Workload-Aware Hardware Accelerator Mining for Distributed Deep Learning Training

Muhammad Adnan
University of British Columbia
Vancouver, Canada
adnan@ece.ubc.ca

Amar Phanishayee
Microsoft Research
Redmond, USA
amar@microsoft.com

Janardhan Kulkarni
Microsoft Research
Redmond, USA
jakul@microsoft.com

Prashant J. Nair
University of British Columbia
Vancouver, Canada
prashantnair@ece.ubc.ca

Divya Mahajan
Georgia Institute of Technology
Atlanta, USA
divya.mahajan@gatech.edu

Abstract

In this paper, we present a novel technique to search for hardware architectures of accelerators optimized for end-to-end training of deep neural networks (DNNs). Our approach addresses both single-device and distributed pipeline and tensor model parallel scenarios, latter being addressed for the first time. The search optimized accelerators for training relevant metrics such as throughput/TDP under a fixed area and power constraints. However, with the proliferation of specialized architectures and complex distributed training mechanisms, the design space exploration of hardware accelerators is very large. Prior work in this space has tried to tackle this by reducing the search space to either a single accelerator execution that too only for inference, or tuning the architecture for specific layers (e.g., convolution). Instead, we take a unique heuristic-based critical path based approach to determine the best use of available resources (power and area) either for a set of DNN workloads or each workload individually. To ensure scalability to for distributed training, we decompose the problem into smaller tasks. First, we perform local search to determine the architecture for each pipeline and tensor model stage. Specifically, the system iteratively generates architectural configurations and tunes the design using a novel heuristic-based approach that prioritizes accelerator resources and scheduling to critical operators in a machine learning workload. Second, to address the complexities of distributed training, the local search selects multiple (k) designs per stage. A global search then identifies an accelerator from the top-k sets to optimize training throughput across the stages. We evaluate this work on 11 different DNN models. Compared to a recent inference-only work Spotlight, our method converges to a design in, on average, 31× less time and offers 12× higher throughput. Moreover, designs generated using our method achieve 12% throughput improvement over TPU architecture.

1 Introduction

Special-purpose hardware is well-suited for deep learning models due to their predictable memory accesses and readily parallelizable dataflow patterns [1–6]. As the models become bigger, training them on a single accelerator is infeasible due to their large memory footprint [7–9]. This mandates pipeline parallel and/or tensor model parallel training that splits the model across multiple devices [10–14]. However, a general scalable approach to determine the optimal solution for the combined exploration of accelerator architecture and operator execution schedule, in the context of *distributed training of deep learning models*, is an important yet open problem.

Prior works in this area have primarily focused on devising solutions for accelerators targeting inference [15, 16]. Some studies scope the architecture search only for matrix multiplication-based operations [17–19]. However, training presents unique challenges compared to inference: the execution graph for training is much larger, it requires greater computational intensity due to the additional backward pass, optimizer, and loss function, and intermediate activations are either stashed or recomputed between forward and backward passes, resulting in a larger memory footprint. Furthermore, all established pipeline parallel training schemes mandate that backward pass operators be placed and executed on the same device as the forward pass to minimize weight movement across accelerators [10–12, 20]. As such, the sheer complexity of the search space not only increases due to training, but it also requires co-optimization across forward and backward pass operators and is further compounded by distributed execution. To tackle the challenge of architecture search in the context of distributed DNN training, we design `WHAM`. It answers the following research questions:

1. **Individual accelerator design optimization**: What is the optimal architecture, given a specific DNN, under certain area and power constraints while maximizing end-to-end training metrics? Additionally, given a set of DNN workloads, can we identify a common architecture that performs well across all of them? How do algorithmically generated architectures by WHAM compare to previous training accelerators?

2. **Global optimization for accelerators in distributed training**: What is the ideal accelerator design for a given set of workloads executing pipeline and/or tensor model parallel training? Are heterogeneous designs obtained by tuning individual accelerators in each stage better than a homogeneous pipeline?

To address these questions, WHAM leverages the insight that accelerator vendors have converged on offering specialized processors, such as tensor and vector cores, that serve a wide range of common DNN operators [3, 21–24]. Tensor cores execute matrix multiplication-based operations, whereas vector cores execute activation and element-wise operators. In WHAM, each operator in the DNN graph executes on a single computation core. As a result, WHAM employs a tunable architectural template to define the scope of its design space exploration. The problem of tuning the hardware architecture boils down to determining the number of tensor and vector cores, their dimensionality, and on-chip buffer sizes. However, even with a template with only tensor core (maximum size 16×16 and maximum quantity 1), the search space for a MobileNet_v2 inference accelerator is on the order of $O(10^{72})$ [25]. This complexity for training increases to $O(10^{216})$ that includes exploring the tensor core dimensions, L1 register file size, global buffer size, and dataflow of operators [17].

Thus, to tackle the scale of accelerator architecture search for distributed training, WHAM breaks down the problem into manageable sub-problems. First, WHAM uses existing techniques to partition a model into stages, where each stage is split based on tensor model and/or pipeline parallel, then uses its novel search mechanism to find multiple suitable architectures for each stage in isolation. Second, to optimize end-to-end training-relevant metrics like throughput or Perf/TDP, WHAM does not simply select the best accelerator across each stage. Instead, it employs the top-k designs for each stage to search for a globally optimized pipeline.

Training is considerably more complex than inference, even for a single accelerator in a stage. For every accelerator architecture search, WHAM performs a novel heuristic-based search that prioritizes resources and scheduling for throughput-critical operators. A critical-path analysis offers a bound on the number of tensor and vector core, but this constraint does not affect output quality, as it corresponds to the model's parallelizability limit. To avoid iterating through all possible options, WHAM strategically trims the search space at every step by eliminating numerous tensor and vector core dimensions based on the feedback from previously explored options. Overall, with a critical-path-based algorithmic approach and configuration pruning, WHAM can reduce the MobileNet_v2 search space to $O(10^{14})$, a significant reduction compared to black-box approaches.

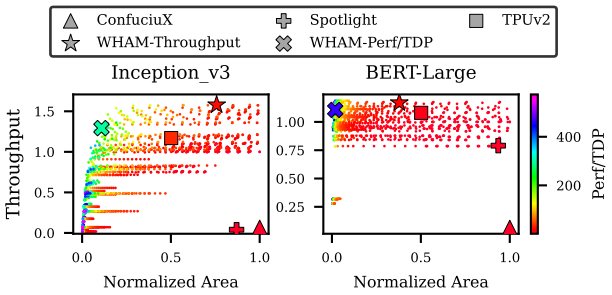

Figure 1. Design space exploration with WHAM.

WHAM's design space exploration for Inception_v3 and Bert-Large, executed on a single accelerator, is illustrated in Figure 1. The figure includes a comparison against designs by previous search approaches and a hand-optimized design [26]. With throughput as a metric, WHAM converges on a design that maximizes this metric. With Perf/TDP as the metric, WHAM maximizes Perf/TDP while maintaining a minimum throughput. Previous work, such as Confuciux and Spotlight, focus solely on inference, thus searches for accelerators based only on the forward pass requirements [17, 19]. As the figure demonstrates, inference designs are unsuitable for training. The metric of interest varies, the compute requirements for the backward pass differ, and the stashing of intermediate activations is not accounted for.

We evaluate WHAM using ten models spanning image classification, translation, and language models. For individual accelerator search, on average, the generated designs per workload, provide 20× and 12× higher training throughput against ConfuciuX and Spotlight suggested designs while taking 174× and 31× less time to converge, respectively. When optimizing an accelerator for a set of DNNs, WHAM's common design yields 2× and 12% better throughput than hand-optimized designs like NVDLA and TPUv2, respectively. WHAM's top-k based global architecture for distributed training with a pipeline depth of 32, the resulting design optimized per model offers 22% higher throughput and 8.1× better Perf/TDP compared to the TPUv2 accelerator.

2 Background

WHAM addresses an important yet open question: How to optimize the accelerator architecture and their corresponding runtime operator schedules for distributed training of large deep learning models? This section provides an overview of training, focusing on the challenges it poses for architecture search. We also outline several established strategies used to facilitate distributed and memory-efficient execution, which, in turn, complicates the search process further.

2.1 Architecture Search for Training vs Inference

The training graph comprises three broad passes - forward, backward, and parameter update. Unlike inference, training stashes activations in the *forward pass* for consumption during the *backward pass*. Intermediate activations are stored in

Table 1. Scope of prior works and WHAM

Works	Search Support and Operators	Search Technique
FAST [15]	Inference (Tensor + Vector Operators)	Black-box (Vizier)
ConfuciuX [17]	Inference (Tensor Operators)	RL and Genetic Algorithm
HASCO [18]	Inference (Tensor Operators)	Multi-objective Bayesian Optimization
PRIME [16]	Inference (Tensor + Vector Operators)	Machine Learning-based Surrogate
Spotlight [19]	Inference (Tensor Operators)	Domian Aware Bayesian Optimization
WHAM	**Training/Inference (Tensor + Vector)**	**Critical-path-based Heuristics and ILP**

the memory and fetched during the backward pass for loss gradient calculation. Thus, intermediate activations have a shorter lifespan in inference, lasting until the next consuming operation. In contrast, they persist for a relatively longer duration in training until the execution of the corresponding operator's backward pass.

Previous research in architecture search has primarily focused on inference-only accelerators, which do not account for training passes or consider the memory needed for storing activations [15, 16]. Other studies have mainly optimized within matrix multiplication-based Convolution and GEMM operations, neglecting other operations [17, 27–29]. However, pointwise vector and non-linear operations cannot be ignored when evaluating end-to-end training, as they can contribute significantly to the total training time. For example, Softmax execution time in BERT [9]scales quadratically with increasing sequence length and can take up to 30% of total training time on a TPU architecture [24]. Moreover, previous approaches to architecture search are computationally expensive as they rely on black-box optimizations [30], Bayesian optimization [18, 19], evolutionary [31], model-based learning (MBO) [32] or machine learning based techniques [16], as shown in Table 1. These methods search through and evaluate a large number of configurations to select the best option, albeit only for inference. WHAM instead accounts for the entire training graph, both tensor and vector operations, co-locates forward, corresponding backward and optimizer nodes on the same accelerator, and optimizes across them.

2.2 Distributed Training

For training, different degrees of parallelism can be employed. Data parallel training is a common strategy where model replicas are executed on multiple devices [33]. While effective in accelerating the training of many models, the recent trend of growing model sizes necessitates splitting the model across devices. This requires other parallelization schemes such as pipeline- and model-parallel training. Pipeline parallel training is an established method for dividing a model across a pipeline of stages, alleviating memory capacity requirements while maintaining training fidelity. Various pipelining strategies, such as Pipedream [11, 12], GPipe [10], and Pipemare [20] exhibit different memory footprints as the order of micro-batches and when the pipeline is flushed, varies. Another technique to further split the model to reduce memory requirements, and support parallel execution, is called model parallelism, specifically tensor model parallelism, where weights of a single operation are split across devices [14, 34]. Training large models involves combining some or all of these parallelization techniques. Pipeline and model parallel training execute different parts of the model across devices, thus is the focus of this work. *As such,* WHAM *is the first work to support architectural exploration for pipeline parallel training through a combined architectural optimization across pipeline stages.*

2.3 Device placement for Distributed Execution

Device placement and determining the distribution strategy is not the focus of this work, thus WHAM leverages existing techniques to split a model in pipeline and model parallel fashion. For pipeline parallel, while WHAM can support complex techniques, such as reinforcement learning [35], dynamic programming [36, 37], and randomized [34] search, in this work, we evaluate WHAM using a memory-capacity-based model partitioning scheme. For model parallel, which is commonly employed for large language models, WHAM considers the well established Megatron strategy [14]. Megatron style model parallel splits the attention layer to reduce the memory footprint per device. Overall, WHAM focuses on conducting an accelerator architecture search independent of the pipeline or tensor model scheme and device placement strategies. Pipeline and tensor model parallel scheme and device placement strategy are inputs to the search.

3 Architectural Search Parameters

An architectural template defines the realm of WHAM's architecture search. This architectural template covers a wide variety of machine learning accelerators from the literature [2–4, 6, 22, 24, 38–40]. This template is based on fundamental units commonly deployed for machine learning execution, tensor core and vector core. Tensor cores are 2-D arrays of Processing Engines (PEs), while vector cores consist of 1-D arrays. Each PE carries out a *scalar operation*, and together as a core, they can execute larger operations, such as convolutions, dot products etc. Each core in the computational unit also has dedicated on-chip storage.

3.1 Architectural Template

To define WHAM's architectural template, we look at the evolution of a well-established deep learning accelerator, TPU [24]. TPUv1 comprises a single 256×256 systolic array in a chip with separate storage for activations and partial sums. TPUv2 [26] is a training accelerator with a reduced systolic array size of 128×128, and TPUv3 [41] is a dual-core chip with each core having two 128×128 systolic arrays for training. Although large systolic arrays provide more compute per byte of High-Bandwidth Memory (HBM) bandwidth, they can be inefficient, as numerous workloads fail to fully utilize the 256×256 systolic array, as demonstrated by Figure 2

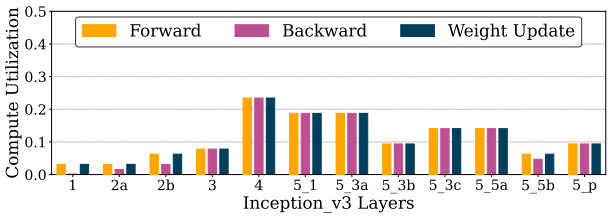

Figure 2. The per-layer utilization of tensor cores and vector cores in Inception_v3 model using a single tensor (256×256) and vector core (256 wide). The y-axis is capped at 50%. Layers with fewer channels have lower utilization.

for Inception_v3. This observation leads to the fundamental question that WHAM explores: what are the appropriate number of cores, size of each core, and number of cores per computational unit for a single or set of machine learning workloads?

WHAM's architectural template encapsulates these search parameters and builds on prior work [3, 22, 24, 38, 40, 42]. Figure 3 illustrates the micro-architecture of the template. It consists of computational units, each containing at most one Tensor Core (TC), one Vector Core (VC), or both, performing dense computations. Each core's inputs, outputs, and activations are stored in the L2 SRAM. Activations are stashed for the backward pass in the HBM. The scheduler generates control signals to execute each operator through the instruction dispatcher and FIFOs. A network-on-chip manages data transfer between cores. Table 2 displays the tunable parameters of this template, which accommodate a wide range of architectures. Each architecture design point is represented as: <*#TC, TC-Dim, #VC, VC-Width*>, indicating the number of TCs, 2-dimensional size of the TC, number of VCs, and 1-dimensional size of the VC, respectively. This flexibility enables WHAM to explore designs based on the model's compute, memory, and dataflow requirements without being limited to a specific family of accelerators.

Table 2. Architecture Configuration Parameters

Parameter Description	Notation	Range of Values
No. of Tensor Cores	#TC	1 to 256
Tensor Core x dim	TC_x	4 to 256
Tensor Core y dim	TC_y	4 to 256
No. of Vector Cores	#VC	1 to 256
Vector Core width	VC_w	4 to 256

Despite using the template, the search space for training workloads is vast, determined by core dimensions, the number of cores, operator schedules on the accelerator, and exploration of hardware dataflow of individual operators on a core. For example, for a moderately sized model like BERT-Base on a single accelerator, the exhaustive search space is on the order of $\sim 10^{2300}$, which is prohibitively expensive. This search complexity is further complicated by distributed training. To address this, WHAM refines the problem by iteratively searching for core dimensions and quantity. It employs

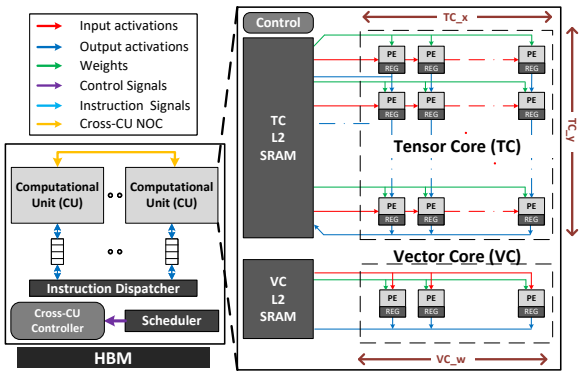

Figure 3. Architectural template. WHAM explores TC dimensions, VC width, and number of TCs and VCs.

an intelligent pruner with a performance feedback loop for the core dimensions to minimize the exploration. For core quantity, critical path analysis determines the count based on the parallizability and density of the training graph, assuming infinite resources and no area or power constraints. WHAM allocates accelerator resources and scheduling to critical operators, significantly impacting the end-to-end metric. These WHAM techniques are described in detail in the next section.

4 Accelerator Search with WHAM

The search process for an accelerator in WHAM takes an architectural template and a training operator graph as input, as illustrated in Figure 4. In deep learning, a model is defined by a pre-set layout that comprises the number of layers, the types of layers (e.g., convolutional, recurrent, dense, transformer), and the connections between them. The training operator graph further breaks down these layers into individual dense computations occurring during the forward pass, back-propagated backward pass, parameter update, and loss function. Each computation in the operator graph is executed on a specific type of hardware core, such as a tensor core, a vector core, or occasionally both. In some instances, operators like GEMM (executed on tensor core) and RELU (executed on vector core) are fused to minimize data transfer between off-chip HBM and on-chip memory [15, 43–45]. These fused operators execute on tensor and vector cores simultaneously, enhancing efficiency and reducing latency.

The WHAM accelerator search tunes the architectural template for the operator graph it hosts. This is either a model partition for distributed training or the entire model for single accelerator execution. The search determines core dimensions and their quantity (represented as <*#TC, TC-Dim, #VC, VC-Width*>), and establishes an effective operator schedule that optimizes end-to-end training metrics. This schedule efficiently utilizes on-chip resources and a tightly coupled HBM.

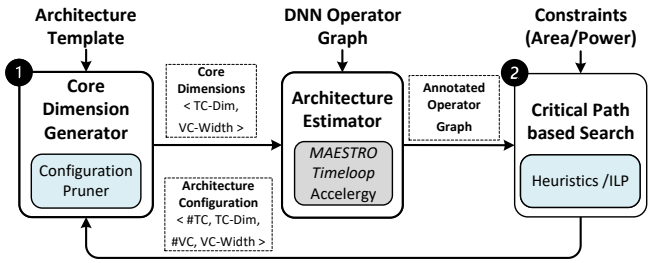

Figure 4. WHAM's accelerator search takes an algorithmic approach to prune the large search space for training.

To perform the search, the module ❶ generates dimensions (<*TC-Dim, VC-Width*>) starting with the largest configuration that fits within the area constraint. For each dimension, the operator graph is annotated with the necessary latencies used by the subsequent critical-path-based search. Then, module ❷ determines the number of cores (<*#TC, #VC*>) best suited for the model by leveraging critical operators in the training graph and prioritizing cores and scheduling for those. This search takes advantage of the insight that the backward pass in training mirrors the dataflow of the forward pass, although with different operators. By resolving resource conflicts in the forward pass, WHAM's search can potentially mitigate conflicts in the backward pass.

To prune the search, WHAM does not explore all possible core dimensions and select the best. Instead, the configuration pruner uses performance feedback from previously explored dimensions to determine early stopping.

4.1 Core Dimension Generator

The Dimension Generator iteratively generates <*TC-Dim, VC-Width*> for Tensor Core Dimension and Vector Core width, respectively. Starting with the largest architecture, dimensions are decreased per iteration until convergence. Following prior work, we use *<256 × 256, 256>* as the largest design and explore dimensions in powers of 2 to accommodate common tensor shapes (batch size, sequence length, hidden size, embedding width) in DNN models. However, WHAM supports any step size. This module utilizes feedback from previous searches to determine whether smaller configurations need evaluation or if the search has concluded, avoiding evaluation across all configurations. The configuration pruner is detailed in Section 4.5 and Figure 6.

4.2 Architecture Estimator

For each <*TC-Dim, VC-Width*>, the Architecture Estimator annotates the operator graph with essential information from the critical-path-based search. Since each operator executes on one or both cores, only TC-Dim and VC-Width are needed to determine this information. Each operator in the graph, across forward and backward passes, is annotated with the core type it executes on, latency for execution on this core, and energy expended. The latency of each operator allows the rest of the flow to identify latency-critical operators and assess if adding more cores of the required type would resolve resource conflicts during execution.

We use established open-source tools like Timeloop [27] and MAESTRO [28] to determine the latency of tensor core operators. For other operators, such as vector and point-wise operations, performance is modeled using a custom model similar to prior work [15]. The operator latency accounts for compute and data movements from HBM and on-chip memories. On-chip memory, denoted by $<TC_{L2-SRAM}, TC_{L1-REG}>$, is determined based on TC-Dim and the dataflow employed by Timeloop/MAESTRO mapping. The $<VC_{L2-SRAM}>$ is based on the VC-Width to ensure full vector core utilization. To avoid stalls in the vector core, L2-SRAM is set according to VC-Width. For energy estimations of each operator, we leverage the established Accelergy [46] tool.

The Architecture Estimator feeds the annotated operator graph with latencies and energy expended across forward and backward operators to the critical-path-based search.

4.3 Critical Path-Based Search

WHAM is the first work to propose a critical-path-based approach for architecture search. This approach is an alternative to black-box optimizers or reinforcement learning. The idea of using such critical path-based heuristics is that training graphs can be large and require optimization across co-located forward and backward operators. This search technique leverages the insight that auto-grad in training mirrors the forward pass dataflow to the backward pass, where the backward operators correspond to partial derivatives of forward operators. Based on this insight, the critical path analyzer identifies latency-critical operators and checks for resource conflicts. If conflicts are observed, the critical-path-based heuristic adds the required core for that operator, potentially resolving the conflict in both forward and backward passes. The scheduler prioritizes critical operators, as delaying them would reduce the training throughput. As a first step, this module determines the theoretically best possible latency and critical operators for each architectural configuration, followed by the search algorithms.

Theoretical Best Latency and critical operators. For every <*TC-Dim, VC-Width*>, WHAM uses operator estimates to determine the theoretically best possible latency a graph can achieve. As Soon As Possible (ASAP) scheduling provides the best latency for the operator graph's forward and backward passes. As Late as Possible (ALAP) scheduling is also required to determine the critical path. Both ASAP and ALAP schedules presume an infinite number of each core type, as illustrated in Figure 5. ASAP scheduling fully exploits parallelization within the graph by scheduling operators as soon as their predecessors are complete. In contrast, ALAP schedules each operator as late as possible without

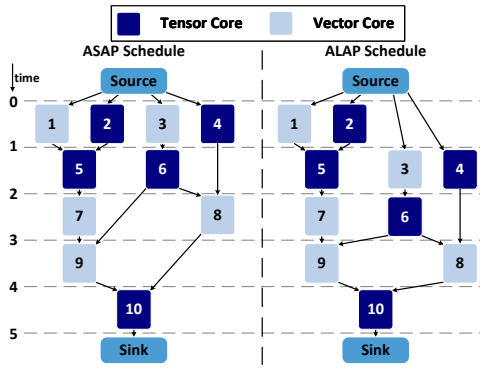

Figure 5. ASAP and ALAP schedules. For simplicity, each operator executes with a unit cycle.

impacting the overall best latency. Operators with the same ASAP and ALAP time are the most critical operators. These operators do not have any slack in their scheduling window.

Critical-path guided Heuristics. For each core dimension, WHAM employs heuristics for tuning the number of cores. It takes as input the ASAP and ALAP schedules for each forward and backward operator, critical operator information, and a bound on the maximum number of cores required by the training graph. These heuristics, called Mirror Conflict Resolution (MCR), start with a single core of <*TC-Dim, VC-Width*> and iteratively determine which core needs to be added. Each iteration adds either one tensor or vector core or an entirely new computational unit with both cores. The criterion for adding a core is: operators are scheduled using a greedy scheduler described below on the current number of TCs and VCs; if a resource conflict causes a delay for an operator beyond its slack in the ALAP schedule, the core that executes the operator is added. Fused operators are executed on a computational unit with both tensor and vector cores. If adding the core/unit to the first conflict does not violate area and power constraints, the change is finalized. The iterative process of MCR builds on this change until an addition is invalidated due to area and power constraints, the architecture converges to the theoretical best possible latency, or no operator is left with a conflict that causes the time to cross the ALAP start time. The MCR heuristics are shown in Algorithm 1.

The rationale behind this heuristic is two fold. First, if an operator's start time is beyond its ALAP schedule time, it would undoubtedly increase the overall latency of the graph execution. Second, the forward and backward pass operators are arranged in a mirror dataflow; hence, resolving earlier conflicts in the forward pass can potentially resolve conflicts in the backward pass, significantly improving the throughput of the overall training iteration.

Greedy Scheduler for Heuristics: The heuristics employs a greedy scheduler through the algorithm. Operators are scheduled greedily, meaning they are scheduled if all their predecessors are completed and the required core is available.

Algorithm 1: Mirror Conflict Resolution Heuristics

Input: $G(V, E)$ *// Annotated Operator graph*
Input: *TC-Dim, VC-Width* *// Current config of the architecture*
Input: $Constraints$ *// Area and power constraints*
Result: $\#TC, \#VC$
$config_{curr}$ = <1, TC-Dim, 1, VC-Width> *// Initialized with 1 core*
$schedule_{time} = GreedyScheduler(G,\ config_{curr})$
while $config_{curr} \neq config_{prev}$ **do**
  foreach $node \in G(V, E)$ **do**
    $delay_{critical} = schedule_{time} - ALAP_{time}$
    if $CheckResourceConflict(node)$ & $delay_{critical} > 0$ **then**
      break
  $config_{prev} = config_{curr}$
  // Adding the necessary core and updating the configuration
  $config_{curr} = AddCoreCheckConstraints(node, constraints)$
  $schedule_{time} = GreedyScheduler(G,\ config_{curr})$
  if $CheckRuntimeIsWorse(schedule_{time})$ **then**
    return $config_{prev}$

If two operators are ready but insufficient cores are available, the order is determined based on operator criticality. The combination of ASAP/ALAP schedules defines the slack for each operator's start time. Operators with zero slack are the most critical. For the remaining operators, higher slack means lower priority, and vice versa. To reduce idle time, a low-priority operator can be added before a critical operator when it doesn't impact the critical operator's start time. As we traverse the graph, the order of operators within a core/unit adhere to the dependencies in the graph. All the operators within a single core/unit are executed in-order. Dependencies across units are maintained using a semaphore block.

4.4 Integer Linear Programming Formulation

The heuristics search for the number of cores takes a deliberate approach towards prioritizing resources towards critical operators. To offer formal guarantees of optimality as an alternative to heuristics, we also formulate our search of the number of cores as an ILP. ILP similar to the heuristics is bounded by critical-path's best latency as that is the limitation of the model. Even with this bound, the integer program is co-optimizing the # of cores and the schedule of the operators, thus can take a non-trivial amount of time.

Problem Definition: Let $G(V, E)$ denote the operator graph with vertex set V and edge set E. A v denotes a single vertex in V and e for a single directed edge in E. Δv denotes the estimated latency of each operator v. Possible types of cores is denoted by C, and in this work we assume $C = [Tensor\ Core,\ Vector\ Core]$. However, our ILP formulation works for any set C. For a core $c \in C$, the variable $x(c)$ denotes the number of cores of type c our solution uses, and we assume that $x(c) \geq 1$ by preprocessing the input. The

function $M : V \rightarrow C$ gives a mapping of operators V to computational core C; an operator $v \in V$ needs to be processed on the core $M(v)$. Let $A(c), P(c)$ denote the area utilization and power consumption of each unit of core c, and let A, P denote the total area and power constraints. We require that the total area and power used by all computational cores is at most A, P. The main decision variables are $y_{(v,t)}$, that indicate when the operator v is scheduled. We assume that time is slotted and entire DAG can be feasibly scheduled in T time slots. We get an estimate of T by doing a binary search. For an operator v, $y_{(v,t)} = 1$ only if v starts its execution at time slot t. If $y_{(v,t)} = 1$, then it means that operator v is scheduled on core $M(v)$ in the contiguous set of time slots between $[t, t + \Delta v - 1]$.

ILP Objectives: As we aim to minimize the training time, area, and power, we formulate a multiple objective ILP.

- First objective minimizes the training iteration time by tuning the number of cores. We formulate it as follows:

$$\text{minimize} \quad \sum_{t \in T} t \cdot y_{(v^*,t)} \tag{1}$$

- Second objective minimizes the area and power consumption whilst keeping it within the constraints.

$$\begin{aligned} &\text{minimize} \quad f(z) = \sum_c x(c) \cdot A(c),\ f(p) = \sum_c x(c) \cdot P(c) \\ &\text{subject to} \quad f(z) \leq A,\ f(p) \leq P \end{aligned} \tag{2}$$

ILP Constraints: The constraints ensure a valid schedule of operators is obtained that respects the graph dependencies.

- First set of constraints enforce that each operator gets scheduled only once and is executed non-preemptively.

$$\sum_{t \in T} y_{(v,t)} = 1 \qquad \forall\, v \in V \tag{3}$$

- Next we enforce capacity constraints. We ensure that the total number of operators that require computational core c at any time t is $\leq$ the tuned number of computational cores.

$$\sum_{v \in V, M(v)=c} \ \sum_{t'=t-(\Delta v-1)}^{t} y_{(v,t')} \leq x(c) \quad \forall\, t \in T,\ c \in C \tag{4}$$

  Above constraint implies that if an operator v has a start time t' (that is, $y_{(v,t')} = 1$) then it would require core $M(v)$ for the entire duration of $[t', t' + \Delta v - 1]$.
- Finally, we want the operators to be scheduled in order of their precedence within the operator graph.

$$\sum_{t \in T} t \cdot y_{(v',t)} - \sum_{t \in T} t \cdot y_{(v,t)} \geq \Delta v \quad \forall\, e : (v, v') \in E, v \rightarrow v' \tag{5}$$

ILP Outputs: As output, the ILP provides the optimal number of cores and optimal schedule (variable $x(c)$) required for the workload within the area and power constraints. We obtain the optimal schedule from variable $y_{(v,t)}$ of each operator v.

4.5 Architecture Configuration Pruner

For each configuration the dimension generator generates, heuristics or ILP are executed to search for the number of cores. Naively exploring every configuration is time-consuming, so WHAM employs a novel pruner to reduce the number of core dimensions explored. The design space is represented as a binary tree, with the largest dimension at the top level and the next level nodes representing dimensions reduced by the step size. The pruner runs for each core type while keeping the other core's configuration constant. Figure 6 shows this tree design space with top-level design point as $< 256 \times 256 >$ and step-size of power of 2 for the tensor core, and fixed vector core width. The pruner uses a breadth-first algorithmic technique that prunes an entire subtree if one child configuration is better than the parent and the other is worse. To avoid selecting a local minimum, a hysteresis level is applied only when all direct child core dimensions perform worse than the parent node. In this case, the children are evaluated for multiple sub-levels, and if all these dimensions are worse than the original parent, the entire subtree is pruned. This detailed pruning algorithm is shown in Algorithm 2.

Algorithm 2: Configuration Pruner for Tensor Core

```
Input: MaxTCDim          // TC dimension range
Input: VCWidth           // Vector core width for this pruning
Input: StepSize          // Step size to decrement the dimensions
Input: HysLevels         // Hysteresis level as input for the pruning
TCDimsToExplore.append(MaxTCDim)   // Starting Config
MinRuntime =
  CriticalPathSearch(CurrentTCDim, VCWidth)
while TCDimsToExplore ≠ ∅ do
    CurrentTCDim = TCDimsToExplore.pop()
    // Generates configs for the next level and ignores duplicates.
    NewTCDims =
      GenerateNextDim(CurrentTCDim, StepSize)
    Runtimes =
      CriticalPathSearch(NewTCDims, VCWidth);
    if min(Runtimes) < MinRuntime then
        MinRuntime = min(Runtimes)
        TCDimsToExplore.append(GetBetterConfigs(NewTCDims))
    else if Check(HysLevels) then
        TCDimsToExplore.append(NewTCDims)
```

This technique is based on the insight that if a smaller core dimension does not offer a better training metric than the parent node, there is either insufficient parallelism in the model's operator graph to exploit or the model tensor shapes are not aligned with the architecture configuration. In either case, smaller configurations are not beneficial and are pruned. The core dimension generator stops generating new configurations when it reaches leaf configurations or the entire subtree is pruned. Ultimately, WHAM's search selects

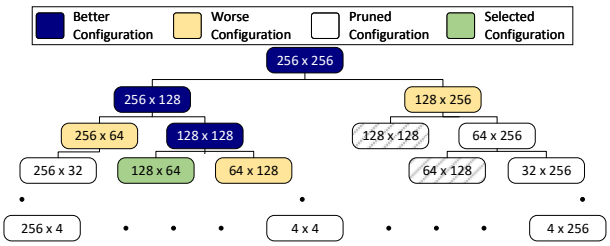

Figure 6. Architectural configuration pruning, each evaluated core dimension executes the heuristics/ILP to determine the # of cores.

the best architecture based on the training metric from all explored configurations.

Search Space Comparisons. WHAM efficiently explores a pruned set of core dimensions, searching for the number of cores and scheduling space for a given operator graph. Existing toolchains are used for per-operator dataflow exploration [27, 28]. Table 3 compares the search space explored with and without WHAM's pruner for both heuristics and ILP, excluding per-operator dataflow search complexity. The table also compares against unconstrained exhaustive search, where neither the pruner nor the critical path-based algorithmic approach is employed.

Table 3. Search space comparisons, excluding per-operator mapping.

Model	Exhaustive	ILP		Heuristics	
		Unpruned	Pruned	Unpruned	Pruned
MobileNet_v3	10^{38}	10^{24}	10^{14}	10^{21}	10^{10}
Inception_v3	10^{39}	10^{25}	10^{14}	10^{22}	10^{12}
ResNeXt-101	10^{40}	10^{26}	10^{15}	10^{23}	10^{13}
BERT-Large	10^{40}	10^{26}	10^{16}	10^{23}	10^{13}

WHAM significantly reduces the search space compared to the exhaustive due to the critical path-based bounds, even when using optimality-guaranteeing techniques like ILP. The pruner eliminates dimensions that cannot offer higher benefits due to graph properties, resulting in the same architecture for both pruned and unpruned searches. Across various models, the pruner reduces the search space by order of 10 compared to unpruned searches, decreasing WHAM's convergence time by 65% and 70% compared to unpruned heuristics and ILP, respectively. Convergence time and quality of results are discussed in detail in Section 6.

4.6 WHAM-Common

WHAM search conducts architecture optimization for each DNN. When optimizing for a set of workloads, the pruner tracks a weighted average of the metric of interest. The allows for homogeneity, targeting ASICs based on common compute, data flow, and memory requirements of the workloads. In our evaluation, equal weight is assigned to each workload.

5 Global Search for Accelerators

The Global Architecture Search module performs an optimization to determine the architecture for a set or single workload across the stages of distributed training.

Partitioning the model. Operator graph is partitioned using existing device placement techniques [34, 36, 37]. This work specifically handles pipeline and model parallel split which impact the operator graph that executes on each device. Data parallel is a replicated pipeline and hosts the same graph across. As a proof of concept, WHAM includes a memory-balanced splitter that partitions the graph based on HBM capacity and memory requirements of training. Based on user inputs such as pipeline training scheme (e.g., Gpipe or Pipedream), pipeline depth, and batch size, along with model properties such as parameter and activation size, the splitter determines the memory footprint of training and partitions the model. For model parallel, the per stage operator graphs are based on Megatron style splits per device. The tensor model parallel width is given as an input to the WHAM search.

Networking. Pipeline parallel training only requires activations to be transferred from one device to the next, and WHAM accounts for the latency of data transfers across neighboring accelerators via interconnects defined in the system configuration. Model parallel requires collective operators such as allreduce in forward and backward pass to collect the intermediate results. Similar to prior works [34, 37], we assume a homogeneous network where all devices communicate with each other. Hierarchical or multi-level network topologies are not considered within the scope of this work.

5.1 Configuration Search using Top-k Designs

This search obtains *top-k* designs for each partition of the operator graph per device using the search described in Section 4. Selecting k instead of a 1 design per stage as the top design for a particular stage in the pipeline may not necessarily yield a balanced pipeline, which is crucial for achieving high throughput and utilization of the entire system. The search for architectures for distributed training presents a challenge in that each model has multiple designs to select from, resulting in $k \times s$ architectures for an s pipeline depth execution. When optimizing for a set of workloads, the number of architectures grows to $k \times s \times m$, where m is the number of models. Evaluating every possible configuration and selecting the best across models would be time-consuming. To address this challenge, the global module employs a top-level pruning policy similar to Configuration Pruner in Section 4.5.

This top-level pruner takes unique configurations from the $k \times s \times m$ designs to construct a search tree. Each configuration comprises both the dimensionality and number of cores. Each level in the tree contains designs of the same area, with root node as the the smallest design. Evaluating smaller to larger architectures ensures that larger configurations in the lower levels of the tree that consume more energy but

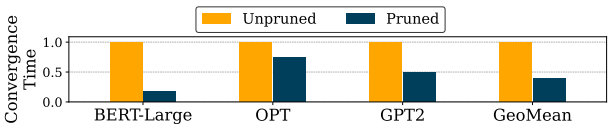

Figure 7. Convergence time comparison to search for a global design in pipeline parallel training with a pipeline depth of 32 and $k = 10$.

do not offer better performance can be pruned. The pruner eliminates sub-trees if a larger configuration is worse in metric across all the models. Alternatively, if all children are worse than the direct parent, subtrees are pruned once the evaluation reaches the hysteresis level and all evaluated child configurations are worse. As illustrated in Figure 7, the pruned distributed search converges 2.5× faster than the unpruned search, which evaluates all top-k configurations across all the models.

6 Evaluation

6.1 Experimental Setup

Models: WHAM is evaluated across a diverse set of workloads, such as Vision [47–51], Translation [52], and Language Modeling [7–9]. Table 4 shows the details of models and their training configurations. Additionally, we evaluated the performance of WHAM for distributed training of language models.

Software Implementations: We use PyTorch-1.9 [53] to obtain the operator graphs [54]. We use readily available training scripts such as torch-vision for image classification, GNMT [52] from NVIDIA [55], and language models from huggingface [56]. WHAM is executed on an 8-core Intel Xeon E5-2673 CPU with Haswell architecture and 28 GB DDR4 main memory. The ILP is solved using Gurobi [57]).

Performance Metric: WHAM optimizes for relevant training metrics, such as throughput or energy efficiency. For efficiency, as established by prior works, we use the correlated Perf/TDP for efficiency [15, 58] due to the proprietary nature of TCO. With throughput as the metric, WHAM designs maximize end-to-end throughput within power and area constraints. With Perf/TDP, WHAM designs maximize Perf/TDP while maintaining a user-specified minimum end-to-end throughput.

6.2 Baselines

We compare WHAM against two types of baselines: prior search frameworks ConfuciuX [17] and Spotlight [19] and established hardware architectures for deep learning. This allows the evaluation to establish both the efficacy of the search technique and the generated accelerators. All baselines assume an HBM of 16 GB [26] and a bandwidth of 900 GB/s [41].

Table 4. DNN models and their training configurations.

Task	Model	Model Parameters	Hyper Parameters	Number of Accelerators
Image Classification	MobileNet_v3 [25]	24 M	batch size: 128	1
	ResNet-18 [49]	30 M	batch size: 128	1
	Inception_v3 [47]	43 M	batch size: 64	1
	ResNeXt-101 [50]	87 M	batch size: 16	1
	VGG-16 [51]	141 M	batch size: 64	1
Translation	GNMT-4 [52]	70 M	batch size:128 hidden size: 512	1
Language Modeling	BERT-Base [9]	110 M	batch size: 4 sequence length: 512	1
	BERT-Large [9]	340 M	batch size: 8 sequence length: 128	1
			batch size: 1/32 sequence length: 512	1/32
	OPT [8]	1.3 B	batch size: 32 num layers: 24 attention heads: 32	32
	GPT2 (XL) [7]	1.5 B	batch size: 32 sequence length: 512 attention modules: 48	32
	GPT3 [59]	175 B	batch size: 4 sequence length: 2048 num layers: 96 attention heads: 96	64

Prior Frameworks. WHAM is the first framework to explore the design space for training. To compare with other approaches, we extend two state-of-the-art frameworks, ConfuciuX and Spotlight, to incorporate training. While ConfuciuX and Spotlight perform search over forward pass (inference) for GEMM and Convolution operators, using reinforcement learning and Bayesian optimization, respectively, we extend these frameworks, **ConfuciuX+** and **Spotlight+**, to support backward pass and weight update pass for all GEMM and Convolution operators. ConfuciuX+ selects the largest configuration across forward, backward, and weight update passes similar to its original version. In contrast, Spotlight+ optimizes for architecture for the backward pass and weight update pass, in addition to the forward pass. To consider the point-wise vector operations ignored by both frameworks, we use the same vector core width as suggested by the framework for the tensor core.

Comparison against hand-optimized accelerators: We assess WHAM against hand-designed accelerators, specifically TPUv2-like [26] and NVDLA-like [38] designs, including their corresponding dataflows. We use a scaled-up version of NVDLA to incorporate training. This design has one 256×256 tensor core and one 256 wide vector core (<*1, 256 × 256, 1, 256*>). TPUv2 [26] contains 2 computational units, each having tensor core with 128 × 128 systolic array and 128 wide vector core (<*2, 128 × 128, 2, 128*>).

Compiler and runtime optimizations. Both WHAM and baselines use common compilation and runtime techniques in deep learning. Op-fusion is applied when a convolution or GEMM operator is followed by an activation function [43, 44] to reduce data movement across the memory subsystem. Additionally, runtime data reuse allows in-flight and ready-to-schedule operators to share intermediate results, reducing costly round trips to HBM as data is directly consumed on the chip.

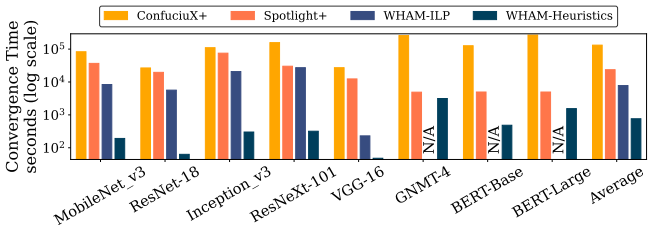

Figure 8. Convergence time comparison of WHAM with prior frameworks. N/A does not converge in 7 days.

Table 5. Per accelerator architecture comparison. WHAM architectures with Heuristics are *optimized for throughput*. Designs are represented as *< # TC, TC-DIM, # VC, VC-Width >*.

Model	ConfuciuX+	Spotlight+	WHAM		
			L2 SRAM	Individual	Common
MobileNet_v3	< 1, 256 × 256, 1, 256 > L2 SRAM : 32 MB	< 1, 12 × 512, 1, 12 >	8 MB	< 1, 256 × 128, 1, 256 >	< 1, 128 × 128, 3, 128 > L2 SRAM : 16 MB
ResNet-18		< 1, 256 × 240, 1, 256 >	18 MB	< 2, 128 × 64, 2, 128 >	
Inception_v3		< 1, 128 × 446, 1, 128 >	8 MB	< 4, 128 × 64, 4, 128 >	
ResNeXt-101		< 1, 244 × 256, 1, 244 >	6 MB	< 2, 128 × 64, 2, 128 >	
VGG-16		< 1, 128 × 264, 1, 128 >	32 MB	< 1, 256 × 128, 1, 256 >	
GNMT-4		< 1, 60 × 896, 1, 60 >	8 MB	< 3, 128 × 64, 3, 128 >	
BERT-Base		< 1, 64 × 552, 1, 64 >	8 MB	< 3, 128 × 64, 3, 128 >	
BERT-Large		< 1, 64 × 960, 1, 64 >	8 MB	< 3, 128 × 64, 3, 128 >	

6.3 Results for Individual Accelerator Search

WHAM can configure for either *throughput* or *Perf/TDP*. For each metric, it can search for a configuration specific to a single workload, WHAM-individual, or a common configuration that works for a set of workloads, WHAM-common. Larger workloads OPT, GPT2-XL and GPT3, are only evaluated for distributed training.

Convergence Time. Figure 8 compares the convergence time of WHAM ILP and heuristics against prior frameworks, ConfuciuX+ and Spotlight+. We run WHAM and prior frameworks for 500 iterations and compare their wall clock times. On average, WHAM converges 174× and 31× faster than ConfuciuX+ and Spotlight+, respectively. This is because WHAM employs a novel algorithmic technique that deliberately reduces the search space using the pruner and the critical-path-based approach. In contrast, prior approaches use reinforcement learning, genetic algorithm, and Bayesian optimization techniques and scale the problem proportional to the problem size.

For ConfuciuX+, the RL converges to a local minima relatively quickly, while the genetic algorithm takes a long time to fine-tune the minima. Spotlight+ reduces the search space by removing duplicate problem dimensions in a DNN graph and thus converges faster, especially for language models with replicated transformer layers. However, Spotlight+ does not prune the architectural search space like WHAM. We observed that the ILP in WHAM could not converge within seven days for a single iteration of architectural configuration for language and translation models due to the large size of the DNN graph.

Architecture comparison with Throughput as Metric. Table 5 shows the architectures proposed by each framework and WHAM. Tensor Core L1-reg has a size of 512 Bytes while vector core and tensor core L2-SRAM sizes are shown in the table. Figure 9 presents the throughput improvement of WHAM-individual and -common over prior frameworks and hand-optimized accelerators with *throughput* as the optimization metric. WHAM-individual is compared against ConfuciuX+ and Spotlight+ generated architectures and TPUv2 and NVDLA to WHAM-common.

On average, WHAM-individual provides 20× and 12× throughput improvement over ConfuciuX+ and Spotlight+, respectively. ConfuciuX+ and Spotlight+ generated configurations are inefficient mainly due to the large training design space. Due to their search techniques, they fail to converge on a design suitable across forward, backward and parameter update pass and mostly rely on the biggest configuration to accommodate the training complexities.

The WHAM-common design addresses the needs of all the evaluated workloads, and offers 2× and 12% higher throughput over NVDLA and TPUv2, respectively. The reason for these benefits is the improved utilization of the cores and the exploited concurrency across operators, allowing them to be scheduled in parallel over multiple cores. For example, in the BERT model, the QKV projection in each encoder layer can be executed in parallel across three tensor cores, which justifies the architectural configuration for this model. For BERT-Base and BERT-Large models, such parallelism is the source of up to 53% of performance improvement over the TPUv2 baseline. For workloads without any branching structure (MobileNet_v3, VGG-16, etc.), the main source of performance improvement is better utilization of the core. WHAM-individual, however, can offer 2× and 15% higher benefits in comparison to NVDLA and TPUv2, respectively. These configurations are specialized for a single model and employ model-specific spatial unrolling of the output and input feature interactions across tensor core dimensions.

Architecture comparison with Perf/TDP as Metric. Figure 10 shows the Perf/TDP benefits of WHAM's proposed architecture compared to TPUv2 like design. WHAM optimizes for Perf/TDP with a throughput constraint of TPUv2. While designs generated by ConfuciuX+, Spotlight+, and NVDLA are not compared in the Figure due to their focus on latency-bound inference, it is worth noting that both WHAM-common and WHAM-individual provide orders of magnitude higher Perf/TDP than all these designs. This is because these frameworks heavily optimize for tensor-core only operators and often select the largest design, leading to low utilization but high energy. In contrast, WHAM deliberately optimizes for training and considers a metric of interest. Compared to TPUv2, WHAM-common provides 19% better Perf/TDP than TPUv2, because two cores do not limit it and can exploit operator concurrency beyond two for many models. For certain

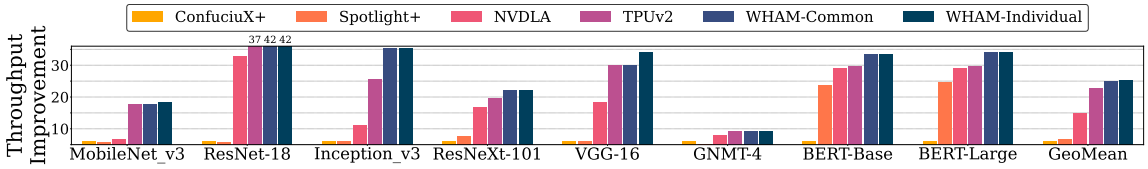

Figure 9. Comparison of WHAM generated acclerators *optimized for throughput* against hand-optimized accelerators and framework suggested designs. All results are compared to ConfuciuX+ generated design.

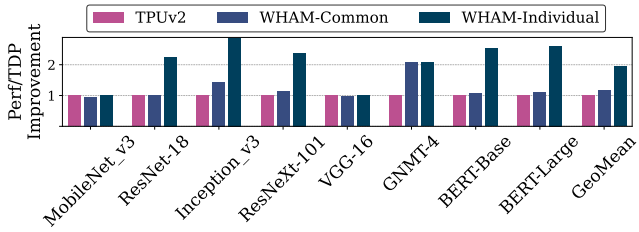

Figure 10. Comparison of WHAM generated designs *optimized for Perf/TDP* with TPUv2 as baseline.

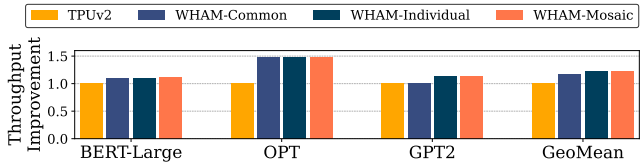

Figure 11. Throughput comparison of WHAM designs for pipeline parallel training compared to TPUv2, optimized for throughput.

models, WHAM-individual does not offer any higher benefits as models exhibit little to no branching.

6.4 Global Search for Distributed Training

This section compares the architectures generated across LLMs for pipeline and model parallel execution.

Pipeline parallel training. We compare global search results across various generated designs: WHAM-common, a common architecture across pipeline stages addressing all models, WHAM-individual is tailored to each model but homogeneous across its pipeline, and WHAM-mosaic top-1 design for each stage in the pipeline for every model, resulting in a heterogeneous pipeline. Our results are presented using a pipeline depth of 32, GPipe [10] pipeline strategy, and activation stashing.

Architecture comparison with Throughput as a Metric. Figure 11 compares the training throughput of the WHAM optimized accelerator with the TPUv2 accelerator, the best-performing baseline. On average, we observe a throughput improvement of 17%, 22%, and 23% for the Common, Individual, and Mosaic configurations, respectively, compared to TPUv2. Among the generated designs, WHAM-individual

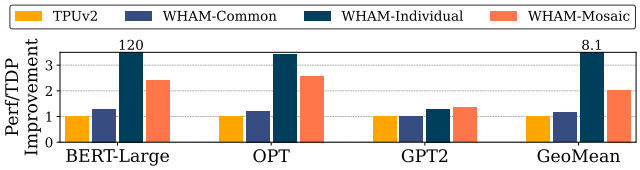

Figure 12. Perf/TDP comparison for pipeline parallel training with WHAM designs compared to TPUv2 and optimized for Perf/TDP. Normalized to TPUv2 TMP-1, PP-64.

provides the most significant benefit because it is specific to each model. Language models have repeated transformer layers, resulting in similar dataflow properties across model partitions. As a result, WHAM-individual can capture a common design across those stages to cater to those properties. This is also why WHAM-mosaic's heterogeneity only provides modest benefits over WHAM-individual.

Architecture comparison with Perf/TDP as a Metric. Figure 12 compares Perf/TDP achieved by WHAM generated designs when optimized for this metric and the minimum throughput of the TPUv2 like architecture. On average, the configurations generated by WHAM exhibit a Perf/TDP improvement of 1.6×, 8.1×, and 2.0× for the Common, Individual, and Mosaic configurations, respectively, compared to TPUv2 design. It is important to note that when optimizing for Perf/TDP, the top-1 architecture optimized for each pipeline stage, WHAM-Mosaic, may not yield a better end-to-end metric. This is because each pipeline stage chooses the best architecture for its own stage, but due to the bottleneck stage, it may not contribute to higher throughput while consuming more energy due to the larger area. In contrast, WHAM-individual considers all pipeline stages to accommodate the end-to-end metric and can generate a homogeneous architecture that provides better Perf/TDP. Its worth noting that WHAM-common must be generalized across workloads.

Model and Pipeline Parallel Training. To address training of large models, model parallelism (TMP) is used with pipeline parallelism. Although our evaluation explores architectures for Megatron-style split, we can support any TMP or pipeline parallel strategy by obtaining the corresponding operator graph that resides on each device. Figure 13 illustrates the throughput improvements achieved by WHAM configurations compared to TPUv2 like design, as TMP scales from

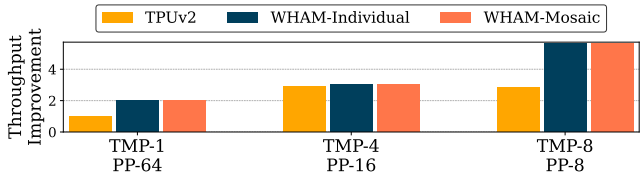

Figure 13. GPT3 throughput comparison between various tensor model (TMP) and pipeline parallel configurations, when using WHAM designs, in contrast to TPUv2. The total devices is 64.

1 to 8. The total number of devices involved in training is 64. WHAM proposed architecture provides 2× throughput improvement over TPUv2 architecture with TMP and pipeline parallelism of 8. As all the stages in GPT3 are uniform due to the model structure, WHAM individual and mosiac results are identical.

Top-k hyper-parameter search. We sweep the top-k hyper-parameter generated for each pipeline stage across three LLMs to determine the optimal value for distributed pipeline parallel training. As Figure 14 shows, naively selecting the top-1 design does not always yield the best metric, however, we observe diminishing returns as the Perf/TDP improvements saturate after $k = 10$.

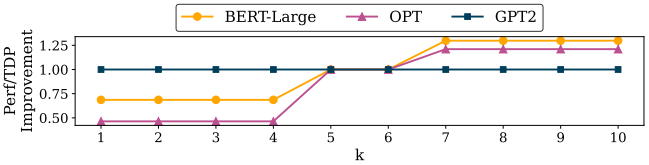

Figure 14. Top-k hyper-parameter sweep, and its impact on Perf/TDP for WHAM-Common (all results compared to TPUv2).

7 Related Work

7.1 Architectural Search Frameworks

Comprehensive architectural search techniques for deep learning are currently only focused on inference [15–19]. FAST [15] maps operators to Tensor and Vector cores for inference accelerators, using a black box optimizer [30] that generates the search hyperparameters. It aims to utilize the extra global buffer memory for subsequent operators through fusion. However, in training, intermediate activations must be stashed for the backward pass, thus the extra memory might not be available for this optimization. PRIME [16] is an offline approach devised for inference that utilizes logged simulation data, to architect hardware accelerators without requiring new simulations. PRIME creates a cost function through a surrogate model but does not address how to generalize a single cost function for the forward, backward, and parameter update passes required in training. Spotlight [19] searches the HW/SW co-design space by injecting hand-provided domain information formulated as a Bayesian optimization. Spotlight optimizes for layer-wise tensor core cost estimation targeting inference. ConfuciuX [17] targets only inference and employs reinforcement learning and genetic algorithms to determine the number of PEs and local buffer size without considering vector operations. It optimizes per layer and selects the largest design across layers for end-to-end inference execution.

Other works in this area [4, 6, 18, 29, 32] provide HW/SW solutions for dense tensor computation. Apollo [31] uses transfer learning, FlexiBO [60] and HASCO [18] apply Bayesian optimization, HyperMapper [61] employs random forests, and others utilize genetic algorithms [62] for design space exploration. dMazeRunner [63] and ZigZag [64] focus mainly on large software design spaces. Design Space Exploration for recommendation models training [65–68] involves embedding exploration while federated learning [69] also benefits from DSE on each device.

In contrast, WHAM prioritizes training, which necessitates hardware accelerator optimized across forward, backward, and weight update passes. Furthermore, none of the mentioned works address distributed execution, whereas WHAM performs architecture search for pipeline parallel training.

7.2 Mapping Frameworks

Various mapping search frameworks [27, 70–72] determine data movement and compute placement across a design for a fixed architecture. Marvel [70] optimizes dataflow for an architecture by reducing off-chip movements. Timeloop [27] employs random pruning to find the mapping for single operations (GEMM or CONV). MindMapping [71] is a gradient-based search method for mapping search exploration. GAMMA [72] uses a genetic algorithm to develop an optimized mapping for a given layer.

In contrast, WHAM aims to identify the architecture design while considering dataflow. To optimize the architecture for training, WHAM utilizes existing open-source dataflow mapping search techniques for deep learning operations. It can integrate with any dataflow search framework for its architecture search.

8 Conclusion

WHAM is the first work to perform architecture search for hardware accelerators in a pipeline and model parallel setting. This is an important problem as models are becoming larger and require multiple accelerators. WHAM solves this problem via a multi-step approach. The distributed search performs multiple isolated searches for an accelerator executing the model partition. Each individual accelerator, the search takes a critical-path based algorithmic approach to determine the number of cores and buffers for each type. WHAM then obtains the top-k designs for each accelerator and combines the results to determine the architectural configuration for a distributed setting. This enables WHAM to scale for training and search for accelerators across across a wide range of DNN workloads.

References

[1] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings, 2023.

[2] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[3] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, , Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38:8–20, March 2018.

[4] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kim, and Hadi Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. March 2016.

[5] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. Scale-out acceleration for machine learnng. October 2017.

[6] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From High-Level Deep Neural Models to FPGAs. October 2016.

[7] Language Models are Unsupervised Multitask Learners. 2019. URL https://openai.com/blog/better-language-models/.

[8] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language msodels. *arXiv preprint arXiv:2205.01068*, 2022.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019.

[10] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[11] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15. ACM, 2019.

[12] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel DNN training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[13] R. E. Kessler. The Alpha 21264 Microprocessor. *MICRO*, 1999.

[14] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *ArXiv*, abs/1909.08053, 2019.

[15] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. *A Full-Stack Search Technique for Domain Optimized Deep Learning Accelerators*, page 27–42. Association for Computing Machinery, New York, NY, USA, 2022. ISBN 9781450392051. URL https://doi.org/10.1145/3503222.3507767.

[16] Aviral Kumar, Amir Yazdanbakhsh, Milad Hashemi, Kevin Swersky, and Sergey Levine. Data-driven offline optimization for architecting hardware accelerators. *arXiv preprint arXiv:2110.11346*, 2021.

[17] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. ConfuciuX: Autonomous Hardware Resource Assignment for DNN Accelerators using Reinforcement Learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 622–636, 2020.

[18] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. Hasco: Towards agile hardware and software co-design for tensor computation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1055–1068. IEEE, 2021.

[19] Chirag Sakhuja, Zhan Shi, and Calvin Lin. Leveraging domain information for the efficient automated design of deep learning accelerators. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 287–301, 2023. doi: 10.1109/HPCA56546.2023.10071095.

[20] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems*, 3: 269–296, 2021.

[21] Nvidia. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/.

[22] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pages 262–263, 2016.

[23] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steve Reinhardt, Adrian Caulfield, Eric Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. ACM, June 2018. URL https://www.microsoft.com/en-us/research/publication/a-configurable-cloud-scale-dnn-processor-for-real-time-ai/.

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *ArXiv*, abs/1704.04861, 2017.

[26] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. The design process for google's training chips: Tpuv2 and tpuv3. *IEEE Micro*, 41(2):56–63, 2021. doi: 10.1109/MM.2021.3058217.

[27] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A

Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019.

[28] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 754–768, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 369–383, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378514. URL https://doi.org/10.1145/3373376.3378514.

[30] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and David Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1487–1495, 2017.

[31] Amir Yazdanbakhsh, Christof Angermueller, Berkin Akin, Yanqi Zhou, Albin Jones, Milad Hashemi, Kevin Swersky, Satrajit Chatterjee, Ravi Narayanaswami, and James Laudon. Apollo: Transferable Architecture Exploration, 2021.

[32] Christof Angermueller, David Dohan, David Belanger, Ramya Deshpande, Kevin Murphy, and Lucy Colwell. Model-based reinforcement learning for biological sequence design. 2020.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM*, 60(6):84–90, May 2017.

[34] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *SysML 2019*, 2019.

[35] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 2430–2439. JMLR.org, 2017.

[36] Jakub M Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of dnn graph operators. *Advances in Neural Information Processing Systems*, 33, 2020.

[37] Jakub Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional Planner for DNN Parallelization. In *NeurIPS 2021*, December 2021. URL https://www.microsoft.com/en-us/research/publication/piper-multidimensional-planner-for-dnn-parallelization/.

[38] Gaofeng Zhou, Jianyang Zhou, and Haijun Lin. Research on NVIDIA Deep Learning Accelerator. In *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, pages 192–195, 2018.

[39] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.

[40] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 14–27, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Google Cloud Documentation. https://cloud.google.com/tpu/docs/system-architecture.

[42] Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. Procrustes: a dataflow and accelerator for sparse deep neural network training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 711–724. IEEE, 2020.

[43] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL https://www.usenix.org/conference/osdi18/presentation/chen.

[44] Sean Kinzer, Joon Kyung Kim, Soroush Ghodrati, Brahmendra Yatham, Alric Althoff, Divya Mahajan, Sorin Lerner, and Hadi Esmaeilzadeh. A computational stack for cross-domain acceleration. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 54–70, 2021. doi: 10.1109/HPCA51647.2021.00015.

[45] Joon Kyung Kim, Byung Hoon Ahn, Sean Kinzer, Soroush Ghodrati, Rohan Mahapatra, Brahmendra Yatham, Shu-Ting Wang, Dohee Kim, Parisa Sarikhani, Babak Mahmoudi, Divya Mahajan, Jongse Park, and Hadi Esmaeilzadeh. Yin-yang: Programming abstractions for cross-domain multi-acceleration. *IEEE Micro*, 42(5):89–98, 2022. doi: 10.1109/MM.2022.3189416.

[46] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[47] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision, 2015.

[48] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3, 2019. URL https://arxiv.org/abs/1905.02244.

[49] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, 2015.

[50] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks, 2016. URL https://arxiv.org/abs/1611.05431.

[51] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, 2014.

[52] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, 2016.

[53] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

[54] Sergey Zagoruyko, Victor Quach, and Will Price. Torchviz. https://github.com/szagoruyko/pytorchviz, 2020.

[55] Gnmt v2 for pytorch. URL https://catalog.ngc.nvidia.com/orgs/nvidia/resources/gnmt_v2_for_pytorch/setup.

[56] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi

Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. HuggingFace's Transformers: State-of-the-art Natural Language Processing, 2020.

[57] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021. URL https://www.gurobi.com.

[58] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2021.

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

[60] Md Shahriar Iqbal, Jianhai Su, Lars Kotthoff, and Pooyan Jamshidi. FlexiBO: Cost-Aware Multi-Objective Optimization of Deep Neural Networks, 2020.

[61] Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. HyperMapper: a Practical Design Space Exploration Framework. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 425–426, 2019.

[62] Yanan Sun, Bing Xue, Mengjie Zhang, Gary G. Yen, and Jiancheng Lv. Automatically Designing CNN Architectures Using the Genetic Algorithm for Image Classification. *IEEE Transactions on Cybernetics*, 50(9):3840–3854, 2020.

[63] Shail Dave, Aviral Shrivastava, Youngbin Kim, Sasikanth Avancha, and Kyoungwoo Lee. dmazerunner: Optimizing convolutions on dataflow accelerators. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1544–1548, 2020. doi: 10.1109/ICASSP40776.2020.9054275.

[64] Linyan Mei, Pouya Houshmand, Vikram Jain, Sebastian Giraldo, and Marian Verhelst. Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators. *IEEE Transactions on Computers*, 70(8):1160–1174, 2021. doi: 10.1109/TC.2021.3059962.

[65] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. Accelerating recommendation system training by leveraging popular choices. *Proc. VLDB Endow.*, 15(1):127–140, sep 2021. ISSN 2150-8097.

[66] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. Heterogeneous acceleration pipeline for recommendation system training. In *Proceedings of the 51st International Symposium on Computer Architecture (ISCA)*. ACM, 2024.

[67] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. Ad-rec: Advanced feature interactions to address covariate-shifts in recommendation networks. *arXiv preprint arXiv:2308.14902*, 2023.

[68] Yassaman Ebrahimzadeh Maboud, Muhammad Adnan, Divya Mahajan, and Prashant J. Nair. Accelerating recommender model training by dynamically skipping stale embeddings, 2024.

[69] Irene Wang, Prashant J. Nair, and Divya Mahajan. FLuID: Mitigating stragglers in federated learning using invariant dropout. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[70] Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. Marvel: A data-centric compiler for dnn operators on spatial accelerators. *arXiv preprint arXiv:2002.07752*, 2020.

[71] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W. Fletcher. Mind mappings: Enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 943–958, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446762. URL https://doi.org/10.1145/3445814.3446762.

[72] Sheng-Chun Kao and Tushar Krishna. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.