.

Solution-μ-two. We present μ-two, a novel compiler that maximizes GPU utilization to efficiently scale concurrent training of multiple models. We show that μ-two achieves up to 3$\times$ faster end-to-end training latency than state-of-the-art approaches for models with memory requirements up to 6$\times$ the GPU memory size. The core insight in μ-two's design is that the training performance (latency), of any given set of models, is a function of the trade-off space determined by compute utilization, peak memory consumption, and the degree of independence between operations, as illustrated in Figure 1. To achieve scalable model training, we need to efficiently navigate this trade-off, for a given set of models and hardware, instead of having a fixed strategy. μ-two's compilation strategy automatically adapts to the properties of input models and the performance characteristics of the target GPU, enabling it to land in the sweet spot of the trade-off curve. To accomplish this, for each training session, μ-two performs static data dependency analysis and lightweight, yet accurate, run-time profiling. It then uses this information to (1) determine the number of operators to fuse (to saturate compute), (2) select the tensors to be swapped and/or recomputed (to reduce peak memory consumption), and (3) generate a tailored training schedule that maximally overlaps data movement with independent compute operations (to eliminate stalling).

Our Contributions are as follows:

1. We show that existing multi-model training and memory optimization strategies do not scale with the size and number of models, resulting in a 50% slowdown (§2 & §5).

2. We derive the design of μ-two, a compiler based on the central idea that the training performance (latency) of a given set of models is a function of the trade-off space determined by compute utilization, peak memory consumption, and the degree of independence between operations (§3).

3. Given a set of input models and a target GPU, we explain how μ-two collects and uses static and run-time information to generate a tailored training schedule that maximally overlaps swaps with independent compute operations (§4).

4. We discuss how to integrate μ-two in the open source framework PyTorch (Paszke et al., 2019). We show how each component of μ-two can be built with latest compiler tools and, can be smoothly plugged into the existing PyTorch execution stack (§B).

5. We conduct a thorough experimental analysis on a diverse set of hardware and several state-of-the-art model architectures spanning vision, natural language processing and recommendation systems. Our results show that compared to the state-of-the-art approaches (HFTA), μ-two enables concurrent training of 3-5$\times$ more models with a memory footprint of up to 6$\times$ the GPU memory size and delivers a 3$\times$ speedup (§5).

2 Background and Motivation

Neural network training. Training happens across several epochs. During every epoch, the neural network processes the data in subsets called mini-batches. For every round of mini-batch training, the computation is divided into two phases: (i) forward pass and (ii) backward pass. Figure 2a shows a computational graph corresponding to a network with 5 parameterized layers.

i) Forward pass. During the forward pass, the mini-batch is passed sequentially through every layer of the network to produce a set of neural network outputs. As shown in Figure 2a, to produce the output tensor $Z2$, we just need the input tensor $Z1$, the weight tensor $W2$, and enough memory to store the output tensor $Z2$.

ii) Backward pass. In the second phase, i.e., the backward pass, we compute the weight gradients. Backward pass is, in principle, application of the derivative chain rule. Like the forward pass, the backward pass is also processed sequentially but in reverse. As shown in Figure 2b, For computing the weight gradient $\nabla W2$ we require $\nabla Z2$ and $Z1$. The tensors, that are produced during the forward pass and are required for calculating the weight gradients in the backward pass, are called as feature maps or intermediate tensors, while the corresponding tensors produced during the backward pass for computing the weight gradients are called as gradient maps.

Inefficient memory usage. The major training frameworks, including Tensorflow (Abadi et al., 2016), PyTorch (Paszke et al., 2019), and MXNet (Chen et al., 2015), suffer from inefficient memory utilization because they store all the intermediate tensors, gradient maps, weights, and weight gradients throughout the processing of a minibatch.

Out-of-memory strategies. For many widely used models, the intermediate tensors produced during forward pass and consumed during backward pass are the major consumers of memory(Rhu et al., 2016; Jain et al., 2018; Peng et al., 2020). Out-of-memory approaches address this problem of fixed and inefficient memory allocation by freeing up memory occupied by intermediate tensors that are not immediately needed. These approaches are motivated by the fact that there is a huge temporal gap between the last use of a tensor during the forward pass and its first use in the backward pass. Figure 3a shows this idle time for every feature map when training BERT (a state-of-the-art language model).

Strategy 1: Tensor swapping. The tensor swapping strategies selectively offload tensors during the forward pass from the smaller GPU memory to the larger host memory. During the backward pass, the offloaded tensors are prefetched before their use to minimize the overall execution time. In scenarios with stringent memory constraints, we may need to offload several tenors to the host memory. In such cases, the backward pass may incur several stall cycles waiting for the required tensors to be fetched. Figure 2d shows what happens when computing ∇W_2: the required tensor Z_1 needs to be swapped in from host memory.

Strategy 2: Tensor recomputation. Tensor Recomputation strategies trade-off compute for memory. These approaches discard a selected number of tensors after their last use in the forward pass. During the backward pass, when these tensors are required for gradient computation, they are recomputed. Tensor recomputation involves the repetition of several forward pass operations to recompute the desired tensors which adds compute overhead in favor of memory savings. Figure 2c shows what happens when computing ∇W_2: the required tensor Z_1 needs to be recomputed. Unlike swapping, tensor recomputation does not add any stalling cycles in the execution path. Although tensor recomputation keeps the GPU busy at all times, the computation is superfluous.

Horizontal fusion. When concurrently training multiple neural networks on a single GPU to better utilize hardware, recent research proposes deeply fusing the neural networks together. In this approach, known as horizontal fusion (Wang et al., 2021), operators corresponding to every layer in the set of concurrently trained neural networks are fused together. These fused neural networks can then be trained simultaneously using a single GPU. For instance, individual convolutional operators across models are mapped

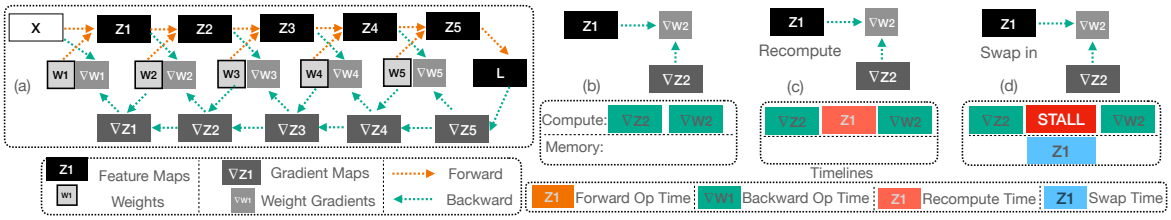
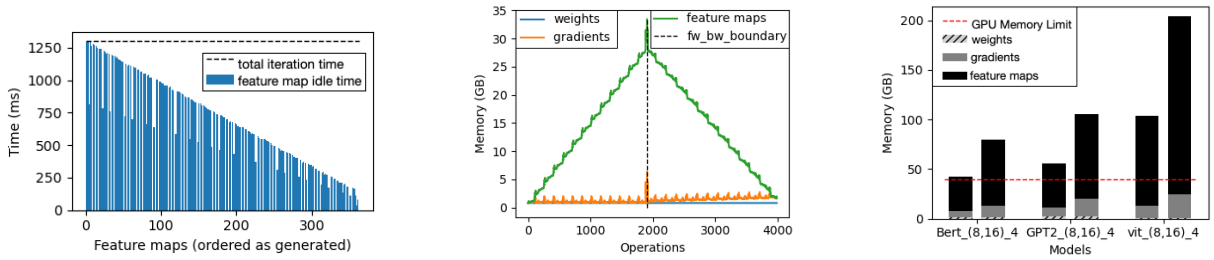


Figure 2. (a) Computation graph of a simple neural network with five layers showing the various intermediate tensors produced and required during the forward and the backward pass. (b), (c), and (d) show the computation of ∇W_2 when no out-of-memory, tensor recomputation, and tensor swapping strategies are employed respectively.



(a) When training BERT (with batch size 32), feature maps stay idle in memory for a long period of time.

(b) Peak memory increases during the forward pass and decreases during the backward pass (BERT with batch size 32).

(c) We exceed the memory limit of the Nvidia A100 GPU when horizontally fusing four models (batch size: 8 and 16).

Figure 3. Memory Footprint Characteristics

to grouped convolutions, while matrix multiplications are mapped to batch matrix multiplications. While horizontal fusion can improve compute utilization, naively fusing neural networks easily lead to a scenario, where the fused network does not fit in memory. As evidence of this, we show the memory requirement of training state-of-the-art models in Figure 3c. We observe that fusing just four models leads to a scenario where we exceed the memory limit of the Nvidia A100 GPU across all these models.

3 μ-TWO : INSIGHTS AND OVERVIEW

We discuss the core insights that drive μ-two's design and provide an overview of our compiler through an example. The next section discusses the core algorithm in detail.

3.1 μ-two Insights and design space

Insight I1: Swapping makes only the backward pass IO-sensitive. Any tensor swapping algorithm changes the IO sensitivity of the backward pass. During the forward pass, the tensors to be offloaded can be sent to the host memory asynchronously without blocking the computation process. However, fetching the swapped tensor back to the GPU during the backward pass lies in the critical path. If the required tensor is not fetched before the corresponding gradient calculation, then training stalls.

Insight I2: The forward and backward passes of two concurrently trained models are independent. When training a single model, the forward and backward passes depend on one another since the feature maps produced during a forward pass are utilized in the next backward pass for computing weight gradients (and the subsequent forward pass uses these gradient updates and so on). However, when concurrently training multiple models, there is no such data dependency across different models.

Insight I3: Peak memory consumption monotonically increases during the forward pass and decreases during the backward pass. During the forward pass, feature maps are created. During the backward pass, these accumulated feature maps are used to calculate the gradients and are released as soon as they are used. Figure 3b shows this phenomenon when training BERT.

Design implications. (I1) motivates that swapping should be scheduled conservatively, and in order to minimize stall overheads, one needs to achieve as much overlap with compute as possible. (I2) suggests that when concurrently training models, we can use compute operations from some models to overlap swapping operations from other models. Since modern interconnects (PCIe and NVLinks) allow full duplex data transfers, the forward pass and backward pass data transfers across models are resource-independent. (I3) suggests that, we should only multiplex backward pass operations with forward pass operations (from other models) due to their contrasting memory consumption patterns. (I1) also supports this observation since the IO sensitivities of the forward pass and backward pass are complementary.

Design trade-off space. Recent work has shown that horizontal operator fusion across multiple models is necessary to achieve maximum compute utilization. Combined with the design implications stated above, this lands us in a design trade-off space where different approaches exist with respect to compute utilization, independence between operations, and memory consumption. There are two extremes.

a) Complete fusion. On one hand, if we fuse all operations horizontally, we get a single and monolithic forward and backward pass, with individual operations from each of the participating models being inseparable. This means that: (i) we can achieve maximum compute utilization, but (ii) with minimal opportunities to multiplex operations, and (iii) with high peak memory consumption, resulting in more swap/recompute operations.

b) No fusion. On the other end of this spectrum, not applying any horizontal fusion gives us a separate forward and backward pass for each model. This means that we get (i) severe under-utilization of compute, but (ii) we get maximum independence across multiple operations from different models for overlapping with swaps, and (iii) the peak memory consumption of non-fused operations is low, resulting in minimal swap/recompute operations.

Design choice for μ-TWO: Sub-array fusion. At the core of μ-two's approach is the identification of the optimal point within the trade-off space by utilizing the degree of fusion as a parameter to strike a balance between competing objectives. This is achieved by partitioning the target model array into multiple sub-arrays and horizontally fusing operations across the models within each sub-array. This sub-array fusion balances desirable properties of the two extremes: μ-two can (i) sufficiently utilize compute, (ii) multiplex operations from the forward pass of one sub-array with backward pass operations of another sub-array (vice versa), providing opportunities to overlap any necessary swaps and (iii) also reduce peak memory consumption.

3.2 μ-two system overview and example

We present the system architecture of μ-TWO (Figure 4) through a running example.

Input. The input to the μ-two system is the set of models to train, i.e., their exact specification: architecture, loss function, mini-batch size, etc. Every model can have different hyperparameters, such as momentum, learning rate, and initialization but the architecture should be the same across all models. We illustrate μ-two's behavior through a running example, where an array of eight models $[M_1...M_8]$ is given as input.

1. Model sub-array constructor The first step is to enumerate all possible sub-array partitions of the input array of models. For our example of eight models, this results in the following four partitions:

(a) 2 sub-arrays of 4 models
(b) 2 sub-arrays of 3 models, 2 sub-arrays of 1 model
(c) 4 sub-arrays of 2 models each
(d) 8 sub-arrays of 1 model each

The input number of models can be odd or even. The number of sub-arrays created will always be even, since sub-arrays are processed in pairs. For example, we show processing of partition (a) in Figure 4: given the model array $[M_1...M_8]$, it creates two sub-arrays, A_1 consisting of models $[M_1...M_4]$ and A_2 consisting of $[M_5...M_8]$.

2. Horizontal fuser For each possible partition of the input model array, the operators across models within each sub-array are horizontally fused. For example, Figure 4 shows how, for partition (a), the models within A_1 and A_2 are horizontally fused to create two horizontally fused training arrays FA_1 and FA_2.

3. Graph tracer. For each fused sub-array, the Graph tracer derives the forward and backward computational graph consisting of nodes as operations and edges describing the data flow dependencies. As shown in Figure 4, for fused sub-array FA_1, the Graph tracer produces the forward pass graph FW_1 and backward pass graph BW_1, and similarly for FA_2, it traces FW_2 and BW_2.

4. Profiler. Then for each partition, the Profiler runs 3 iterations (after 1 warm-up) to collect performance profiling statistics so we can compare those partitions. There is an extensive set of data collected - a detailed description is provided in Section 4.1. Profiling data includes:

(i) Static analysis statistics such as the uses of feature maps in the forward and backward pass.
(ii) Run-time statistics such as the run-time of each operation, and swap-time of each feature map.
(iii) Memory usage statistics such as the active and peak memory consumption during execution of each node, and size of feature maps.

5. Scheduler. The collected profiling information is used to make scheduling and memory optimization decisions by the Scheduler. The scheduler considers all possible partitions (e.g., (a) through (d) in our example) and simulates the expected training time for each partition when making the best possible memory and computation utilization decisions for each one. If a partition does not satisfy the memory limit, it is discarded. At the end, the scheduler picks the partition with the shortest expected training time.

The Scheduler considers two sub-arrays at a time (closest iteration time). For example, for partition (a) in our running example (2 sub-arrays of 4 models), the Scheduler first operates on the backward graph BW_1 of fused model sub-array FA_1 and the forward graph FW_2 of fused model sub-

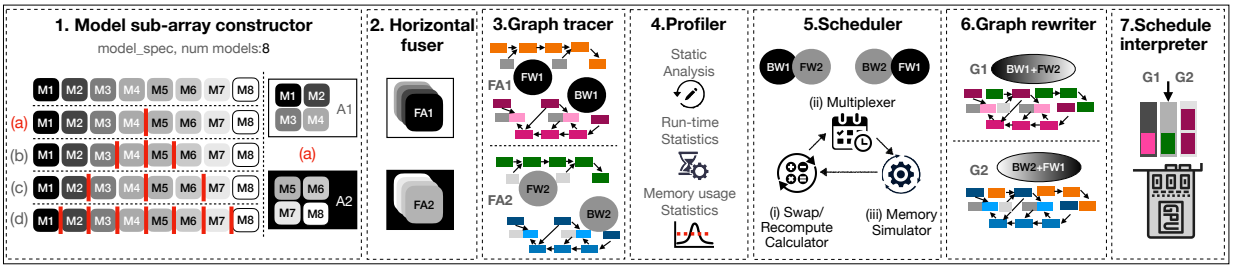


Figure 4. μ-two System architecture: the the end-to-end operations flow for 2 sub-arrays of 4 models.

array FA_2, since their operations are pair-wise independent. It then operates on $[BW_2, FW_1]$. The Scheduler utilizes the following components to analyze sub-arrays.

(i) Swap/recompute calculator. It makes a greedy decision regarding whether a feature map needs to be swapped or recomputed and calculates the cost for each case.
(ii) Multiplexer. It multiplexes operations from the forward graph to maximize the overlap of compute with swapping operations in the backward graph.
(iii) Memory simulator. It validates the decisions made by the Swap/recompute calculator and Multiplexer by ensuring that they do not violate GPU memory constraints.
We describe these components in detail in Section 4.2.

6. Graph rewriter. The Graph rewriter processes the graph pairs corresponding to the partition selected by the Scheduler. In our running example, it is partition (a). As illustrated in Figure 4 the Graph rewriter takes in graphs $[BW_1, FW_2]$ to produce a merged graph G_1 that reflects the decisions made by the Scheduler. The merged graph includes *hints* at various nodes to enforce decisions like: when a tensor should be swapped in/out, when a tensor should be discarded etc. It also extracts sub-graphs for regenerating the tensors selected for recomputation and inserts them at appropriate locations. It then repeats the same process for $[BW_2, FW_1]$ to produce G_2.

7. Schedule interpreter. The Schedule interpreter enforces the decisions made by the Scheduler and utilizes the hints provided by the graph re-writer to drive the execution of the merged graphs, e.g., G_1 and G_2 in our running example.

4 THE μ-TWO ALGORITHM

In this section, we describe the algorithms behind the core components of Profiler and Scheduler. Appendix A provides detailed algorithms for all μ-two components.

4.1 Profiling algorithm

The Profiler executes the computational graph to collect metrics and populates them as node attributes, as listed in Tables A and B in the Appendix. Figure 5a depicts the two-phase flowchart for profiling.

(i) Static data flow analysis gathers metrics that can be inferred from the computational graph without running it. These metrics capture information about the order in which tensors are accessed. For instance, $last\_fw\_uses$ denotes the set of all tensors that had their last forward use at this node, while $first\_bw\_uses$ denotes the set of tensors that had their first backward use at this node.

(ii) Run-time analysis is conducted over three iterations, with one warm-up iteration. For all run-time data collected, the median statistic is used. The profiling process involves three stages. In stage (1), before executing an operation during the backward pass, all intermediate tensors required for the operation and offloaded to host memory are swapped-in back to GPU memory. In stage (2), the operation is executed, and its end-to-end run time and memory usage are collected. In stage (3), after executing an operation in the forward pass, all intermediate tensors are swapped-out to host memory after their last use. Stages (1) and (3) enable profiling of computational graphs of models that exceed the GPU memory limit with the minimum assumption that inputs, outputs, and operation workspaces must fit on GPU memory in isolation.

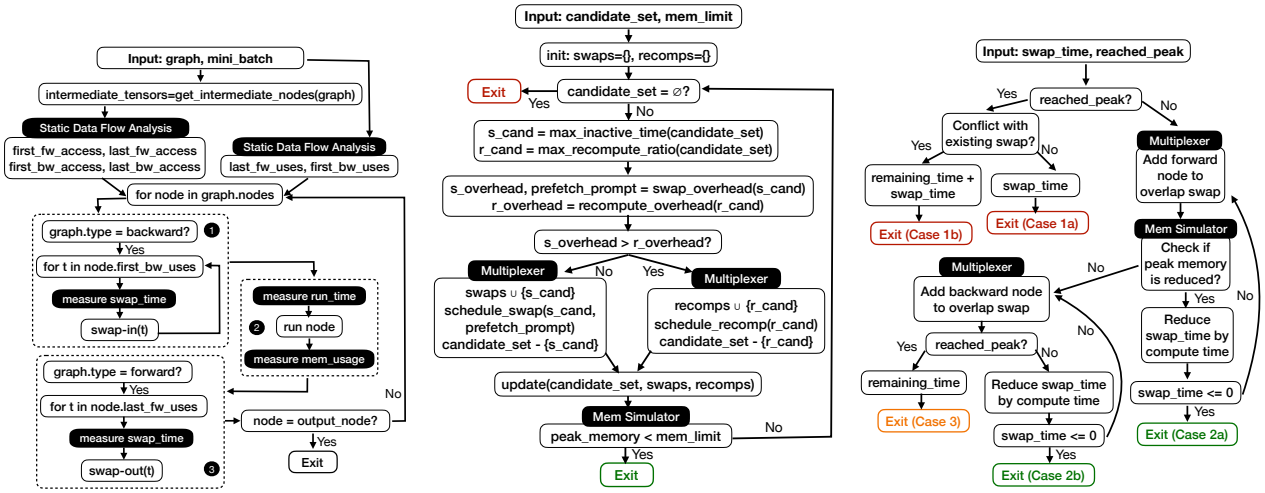
For the intermediate tensors we collect additional run-time information required for choosing the memory optimization strategy, such as $swap\_time$, in stages (1) and (3). Subsequent to profiling we populate the attribute $inactive\_time$, which denotes the time elapsed between the last forward and first backward access of the intermediate tensor.

4.2 Scheduling algorithm

The Scheduler takes as input a backward graph BW_j and a forward graph FW_i corresponding to fused model sub-arrays FA_j and FA_i and their profiling information with the objective of minimizing GPU idle time and superfluous compute under the given GPU memory constraints.

4.2.1 Scheduling policy

The scheduling algorithm determines whether to recompute or swap intermediate tensors for the forward and backward pass graph [FW_i,BW_j]. It uses two metrics: (1) the time elapsed between last use in forward pass and first



(a) Profiling (b) Scheduling Policy (c) Swap Overhead Calculation

Figure 5. Flow diagrams representing the profiler, scheduling policy and swap simulation for overhead calculation algorithms.

use in backward pass ($inactive\_time$) and, (2) the ratio of memory occupied by a tensor over the time required to recompute it ($recompute\_ratio$). These metrics capture an approximation of the overhead due to swapping or recomputing tensors. Tensors are selected for swapping or recomputation in decreasing order of their $inactive\_times$ and $recompute\_ratios$ using a greedy approach. See Figure 5b for the detailed steps.

The policy iterates over a set of candidate tensors and selects the best swap and recompute candidates (s_cand and r_cand, respectively) using the metrics defined above. Each memory optimization strategy has an associated overhead. For swapping, if the candidate tensor is not fetched before it is required in the backward pass, the processing stalls and the overhead is equal to the stall time. The Scheduler first attempts to schedule the candidate with zero overhead by overlapping it with compute, and only encounters a stall if it is unavoidable. For recomputation, the overhead is the minimum time required to recompute the tensor. We calculate the swap overhead ($s\_overhead$) for s_cand and recompute overhead ($r\_overhead$) for r_cand. The Multiplexer schedules the candidate with a lower overhead. It is then removed from the candidate set, and the side-effects of its selection are accounted for, by updating the candidates already chosen for swap or recompute, as well as the remaining candidates. The memory simulator then simulates the new schedule to calculate the peak memory consumption. If it is less than the specified limit, the process exits; otherwise, the steps are repeated.

4.2.2 *Swap simulation for overhead calculation*

To calculate the swap overhead we simulate the swap using the current state of the schedule and attempt multiplexing nodes from forward and backward graph to maximally overlap the swap. A step-by-step flowchart for calculating the swap overhead is shown in Figure 5c, while the timeline snapshots of the simulation algorithm are shown in Figure 6 and explained using an example.

The swap overhead calculation involves two terms: (1) the set of consecutive operations where peak memory consumption exceeds the GPU memory limit ($peak\_memory\_interval$) and, (2) the node in the backward graph ($prefetch\_prompt$) that begins swapping-in the intermediate tensor before its $first\_bw\_access$. Swapping is not possible in the $peak\_memory\_interval$ as there is no memory available for the tensor being swapped in. Any delay in this process causes a stall, which is measured as the swapping overhead. The algorithm takes in the $swap\_cand$, its $swap\_time$, and a flag indicating whether the peak memory interval has been reached ($reached\_peak$).

Based on when we enter the peak memory interval, the calculation of the swap overhead can be classified into three cases:

1. **No overlap.** When the peak interval is already reached before scheduling the swap, we cannot overlap this swap with compute and the swap overhead is calculated based on whether this swap: (a) does not conflict with existing swap or (b) conflicts with an existing swap. In case 1(a), the swap overhead is the total $swap\_time$ of the candidate. In case 1(b), the swap overhead is the remaining time of the the conflicting swap ($remaining\_time$), plus the $swap\_time$ of this candidate.

2. **Complete overlap.** When peak interval is not reached after scheduling swap, the swap can be completely overlapped by using: (a) forward pass operations only or (b) mix of forward and/or backward pass operations. For both the

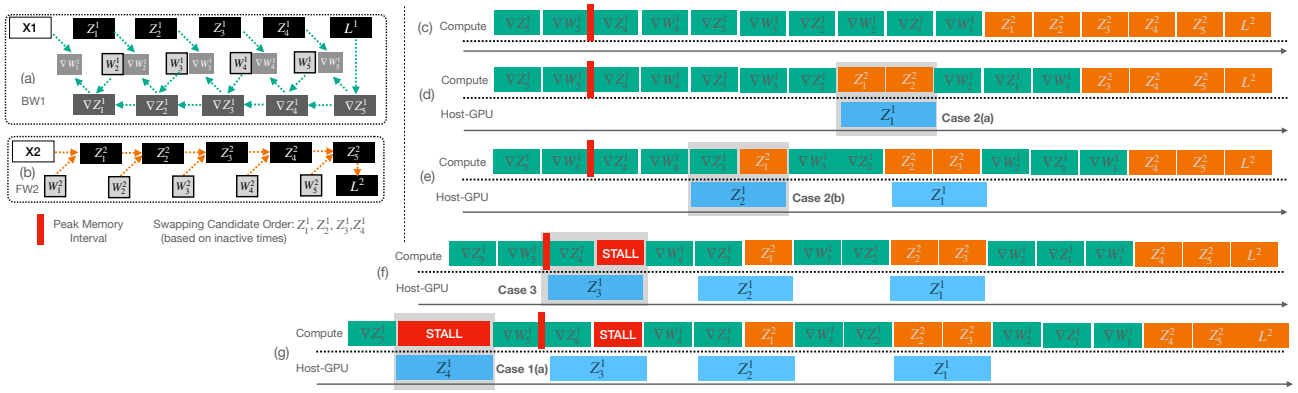


Figure 6. Timeline snapshots of swap simulations for calculating the swap overhead of intermediate tensors.

cases 2(a) and (b), we are able to completely overlap the swap with useful compute resulting in zero overhead.

3. **Partial overlap.** When peak interval is reached while scheduling swap, the swap is partially overlapped by using a mix of forward and/or backward pass operations. In case 3, any $remaining\_time$ that we are not able to overlap with compute is the swapping overhead.

Example. Figure 6a shows the backward computational graph BW_1 for fused sub-array 1, while Figure 6b, shows the forward computational graph FW_2 for fused sub-array 2. Figure 6c shows the topologically sorted compute operation timeline of BW_1 followed by FW_2. Based on the $inactive\_times$, the intermediate tensors to be scheduled for swapping for BW_1 are ordered as $Z_1^1, Z_2^1, Z_3^1, Z_4^1$ with their first backward accesses being $\nabla W_2^1, \nabla W_3^1, \nabla W_4^1, \nabla W_5^1$ respectively. First we try scheduling swap-in of Z_1^1 whose $first\_bw\_access$ is at ∇W_2^1. We then try to overlap it with forward operations Z_1^2, Z_2^2 and are able to do so with zero overhead, resulting in Case 2(a) as shown in Figure 6d. Next, we try scheduling the swap-in of Z_2^1. We are only able to use one forward operation Z_1^2 due to memory constraints. Hence, we also make use of the backward operation ∇Z_3^1 to overlap it. Since we moved Z_1^2 ahead in our timeline, we need to adjust the the previous overlap of Z_1^1 using the subsequent forward operations Z_2^2, Z_3^2. We are successful in doing so with zero overhead resulting in Case 2(b) as shown in Figure 6e. Next we try scheduling the swap-in of Z_3^1. We cannot use any more forward operations due to memory constraints. Hence, we try to make use of backward operations. We see that we could use only one backward operation ∇Z_4^1, since we reach the peak memory interval (indicated by red marker). We are only able to partially overlap this swap-in resulting in swap overhead equal to its $remaining\_time$ (stall time) resulting in Case 3, as shown in Figure 6f. At this point we can also switch to recompute Z_3^1 if its recompute overhead is lower than the stall time. Finally, we try to schedule the swap-in of Z_4^1. Since its $first\_bw\_access$ (∇W_5^1) happens during peak-interval, we cannot overlap it with compute operations. Hence, we encounter a stall equal to its $swap\_time$. This results in Case 1(a) as shown in Figure 6g. Similar to the previous case, we can decide to recompute Z_4^1 instead, if its recompute overhead is lower.

Table 1. We evaluate μ-two on state-of-the-art models covering a large space of use cases, arch. features, and model/batch sizes.

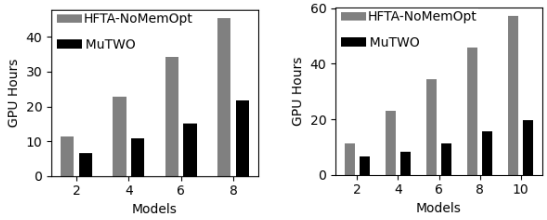
Application	Model Name	Functionality	Architectural Features	Params	Batch Sizes
Vision	Vision Transformer	Image Classification, Image Segmentation, Action Recognition	Positional image embeddings, transformers	60M	8 16
	Mobilenet v3 large		Depthwise separable convolutions	5.4M	64 128
	Resnet101		Convolutions, Skip Connections	44.5M	48 64
Natural Language Processing	Bert	Predict Next Sentence	Transformer Encoders	100M	16 24
	GPT2	Predict Next Token	Transformer Decoders	124M	8 16
Recommender Systems	NVIDIA DLRM	Item Recommendation	Encoders, Decoders, sparse embeddings	40M	512 1024

5 Experimental Analysis

We show how μ-two improves training time up to 3× across several state-of-the-art and diverse models.

Neural networks. We evaluate μ-two on six state-of-the-art neural network models from computer vision, natural language processing, and recommendation systems (Table 1). These models cover a diverse array of use cases, architectures, and model/batch sizes.

Experimental setup. We report results on setups with two different classes of Nvidia GPUs: A100 with 40 GBs of memory and V100 with 32 GBs of memory (more details in Table C in the Appendix). All experiments are run on the machine with the latest A100 GPU unless stated otherwise.



(a) Bert (batch size: 24) (b) ViT (batch size: 8)

Figure 7. μ-two saves between 5 to 40 hours in end-to-end training time when concurrently training several BERT and ViT models.

Baselines. We compare against two baselines:

1. HFTA-NoMemOpt. HFTA is the state of the art and has shown 3-11x speed-ups over all other concurrent training techniques (Wang et al., 2021). It offers no memory optimization and requires the allocation of peak memory during every training iteration. To provide a fair comparison, when the model array we want to train doesn't fit in memory, we break them down into subsets that do and sequentially train each subset.

2. HFTA-Capuchin. We construct this baseline by applying a state-of-the-art memory optimization algorithm Capuchin (Peng et al., 2020) to HFTA. Capuchin is a hybrid strategy for memory optimization, developed in the context of single model training, that uses both tensor swapping and tensor recomputation strategies to train models having peak memory consumption more than the GPU memory capacity.

The purpose of this baseline is to evaluate the performance of μ-two relative to naively applying state-of-the-art memory optimization strategy to concurrently training models.

Metrics. We report: (i) the improvement in end-to-end training time (i.e., decrease in GPU hours) and (ii) improvement in one round of mini-batch training (i.e., normalized speedup). We do not report accuracy or show convergence curves since the the training algorithm remains unchanged.

5.1 μ-two saves upto 40 out of 60 GPU hours in end-to-end training time

First, we show the impact of μ-two on end-to-end training time of two large scale state-of-the-art models – Vision Transformer and Bert. We vary the number of concurrently trained models and train each set of models for 10k iterations[1]. As shown in Figure 7, μ-two saves between 5 to 40 hours in training time when compared with state-of-the-art HFTA baseline. This result indicates that concurrently training models on a single GPU can significantly reduce the time (and the dollar cost) of training deep learning models thereby enabling model design and training in low-resource environments.

5.2 μ-two achieves 3x speedup in iteration latency

Next, we show how μ-two speeds up training iteration latency. In addition to the HFTA baseline, we compare against *HFTA-Capuchin* a variant of the μ-two system that fuses all models together to create a single execution graph (instead of the sub-array fusion employed by μ-two). Comparing against *HFTA-Capuchin* help us evaluate the additional benefit provided by the sub-array fusion technique introduced in μ-two. To measure the end to end latency, we perform 100 warm-up iterations to stabilize training and then measure 10 iterations using the PyTorch profiler and record the median.

We report the normalized speed up (over the HFTA baseline) in end-to-end iteration latency achieved by all three systems across all six models in Figure 8. μ-two consistently outperforms both *HFTA-NoMemOpt* and *HFTA-Capuchin*. This speedup scales as we increase the number of models. Overall, μ-two results in upto 3× speedup compared to state-of-the-art HFTA. When comparing μ-two and *HFTA-Capuchin*, we observe that μ-two consistently outperforms *HFTA-Capuchin* indicating that the sub-array fusion technique employed by μ-two is able to keep the memory optimization overhead to a minimum. In case of *HFTA-Capuchin*, we observe that as we increase the number of models the memory optimization overhead (stemming from complete fusion of models) takes precedence and performance takes a hit. This performance dip is particulary pronounced for compute-intensive vision models (i.e., Mobilenet, Resnet and NV DLRM). Overall, μ-two speeds up per iteration training latency establishing a new baseline for concurrent training of multiple models on a single GPU.

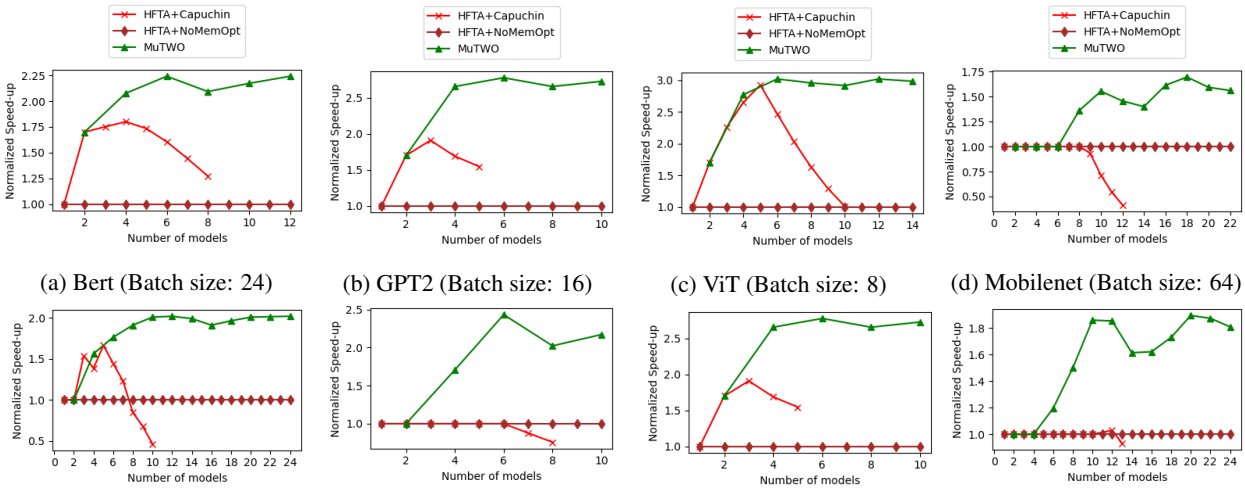
Diverse hardware. We repeat our experiments on a different set of hardware with V100 GPU (from Table C). We scale the batch size appropriately to account for the smaller GPU memory size of this setup. Figure 8g and Figure 8h show results for Bert and DLRM respectively. The trend continues to hold – μ-two provides highest speed-up.

5.3 Performance breakdown

To understand in detail how and why μ-two achieves significant speed-up, we break down its performance using the following metrics. We do the case study for two models, Bert 8a and Vision Transformer 8c, since they present an interesting behavior; they both show good speed-up on the lower range of models with *HFTA-Capuchin* but then show a performance dip later on. μ-two , on the other hand, shows consistent speed-up in both cases.

(a) Iteration latency breakdown. We report how much time μ-two spends in useful compute and recompute oper-

[1]Bert and Vision Transformer both require more than 10k iterations to converge, we cap-off our training at 10k, since it is sufficient to show the absolute benefit.



(a) Bert (Batch size: 24) (b) GPT2 (Batch size: 16) (c) ViT (Batch size: 8) (d) Mobilenet (Batch size: 64)

(e) Resnet101 (Batch size: 48) (f) DLRM (Batch size: 1024) (g) Bert (Batch size: 16) (h) DLRM (Batch size: 512)

Figure 8. μ-two results in up to 3× improvement in latency for one round of mini-batch training (forward and backward pass) when compared to state-of-the-art approaches across a diverse array of models. This improvement scales with the number of models.

ations compared to *HFTA-NoMemOpt*. And we see that by applying horizontal fusion to larger set of models, μ-two is able to extract more compute utilization than *HFTA-NoMemOpt*. Although, μ-two does some recompute due to fusing larger arrays of models, it does not outweigh its benefit.

(b) Recomputation ratio. (lower is better) We measure the ratio of the time spent in recompute with respect to the total compute time and then compare μ-two with *HFTA-Capuchin*'s recompute ratio. Clearly, μ-two spends less than half of the relative time in recomputation than *HFTA-Capuchin*. Since recompute is redundant, μ-two performs better by minimizing it.

(c) Swap overlap ratio. (higher is better) We measure the ratio of the time μ-two and *HFTA-Capuchin* are able to overlap their swaps behind total compute time. μ-two is able to overlap its swaps with useful compute, 2-3x more than *HFTA-Capuchin*. This shows that μ-two's multiplexer does better job at using operations from multiple model sub-batches to overlap swaps. This reduces the need for recomputing as well.

(d) Peak memory consumption ratio. (lower is better) We measure the amount of peak memory that exceeds the GPU memory limit, before optimizing it. Our goal is to show the ramifications of horizontally fusing all the operators together vs fusing them in sub-batches for *HFTA-Capuchin* and μ-two respectively. μ-two's sub-batching approach reduces the peak memory consumption, that further allows in reducing the number of swap and recompute operations.

5.4 μ-two profiling and scheduling overhead

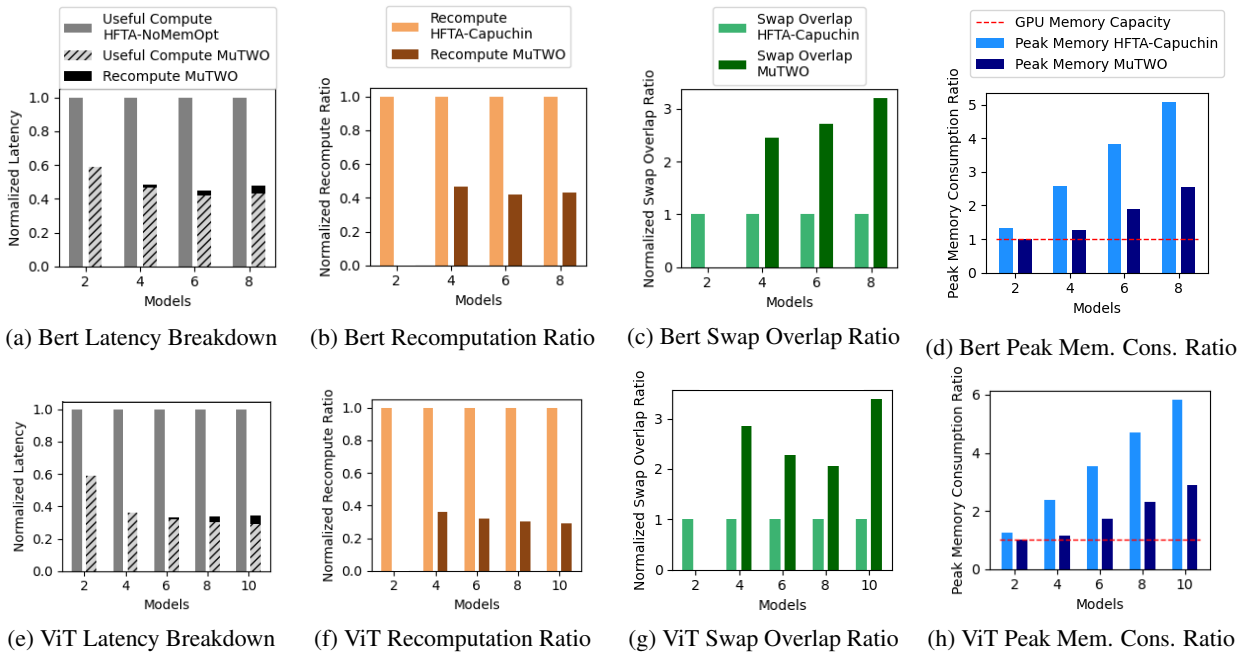
Since the profiling is done only for 4 iterations (3 measured + 1 warm-up) per choice of model array partition, it is lightweight and takes on the order of seconds to complete. Typically models take around thousands of iterations to converge so this overhead is negligible.

Choosing which tensors to swap or recompute has shown to be an NP-Hard problem (Jain et al., 2020; Peng et al., 2020; Wahib et al., 2020). Hence, we greedily decide to swap or recompute a tensor in every iteration. Overall our algorithm is quadratic in terms of the number of intermediate tensors. The scheduler makes all its choices by simulations based on profiling data. Therefore, it has has zero run-time overhead and typically completes scheduling within a minute.

6 Related Work

Multi-model training. Model and data sharing techniques help to improve hardware utilization. For example, HiveMind avoids redundant data transfer costs, performs kernel fusions, and concatenates inputs for models with shared weights to increase the computational intensity of the kernels (Narayanan et al., 2018). ModelPacking rewrites multiple neural networks as a single concatenated network (Liu et al., 2020). Similarly, Horizontally Fused Training Array (HFTA) maps kernels across models into a single highly optimized kernel (Wang et al., 2021).

However, all such approaches require the models to fit onto the accelerator's memory. On the other hand, μ-two supports out-of-memory execution, enabling larger batch sizes, larger model sizes and larger number of models being trained on the same accelerator. A comparison of multi-



(a) Bert Latency Breakdown (b) Bert Recomputation Ratio (c) Bert Swap Overlap Ratio (d) Bert Peak Mem. Cons. Ratio

(e) ViT Latency Breakdown (f) ViT Recomputation Ratio (g) ViT Swap Overlap Ratio (h) ViT Peak Mem. Cons. Ratio

Figure 9. Performance breakdown of the BERT and ViT model shows that: (i) μ-two results in lower compute latency, (ii) μ-two provides a lower recompute ratio (i.e., fewer tensors need to be recomputed), (iii) μ-two is able to better overlap swap with useful compute, and (iv) overall results in less peak memory consumption.

model training techniques is shown in Table D.

Out-of-memory approaches. We classify out-of-memory approaches into three categories. *1.Tensor rematerialization:* Gradient-checkpointing employs recomputation of feature maps discarded at specific checkpoints during the forward pass thereby enabling training neural nets at sub-linear memory cost at the expense of an extra forward pass computation (Chen et al., 2016; Jain et al., 2020). *2.Tensor swapping:* vDNN offloads tensors to the larger host memory during the forward pass and fetches them before they are required in the backward pass. *3. Hybrid strategies:* Recent work combines these two techniques. Capuchin dynamically decides which tensors to discard or offload based on their idle time in GPU memory and their recompute vs. transfer time ratio (Peng et al., 2020). KARMA proposes to interleave recompute with tensor prefetching to utilize idle GPU cycles when an offloaded tensor is being fetched back into memory (Wahib et al., 2020). Zero-Infinity and vPipe apply the memory swapping techniques not only to the activations but also to the model parameters to allow scaling of large networks across multiple GPUs in context of model-parallel training. (Rajbhandari et al., 2021; Zhao et al., 2022).

Although hybrid strategies can overlap some stalls incurred during swapping by recompute, the recomputation of these layers is still redundant computation. μ-two avoids this problem. It overlaps the stalls incurred during fetching of offloaded layers in the backward pass of some models, with the forward pass operations of others, thereby performing *useful* compute during the stalling period and employing recompute only if there is no sufficient useful compute to overlap the memory transfers. Hence, it eliminates the stalling of pure tensor offloading approaches and provides a superior compute-memory trade-off for hybrid approaches resulting in improved throughput. A comparison of out-of-memory approaches with μ-two is shown in Table E. Appendix D contains additional discussion on related topics.

7 Conclusion

In this paper, we tackle the problem of slow neural network training due to compute underutilization and inefficient memory usage. These problems become ever more critical as networks become more complex (larger) and as applications need to consider numerous networks simultaneously (e.g., in Auto-ML). We introduce a compiler μ-two, that is designed to efficiently navigate the performance trade-off of compute utilization, memory consumption, and number of independent operations. Augmented with lightweight profiling and static analysis, μ-two saturates compute through fusion, efficiently utilizes memory via swap/recompute, and maximally overlaps data movement with independent compute operations. μ-two generates tailored training schedules for any given set of models and target GPUs. Compared to the state-of-the-art approaches, μ-two enables concurrent training of 3-5× more models with a memory footprint of up to 6× the GPU memory size and delivers a 3× speedup.

REFERENCES

Nvidia. nvidia multi-instance gpu, 2020. URL https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

Nvidia. multi-process service, 2020. URL https://docs.nvidia.com/deploy/mps/.

Aot autograd: Ahead of time tracing pytorch autograd engine, 2021. URL https://pytorch.org/functorch/stable/notebooks/aot_autograd_optimizations.html.

Nvidia cuda profiling tools interface (cupti) - cuda toolkit, 2021. URL https://developer.nvidia.com/cupti.

Pytorch cuda memory statistics, 2022a. URL https://pytorch.org/docs/stable/generated/torch.cuda.memory_stats.html.

Cuda semantics pytorch, 2022b. URL https://pytorch.org/docs/stable/notes/cuda.html.

Fake tensor, 2022. URL https://pytorch.org/torchdistx/latest/fake_tensor.html.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012. URL http://jmlr.org/papers/v13/bergstra12a.html.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. In *NIPS*, 2011.

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost, 2016.

Coleman, C. A., Narayanan, D., Kang, D., Zhao, T., Zhang, J., Nardi, L., Bailis, P., Olukotun, K., Ré, C., and Zaharia, M. Dawnbench : An end-to-end deep learning benchmark and competition. In *Advances in Neural Information Processing Systems*, 2017.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. *ArXiv*, abs/1808.05377, 2019.

Ganaie, M. A., Hu, M., Tanveer, M., and Suganthan, P. N. Ensemble deep learning: A review. *CoRR*, abs/2104.02395, 2021. URL https://arxiv.org/abs/2104.02395.

He, H. and Zou, R. functorch: Jax-like composable function transforms for pytorch. https://github.com/pytorch/functorch, 2021.

Jain, A., Phanishayee, A., Mars, J., Tang, L., and Pekhimenko, G. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 776–789, 2018. doi: 10.1109/ISCA.2018.00070.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., and Stoica, I. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 497–511, 2020. URL https://proceedings.mlsys.org/paper/2020/file/084b6fbb10729ed4da8c3d3f5a3ae7c9-Paper.pdf.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization, 2017.

Kwon, W., Yu, G.-I., Jeong, E., and Chun, B.-G. Nimble: Lightweight and parallel gpu task scheduling for deep learning, 2020.

Liu, R., Krishnan, S., Elmore, A. J., and Franklin, M. J. Understanding and optimizing packed neural network training for hyper-parameter tuning, 2020.

Lukiyanov, M., Hua, G., Chauhan, G., and Dankel, G. Pytorch profiler, 2021. URL https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

Narayanan, D., Santhanam, K., Phanishayee, A., and Zaharia, M. Accelerating deep learning workloads through efficient multi-model execution. In *NeurIPS Workshop on Systems for Machine Learning*, December 2018.

Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., and Smelyanskiy, M. Deep learning recommendation model for personalization and recommendation systems, 2019.

Nguyen, V., Carilli, M., Eryilmaz, S. B., Singh, V., Lin, M., Gimelshein, N., Desmaison, A., and Yang, E. Cuda-graphs-pytorch, 2021. URL https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

Peng, X., Shi, X., Dai, H., Jin, H., Ma, W., Xiong, Q., Yang, F., and Qian, X. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pp. 891–905, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378505.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476205. URL https://doi-org.ezp-prod1.hul.harvard.edu/10.1145/3458817.3476205.

Reed, J., DeVito, Z., He, H., Ussery, A., and Ansel, J. torch.fx: Practical program capture and transformation for deep learning in python. In Marculescu, D., Chi, Y., and Wu, C. (eds.), *Proceedings of Machine Learning and Systems*, volume 4, pp. 638–651, 2022. URL https://proceedings.mlsys.org/paper_files/paper/2022/file/ca46c1b9512a7a8315fa3c5a946e8265-Paper.pdf.

Ren, J., Luo, J., Wu, K., Zhang, M., Jeon, H., and Li, D. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 598–611, 2021. doi: 10.1109/HPCA51647.2021.00057.

Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W. Vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

Ruder, S. An overview of gradient descent optimization algorithms, 2017.

Strubell, E., Ganesh, A., and McCallum, A. Energy and policy considerations for deep learning in nlp, 2019.

Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. Inception-v4, inception-resnet and the impact of residual connections on learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), Feb. 2017. URL https://ojs.aaai.org/index.php/AAAI/article/view/11231.

Wahib, M., Zhang, H., Nguyen, T. T., Drozd, A., Domke, J., Zhang, L., Takano, R., and Matsuoka, S. Scaling distributed deep learning workloads beyond the memory capacity with karma. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986.

Wang, S., Yang, P., Zheng, Y., Li, X., and Pekhimenko, G. Horizontally fused training array: An effective hardware utilization squeezer for training novel deep learning models. In Smola, A., Dimakis, A., and Stoica, I. (eds.), *Proceedings of Machine Learning and Systems*, volume 3, pp. 599–623, 2021. URL https://proceedings.mlsys.org/paper_files/paper/2021/file/a97da629b098b75c294dffdc3e463904-Paper.pdf.

Wasay, A., Hentschel, B., Liao, Y., Chen, S., and Idreos, S. Mothernets: Rapid deep ensemble learning. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 199–215, 2020.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pp. 2, USA, 2012. USENIX Association.

Zhao, S., Li, F., Chen, X., Guan, X., Jiang, J., Huang, D., Qing, Y., Wang, S., Wang, P., Zhang, G., Li, C., Luo, P., and Cui, H. vpipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel dnn training. *IEEE Transactions on Parallel and Distributed Systems*, 33(3):489–506, 2022. doi: 10.1109/TPDS.2021.3094364.

Zhu, H., Akrout, M., Zheng, B., Pelegris, A., Jayarajan, A., Phanishayee, A., Schroeder, B., and Pekhimenko, G. Benchmarking and analyzing deep neural network training. *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 88–100, 2018.

A μ-TWO ALGORITHMS

A.1 Profiling Attributes and Algorithm

Table A. Profiling and Scheduling attributes for nodes

Atrribute	Definition
Profiling Attributes	
rank	the position of the node in the topological sort of the graph
gtype	type of graph this node belongs to [forward/backward]
run_time	the run-time of the node in milliseconds
peak_mem	the peak memory usage in bytes
active_mem	the active memory usage in bytes (minimum required memory)
Scheduling Attributes	
to_offload	list of nodes to be offloaded to host memory after this node is executed
to_delete	list of nodes to be deleted after this node is executed
to_prefetch	list of nodes to be prefetched from the host memory before this node is executed
to_recompute	list of nodes to be recomputed before this node is executed

Table B. Profiling, Swapping and Recomputation attributes for intermediate nodes (feature map tensors)

Atrribute	Definition
Profiling Attributes	
inactive_time	the time duration elapsed between last forward access and first backward access
swap_time	time required to swap the tensor to/fro host memory and device memory
memory_size	the size of the tensor in bytes
last_fw_access	reference to the node that serves the last access of this tensor in the forward pass
first_bw_access	reference to the node that serves the first access to this tensor in the backward pass
last_bw_access	reference to the node that serves the last access of this tensor in the backward pass
Attributed for swapping	
prefetch_prompt	the node that serves as the prefetch prompt if this tensor is swapped
active_fw_interval	the first and last nodes in the forward pass during which the tensor resides in memory
active_bw_interval	the first and last nodes in the backward pass during which the tensor resides in memory
Attributes for recomputation	
recomp_srcs	intermediate tensors that serve as sources if the tensor needs to be recomputed
recomp_graph	the extracted sub-graph that needs to be executed to regenerate this tensor
recomp_cnt	the number of times this tensor needs to be recomputed during its lifetime
recomp_time	the time required to recompute this tensor from its current sources
total_recomp_time	the total time spent in recomputation of this tensor in its lifetime
recomp_memory	the peak memory required during a single recomputation of this tensor
recompute_ratio	memory_size/total_recomp_time

The steps for run-time profiling are shown in Algorithm A. For each node in the graph we collect memory consumption of the operation (line 11) and the end-to-end time required for it to complete by executing the operations in the graph one by one (lines 8-10). Subsequent to the execution of an operation in the forward pass, we swap-out all the intermediate tensors, after their last use, to the CPU memory (lines 12-15). Prior to the execution of an operation during backward pass, we swap-in all intermediate tensors, required for this operation, offloaded to the CPU memory back to the GPU memory (lines 4-7). This allows us to profile computational graphs of models that exceed the GPU memory limit with the bare minimum assumption that the inputs, outputs and workspace of every operation must fit on the GPU memory in isolation.

Algorithm A Run-time Profiler

```
1:  Input: graph
2:  #Perfrom static data-flow analysis
3:  for node in graph.nodes do
4:      for t in node.first_backward_uses do
5:          swap_in(t)
6:          #Measure swap-in time here
7:      end for
8:      #Start run-time measurement
9:      Execute node
10:     #End run-time measurement
11:     #Measure memory consumption here
12:     for t in node.last_forward_uses do
13:         swap_out(t)
14:         #Measure swap-out time here
15:     end for
16: end for
```

A.2 Scheduling Policy Algorithm

The scheduling policy is outlined in Algorithm B, that we now explain in detail. We first initialize $last\_prompt$ (line 2), that is the node in the backward graph at which the last swap-in was scheduled. It is initialized to be the last node in the backward graph. We choose the $swap\_candidate$ to be the intermediate tensor with largest $incative\_time$ (line 6) and calculate the swap overhead for this candidate (line 7). The calculation for swap overhead is explained in Section 4.2.2. We then choose our recomputation candidate that has the maximum $recompute\_ratio$ and then calculate the recompute overhead for this candidate (lines 8-9), explained in detail in Section A.4. We then make the decision to either swap s_cand or recompute r_cand (lines 10-18). If a candidate is chosen to be swapped, then we set the $to\_prefetch$ attribute of its $prefetch\_prompt$, and the $to\_offload$ attribute of its $last\_fw\_acesss$ node to be this candidate (lines 11-12). If the candidate is chosen to be

Algorithm B Scheduling Policy

1: **Input:** $candidate\_set$, mem_limit
2: $init(last\_prompt)$
3: $swaps = \{\}$
4: $recomps = \{\}$
5: **while** $candidate\_set \neq \emptyset$ **do**
6: $s\_cand = max\_idle\_candidate(candidate\_set)$
7: $s\_overhead, prompt\_node = SwapOverhead(s\_cand, last\_prompt)$
8: $r\_cand = max\_recomp\_candidate(candidate\_set)$
9: $r\_overhead = RecomputeOverhead(r\_cand)$
10: **if** $s\_overhead < r\_overhead$ **then**
11: $last\_prompt = Swap(s\_cand, prompt\_node)$
12: $swaps.add(s\_cand)$
13: $cand = s\_cand$
14: **else**
15: $Recompute(r\_cand)$
16: $recomps.add(r\_cand)$
17: $cand = s\_cand$
18: **end if**
19: $candidates.remove(cand)$
20: $recomp\_cnt = update\_recomps(cand, recomps)$
21: $update\_candidates(cand, recomp\_cnt, candidates)$
22: $update\_swap\_prompts(swaps, candidates)$
23: $mem\_consumption = get\_mem\_consumption()$
24: **if** $(mem\_consumption - mem\_limit) \leq 0$ **then**
25: **break**
26: **end if**
27: **end while**

recomputed then, we simply add it to the recomputation set, we process all of the recomputations together while graph rewriting. We account for the side-effects of this candidate on other candidates already chosen for swap or recompute (lines 20-23) and explain them in detail in Sections A.4 and A.5. Finally we obtain the peak memory consumption from the memory simulator and if it is lower than the memory limit we exit (lines 23-26).

A.3 Swap Overhead Calculation

The detailed steps for swap overhead calculation are outlined in Algorithm C. The input to the algorithm is the candidate to be swapped ($swap\_cand$), the last node in the backward graph that was used as a prefetch prompt ($last\_prompt$) and a flag that indicates that we have already reached the peak memory interval in swapping ($reached\_peak$). The peak memory interval is the set of consecutive operations in which the peak memory consumption exceeds GPU memory limit, we cannot perform any swaps in this interval since there is no memory left to allocate for the tensor being swapped in. Based on when the $prefetch\_prompt$ enters the peak memory interval the

Algorithm C Swap Overhead Calculation

1: **Input:** $swap\_cand$, $last\_prompt$, $reached\_peak$
2: $bw\_access = swap\_cand.first\_bw\_access$
3: $swap\_time = swap\_cand.swap\_time$
4: $r\_time = get\_recomp\_time(bw\_access)$
5: $swap\_time- = r\_time$
6: **if** $reached\_peak$ **then**
7: *# Case 1(a): Swap happens during peak interval*
8: **if** $bw\_access.rank < last\_prompt.rank$ **then**
9: $swap\_overhead = swap\_time$
10: **return** $swap\_overhead$, bw_access
11: **else**
12: *# Case 1(b): Swap happens during other swap*
13: $rem\_time = get\_remaining\_time(bw\_access)$
14: $swap\_overhead = swap\_time + rem\_time$
15: **return** $swap\_overhead$, bw_access
16: **end if**
17: **end if**
18: *# Cases 2, 3: Add forward graph nodes to overlap swap*
19: $fw\_node = first\_fw\_node$
20: **while** $swap\_time > 0$ **do**
21: $add\_forward\_node(bw\_access, fw\_node)$
22: $adjust\_graph(bw\_access, fw\_node)$
23: $mem\_safe = check\_mem\_safety()$
24: **if** mem_safe **then**
25: $swap\_time- = fw\_node.run\_time$
26: $fw\_node = fw\_node.next$
27: **continue**
28: **else**
29: **break**
30: **end if**
31: **end while**
32: *#Cases 2, 3:Use backward graph nodes to overlap swap*
33: **if** $bw\_access.rank < last\_prompt.rank$ **then**
34: $prefetch\_prompt = bw\_access.prev$
35: **else**
36: $prefetch\_prompt = last\_prompt$
37: **end if**
38: **while** $swap\_time > 0$ and $not\_reached\_peak(prefetch\_prompt)$ **do**
39: $r\_time = get\_recomp\_time(prefetch\_prompt)$
40: $swap\_time- = (prefetch\_prompt.run\_time + r\_time)$
41: $prefetch\_prompt = prefetch\_prompt.prev$
42: **end while**
43: $swap\_overhead = swap\_time$
44: **return** $swap\_overhead$, $prefetch\_prompt$

calculation of the swap overhead can be classified into the following cases:

1. When the peak interval is already reached before scheduling the swap, we cannot overlap this swap with compute and the swap overhead is calculated based on whether

this swap:
   (a) does not conflict with existing swap
   (b) conflicts with an existing swap
2. When peak interval is not reached after scheduling swap, the swap can be completely overlapped by using:
   (a) forward pass operations only
   (b) mix of forward and backward pass operations or backward pass operations only
3. When peak interval is reached while scheduling swap, the swap is partially overlapped by using a mix of forward and backward pass operations or backward pass operations only

We first obtain the node on the backward pass before which the swap should complete (bw_access) and the time required to swap the candidate (lines 2-3). For Case 1(a), the swap overhead is the actual $swap\_time$ (lines 7-10). For case 1(b), the swap overhead is the actual $swap\_time$ plus the remaining swap time of an existing in-flight swap that is already scheduled (lines 11-16). If the peak memory memory interval is not reached, we first try to add nodes from the forward graph one by one before bw_acess to overlap the swap. After adding a node from the forward graph we check if this actually reduces the peak memory consumption (using Memory Simulator, Section A.6), since adding a forward graph node comes at the cost of increased memory consumption. If that is the case, only then we proceed and reduce the $swap\_time$ by the forward node's computation time. Case 2(a) happens if the $swap\_time$ reaches zero (lines 19-31). In case the $swap\_time$ has not reached zero and we cannot use any more forward graph nodes to overlap, we try to see if we can use the backward graph nodes prior to bw_acess to overlap this swap. We reduce the $swap\_time$ by a backward node's computation time as we iterate in a reverse fashion through the backward pass graph. Case 2(b) happens if the remaining $swap\_time$ reaches zero, else we are left with some $swap\_time$ that cannot be overlapped and we incur a swap overhead resulting in case 3 (lines 33-43). The swap overhead calculation also takes into account any recomputation time that can be used to overlap the swaps (lines 4-5, 39-40), we explain how we do this in Section A.5.

A.4 Recompute Overhead Calculation

We largely adapt the recomputation algorithm from (Peng et al., 2020) which is based on the Spark's RDD lineage (Zaharia et al., 2012). A candidate might be recomputed either once when it is required or while recomputing some other candidate that requires it during its own recomputation. Hence, the recomputation overhead for a candidate is calculated as the time required to recompute the candidate ($recomp\_time$) multiplied by the number of times it will be recomputed ($recomp\_cnt$) in its lifetime. It is tracked as $total\_recomp\_time$ (Algorithm D) and used to compute the $recompute\_ratio$ for choosing candidates to recompute as explained Section 4.2.1.

Algorithm D Recomputation Overhead

input $recomp\_cand$
output $r\_overhead$
1: **return** $recomp\_cand.total\_recomp\_time$

For each recomputation candidate we maintain a set of $recomp\_srcs$, which denote the ancestor nodes of the candidates using which we can recompute the candidate. When a candidate is chosen for recomputation, it may affect the candidates (1) that are already chosen for recomputation and/or (2) candidates that maybe be chosen for recomputation in future.

In Algorithm E (Case 1), we first iterate through the existing set of recomputations, if the chosen candidate ($cand$) is one of the recomputation sources ($rp.srcs$) of an existing recomputation (rp), then we remove it from $rp.srcs$ and add the recomputation sources of the candidate ($cand.srcs$) to $rp.srcs$ (lines 4-6). We also count the number of times this candidate will be recomputed in its lifetime (line 7).

In Algorithm F (Case 2), t is the candidate chosen for recomputation. We iterate through the list of future candidates ($cand$) and check if either (a) t exists in recomputation sources ($cand.srcs$) of the $cand$ or (b) $cand$ exists in recomputation sources ($t.srcs$) of t. In Case 2(a), we remove t from $cand.srcs$ and add $t.srcs$ to $cand.srcs$ (lines 6-7). We then add the recomputation time of t to $cand$'s recomputation time (line 8). We then calculate the number of times the candidate may be recomputed for the already chosen recomputations and accumulate that in its potential total recomputation time (lines 10-14). For Case 2 (b), the potential total recomputation time is calculated as the number of times t is recomputed ($recomp\_cnt$) multiplied by the recomputation time of $cand$ (lines 17-19). Finally we update the recomputation ratio of all the remaining candidates.

Algorithm E Updating existing recomputations.

input $recomps$, $cand$
output $recomp\_cnt$
1: $recomp\_cnt = 1$
2: **for** rp in $recomps$ **do**
3: **if** $cand$ in $rp.recomp\_sources$ **then**
4: $rp.recomp\_srcs.remove(cand)$
5: $rp.recomp\_srcs.add(cand.recomp\_srcs)$
6: $rp.recomp\_time+ = cand.recomp\_time$
7: $recomp\_cnt+ = 1$
8: **end if**
9: **end for**
10: **return** $recomp\_cnt$

Algorithm F Updating remaining candidates.

input t, $recomp\_count$, $candidates$
1: **for** $cand$ in $candidates$ **do**
2: **if** t in $cand.recomp\_srcs$ **then**
3: **if** $cand.first\_bw\_access$ in $t.active\_bw\_interval$ **then**
4: **continue**
5: **else**
6: $cand.recomp\_srcs.remove(t)$
7: $cand.recomp\_srcs.add(t.recomp\_srcs)$
8: $cand.recomp\_time+ = t.recomp\_time$
9: $cand.total\_recomp\_time = cand.recomp\_time$
10: **for** rp in $recomps$ **do**
11: **if** $cand$ in $rp.recomp\_srcs$ **then**
12: $cand.total\_recomp\_time+ = cand.recomp\_time$
13: **end if**
14: **end for**
15: **end if**
16: **end if**
17: **if** $cand$ in $t.recomp\_srcs$ **then**
18: $cand.total\_recomp\_time = recomp\_cnt * cand.recomp\_time$
19: **end if**
20: $cand.updateRecomputeRatio()$
21: **end for**

A.5 Effects of swap and recompute on one another

Let's define two intervals to understand the effects of swapping on recompute. For an intermediate tensor we define $active\_bw\_interval$ as the set of operations that occur between its first use ($first\_bw\_access$) and last use ($last\_bw\_access$) in the backward pass. For a swapped tensor we define the prefetch interval as the set of operations that occur between it's prefetch start ($prefetch\_prompt$) and $first\_bw\_access$.

We first discuss the effect of swapping on recomputation. If a tensor is chosen to be swapped, then it may affect any remaining candidates that might be chosen for recomputation in future. If the swapped tensor is one of the recomputation sources of these candidates and their $first\_bw\_access$ does not lie in the swapped tensor's $active\_bw\_interval$, then it cannot be used as a recomputation source and the candidate's recomputation sources need to be updated. This is accounted for in Algorithm F (lines 3-16).

Now we explain the effect of recomputation on swapping. If a candidate is chosen for recomputation then it needs its sources to be available in memory. Firstly, if one of the remaining candidates is a source for this recomputation and is chosen for swapping in future then it must be made available before this recomputation takes place. Secondly, the recomputation time of this candidate can be used for overlapping already scheduled swaps if it lies in their prefetch interval or any future swaps while scheduling. Both these effects are accounted for by Algorithm B (line 22) and Algorithm C (lines 4-5, 39-40).

A.6 Memory Simulator

Algorithm G Memory Consumption Simulator

1: **Input:** graph, $static\_mem$
2: $fw\_inter\_mem = 0$
3: $bw\_inter\_mem = graph.inter\_mem$
4: $fw\_active\_mem = 0$
5: $bw\_active\_mem = 0$
6: $peak\_mem = 0$
7: **for** $node$ in graph.nodes **do**
8: **if** $node.gtype = backward$ **then**
9: $bw\_active\_mem = node.active\_mem$
10: **for** $pnode$ in $node.to\_prefetch$ **do**
11: $bw\_inter\_mem+ = pnode.memory\_size$
12: **end for**
13: **for** $rnode$ in $node.to\_recompute$ **do**
14: $bw\_inter\_mem+ = rnode.memory\_size$
15: **end for**
16: **for** $tnode$ in $node.first\_backward\_uses$ **do**
17: $bw\_inter\_mem- = tnode.memory\_size$
18: **end for**
19: **end if**
20: **if** $node.gtype = forward$ **then**
21: $fw\_active\_mem = node.active\_mem$
22: **end if**
23: $current\_mem = fw\_active\_mem+ bw\_active\_mem + bw\_inter\_mem+ fw\_inter\_mem - static\_mem$
24: $peak\_mem = max(peak\_mem, current\_mem)$
25: **if** $node.gtype = forward$ **then**
26: **for** $dnode$ in $node.to\_delete$ **do**
27: $fw\_inter\_mem- = dnode.memory\_size$
28: **end for**
29: **for** $onode$ in $node.to\_offload$ **do**
30: $fw\_inter\_mem- = onode.memory\_size$
31: **end for**
32: **for** $tnode$ in $node.last\_forward\_uses$ **do**
33: $fw\_inter\_mem+ = tnode.memory\_size$
34: **end for**
35: **end if**
36: **end for**
37: **return** $peak\_mem$

The memory simulation algorithm takes in the current state of schedule for a backward graph BW_j and a forward graph FW_i corresponding to the fused sub-arrays FA_j and FA_i

respectively. It maintains five variables to simulate the memory consumption at any current step:

(a) $static\_mem$: the memory occupied by weights and weight gradients.
(b) fw_inter_mem: the memory occupied by the intermediate tensors, between their $last\_fw\_access$ and end of forward pass, in FW_i.
(c) fw_active_mem: the active memory consumption during the forward pass excluding fw_inter_mem.
(d) bw_inter_mem: the memory occupied by the intermediate tensors, between the beginning of backward pass to their $prefetch\_prompts$ or $first\_bw\_access$, in BW_j.
(e) bw_active_mem: the active memory consumption during the backward pass excluding the bw_inter_mem.

When an intermediate tensor is prefetched or recomputed its memory is added to the bw_inter_mem (lines 10-15). When we encounter the $first\_bw\_acess$ of an intermediate tensor, we subtract its memory from bw_inter_mem since it is accounted for in bw_active_mem (lines 16-18). We then measure the current memory consumption as $current\_mem = (b) + (c) + (d) + (e) - (a)$ (line 23). We subtract $static\_mem$ since it is already accounted for twice in (b) and (c). Then we update the peak memory consumption ($peak\_mem$) (line 24). We subtract the memory of an intermediate tensors from fw_inter_mem that are deleted or swapped out after their $last\_fw\_acess$ and add the ones that are retained (lines 26-34). Finally, we return $peak\_mem$.

B Implementation Details

B.1 μ-two Implementation

B.1.1 Horizontal fuser

The horizontal operator fusion is implemented in μ-two using PyTorch's `vmap` library (He & Zou, 2021). It takes an array of models to be fused together, with the requirement that they must have identical architecture, and outputs a single fused model with horizontally fused operators. The models can have different loss functions, weight initialization, batch size, learning rate etc.

B.1.2 Graph tracer

After obtaining a fused sub-array of models we proceed to graph tracing. We use PyTorch `FX` to represent computational graphs (Reed et al., 2022). `FX` provides tools for graph representation, modification, transformation and execution. `FX` graphs are obtained by using a PyTorch library `AOT Autograd` (aot, 2021). `AOT Autograd` records the forward and backward operations performed by the fused model using a sample mini-batch. To enable tracing graphs of models having a memory footprint larger than the GPU memory capacity, we make use of PyTorch `Fake Tensor Mode` (fak, 2022). Fake Tensors are initialized on a meta device, they contain no actual data and only have meta data information like data type, size, memory layout, stride etc. We use `AOT Autograd` under the `Fake Tensor Mode` to obtain the `FX` Graphs.

B.1.3 Optimizers

In deep learning training, optimizers are used to update the model weights with weight gradients, following the backward pass. Currently, neither the `vmap` library allows horizontally fusing the optimizer calls nor does `AOT Autograd` allow tracing of optimizer calls. To circumnavigate this problem we provide a custom implementation of the SGD Optimizer using the point-wise `multiply` and `add` functions (Ruder, 2017). We then batch these function calls using `vmap` and attach them to the weight gradients in the backward pass graph. We do not implement other advanced optimizers like Adam in our work, as implementation of any optimizer is sufficient to establish a proof-of-concept of our scheduling mechanism (Kingma & Ba, 2017).

B.1.4 Profiler

We extend PyTorch `FX` Interpreter to implement the μ-two profiler (Reed et al., 2022). The `FX` Interpreter allows node by node execution of the `FX` Graphs. We override the `run_node` method of the `FX` Interpreter and wrap the run call using the PyTorch Profiler context manager (Lukiyanov et al., 2021). The PyTorch Profiler is a GPU profiling engine, built using Nvidia CUPTI APIs, and is able to capture GPU kernel events with high fidelity (nvi, 2021). We extract the latency of the CUDA kernel calls made by each node to calculate its run-timeto eliminate the host-side overhead of CUDA kernel launches. For calculating the memory consumption we use the CUDA Memstats tool subsequent to each `run_node` call (cud, 2022a). Finally, to measure the swap-out and swap-in times of the intermediate tensors we use CUDA Events to measure the Device-to-Host (D2H) and Host-to-Device (H2D) `memcpy` calls (cud, 2022b). To optimize the memory allocation we use pinned memory buffers on the host side. To get stable profiling measurements we warm-up the CUDA caching allocator using a warm-up run before actual profiling.

B.1.5 Schedule interpreter

The Schedule interpreter creates three CUDA Streams to represent compute operation queue, CPU-GPU and GPU-CPU swapping operation queues (cud, 2022b). The CUDA Streams provide a guarantee that all operations enqueued in a stream will be processed sequentially. However, it pro-

vides no guarantees for operations across streams. To add ordering for operations across streams we uses CUDA events to create synchronization markers. The CUDA Events API provides us with `record` and `wait` calls for each event. A CUDA event can be recorded in one stream and can be waited upon in another stream. This allows us to create operation ordering across different streams and controlled asynchronous processing of compute and swapping operations. Like the profiler, the μ-two Schedule interpreter also extends the `FX` Interpreter. It identifies and enqueues nodes in appropriate streams. Upon encountering a $prefetch\_prompt$ it enqueues the prefetch operations in the CPU-GPU stream, upon encountering $last\_fw\_access$ of swapped tensors it enqueues them in the GPU-CPU stream and all the compute and recompute operations are enqueued in the execution stream. Finally to overcome the host-side kernel launch and memory allocation overhead we record the operations of the Schedule interpreter into CUDA Graphs and then just replay them (Nguyen et al., 2021).

Algorithm H presents the detailed execution methodology. The input to the Schedule interpreter is the merged graph produced by the graph re-writer with scheduling hints. It executes the graph node-by-node and does the following: It first checks if the node to be executed is the $prefetch\_prompt$ for a tensor to be swapped-in, if yes, it adds a $prefetch\_begin$ event in the execution stream. It then waits for the event in the CPU-GPU stream and adds the swap-in operation for the tensor in the CPU-GPU stream. It then adds a $prefetch\_end$ event (lines 3-8). Subsequently, it adds all the recomputation nodes in the execution stream, if any (lines 9-11). Prior to the execution of the node, it waits for any $prefetch\_end$ events for its inputs and then enqueues the operation in execution stream lines(13-16). It then deletes any tensors that are to be recomputed (lines 18-20). Finally, if there are any tensors to be swapped-out, and have their $last\_fw\_uses$ at this node, then their swap-out operations are enqueued in the GPU-CPU stream (lines 21-25) by using appropriate events and waits.

Algorithm H Execution Engine

1: **for** $node$ in graph.nodes **do**
2: **if** $node.gtype = backward$ **then**
3: **for** $pnode$ in $node.to\_prefetch$ **do**
4: Execution Stream: $pnode.prefetch\_begin.record()$
5: CPU-GPU Stream: $wait(pnode.prefetch\_begin)$
6: $prefetch(pnode.cpu\_ref)$
7: $pnode.prefetch\_end.record()$
8: **end for**
9: **for** $rnode$ in $node.to\_recompute$ **do**
10: Execution Stream: $execute(rnode.recomp\_graph)$
11: **end for**
12: **end if**
13: **for** inp in $node.input\_nodes$ **do**
14: $wait(inp.prefetch\_end)$
15: **end for**
16: Execution Stream: $execute(node)$
17: **if** $node.gtype = forward$ **then**
18: **for** $dnode$ in $node.to\_delete$ **do**
19: Execution Stream: $delete(dnode)$
20: **end for**
21: **for** $onode$ in $node.to\_offload$ **do**
22: Execution Stream: $onode.offload\_begin.record()$
23: GPU-CPU Stream: $wait(onode.offload\_begin)$
24: $offload(onode)$
25: **end for**
26: **end if**
27: **end for**

B.2 Baseline Implementation

B.2.1 HFTA

We uses PyTorch's `vmap` library to implement *HFTA-NoMemOpt* (He & Zou, 2021). If the want to train 8 models and let's say only 4 models can be concurrently trained and fused by HFTA at a time due to peak memory consumption exceeding the GPU memory capacity, we run it twice to reflect the total running time. We note that, to the best of our knowledge no prior work applies memory optimization techniques to concurrently training models on a single GPU and hence it reflects the state-of-the-art baseline for evaluating relative performance. Although HFTA provides an open source implementation, it requires manual changes to the model source code, and manually converting all the operators in the base model to their horizontally fused version. First, the HFTA operators are not exhaustive and do not cover all implementations. The examples provided in HFTA represent only a sub-set of our workload models. The `vmap` library presents a fully automated way of horizontally fusing operators across models and hence we use that to reflect HFTA performance. We observe that `vmap` sometimes introduces additional transpose and cloning operations which may cause it to be slower than the original HFTA implementation. However, firstly we use the same library for implementing horizontal fusion for μ-two (Appendix B.1.1). Secondly, our scheduling algorithm is independent of the fusion strategy. Assuming that original HFTA implementation is faster, that will cause the compute latency to be lower and will provide fewer opportunities to overlap swaps. At the same time lower compute latency implies lower recompu-

tation cost, hence our scheduling policy will automatically choose more tensors to recompute to balance this. Thirdly, another significant difference between HFTA and `vmap` is that HFTA allows horizontal fusion of optimizers while `vmap` does not. We explain how we circumnavigate this limitation in Appendix B.1.3. Finally, HFTA open-source implementation is not fully composable with other PyTorch components.

B.2.2 HFTA-Capuchin

Capuchin does not provide an open-source implementation, and hence, we thoroughly implement Capuchin and then extend their algorithm to work with HFTA and call it *HFTA-Capuchin*.

C Additional Experimental Details

C.1 Experimental Setup Details

We conduct our experiments on the latest powerful NVIDIA GPUs. The first machine is the top-end AWS *pd24-xlarge* instance, having A-100 GPU with 40 GB of high bandwidth memory. It is connected to the host machine via full duplex PCI-e gen4 interconnect offering upto 32GB/s bidirectional transfer speed. The host memory size is 1152 GBs, shared across 8 GPUs. We only make use of 1/8th host memory respecting the proportion per GPU. Our second machine is the *Dell Claudron Server*, featuring Tesla V-100 GPU with 32 GB high bandwidth memory. It uses the same interconnect to the host machine as above. The host memory size is 384 GBs divided across 4 GPUs and we make use of 1/4th the host memory. Note that A100 has more compute capability and a larger GPU memory capacity than V100.

Table C. We experiment with two diverse hardware setups.

Instance	Nvidia GPU Version	GPU Mem (GB)	Tensor Cores	CPU-GPU Link	CPUs	CPU Mem
AWS p4d24-large	A-100	40	Yes	PCI-e Gen 4 x16 (32GB/s)	16	1152
Dell Claudron DSS 8440	Tesla V-100	32	Yes	PCI-e Gen 4 x16 (32GB/s)	16	384

D Additional Related Work

Other Scheduling Approaches: Nimble scheduler optimizes for executing the computation graph of a single model in parallel by partitioning the independent paths in the graph across different GPU streams (Kwon et al., 2020), while Hivemind runtime does the same for a multi-model execution graph (Narayanan et al., 2018). The goal of Nimble and Hivemind differ from μ-two since they use concurrent kernel execution to improve compute utilization for small kernels whereas μ-two uses multiple streams for overlapping data transfers with compute for Out-of-memory approaches for large kernels (which already have high compute utilization due to fusion). Further, Nimble optimizes the Kernel launch overhead by pre-allocation of memory, μ-two does the same using the CUDAGraph API in Pytorch (Nguyen et al., 2021). All other schedulers assume that the model/s fit on device memory hence only schedule the compute and not the data transfers.

Table D. μ-two outperforms all multi-model training techniques.

		HFTA	μ-two
Feature	Out-of-memory support	*No*	*Yes*
Parameters essential for hardware utilization	large minibatch size	*No*	*Yes*
	Large model size	*No*	*Yes*
	Large number of models	*Yes*	*Yes*
Hardware utilization	High Memory utilization	*No*	*Yes*
	High Compute utilization	*Yes*	*Yes*

Table E. μ-two achieves low overhead amongst all out-of-memory approaches.

Technique	Out of Memory Strategy	Compute Overhead	Stalling
vDNN	*Swapping*	*None*	*High*
Checkmate	*Recomputation*	*High*	*None*
Capuchin	*Hybrid*	*High*	*Low*
μ-two	*Hybrid*	*Low*	*None*

Other Hardware Sharing Approaches: MPS allows CUDA kernels from different processes to potentially run concurrently on the same GPU via a hardware feature called Hyper-Q (mps, 2020). MIG, which is currently only available on the most recent A100 GPUs, partitions a single GPU into multiple (up to 7) isolated GPU instances (GIs) where each job now run on a single GI (mig, 2020). HFTA has already shown better performance than MIG and MPS since they both do not horizontally fuse operators across neural networks. Further they have a high memory footprint, no memory optimization, restriction on the number of independent processes, and no multiplexing across processes. Since neural network operations are highly deterministic, repetitive, and exhibit very specific memory usage patterns, it is important that schedulers make use of this information to drive the GPU utilization to high levels for achieving high efficiency, lowering training cost, and enabling the training of models larger than GPU.

E Future Work

The question we address in this paper is when several models with identical architecture being trained, how can we

perform horizontal fusion for addressing compute underutilization and use memory optimization techniques to address memory limitation? This question maps to numerous critical problems such as hyper-parameter tuning, ensemble learning, and neural architecture search which are typical cases where the model architecture stays the same but soft hyperparameters like learning rate, learning rate decay momentum, loss functions, and weight initializations need to vary. However, training multiple models with heterogeneous architectures is also an exciting problem to pursue. Firstly, if all models in a batch have different architectures, then horizontal fusion is not possible. Secondly, technically only each sub-array should have identical architecture, and it can vary across sub-arrays. But if we have multiple instances of different architectures, the problem of finding the sub-array splits itself explodes combinatorially and is an interesting direction for future work.