DLAS: A Conceptual Model for Across-Stack Deep Learning Acceleration

PERRY GIBSON, School of Computing Science, University of Glasgow, Glasgow, United Kingdom of Great Britain and Northern Ireland
JOSE CANO, School of Computing Science, University of Glasgow, Glasgow, United Kingdom of Great Britain and Northern Ireland and The University of Edinburgh, Edinburgh, United Kingdom of Great Britain and Northern Ireland
ELLIOT CROWLEY, The University of Edinburgh, Edinburgh, United Kingdom of Great Britain and Northern Ireland
AMOS STORKEY, The University of Edinburgh, Edinburgh, United Kingdom of Great Britain and Northern Ireland
MICHAEL O'BOYLE, The University of Edinburgh, Edinburgh, United Kingdom of Great Britain and Northern Ireland

Deep Neural Networks (DNNs) are very computationally demanding, which presents a significant barrier to their deployment, especially on resource-constrained devices. Significant work from both the machine learning and computing systems communities has attempted to accelerate DNNs. However, the number of techniques available and the required domain knowledge for their exploration continue to grow, making design space exploration (DSE) increasingly difficult. To unify the perspectives from these two communities, this article introduces the Deep Learning Acceleration Stack (DLAS), a conceptual model for DNN deployment and acceleration. We adopt a six-layer representation that organizes and illustrates the key areas for DNN acceleration, from machine learning to software and computer architecture. We argue that the DLAS model balances simplicity and expressiveness, assisting practitioners from various domains in tackling co-design acceleration challenges. We demonstrate the interdependence of the DLAS layers, and thus the need for co-design, through an across-stack perturbation study, using a modified tensor compiler to generate experiments for combinations of a few parameters across the DLAS layers. Our perturbation study assesses the impact on inference time and accuracy when varying DLAS parameters across two datasets, seven popular DNN architectures, four compression techniques, three algorithmic primitives (with sparse and dense variants), untuned and auto-scheduled code generation, and four hardware platforms. The study observes significant changes in the relative performance of design choices with the introduction of new DLAS parameters (e.g., the fastest algorithmic primitive varies with the level of quantization). Given the strong evidence for the

This work was partially supported by the EU Project dAIEDGE (Grant Agreement Number 101120726) and the Innovate UK Horizon Europe Guarantee (Grant Agreement Number 10090788).
Authors' Contact Information: Perry Gibson, School of Computing Science, University of Glasgow, Glasgow, Scotland, United Kingdom of Great Britain and Northern Ireland; e-mail: p.gibson.2@research.gla.ac.uk; Jose Cano, School of Computing Science, University of Glasgow, Glasgow, United Kingdom of Great Britain and Northern Ireland and The University of Edinburgh, Edinburgh, United Kingdom of Great Britain and Northern Ireland; e-mail: Jose.CanoReyes@glasgow.ac.uk; Elliot Crowley, The University of Edinburgh, Edinburgh, United Kingdom of Great Britain and Northern Ireland; e-mail: elliot.j.crowley@ed.ac.uk; Amos Storkey, The University of Edinburgh, Edinburgh, United Kingdom of Great Britain and Northern Ireland; e-mail: a.storkey@ed.ac.uk; Michael O'Boyle, The University of Edinburgh, Edinburgh, United Kingdom of Great Britain and Northern Ireland; e-mail: mob@inf.ed.ac.uk.

ACM 1544-3973/2025/03-ART1
https://doi.org/10.1145/3688609

need for co-design, and the high costs of DSE, DLAS offers a valuable conceptual model for better exploring advanced co-designed accelerated deep learning solutions.

CCS Concepts: • **Computing methodologies** → **Machine learning**; **Artificial intelligence**; • **Software and its engineering** → *Compilers*; • **General and reference** → **Empirical studies;**

Additional Key Words and Phrases: DNNs, Convolutional Neural Networks, Pruning, Quantization, Sparse Tensors, Auto-Scheduling, Auto-Tuning, DNNs, Tensor Compilers, Tensor Programs

ACM Reference Format:
Perry Gibson, Jose Cano, Elliot Crowley, Amos Storkey, and Michael O'Boyle. 2025. DLAS: A Conceptual Model for Across-Stack Deep Learning Acceleration. *ACM Trans. Arch. Code Optim.* 22, 1, Article 1 (March 2025), 28 pages. https://doi.org/10.1145/3688609

1 Introduction

Recent years have yielded rapid advances in deep learning, largely due to the unparalleled effectiveness of **Deep Neural Networks (DNNs)**, such as **Convolutional Neural Networks (CNNs)** and Transformer architectures [81], on a variety of difficult problems [42, 49, 87]. Despite increases in algorithmic efficiency [35], the trend with DNN architectures is increased size and deployment costs as demands for more powerful and general solutions grows [7, 77]. As such, creative approaches are required to deploy DNNs on hardware with limited resources to enable a variety of emerging applications (e.g., autonomous driving [75], collision avoidance for quadcopters [2]). However, often these optimization approaches come with limited benchmarks and few comparisons, and there may be a disconnect between machine learning-based and systems-based optimizations due to disparate communities with varying core competencies.

Even in a simplified view of the relevant components of deep learning deployment (e.g., machine learning, software, and hardware), it is evident that choices in one area can have consequences for the choices in others [79]. For example, constrained CPU hardware will require appropriately resource-conservative software and DNN models that fit limited memory and latency budgets. Alternatively, a new DNN architecture with a novel operation will need an optimized software kernel to execute it on hardware that can provide the required inference time. Without broad awareness of these interactions from practitioners, potential performance may be lost, since novel machine learning techniques may not be fully exploited by systems techniques [4, 28], or machine learning practitioners may not be aware of under-utilized resources available on their hardware platforms.

In this work, we outline a high-level "stack-based" conceptual model of the relevant techniques pertaining to DNN acceleration and demonstrate how they are linked with a multi-level experimental analysis. Our goal is to enable a more comprehensive understanding of the performance available under different constraints of inference accuracy, execution time, memory space, and energy consumption. We introduce the ***Deep Learning Acceleration Stack* (DLAS)**, which provides a model for both machine learning and systems researchers to reason about the impact of their performance optimizations. We propose DLAS as six layers,[1] as shown in Figure 1, covering parameters more relevant to machine learning (Datasets & Problem Spaces, Models & Neural Architectures, and Model Optimizations) and parameters more relevant to systems (Algorithms & Data Formats, Systems Software, and Hardware). Each of the layers can be further decomposed into sub-layers, however, the intent of DLAS is to provide a starting point for reasoning about across-stack optimization and encourage co-design of accelerated DNN deployments. The core contributions of this work include:

[1]These layers are not to be confused with the computational layers that make up a neural network.

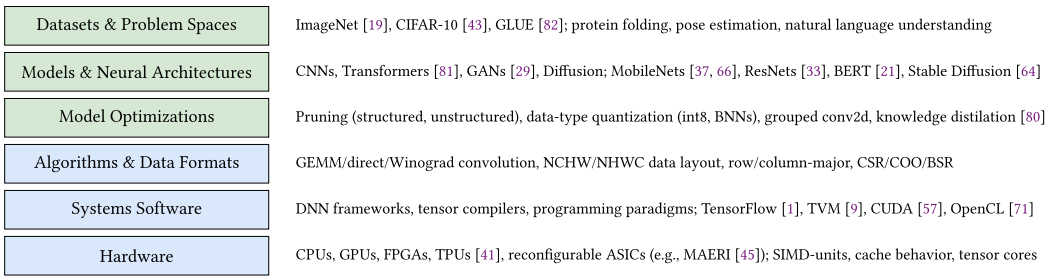


Fig. 1. Overview of DLAS, split between machine learning and systems techniques, with examples.

— We introduce the *Deep Learning Acceleration Stack* to reflect the different layers of optimization, from both machine learning and systems, that can be applied to run a DNN model more efficiently on a given target device.
— We select parameters to vary at each layer of DLAS (two datasets, four models, three compression techniques, three algorithms, two compilation techniques, four hardware devices), which gives us a vertical slice of DLAS to explore, presenting results on the inference time and accuracy impacts of our variations.
— We develop an experimental framework based on Apache TVM [9] to evaluate our parameters in a consistent environment, extending TVM where required and possible.
— We explore across-stack interactions and make 13 key observations from our results, discussing their consequences and highlighting potential improvements that could be made from other works.

The rest of the article is organized as follows: In Section 2, we motivate and describe DLAS. Section 3 gives the necessary background to understand the optimization techniques we explore across DLAS. For our evaluation, we describe the experimental setup in Section 4, and the results are presented in Section 5. In Section 6, we discuss the results of our evaluation as well as related work that could be exploited in further exploration.

2 Deep Learning Acceleration Stack

2.1 Motivation

The recent growth of deep learning has been partially facilitated by the computational power of high-end GPUs as well as improvements in algorithmic representations [35]. When combined with a tendency to focus narrowly on higher accuracies and the availability of large server-class GPUs, this has led to state-of-the-art DNN models to explode in size [60]. This presents a large barrier to deploying many modern machine learning applications on constrained devices.

Both machine learning researchers and systems engineers have proposed innovative solutions to overcome this barrier. However, these solutions are typically developed in isolation, meaning that machine learning practitioners may not explore the systems consequences of their approach and vice versa. For instance, sparsity is regarded by some in the machine learning community as a silver bullet for compressing models, whereas exploiting parallelism is generally seen as essential by system architects. Challenging these isolated preconceptions reveals that sparsity does not always excel at reducing the number of operations during inference, and parallelism does not necessarily bring the expected speedups. These observations are presented in greater detail in Section 6.

The goal of introducing DLAS as a conceptual model is to make it clearer to both machine learning and systems practitioners what the relevant contributors to performance for their DNN workloads are, allowing greater opportunities for co-design and co-optimization. This is not to

advocate for machine learning experts to re-train as systems experts and vice versa. Rather, we aim to provide a framework of reasoning so practitioners can understand the context in which their area of expertise exists in and give a "checklist" of other relevant performance contributing factors to be aware of. By exposing the wide range of choices and highlighting the impact of across-stack interaction, we also hope to encourage better tooling so practitioners can more easily experiment with perturbations.

DLAS is an instantiation of the "systems stack," whose layers have been chosen to highlight the most relevant components for deep learning acceleration. These goals are similar to other conceptual models such as the OSI model [17] and the LAMP stack for web applications, which present an abstraction that organizes and illustrates the critical areas of each system while allowing choice of concrete implementation.

Practically, by using DLAS as a conceptual model, multi-disciplinary teams may be able to align on a common language for accelerating their workloads, especially in constrained environments such as the edge. DLAS could help practitioners communicate how their techniques might interact with others and also act as a quick reference for new techniques that could be included in a given deployment to further accelerate their workloads, especially when a given acceleration strategy is giving diminishing returns.

2.2 Description of the Stack

We introduce the ***Deep Learning Acceleration Stack* (DLAS)**, which spans from the machine learning domain all the way down to the hardware domain. Each layer can be tuned to optimize different goals (i.e., inference accuracy, execution time, memory footprint, power) or to yield further improvements in adjacent layers. However, for their potential to be fully realized, many optimizations are required to be implemented using techniques across several layers, i.e., co-design and co-optimization. DLAS contains the following six layers, with examples given in Figure 1:

(1) *Datasets & Problem Spaces*: the top-level of the stack defines the problem and/or environment that the machine learning solution must solve.
(2) *Models & Neural Architectures*: DNN models and families of architectures, as well as their training techniques.
(3) *Model Optimizations*: approaches to reduce the size and costs of a DNN model (e.g., memory, inference time), while attempting to maintain the accuracy.
(4) *Algorithms & Data Formats*: DNN layers (e.g., convolutions) can be implemented using various algorithms, with myriad tradeoffs in space and time. Interlinked with algorithms are data formats, i.e., how data is laid out in memory. These choices can be consistent across a DNN model or vary per layer.
(5) *Systems Software*: software used to run the DNNs, such as DNN frameworks, algorithmic implementations, supporting infrastructure, tensor compilers, and code-generators.
(6) *Hardware*: devices the DNN is deployed on, from general purpose (e.g., CPUs, GPUs) to application-specific (e.g., FPGAs, NPUs, TPUs). It also includes hardware features, such as SIMD-units, cache behavior, and tensor cores.

Although we have delineated the layers of the stack, it is critical to highlight that design decisions made at each layer of DLAS can directly impact adjacent layers and those across the stack. In addition, a domain expert may divide a given layer into more detailed sub-layers, and increased co-design may blur the separation between layers. However, we believe this six-layer structure strikes a balance between descriptiveness and simplicity. The six layers of DLAS should be understandable by experts at opposite ends of the stack, e.g., machine learning experts can understand

that hardware choices can be important but may not want to reason about the tradeoffs of different cache policies or ISA extensions.

In this article, we perform an across-stack perturbation of some parameters from each layer to determine their impact on inference and accuracy performance, and any interactions between parameters. In the future, practitioners will increasingly need to be aware of these across-stack interactions, as Moore's law scaling can no longer be relied upon by machine learning engineers [34], and increased competition between hardware designers will require progressively more innovative workload-aware approaches.

2.3 Case Study

In the remainder of this article, we demonstrate the value of DLAS with a case study, choosing a small subset of popular parameters at each layer and showing how they can influence each other. Note that even examining a small number of parameters can result in a large number of experiment variants, due to the combinatorial growth for each new parameter added—there are nearly 1,000 combinations in our study alone. Considering a wider range of parameters poses significant research challenges regarding efficient **design space exploration (DSE)**, which is a broader problem the community continues to tackle from a number of directions [3, 6, 27, 78].

We implement our case study using PyTorch and a modified version of Apache TVM. However, DLAS evaluations can be performed with any combination of software frameworks and techniques. Our case study highlights how an across-stack DLAS evaluation can be conducted with a narrow but deep investigation. We encourage readers to consider how the results could change if additional parameters were included, for example, Winograd convolution [48], Transformers [81], TPUs [41], or some other acceleration technique of interest. However, including these additional parameters will not change the core purpose of the case study, namely, to highlight that DLAS can be a useful delineation for DNN acceleration research, and across-stack thinking will be increasingly important to unlock the next generation of acceleration techniques. Section 3 gives the necessary background to understand the details of our case study, Section 4 describes the experimental setup, and Sections 5 and 6 provide the results and discussion, respectively.

3 Background

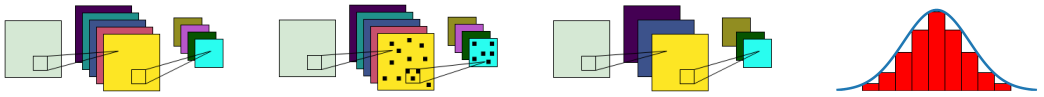
In this section, we discuss the necessary background of the techniques in DLAS explored in our experiments. Note that providing a complete background of every technique in DLAS is beyond the scope of this article, as deep learning innovations are rapid and continuous, making any attempt at comprehensiveness quickly outdated.

3.1 Datasets & Problem Spaces

We focus on two common image classification datasets, CIFAR-10 [43] and ImageNet [19]. CIFAR-10 images are 32×32 pixels across 10 classes, while ImageNet images are 224×224 pixels across 1,000 classes.

3.2 Models & Neural Architectures

There is a wide range of DNN architectures available, with a variety of popular models for each. **Convolutional neural networks (CNNs)** are commonly leveraged for image classification tasks and are characterized by their use of convolutional layers. Other layers may include batch normalization and pooling layers, fully connected layers, and activation functions such as ReLU. The topology of these networks are generally deterministic, directed acyclic graphs. Some neural architectures may include skip-connections [33], where activations from previous layers are reused



(a) Dense Convolutions (b) Unstructured Pruning (c) Structured Pruning (d) Data-type Quantization

Fig. 2. A visual representation of different Model Optimization techniques: (a) shows a slice of a typical DNN. Input on the left is computed with filters in the center to produce output on the right; (b) shows the same network slice with weight pruning applied. A subset of parameters in the filters is forced to zero (visually represented with black holes) producing sparse matrices; (c) shows the network slice with channel pruning applied onto the filters, where there are fewer channels; (d) shows data-type quantization, where the range of values that a given parameter of a DNN has been reduced.

in later layers, or depthwise separable convolutions [68], which can reduce memory and computational requirements.

When training DNNs, we repeatedly present the full dataset (each round called an epoch) and assess ultimate performance against a separate test set. Algorithms like **stochastic gradient descent (SGD)** iteratively update the DNN weights to enhance accuracy between epochs. Hyperparameters like the **learning rate (LR)** influence weight adjustments each epoch, usually reducing as the network learns.

3.3 Model Optimizations

It has been observed that DNN models are overparameterized, and similar accuracies can be achieved with smaller models [24]. As a result, a wide range of model optimization and compression techniques has been proposed. Figure 2 shows some common compression techniques, with an uncompressed (or "dense") convolution shown in Figure 2(a).

Pruning is a family of techniques that sets weights/parameters in a DNN to zero. This can reduce the computational or memory overheads of a given DNN model, with potential accuracy loss. There are two general types of parameter pruning: unstructured and structured, each of which can be used in either global or layer-wise configurations [5]. Unstructured pruning removes individual parameters, as seen in Figure 2(b), whereas structured pruning removes whole groups of parameters, such as blocks, or channels, as seen in Figure 2(c). With global pruning, given some pruning target (e.g., 50% of parameters should be pruned), the pruning algorithm will find the best parameters to prune across the whole DNN model, meaning that some layers will be more or less pruned than others. For layer-wise pruning, we prune by a pre-defined amount per layer, e.g., 50% pruning. All types of pruning apply some *scoring* function to the weights to determine which are the least important and therefore are likely to have the lowest impact on accuracy if removed. A popular approach is "L1-pruning," where we prune parameters that have the lowest absolute value, i.e., closest to zero. Other ranking approaches include gradient-based methods and Taylor series expansion.

Another compression technique is *data-type quantization*, which reduces the number of bits used to represent data, visualized in Figure 2(d), where a continuous function is approximated with nine distinct values. Typically, DNNs are trained using `float32`, however, when they are deployed, we can reduce the precision. Common quantized data-types include `float16` and `int8`, as well as emerging machine learning specific types such as `bfloat16`. For some types of data-type quantization, it may be necessary to add additional operations to the DNN, e.g., in many `int8` DNNs, we need to store the partial-sums of **MACs (multiply-accumulate operations)** using up to 16 bits, and then quantize back into `int8`. Other layers, such as layer norm and softmax, may also require higher precision.

Note that for many model optimization techniques, including pruning and data-type quantization, it may be necessary to use some form of post-training fine-tuning or calibration to try and recover some of the lost accuracy. This can involve retraining the non-pruned parameters of the model in the case of pruning, or adjusting constants used in the rescaling operations in the case of data-type quantization, or some combination of the two.

From a systems perspective, many model optimization techniques do not necessarily provide speedups unless lower levels of the stack adequately support it; for example, hardware that can compute using data-types with fewer bits or algorithms that exploit the pruning to skip operations.

3.4 Algorithms & Data Formats

The layers of a given DNN model can be implemented in a variety of ways, as long as they still provide the same output. However, the behavior of a given implementation is influenced by the size and shape of the data, the properties of the hardware we are running on, as well as the choices around how we exploit model optimization techniques.

For the CNN models we evaluate in our work, the most important component to optimize is the convolutional layers, since they are generally the most compute- and memory-intensive. Algorithms used in this work implementing convolutional layers include *direct*, *GEMM*, and *spatial pack* convolution. *Direct* convolution applies the convolution in a manner similar to the textbook definition "sliding-window" and does not reshape input data and weights. *GEMM* convolution reshapes input data into a 2D array, potentially replicating elements using a reshaping algorithm known as "im2col." This means that the convolution can be computed as a matrix multiplication. *Spatial pack* convolution reshapes both in input data and weights, although the weights can be reshaped offline, with data packed into tiles that are ideally loaded once. Unlike GEMM convolution, the size of the reshaped input data is the same size, and tiling on inputs and weights is intended to exploit data reuse and SIMD vectorization. Simplified implementations of these three algorithms are given in the Appendix using the TVM tensor expression language.

Another important component of how we format data is the layout, which is how we order the data in memory, since memory may have a different structure to a given tensor (e.g., 1D memory but 4D tensors). For 2D arrays, common formats include "row-major" and "column-major" order, with the former meaning that data in the same row is contiguous in memory, and the latter meaning data in the same column is contiguous in memory. Similarly, for 4D data, which is more relevant to our image classification CNNs, two common formats are `NCHW` and `NHWC`, with `N` representing the batch size and `C`, `H`, and `W` representing the number of input channels, the input height, and input width, respectively. The algorithms in listings 1–3 are in the `NCHW` format, however, spatial pack (Listing 2) temporarily reshapes data to `NCHWc`, where `c` is an inner tile dimension to exploit vectorization.

To achieve savings from pruning, the Algorithms & Data Formats layer is critical, since the chosen algorithm must support "sparsity," i.e., exploiting the zeros generated by pruning to skip computation or reduce memory usage. The computational savings are enabled by the fact that, regardless of the value of an input, a multiplication by zero will have no impact on the final output; and the memory savings come from representing sequences of zeros in a more compressed format. In this work, we use the popular **CSR (compressed sparse row)** format, which represents 2D data using three arrays: (1) The non-zero elements of the parameters (*data*); (2) the original column index of the corresponding parameters (*indices*); and (3) the first non-zero elements in each row, as well as the final non-zero element (*indptr*). Other formats include **BSR (block sparse row)** and **COO (coordinate list)**.

3.5 Systems Software

The systems software most commonly exposed to machine learning practitioners is the DNN framework. Common frameworks include PyTorch [59], TensorFlow [1], JAX [25], and MXNet [8], all of which are focused on DNN training. Other frameworks may focus exclusively on deployment, such as TensorFlow Lite and TensorRT [15]. These frameworks include utilities for defining, saving, and loading models; passing, processing, and inspecting data; and invoking and profiling training and inference. Underlying these DNN frameworks are the kernel libraries that execute the critical computations of the DNNs (e.g., the convolutional layers). For example, for Nvidia GPUs, many frameworks leverage the cuDNN library [11], which is a collection of optimized CUDA kernels to run common DNN operations.

An alternative to using optimized vendor libraries is using a tensor compiler such as TVM [9] or IREE [76]. They generate code for a specific DNN and hardware backend, and when leveraged correctly can outperform vendor libraries, especially for operations that may be less popular or optimized. TVM uses a "compute schedule" programming paradigm, similar to Halide [63], where a high-level description of the computation is complemented by "schedules." Schedules are platform-specific transformations that are applied to the code for each algorithm to enable better performance. Examples of schedule language primitives include parallelism, loop unrolling/splitting/reordering, and vectorization.

Tensor compilers can be taken even further by tuning the code (i.e., schedule) for each layer, using tools such as AutoTVM [10] and Ansor [90]. Specifically, Ansor is an auto-scheduling system built on top of TVM that automatically searches for optimized schedules for a given DNN model on a given hardware platform. It leverages a genetic algorithm and learned cost model to iteratively explore schedule transformations. A tuned schedule alters a given operation to exploit how the data sizes and access patterns interact with the target hardware, e.g., changing the schedule of a DNN layer of a given size to better exploit the cache memory. This is contrasted to an untuned schedule, which is designed to be more generic, and does not exploit knowledge of how the hardware and a particular DNN layer interact.

Below the level of tensor compilers are programming paradigms and general-purpose compilers. LLVM [47] is a cross-platform compiler infrastructure that higher-level compilers such as TVM can lower their optimized code to, which is then converted to a binary. For hardware accelerators, systems such as CUDA [57] and OpenCL [71] provide a programming interface for GPUs, and more specialized accelerators may define their own libraries. Generally, when using accelerators, we still require CPU-side host code to manage accelerator calls and data transfers.

3.6 Hardware

Hardware devices that DNN models commonly execute on include CPUs and GPUs, as well as more specialized accelerators. CPUs are generally complex independent processing cores, typically with one to several dozen cores on a single chip. GPUs are generally simpler interdependent processing cores, typically with dozens to several thousand cores on a single chip. Note that this comparison is a simplification, since these cores are not equivalent, and they may vary in speed, programmability, and other features.

Vector or SIMD instructions allow multiple data to be loaded or computed upon using a single instruction. For example, the Intel instruction `MOVAPS` loads four `float32` values in a single instruction, which in theory represents a 4× speedup compared to loading them one-by-one. Practically, micro-architectural considerations means that this speedup may vary. Different architectures may support varying maximum SIMD length, e.g., Intel's AVX instructions support up to 256 bits, whereas Arm Neon supports up to 128 bits. Different hardware may also have varying support

and levels of optimization for different data-types, e.g., a CPU may support `float16` data-types but actually execute them as `float32` instructions; whereas a GPU may have explicit `float16` instructions that can be exploited.

Another important aspect of hardware is the memory system, with fast, small caches being close to computation and larger, slower memories higher up the hierarchy. As well as memory management within processor hardware, data transfers between CPU and an accelerator can also be a critical bottleneck to efficient processing.

4 Experimental Setup

Our experiments represent a vertical slice of DLAS, which demonstrates the design choices available and interactions that occur. Therefore, we do not optimize for every technique that may influence a given result. Additional techniques from the literature that could be used to further optimize performance, as well as discussion of the trends in our results, are given in Section 6. We develop our case study using Apache TVM (extending it where necessary), a state-of-the-art tensor compiler that is competitive with other optimized runtimes [9, 90] and has the advantage of being able to generate code for multiple targets (e.g., GPUs and CPUs of various architectures).

4.1 Models & Neural Architectures

As highlighted in Section 3.1, we investigate two image classification datasets: CIFAR-10 and ImageNet. For CIFAR-10, we use model definitions from a PyTorch-based library [44], which we train from scratch. We consider four architectures: ResNet18 [33], MobileNet V1 [37] and V2 [66], and VGG-16 [69]. ResNet18 and VGG-16 are larger models, and MobileNets V1 and V2 are designed to be more resource-efficient. To train the models, we used SGD to minimize the cross-entropy loss (averaged across all data items), which penalizes the network for making incorrect classifications. We used a 1cycle LR scheduler [70] with momentum 0.9, weight decay 5×10^{-4}, and an initial LR of 5×10^{-2}, trained for 200 epochs.

For our ImageNet models, we use pre-trained models from the TorchVision repository [55]. We consider four architectures: DenseNet161 [39], EfficientNetB0 [73], ResNet50 [33], and MobileNetV2 [66]. ResNet50 and DenseNet161 are larger models, and MobileNetV2 and EfficientNetB0 are designed to be more resource-efficient. These models are pre-trained with the training configurations described in the TorchVision documentation.

4.2 Model Optimizations

We explore three approaches to compression: (1) global L1 unstructured pruning, which we call "weight pruning" (Figure 2(b)); (2) global L1 structured pruning over convolutional channels, which we call "channel pruning" (Figure 2(c)); and (3) data-type quantization, exploring `float16` and `int8` quantization (Figure 2(d)). Our pruning techniques explore the impact of increasingly higher compression ratios,[2] thus for the evaluation of inference time and for each model and pruning technique, we select the pruning level that has the highest compression ratio before a significant accuracy drop (elbow point) occurs. To implement our pruning, we leverage PyTorch Lightning [23], a wrapper of PyTorch that simplifies pruning. We apply pruning iteratively, starting with the pre-trained unpruned dense models. To reduce accuracy loss, at each pruning step, we apply fine-tuning. For weight pruning, we start by pruning at 50%, then increase in step sizes of 10%, additionally pruning at 95% and 99%. For channel pruning, we start by pruning at 5%, then increase in step sizes of 5%, additionally pruning at 99%. In total, each pruning technique gets the same number of fine-tuning epochs shared evenly in each pruning step. However, we perform

[2]We define compression ratio as: $(\textit{original size} - \textit{compressed size})/\textit{original size} \times 100$.

channel pruning in a more fine-grained way to compensate for its more coarse-grained approach to removing weights. For our CIFAR-10 models, we use: 210 epochs of fine-tuning; an initial LR of 5×10^{-2}; and SGD with momentum 0.9, a weight decay 5×10^{-4}, and the 1cycle LR scheduler [70]. For our ImageNet models, we use: 140 epochs of fine-tuning; an initial LR of 1×10^{-3}; and SGD with momentum 0.9, weight decay 5×10^{-4}, and the cosine annealing LR scheduler [52].

For data-type quantization, we use TVM's native conversion tool. To recover the accuracy from quantizing from `float32` to `int8`, we use ONNXRuntime's [20] post-training quantization tool (with the default static quantization configuration) and the relevant validation dataset. For `float16`, as we show later in our results, there is no accuracy loss, thus, we do not perform any additional steps to recover lost accuracy.

4.3 Algorithms & Data Formats

We evaluate three algorithmic primitives for the convolutional layers: (1) *direct*, (2) *GEMM*, and (3) *spatial pack* convolution. We use both dense and sparse versions of these algorithms, which we implement or extend within TVM v0.8.0. The same high-level algorithm implementation is used for both CPU and GPU. All of our algorithms use the `NCHW` data layout, and for both weight and channel pruning, we use the CSR sparse data format.

4.4 Systems Software

For each convolution algorithm, we implement a minimal TVM schedule that uses thread parallelism. However, since TVM's performance comes from optimized schedules, unoptimized algorithms may give an unrealistic indication of the best algorithm. Thus, rather than hand-optimize the schedules and risk introducing bias from inconsistent levels of optimization, we leverage the Ansor auto-scheduler [90] to generate optimized schedules for each DNN layer and algorithm. For CPU code, we use TVM's LLVM backend with AVX and Neon extensions for the Intel and Arm CPUs, respectively. For GPU code, we generate OpenCL and CUDA kernels for the Arm and Nvidia GPUs, respectively.

For our auto-scheduling (or "tuned") experiments, we allow Ansor to explore up-to 20,000 program variants, with early stopping permitted if no speedups have been observed after 1,000 variants. Auto-scheduling sparse computations is not fully supported by TVM. Thus, we employ an approach in TVM called "sparse sketch rules," where we describe a starting point for the auto-scheduler to begin schedule generation. This works for the CPU, however, TVM is unable to support auto-scheduled sparse computations on GPUs in the versions of TVM we have evaluated. This is because the auto-scheduler has two conflicting requirements: (1) cross-thread reduction, requiring partial sums that must be computed across GPU threads simultaneously; and (2) loops parallelized over threads must request a static number of threads. Both of these conditions cannot be satisfied, since the size of our reduction loop for our algorithms varies, depending on how many non-sparse elements there are in a given portion of the computation. Thus, we cannot tune pruned models on the GPU in our evaluation, which highlights a type of barrier faced by across-stack acceleration researchers, i.e., limited software support for a given combination of DLAS parameters.

4.5 Hardware

Table 1 shows the hardware platforms used in our experiments. For CPU experiments, we use an Intel i7 workstation machine and the HiKey 970 development board. The i7 has 6 cores, but due to hyper-threading, 12 threads are exposed. By default, TVM uses 1 thread per core, a default we follow in our experiments. The HiKey board has an Arm big.LITTLE architecture, meaning that it has 4 more powerful cores (A73@ 2.4 GHz) and 4 less powerful cores (A53@1.8 GHz). For our experiments, we use only the A73 (big) cores, which is the default for TVM. In principle, with

Table 1. Hardware Features of the Devices Used in the Experiments

Device	CPU	L1 Cache (I+D)	L2 (+L3) Cache	RAM	GPU	GPU API
Intel i7	Intel i7-8700 (6 cores) @ 3.2 GHz	192K + 192K	1.5M (+12M)	16 GB DDR3	–	–
HiKey 970	*Arm Cortex-A73 (4 cores) @ 2.4 GHz* Arm Cortex-A53 (4 cores) @ 1.8 GHz	*256K + 256K* 128K + 128K	*2M shared* 1M shared	6 GB LPDDR4	Mali-G72 (12 cores)	OpenCL
AGX Xavier	Arm v8.2 Carmel (4 cores) @ 2.19 GHz	64K + 128K	2M (+ 4M)	16 GB LPDDR4x	Volta (512 cores)	CUDA

For the HiKey 970, we only use the A73 CPU.

appropriately configured load balancing between cores, using all cores could bring a performance improvement. However, this is outside the scope of this work, and as we will discuss in Section 6.6, exposes further across-stack considerations. For our GPU experiments, we leverage the GPU cores of the HiKey 970 and an Nvidia AGX Xavier devices.

4.6 Evaluation Methodology

On both CPU and GPU experiments, we ensure that devices are single-tenant and repeat inference experiments 150 times. We use TVM's `time_evaluator` function with a single input image (i.e., batch size 1). For auto-scheduling, we run our search across 20,000 program variants once per experiment and evaluate the optimized binary 150 times. We report the median inference time, disregarding an initial warm-up run.

5 Evaluation

We split the results between CIFAR-10 (Section 5.1) and ImageNet (Section 5.2). We first analyze the accuracy impact of the optimization techniques and choose maximally compressed models for each technique that maintains accuracy. Then, we analyze the inference performance of these models using the algorithms and compilation configurations on the CPUs and GPUs. Section 6 gives a high-level discussion of the results.

5.1 CIFAR-10

5.1.1 Accuracy. The models' accuracy with varying levels of compression can be seen in the first row of Figure 3. For the four models, Table 2 shows the baseline (dense) top-1 accuracy on CIFAR-10. We observe that, for weight pruning (Figure 3(a)), the accuracy is maintained for all models until 95% pruning, at which point all models see a drop in accuracy at 99%. However, the drop in accuracy for MobileNetV1 and V2 is higher, likely because they have fewer parameters.

We also observe this trend in channel pruning (Figure 3(b)), where VGG-16 and ResNet18 maintain their accuracy for longer compared to the MobileNets. However, for all models, the drop in accuracy is earlier compared to weight pruning, with the elbow appearing at 50% pruning for the MobileNet models and 80% for VGG-16 and ResNet18. We also observe that after the elbow the drop is higher, to around 10% accuracy or equivalent to random guessing.

For data-type quantization in Figure 3(c), we observe almost no change in accuracy for `float16` across the four models. The output is not bit-wise identical, however, at most, this represents a 0.03% difference in top-1 accuracy. For uncalibrated `int8` quantization, all models see a drop in accuracy, with MobileNets V1 and V2 seeing the highest drops, to 10.0% and 16.4%, respectively. However, with calibration, all models recover significantly, with MobileNetV1 losing the most accuracy at 1.7%. Table 2 shows the elbow points of accuracy we take for the inference experiments.

5.1.2 Inference – CPU (Untuned). The first two rows of Figure 4 show the untuned performance of the CIFAR-10 models when running on the CPUs of the HiKey (4(a)–4(d)) and i7 (4(e)–4(h)) platforms, with varying compression strategies and convolutional primitives. The overall trends, including the fastest combination of parameters under different settings, are shown in Table 3.

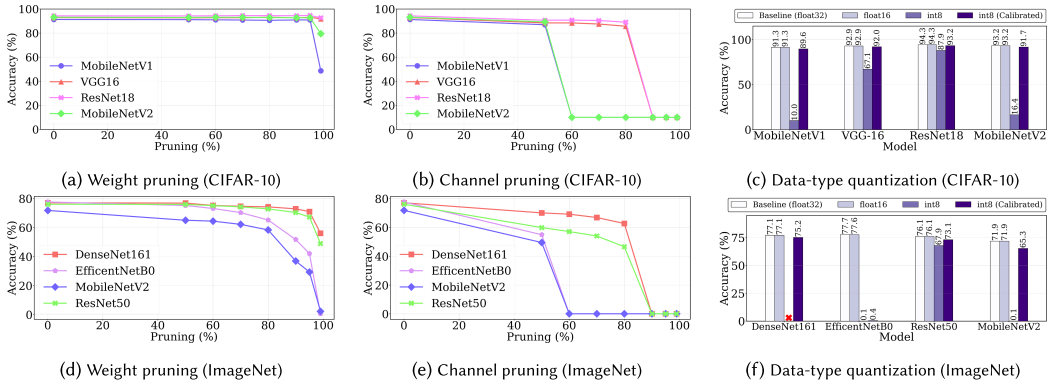


Fig. 3. Accuracy and compression tradeoffs for our CIFAR-10 (a–c) and ImageNet (d–f) models: (a/d) show the accuracy of each model after iterative weight pruning of the convolutional layers, and fine-tuning the model; (b/e) show a similar setup for channel pruning; and (c/f) show `float16` and `int8` quantization accuracy, with `int8` accuracy shown before and after calibration.

Table 2. CIFAR-10 Models, Including Baseline Accuracy (Top1) and Our Chosen Compression Ratios and Corresponding Accuracies

Model	Params	MACs	Top1	Model Optimization Accuracy (& Compression Ratio)			
				Weight Pruning	Channel Pruning	`float16`	`int8`
MobileNetV2	2.3M	98M	93.2%	92.7% (95%)	89.3% (50%)	93.2% (50%)	91.7% (75%)
ResNet18	11.2M	557M	94.3%	94.7% (95%)	89.0% (80%)	94.3% (50%)	93.2% (75%)
VGG-16	14.7M	314M	92.9%	93.1% (95%)	85.7% (80%)	92.9% (50%)	92.0% (75%)
MobileNetV1	3.2M	48M	91.3%	90.9% (95%)	86.9% (50%)	91.3% (50%)	89.6% (75%)

For the untuned baseline, we observe that the i7 and HiKey differ in fastest algorithm, with *direct* and *GEMM* for all models, respectively. However, in most cases across devices, *GEMM* algorithms are the fastest, with one exception for VGG-16 on the HiKey. In terms of compression techniques, `int8` has the fastest overall time in three out of four cases on the i7, whereas weight pruning is the fastest on the Hikey in three out of four cases.

If we take the *best* baseline time for each model, then we can compute an expected speedup given the compression ratio of each model optimization technique. For example, on the HiKey for MobileNetV1, with a pruning rate of 95%, we could expect an ideal speedup of 20×. However, in this case, we only achieve a speedup of 2.6× for the best weight pruning algorithm (*GEMM*), i.e., 13.0% of the expected speedup. On average, for weight pruning, we achieve 11.5% and 21.8% of the expected speedup on the HiKey and i7, respectively.

For channel pruning, the elbow points for the models (see Table 2) show that channel pruned models are less compressed than the weight pruning models, means that we would [expect] the latter to always be slower than the former. However, we observe several cases where a channel pruning model is *faster*, namely, for all VGG-16 variants, except for *GEMM* on the HiKey, which is slightly slower. On average, for channel pruning, we achieve 77.9% and 83.9% of the expected speedup on the HiKey and i7, respectively.

For `float16`, we observe a slowdown when compared to the baseline (`float32`) in every case. For `int8`, we generally see a speedup relative to the baseline, with some notable exceptions using *spatial pack*. In some cases, `int8` gives the best time overall, as seen in the last column of Table 3. On average, we achieve 33.6% and 157.2% of the expected speedup on the HiKey and i7, respectively.

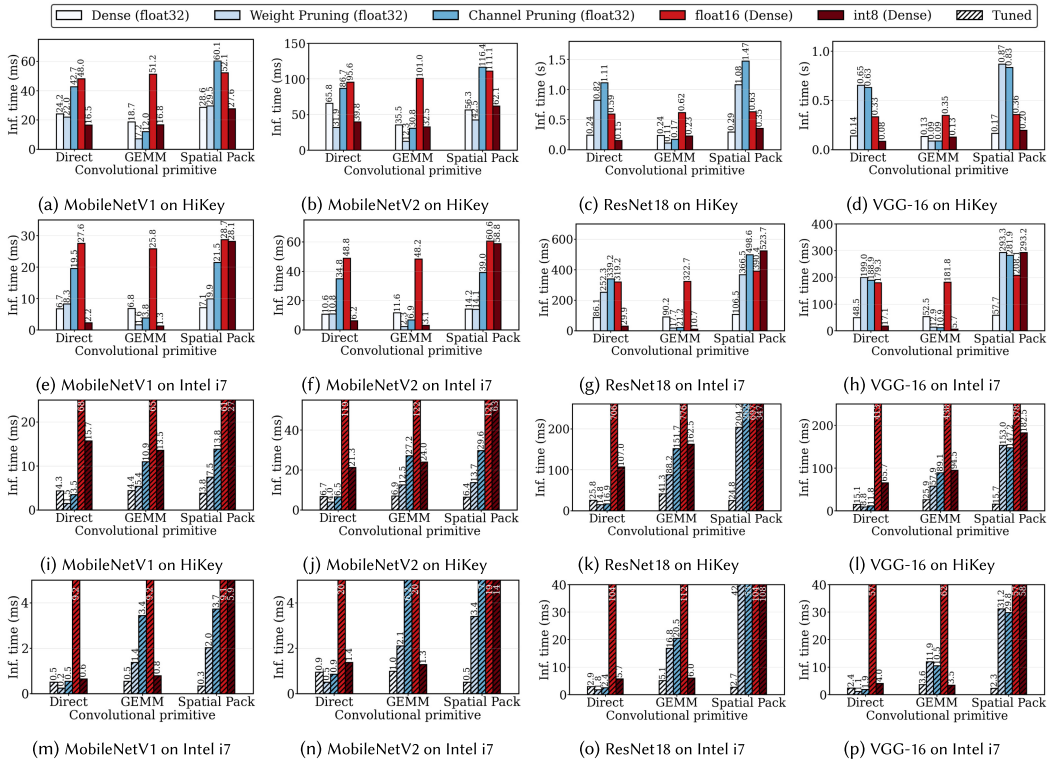


Fig. 4. Experiments comparing the compressed CIFAR-10 models chosen from obvious elbows of accuracy, with varying algorithmic primitives, benchmarked on the i7 and HiKey CPU platforms, with and without auto-scheduling.

5.1.3 Inference – CPU (Tuned). The last two rows of Figure 4 show the tuned performance of the CIFAR-10 models when running on the HiKey (4(i)–4(l)) and i7 (4(m)–4(p)) CPUs, with overall trends shown in Table 3. Comparing to the untuned results in Section 5.1.2, we observe that tuning has created some significant differences in the relative performance of the experiments, beyond reducing the inference time significantly. For example, *spatial pack* is now predominantly the best algorithm for dense models on both CPUs. For weight pruning, *direct* is the best algorithm, and this combination is predominantly the fastest inference time across models and CPUs. On average, we achieve 9.5% and 6.9% of the expected speedup on the HiKey and i7, respectively, less than untuned but faster in absolute terms. For channel pruning, sparse *direct* is also the best algorithm, and we still observe that channel pruning is faster than weight pruning in most cases of VGG-16. On average, we achieve 39.8% and 26.6% of the expected speedup on the HiKey and i7, respectively. Also, `float16` still gives a consistent slowdown, and these differences are exacerbated when compared to the untuned results. The relative speedups of `int8` on the i7 have disappeared with tuning, with an average slowdown of 2.0×.

5.1.4 Inference – GPU (Untuned). The first two rows of Figure 5 show the untuned performance of the CIFAR-10 models when running on the GPUs of the HiKey (5(a)–5(d)) and Xavier (5(e)–5(h)) devices, with overall trends shown in Table 4. We note that the HiKey's inference time on the GPU is much higher than on the CPU, e.g., the dense *direct* MobileNetV1 is almost 7× slower on the GPU. For the pruned experiments, we see speedups most consistently using *spatial pack*,

Table 3. Analysis of CIFAR-10 CPU Results for Varying Combinations of Parameters, Summarizing Figure 4

Platform	Model	Fastest algorithm					Fastest compression technique			Overall fastest
		Dense	WP	CP	i8	f16	GEMM	Direct	Spatial (Pack)	
HiKey (untuned)	MobileNetV2	GEMM	**GEMM**	GEMM	GEMM	Direct	**WP**	WP	WP	WP+GEMM
	ResNet18	GEMM	**GEMM**	GEMM	Direct	Direct	**WP**	i8	Dense	WP+GEMM
	VGG-16	GEMM	GEMM	GEMM	**Direct**	Direct	WP	**i8**	Dense	i8+Direct
	MobileNetV1	GEMM	**GEMM**	GEMM	Direct	Direct	**WP**	i8	i8	WP+GEMM
i7 (untuned)	MobileNetV2	Direct	**GEMM**	GEMM	GEMM	GEMM	**WP**	i8	WP	WP+GEMM
	ResNet18	Direct	GEMM	GEMM	**GEMM**	Direct	**i8**	i8	Dense	i8+GEMM
	VGG-16	Direct	GEMM	GEMM	**GEMM**	Direct	**i8**	i8	Dense	i8+GEMM
	MobileNetV1	Direct	GEMM	GEMM	**GEMM**	GEMM	**i8**	i8	Dense	i8+GEMM
HiKey (tuned)	MobileNetV2	Spatial	**Direct**	Direct	Direct	Direct	Dense	**WP**	Dense	WP+Direct
	ResNet18	Spatial	**Direct**	Direct	Direct	Spatial	Dense	**WP**	Dense	WP+Direct
	VGG-16	Direct	**Direct**	Direct	Direct	Spatial	Dense	**WP**	Dense	WP+Direct
	MobileNetV1	Spatial	**Direct**	Direct	GEMM	Spatial	Dense	**WP**	Dense	WP+Direct
i7 (tuned)	MobileNetV2	**Spatial**	Direct	Direct	GEMM	Spatial	Dense	WP	**Dense**	Dense+Spatial
	ResNet18	Spatial	**Direct**	Direct	Direct	Spatial	Dense	**WP**	Dense	WP+Direct
	VGG-16	Spatial	**Direct**	Direct	GEMM	Direct	i8	**WP**	Dense	WP+Direct
	MobileNetV1	Spatial	**Direct**	Direct	Direct	Direct	Dense	**WP**	Dense	WP+Direct

The best inference times are shown in **bold**. Shortened names are used for brevity: WP (weight pruning), CP (channel pruning), i8 (`int8`), f16 (`float16`).

which is different from the CPU where we almost always saw a slowdown. In terms of expected speedup, for the HiKey and Xavier, respectively, for weight pruning, we achieve 7.4% and 2.2%, and for channel pruning, we achieve 41.1% and 26.0%. For `float16`, unlike the CPU, we observe speedups in several cases, however, this behavior is not consistent across models, algorithms, or devices. For example, on the HiKey using *GEMM* with MobileNetV1 (Figure 5(a)), `float16` provides a slowdown, whereas, for other models on the HiKey, we observe a speedup (similar to the Xavier). On the Xavier, `float16` provides a speedup in all cases except for *direct* convolution where we observe small slowdowns. On average, `float16` achieves 51.9% and 49.4% of its potential speedup on the HiKey and Xavier, respectively.

5.1.5 Inference – GPU (Tuned). The last two rows of Figure 5 show the tuned performance of the CIFAR-10 models on GPUs, with overall trends shown in Table 4. As noted in Section 4.4, we cannot provide tuned results for sparse models on the GPU. The HiKey GPU is still slower than the HiKey CPU (tuned), with the best dense result being 3.1× slower on average. The Xavier does not get any improvement when tuning; we discuss this issue in Section 6.5. For quantization, on the HiKey, taking the best result for each model, we achieve 58.9% and 44.4% of the expected speedup on average for `float16` and `int8`, respectively; the Xavier achieves 49.0% and 28.4% of its expected speedups.

5.2 ImageNet

5.2.1 Accuracy. For the four models, the baseline (dense) top-1 accuracy on ImageNet is shown in Table 5. EfficientNetB0 has the highest accuracy, which may be surprising, given it has fewer parameters. However, EfficientNetB0 is more recent and thus exploits a number of newer machine learning techniques to improve its parameter and training efficiency.

The accuracy on ImageNet with varying levels of compression can be seen in the second row of Figure 3. We observe a similar trend to the CIFAR-10 models, namely, the smaller models (EfficientNetB0 and MobileNetV2) lose their accuracy more quickly than the larger ones (DenseNet161 and ResNet50). We also observe that *all* models lose more accuracy earlier when compared to CIFAR-10 pruning. This suggests that the CIFAR-10 models are more overparameterized.

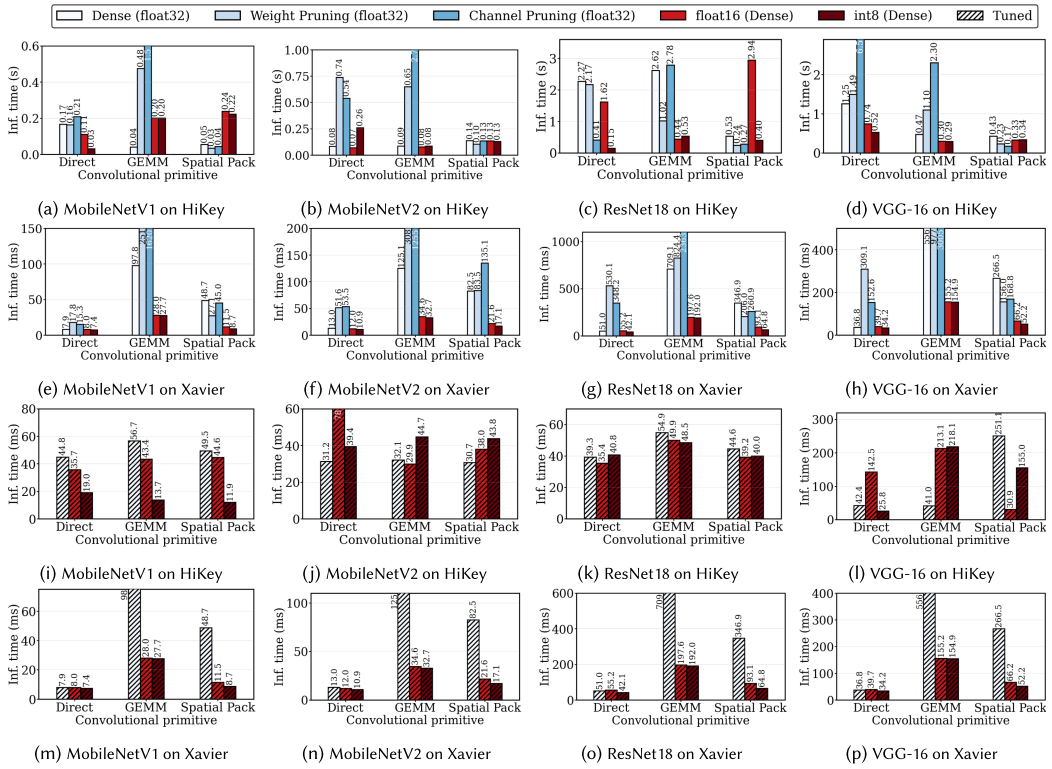


Fig. 5. Experiments comparing the compressed CIFAR-10 models chosen from obvious elbows of accuracy, with varying algorithmic primitives, benchmarked on the HiKey and Xavier GPU platforms, with and without auto-scheduling.

Table 4. Analysis of CIFAR-10 GPU Results for Varying Combinations of Parameters, Summarizing Figure 5

Platform	Model	Fastest algorithm					Fastest compression technique			Overall fastest
		Dense	WP	CP	i8	f16	GEMM	Direct	Spatial (Pack)	
HiKey (untuned)	MobileNetV2	GEMM	Spatial	Spatial	GEMM	**Direct**	f16	**f16**	WP	f16+Direct
	ResNet18	Spatial	Spatial	Spatial	**Direct**	GEMM	f16	**i8**	CP	i8+Direct
	VGG-16	Spatial	Spatial	**Spatial**	GEMM	GEMM	i8	i8	**CP**	CP+Spatial
	MobileNetV1	GEMM	Spatial	Spatial	**Direct**	Direct	Dense	i8	WP	i8+Direct
Xavier (untuned)	MobileNetV2	Direct	Direct	Direct	**Direct**	Direct	i8	**i8**	i8	i8+Direct
	ResNet18	Direct	Spatial	Spatial	**Direct**	Direct	i8	**i8**	i8	i8+Direct
	VGG-16	Direct	Spatial	Direct	**Direct**	Direct	i8	**i8**	i8	i8+Direct
	MobileNetV1	Direct	Direct	Direct	**Direct**	Direct	i8	**i8**	i8	i8+Direct
HiKey (tuned)	MobileNetV2	Direct	–	–	Direct	**GEMM**	**f16**	Dense	f16	f16+GEMM
	ResNet18	Direct	–	–	Spatial	**Direct**	i8	**f16**	f16	f16+Direct
	VGG-16	GEMM	–	–	**Direct**	Spatial	Dense	**i8**	f16	i8+Direct
	MobileNetV1	Direct	–	–	**Spatial**	Direct	i8	i8	**i8**	i8+Spatial
Xavier (tuned)	MobileNetV2	Direct	–	–	**Direct**	Direct	i8	**i8**	i8	i8+Direct
	ResNet18	Direct	–	–	**Direct**	Direct	i8	**i8**	i8	i8+Direct
	VGG-16	Direct	–	–	**Direct**	Direct	i8	**i8**	i8	i8+Direct
	MobileNetV1	Direct	–	–	**Direct**	Direct	i8	**i8**	i8	i8+Direct

The best inference times are shown in **bold**. Shortened names are used for brevity: WP (weight pruning), CP (channel pruning), i8 (`int8`), f16 (`float16`).

Table 5. ImageNet Models, Including Baseline Accuracy (Top1), and Our Chosen Compression Ratios and Corresponding Accuracies

Model	Params	MACs	Top1	Model Optimization Accuracy (& Compression Ratio)			
				Weight Pruning	Channel Pruning	`float16`	`int8`
MobileNetV2	3.5M	327M	71.9%	58.4% (80%)	49.9% (50%)	71.9% (50%)	65.3% (75%)
ResNet50	25.6M	4.1G	76.1%	67.3% (95%)	46.6 (80%)	76.1% (50%)	73.1% (75%)
DenseNet161	27.7M	7.8G	77.1%	74.3% (95%)	62.7 (80%)	77.1% (50%)	75.2% (75%)
EfficientNetB0	5.3M	415M	77.7%	65.3% (80%)	54.9 (50%)	77.6% (50%)	0.4% (75%)

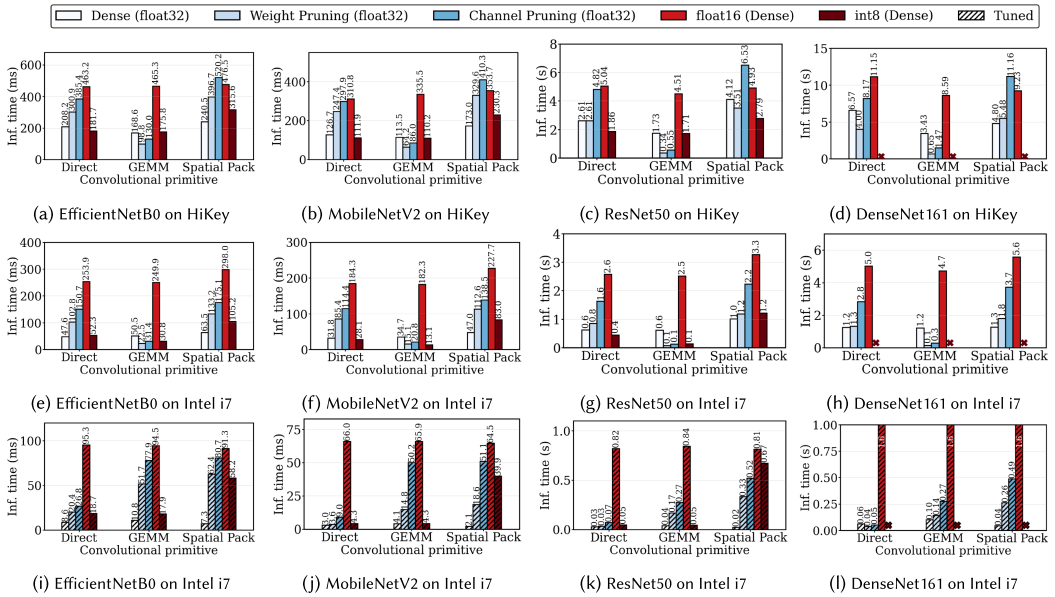


Fig. 6. Experiments comparing the compressed ImageNet models chosen from obvious elbows of accuracy, with varying algorithmic primitives, benchmarked on the i7 and HiKey CPU platforms, with and without auto-scheduling.

For data-type quantization, we observe a similar trend as CIFAR-10, namely, a negligible difference in accuracy for `float16`. For ResNet50, we see a large drop in accuracy for uncalibrated `int8` quantization, recovering to around a 3.0% accuracy reduction. For MobileNetV2, we observe a huge drop in accuracy for the uncalibrated model, down to around 0.09%, recovering to around a 6.6% accuracy reduction. This drop was much higher than we expected, so we also tried importing the Keras [12] definition of MobileNetV2 and observed the same behavior.

For EfficientNetB0, we also observe a huge drop in accuracy to 0.08%, however, the recovery is much smaller than MobileNetV2's, reaching only 0.43% accuracy. This is due to architecture features of the model that make it less suitable for quantization, which we discuss in Section 6.2. For DenseNet161, we cannot run the `int8` model in TVM due to an unsupported quantized operation. This excludes it from collection of uncalibrated accuracy and inference time results. However, when calibrated in ONNXRuntime, we reduce accuracy by 1.9%.

5.2.2 Inference – CPU (Untuned). The first two rows of Figure 6 show the untuned performance of the ImageNet models when running on the i7 and HiKey CPUs, with overall trends shown in Table 6. For dense models, we observe that in all cases on the HiKey *GEMM* gives the best performance, which matches its behavior as seen for CIFAR-10. For the i7, *GEMM* is fastest for the large

Table 6. Analysis of ImageNet Results for Varying Combinations of Parameters, Summarizing Figures 6 and 7

Platform	Model	Fastest algorithm					Fastest compression technique			Overall fastest
		Dense	WP	CP	i8	f16	GEMM	Direct	Spatial (Pack)	
HiKey CPU (untuned)	MobileNetV2	GEMM	**GEMM**	GEMM	GEMM	Direct	**WP**	i8	Dense	WP+GEMM
	ResNet50	GEMM	**GEMM**	GEMM	GEMM	GEMM	**WP**	i8	i8	WP+GEMM
	DenseNet161	GEMM	**GEMM**	GEMM	–	GEMM	**WP**	WP	Dense	WP+GEMM
	EfficientNetB0	GEMM	**GEMM**	GEMM	GEMM	Direct	**WP**	i8	Dense	WP+GEMM
i7 (untuned)	MobileNetV2	Direct	GEMM	GEMM	**GEMM**	GEMM	**i8**	i8	Dense	i8+GEMM
	ResNet50	GEMM	**GEMM**	GEMM	GEMM	GEMM	**WP**	i8	Dense	WP+GEMM
	DenseNet161	GEMM	**GEMM**	GEMM	–	GEMM	**WP**	Dense	Dense	WP+GEMM
	EfficientNetB0	Direct	**GEMM**	GEMM	GEMM	GEMM	**WP**	Dense	Dense	WP+GEMM
i7 (tuned)	MobileNetV2	**Spatial**	Direct	Direct	Direct	Spatial	Dense	Dense	**Dense**	Dense+Spatial
	ResNet50	**Spatial**	Direct	Direct	GEMM	Spatial	Dense	Dense	**Dense**	Dense+Spatial
	DenseNet161	Spatial	**Direct**	Direct	–	Spatial	Dense	**WP**	Dense	WP+Direct
	EfficientNetB0	**Spatial**	Direct	Direct	GEMM	Spatial	Dense	Dense	**Dense**	Dense+Spatial
HiKey GPU (untuned)	MobileNetV2	GEMM	Spatial	Direct	Direct	**GEMM**	**f16**	i8	i8	f16+GEMM
	ResNet50	Direct	Spatial	Spatial	Direct	Direct	WP	i8	**WP**	WP+Spatial
	DenseNet161	Direct	**Spatial**	Spatial	–	GEMM	WP	WP	**WP**	WP+Spatial
	EfficientNetB0	GEMM	Spatial	Spatial	**Direct**	GEMM	f16	**i8**	i8	i8+Direct
Xavier (untuned)	MobileNetV2	Direct	Spatial	Spatial	Direct	**Direct**	f16	**f16**	f16	f16+Direct
	ResNet50	Spatial	Spatial	Spatial	**Direct**	Direct	i8	**i8**	i8	i8+Direct
	DenseNet161	Spatial	**Spatial**	Spatial	–	Direct	WP	f16	**WP**	WP+Spatial
	EfficientNetB0	Direct	Spatial	Spatial	Direct	**Direct**	f16	**f16**	f16	f16+Direct

The best inference times are shown in **bold**. Shortened names are used for brevity: WP (weight pruning), CP (channel pruning), i8 (`int8`), f16 (`float16`).

models (ResNet50 and DenseNet161), however, *direct* is fastest for the small models (MobileNetV2 and EfficientNetB0); on CIFAR-10, *direct* was consistently the fastest on this CPU.

For weight pruning, we find that by taking the best-performing variants as before, we achieve 30.3% and 41.8% of the potential speedup for the HiKey and i7, respectively; significantly higher than CIFAR-10. For channel pruning, this is 60.2% and 84.2%, respectively, which is 16.7% less than CIFAR-10 for the HiKey and 0.3% more for the i7. For quantization, we see similar trends to CIFAR-10, namely, a slowdown using `float16` and a speedup using `int8`. For `int8`, we achieve 25.0% and 73.0% of the expected speedup on the i7 and HiKey, respectively, lower than CIFAR-10.

5.2.3 Inference – CPU (Tuned). The last row of Figure 6 shows the tuned performance of the ImageNet models when running on the i7 CPU. We note that tuning on the HiKey CPU (and GPU) was not practical, since the two variants we attempted took over 140 hours each, so we do not include any of the 57 variants required for each device. For the dense case on the i7, we see that *spatial pack* is consistently the best, matching the observed trends on tuned CIFAR-10. For the pruned models, we do not see any cases where pruned models are faster than a dense `float32` implementation. This is contrasted with CIFAR-10, where we observe this in every case.

5.2.4 Inference – GPU (Untuned). On the Xavier in the dense case, *spatial pack* is the best algorithm for the larger models (ResNet50 and DenseNet161), and *direct* is the best for the smaller models (MobileNetV2 and EfficientNetB0). On the Hikey, for the smaller models, *GEMM* was the best, and for the larger models, *direct* was the best. However, on the HiKey, dense ResNet50 and DenseNet161 experiments using *spatial pack* crashed with the error `CL_INVALID_WORK_GROUP_SIZE`. This means that TVM is exceeding the number of supported work items (see the OpenCL specification for more details [71]). If we run auto-tuning, then TVM can configure the work group size, which could avoid this issue.

For the sparse experiments, *spatial pack* using weight pruning was consistently the best across both GPUs, however, only outperformed the baseline in one case, ResNet50 on the Xavier. On the Xavier, we found that sparse *direct* experiments did not halt, even allowing hours for a

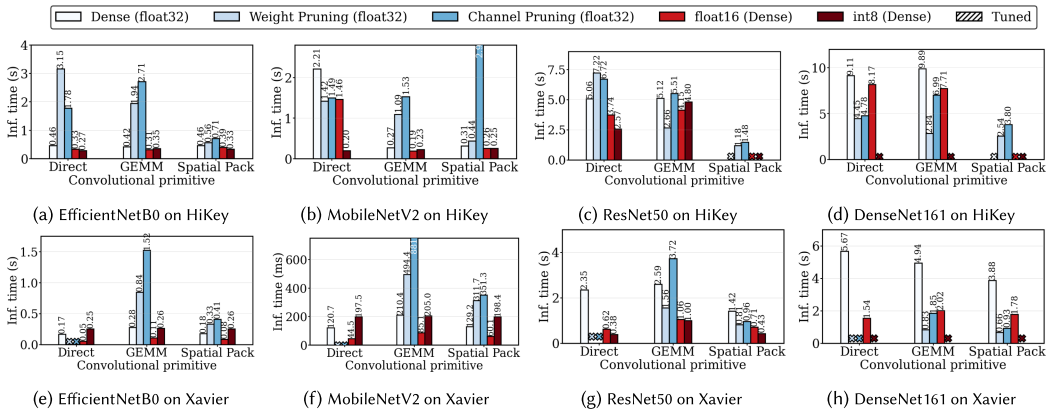


Fig. 7. Experiments comparing the compressed ImageNet models chosen from obvious elbows of accuracy, with varying algorithmic primitives, benchmarked on the HiKey and Xavier GPU platforms, without auto-scheduling.

single run. GPU memory utilization was at its maximum, suggesting that some inefficiency in this algorithm/hardware combination. Again, auto-tuning may make this variant viable, however, as highlighted in Section 4.4, we cannot tune sparse models on GPUs.

For ImageNet, quantization was not consistently the best compression technique for the GPUs, unlike CIFAR-10. For several cases, weight pruning performed best. On the HiKey, using `int8` for EfficientNet was the fastest approach, however, it should be noted that in this case the accuracy drop was prohibitively large, as discussed in Section 6.2.

5.2.5 Inference – GPU (Tuned). As discussed in Section 5.2.3, collecting tuned results for the HiKey GPU was not practical. In addition, we again observed no speedup on the Xavier when tuning, hence, we do not include the graphs.

6 Discussion and Related Work

Our experiments show a large number of results, even from varying a small number of parameters across DLAS, which highlights a critical challenge: the huge design space for across-stack DNN acceleration. Other studies try to make more generalized claims of the impact of a specific parameter by exploring a significantly reduced design space, limiting the number of parameters explored at some layers of DLAS or keeping them fixed. For example, Hadidi et al. [32] do not explore the impact of compression techniques or algorithms. Similarly, the stack presented by VTA [56] focuses on a given compiler framework and accelerator, which is less general than DLAS.

Our results show that the interaction between different layers of DLAS can cause significant performance variations. This is our core observation: The number of design choices available for DNN acceleration continues to grow, and including additional parameters in a study can significantly change the performance optimization landscape. Some examples of this include:

— For MobileNetV2 on CIFAR-10, the fastest overall combination of algorithm and compression technique is weight pruning and GEMM (see Table 3). This is true in the untuned cases of both the HiKey and i7 CPUs. However, adding in just one additional DLAS parameter (tuning) changes this, such that for the HiKey, weight pruning and direct convolution is the fastest, whereas for the i7, dense and spatial pack convolution is the fastest. This example shows that introducing just one new DLAS parameter can expose differences between other parameters (the hardware devices) not previously evident.

— Similarly, for ImageNet in the untuned case on the CPU, weight pruning is consistently the fastest compression technique on both devices. However, on the i7, for MobileNetV2 only, `int8` quantization is the fastest option.
— Finally, for CIFAR-10 on the GPUs, direct convolution is consistently the fastest algorithm in the untuned case, however, on the HiKey for VGG-16, spatial pack convolution is the fastest.

Overall, these examples show how multiple across-stack interactions can both unlock higher performance, but also present the challenge of how to efficiently explore the design space, i.e., choosing DLAS parameters that are likely to give high-performance improvements without significantly increasing the number of experiments required.

Identifying the across-stack problems relevant to DNN acceleration has been explored in other works. For example, Sze et al. [72] give a comprehensive overview of relevant DNN acceleration parameters, but do not simplify this overview into a concise structure, which is critical for efficient discussion and analysis. Accelerating DNNs is the combination of several NP-hard optimization problems, so, at best, researchers can create principled heuristics that try to give the highest speedups, while keeping the evaluation costs low. Our experiments used a **modified tensor compiler (TVM) [9]**, which helped reduce the cost of DSE by automatically generating code for different combinations of parameters. Techniques and software like this will be increasingly relevant as the number of design choices continues to grow [26].

In Section 5, we observed many variations across our experiments with a number of non-trivial dynamics emerging. Our first observation is *(I) MACs, accuracy, and inference time are not strongly correlated*, along with 12 additional observations, *(II)–(XIII)*. These observations are not intended to be definitive, and large changes in the results could be expected by bringing in new techniques from DLAS. However, the key point of this work is that across-stack interactions of machine learning and systems optimizations can be non-trivial, and bringing in additional features may significantly accelerate or impede a given technique. We refer the reader to other characterization works and surveys that highlight cutting-edge techniques from across DLAS [5, 32, 36, 50, 72, 84].

6.1 Datasets & Problem Spaces

Between datasets, models showed similar trends in terms of accuracy, with the accuracy losses due to compression being higher for ImageNet models. The computational requirements of models for each dataset varied, i.e., CIFAR-10 models have fewer MACs and parameters than ImageNet models (Tables 2 and 5), which we would expect to have effects on the inference behavior. For instance, *(II) tuned sparse CIFAR-10 models were faster than the dense baseline, but this was not the case for ImageNet models.* Possible explanations include overheads in the sparse algorithm and data format choices, which could be exacerbated by the larger ImageNet models; and for tuning, both sets of models were given the same number of trials despite ImageNet models being larger.

6.2 Models & Neural Architectures

As a related observation to *(I)*, we note that *(III) model size is not strongly correlated with accuracy, however, smaller models are more vulnerable to compression.* For example, EfficientNetB0 has a higher baseline accuracy than both DenseNet161 and ResNet50, despite having at most 21% of the parameters. This can be understood as EfficientNetB0 being a more recent model that exploits novel architectural and training techniques to achieve better parameter efficiency. However, other works have shown that if we keep the model architecture the same, then more parameters generally means higher accuracy [16, 28, 73]. For CIFAR-10, we observed that ResNet18 and MobileNetV2 had higher baseline accuracies than VGG-16, despite both having fewer parameters. A similar

explanation of using techniques such as residual blocks can explain this behavior. However, as *(III)* notes, models with fewer parameters were more vulnerable to compression.

Another related observation is that EfficientNetB0 had the highest accuracy drops for `int8` quantization out of any model. We understand this to be due to EfficientNetB0's architecture not being amenable to post-training quantization. These issues were highlighted and corrected by EfficientNet-Lite [51], which removes the squeeze-and-excitation networks and replaces the swish activation functions with ReLU6 activations [37].

6.3 Model Optimizations

Overall, the best model optimization technique varied. *(IV) Weight pruning tended to give better compression ratios and speedups than channel pruning, however, achieved less of its expected speedups.* We also observed that *(V) quantization's speedup varied by hardware platform and data-type.*

6.3.1 Pruning. As observation *(IV)* notes, weight pruning was generally faster than channel pruning, however, the former achieved a lower proportion of its expected speedups. There were several cases where a less compressed channel pruning model was faster than a more compressed weight pruning model. However, this appears to come [from] algorithmic interactions, since taking the best variant across algorithms it was rare for channel pruning to outperform weight pruning. Overall, the relative performance of pruned models on the GPUs was worse than on the CPUs. This is because sparse computations are irregular and thus cannot easily take full advantage of the greater number of cores available in a GPU that dense computations can. However, some across-stack techniques can improve sparse performance on GPUs [30, 62]. Since we could not tune sparse computations on the GPUs, we cannot comment how well they would perform if the code further optimized. Other pruning techniques that we did not explore include layer-wise pruning and gradient-based methods, tradeoffs of which are discussed in Blalock et al. [5]. We only evaluated pruning techniques in terms of accuracy and compression, however, as other work has noted, pruning also comes with non-negligible costs due to retraining, which may be a bottleneck to DSE [38, 61].

6.3.2 Data-type Quantization. (VI) `float16` *had a negligible impact on accuracy, when calibrated* `int8` *lost 1.8% and 22.2% for CIFAR-10 and ImageNet models on average, respectively.* Excluding EfficientNetB0, ImageNet models lost 3.8% of accuracy on average. With regards to inference time, as observation *(V)* notes for `float16` on the CPUs, we observed a consistent slowdown when compared to `float32`. This is because the hardware runs `float16` computation using software emulation. Although there are savings in memory footprint, the overhead involved in the emulation clearly outweighs these savings, with the differences exacerbated when tuning. For `int8` on the CPUs, we generally observed a speedup, since the CPU ISAs can use SIMD instructions to compute `int8` computations relatively efficiently.

In the untuned `int8` case on the CPU, we observe some cases where we get *higher* than expected speedups; for instance, many of the *GEMM* and *direct* cases on the i7 in Figure 4. The highest of this is for VGG-16 with CIFAR-10 using *GEMM*, where we observe a speedup of over 9.2×. We hypothesize that using smaller data sizes is reducing cache usage, meaning that the algorithm has to fallback to slower caches less in the `int8` case. However, this advantage is significantly reduced when we tune. We believe that Ansor is not fully exploiting the performance potential and search space of `int8`, since it was (1) initially designed with `float32` computation in mind, and (2) `int8` can require a more complex sequence of instructions to be generated to run efficiently, which Ansor does not appear to factor in.

For `float16` on the GPUs, contrasting to the CPUs, we observed in general a reduction in inference time on both GPUs, as they have direct hardware support for `float16` instructions. Like the

CPU, we also saw speedups with `int8` models, in general, marginally higher than with `float16`. However, if we take the best times across all algorithms, we did not see any cases where `float16` or `int8` achieved close to an ideal speedup of 2× and 4× relative to `float32`.

6.4 Algorithms & Data Formats

We made several observations for algorithms and how they interacted with other layers of DLAS. In the dense case, when tuned, *(VII) spatial-pack convolution is generally the best algorithm on the CPU (when tuned)*, and *(VIII) direct convolution is generally the best algorithm on the GPU (when tuned).* Observation *(VIII)* goes against conventional wisdom, where we would normally expect *GEMM* to be faster on the GPU. However, we should note that we are using a custom implementation of GEMM within TVM, rather than an optimized BLAS library. When the algorithm was not tuned, the best algorithm varied more, especially on the GPUs. For example, with ImageNet on the HiKey, *GEMM* was faster for smaller models, and *direct* was faster for larger models. In the sparse case, *(IX) GEMM is generally the best algorithm on CPUs, and direct is generally the best on GPU.* For quantized models, the best algorithm varied on the CPUs and HiKey GPU, whereas on the Xavier, *direct* was generally the fastest.

In the evaluation, we used the same convolution algorithm for all layers of a given model, but we could vary the algorithm used per-layer, which could bring significant performance speedups. However, as other work has noted, data format transformation overheads between layers need to be considered [3, 18]. We also did not explore all possible algorithms available, such as Winograd convolution [48]. However, we chose three common algorithms to keep the across-stack evaluation tractable. Other works have explored algorithmic tradeoffs in more detail [14, 22, 65, 85].

As discussed in Section 6.3.1, *(X) for the pruning techniques, we rarely observed the expected performance improvements* when compared to the dense implementations. Partly this can be attributed to the inherent overheads of sparse data formats—CSR must store up to three values for every non-zero element. This, coupled with irregular data access patterns means that the sparse algorithms do not realize their full potential. CSR is not the only way to represent sparsity, and alternative data formats (and their complementary algorithms) may provide different tradeoffs [13, 31, 46]. For example, for channel pruning, we could store a list of the indices of pruned channels and store the non-pruned channel data in a dense format. This could allow us to leverage a more "dense-like" algorithm with an overhead for looking up pruned indices. Alternatively, we could use a format called **block-sparse row (BSR)**, which is similar to CSR but represents blocks of sparse parameters, rather than individual weights. This could allow us to reduce the overheads of the sparse storage format, with greater savings with larger block sizes at the risk [of] higher accuracy loss. However, to fully exploit this, we would need to change the pruning method to prune blocks.

6.5 Systems Software

(XI) auto-tuned code dramatically accelerates inference time and can change the best algorithm or compression technique. However, we observed no speedup tuning the Xavier. When testing with server-class Nvidia GPU platforms, we observed speedups using auto-scheduling and the same evaluation code. Our conclusion is that some aspect of the AGX Xavier's software stack was incompatible with Ansor, but we could not detect it. It is well known that auto-scheduling can provide significant speedups [89, 90], but the search time required is non-negligible. We could not collect auto-scheduled results on the HiKey for the ImageNet models, since they took too long to tune (over 140 hours each). This highlights a key issue with auto-tuning, namely, the cost of search, especially on constrained devices. Approaches that significantly prune the search space [78] and techniques such as transfer-tuning [27] can reduce the search time.

As highlighted in Section 6.3.2, we observed that *(XII) Pruned models saw a lower relative speedup when tuned.* A more specialized sparse compiler, such as the emerging SparseTIR system [88], could reduce the impact of these overheads. This was also observed for the quantized models, which may require similar optimization support.

We initially experienced an issue compiling `int8` EfficientNet, as TVM assumed that in a multiplication operation only the left-hand operand would be pre-quantized. However, the structure of EfficientNet violated this assumption, which necessitated a bug-fix that we pushed upstream. This highlights that assumptions that systems software make about the properties of workloads may not always hold, especially when novel DNN architectures emerge.

All of the models were defined in PyTorch and evaluated in TVM, however, there are other DNN frameworks available, and the relative performance of different DNN frameworks has been well studied [28, 32, 54]. Other systems-software dimensions to consider are hand-tuned kernel libraries such as oneDNN [58], cuDNN [11], or other deep learning compilers such as TensorRT [15] or IREE [76]. Collage [40] can explore varying the backend for different subgraphs of the same DNN, which can bring significant performance improvements.

6.6 Hardware

As expected, *(XIII) the i7 CPU was generally faster than the HiKey CPU, and the Xavier GPU was generally faster than the HiKey GPU.* Unfortunately, we did not see improvements when tuning on the Xavier GPU. Other aspects of the hardware that could be better utilized include the big.LITTLE architecture of the HiKey CPU (we only leveraged the big cores), hyper-threading on Intel CPU (we ran one thread per core), or leveraging both the CPU and GPU in parallel for the Xavier and HiKey. However, across-stack optimizations would be required to exploit these features properly [53, 83]. We also did not leverage the Xavier GPU's 64 tensor cores in addition to its 512 general purpose CUDA cores. TVM supports tensor cores but requires manual schedule re-design for each algorithm. MetaSchedule [67] can expand auto-scheduler search spaces to include hardware features such as tensor cores, which could ease [investigation of] this dimension.

6.7 Evaluation Methodology

For the experiments, we kept the batch size as 1, took the median of 150 runs, disregarding the first warm-up run. Although this is a common deployment and evaluation scenario, it is important to be aware that this is not the only one, and experimental design should reflect which deployment case is being considered when evaluating models [74, 86]. For instance, for edge deployment, we may expect the batch size to be small, whereas on the cloud it may be large. Increased batch sizes mean increased memory requirements and inference latency, but also potentially higher throughput. For the use of 150 runs, disregarding the first run, there could be deployment scenarios where we are more interested in the performance of these initial warm-up runs, before the cache behavior becomes more regular.

7 Conclusions

This article first motivates and introduces the **Deep Learning Acceleration Stack (DLAS)** and then presents a perturbation study with an exploration of the impact of varying a small number of parameters at each layer of the stack. In our study, we find a variety of across-stack interactions and scenarios where theoretical performance improvements were not achieved due to lack of full exploitation across the stack. Our work is not intended to propose solutions to all of these limitations, instead highlights some complexities that emerge in deep learning acceleration and presents a conceptual framework (DLAS) for practitioners to approach their studies in the future.

We believe this can be achieved through closer collaboration across the layers of DLAS to enable more holistic co-design and co-optimization.

A Appendix

```
N, K, M = OC, (KH * KW * IC), (OH * OW)
B = te.compute( # perform im2col transformation
   (batches, M, K), lambda n, m, k: data[
       n, (k // (KH * KW)) % IC, (k // KH) % KW + ((m // OW) * HSTR), (k % KW) + ((m % OW) * WSTR),
   ],
)
k = te.reduce_axis((0, K), "k")
C = te.compute((batches, OC, OH, OW), lambda b, c, h, w: te.sum(A[c, k] * B[b, h * OH + w, k], axis=k))
```

Listing 1: Definition of GEMM convolution using TVM's tensor expression language.

```
# weight data is reshaped offline to 6-D with shape
# [num_filter_chunk, in_channel_chunk, filter_height, filter_width, in_channel_block, num_filter_block]
data = te.compute( # reshape data to 5-D
   (n, ic_chunk, ih, iw, ic_bn), lambda bs, c, h, w, vc: data[bs, c * ic_bn + vc, h, w]
)
ic = te.reduce_axis((0, in_channel), name="ic")
kh = te.reduce_axis((0, kernel_height), name="kh")
kw = te.reduce_axis((0, kernel_width), name="kw")
oshape = (n, oc_chunk * oc_bn, oh, ow)
packed_out = te.compute(
   oshape, lambda n, oc_chunk, oh, ow, oc_block: te.sum( # compute to 5-D output volume
       data[
           n, idxdiv(ic, ic_bn), oh * HSTR + kh * dilation_h, ow * WSTR + kw * dilation_w, idxmod(ic, ic_bn),
       ] * kernel[oc_chunk, idxdiv(ic, ic_bn), kh, kw, idxmod(ic, ic_bn), oc_block], axis=[ic, kh, kw],
   ),
)
out = te.compute( # reshape to NCHW
   oshape, lambda n, c, h, w: packed_out[n, idxdiv(c, oc_bn), h, w, idxmod(c, oc_bn)]
)
```

Listing 2: Definition of spatial pack convolution using TVM's tensor expression language.

```
rc = te.reduce_axis((0, in_channel), name="rc")
ry = te.reduce_axis((0, kernel_h), name="ry")
rx = te.reduce_axis((0, kernel_w), name="rx")
out = te.compute((batch, out_channel, out_height, out_width),
   lambda nn, ff, yy, xx: te.sum(
       data_pad[nn, rc, yy * stride_h + ry * dilation_h, xx * stride_w + rx * dilation_w]
       * Filter[ff, rc, ry, rx], axis=[rc, ry, rx],
   ),
)
```

Listing 3: Definition of direct convolution using TVM's tensor expression language.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*, USENIX Association, USA, 265–283.

[2] H. Alvarez, L. M. Paz, J. Sturm, and D. Cremers. 2016. Collision avoidance for quadrotors with a monocular camera. In *Proceedings of the 14th International Symposium on Experimental Robotics*, M. Ani Hsieh, Oussama Khatib, and Vijay Kumar (Eds.). Springer International Publishing, Cham, 195–209. DOI: https://doi.org/10.1007/978-3-319-23778-7_14

[3] Andrew Anderson and David Gregg. 2018. Optimal DNN primitive selection with partitioned boolean quadratic programming. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'18)*. ACM, New York, NY, USA, 340–351. DOI: https://doi.org/10.1145/3168805

[4] Paul Barham and Michael Isard. 2019. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19)*. Association for Computing Machinery, 177–183. DOI: https://doi.org/10.1145/3317550.3321441

[5] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. 2020. What iss the state of neural network pruning? In *Proceedings of Machine Learning and Systems*. 129–146. Retrieved from https://proceedings.mlsys.org/paper/2020/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf

[6] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once-for-all: Train one network and specialize it for efficient deployment. In *Proceedings of the 8th International Conference on Learning Representations*.

[7] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. 2017. An Analysis of Deep Neural Network Models for Practical Applications. DOI: https://doi.org/10.48550/arXiv.1605.07678

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems*.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*, USENIX Association, USA, 579–594.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 31 (2018), 3393–3404. DOI: https://doi.org/10.5555/3327144.3327258

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *arXiv:1410.0759 [cs]* (Oct. 2014).

[12] François Chollet. 2015. *Keras*. Retrieved April 4, 2019 from https://keras.io/

[13] Antonio Cipolletta and Andrea Calimera. 2021. On the efficiency of sparse-tiled tensor graph processing for low memory usage. In *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC'21)*. 643–648. DOI: https://doi.org/10.1109/DAC18074.2021.9586154

[14] Jason Cong and Bingjun Xiao. 2014. Minimizing computation in convolutional neural networks. In *Proceedings of the International Conference on Artificial Neural Networks (ICANN'14)*, Stefan Wermter, Cornelius Weber, et al. (Eds.). Springer International Publishing, Cham, 281–290. DOI: https://doi.org/10.1007/978-3-319-11179-7_36

[15] Nvidia Corporation. 2016. NVIDIA TensorRT: Programmable Inference Accelerator. Retrieved from https://developer.nvidia.com/tensorrt

[16] Elliot J. Crowley, Gavin Gray, and Amos J. Storkey. 2018. Moonshine: Distilling with cheap convolutions. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 2888–2898.

[17] J. D. Day and H. Zimmermann. 1983. The OSI reference model. *Proc. IEEE* 71, 12 (1983), 1334–1340. DOI: https://doi.org/10.1109/PROC.1983.12775

[18] Miguel de Prado, Nuria Pazos, and Luca Benini. 2019. Learning to infer: RL-based search for DNN primitive selection on heterogeneous embedded systems. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE'19)*. 1409–1414. DOI: https://doi.org/10.23919/DATE.2019.8714959

[19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. DOI: https://doi.org/10.1109/CVPR.2009.5206848

[20] ONNX Runtime developers. 2018. ONNX Runtime.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. DOI: https://doi.org/10.18653/v1/N19-1423

[22] Manuel F. Dolz, Sergio Barrachina, Héctor Martínez, Adrián Castelló, Antonio Maciá, Germán Fabregat, and Andrés E. Tomás. 2023. Performance–Energy Trade-Offs of deep learning convolution algorithms on ARM processors. *The Journal of Supercomputing* 79, 9 (June 2023), 9819–9836. DOI: https://doi.org/10.1007/s11227-023-05050-4

[23] William Falcon and The PyTorch Lightning team. 2019. PyTorch Lightning. DOI: https://doi.org/10.5281/zenodo.3828935

[24] Jonathan Frankle and Michael Carbin. 2019. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR'19)*.

[25] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Syst. Mach. Learn.* 4, 9 (2018).

[26] Perry Gibson. 2023. *Compiler-centric Across-stack Deep Learning Acceleration*. Ph. D. Dissertation. College of Science and Engineering, School of Computing Science, University of Glasgow. DOI: https://doi.org/10.5525/gla.thesis.83959

[27] Perry Gibson and José Cano. 2023. Transfer-tuning: Reusing auto-schedules for efficient tensor program code generation. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT'22)*. Association for Computing Machinery, New York, NY, USA, 28–39. DOI: https://doi.org/10.1145/3559009.3569682

[28] Perry Gibson, José Cano, Jack Turner, Elliot J. Crowley, Michael O'Boyle, and Amos Storkey. 2020. Optimizing grouped convolutions on edge devices. In *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'20)*. 189–196. DOI: https://doi.org/10.1109/ASAP49362.2020.00039

[29] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Communications of the ACM* 63, 11 (2020), 139–144.

[30] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse DNN Models without Hardware-support via tile-wise sparsity. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. DOI: https://doi.org/10.1109/SC41405.2020.00020

[31] Fred G. Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.* 4, 3 (Sept. 1978), 250–269. DOI: https://doi.org/10.1145/355791.355796

[32] Ramyad Hadidi, Jiashen Cao, Yilun Xie, Bahar Asgari, Tushar Krishna, and Hyesoon Kim. 2019. Characterizing the deployment of deep neural networks on commercial edge devices. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 35–48. DOI: https://doi.org/10.1109/IISWC47752.2019.9041955

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 770–778. DOI: https://doi.org/10.1109/CVPR.2016.90

[34] John L. Hennessy and David A. Patterson. 2018. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In *Proceedings of the International Symposium on Computer Architecture (ISCA'18)*. 27–29. DOI: https://doi.org/10.1109/ISCA.2018.00011

[35] Danny Hernandez and Tom B. Brown. 2020. Measuring the algorithmic efficiency of neural networks. *arXiv:2005.04305* (May 2020).

[36] Sara Hooker. 2021. The hardware lottery. *Commun. ACM* 64, 12 (Nov. 2021), 58–65. DOI: https://doi.org/10.1145/3467017

[37] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861 [cs]* (Apr. 2017).

[38] Wenhao Hu, Perry Gibson, and Jose Cano. 2023. ICE-Pick: Iterative cost-efficient pruning for DNNs. In *Neural Compression: From Information Theory to Applications – Workshop @ ICML*.

[39] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4700–4708.

[40] Byungsoo Jeon, Sunghyun Park, Peiyuan Liao, Sheng Xu, Tianqi Chen, and Zhihao Jia. 2023. Collage: Seamless integration of deep learning backends with automatic placement. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'22)*, Association for Computing Machinery, New York, NY, USA, 517–529. DOI: https://doi.org/10.1145/3559009.3569651

[41] 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. 2017. In-Datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International*

Symposium on Computer Architecture (ISCA'17), Association for Computing Machinery, Toronto, ON, Canada, 1–12. DOI: https://doi.org/10.1145/3079856.3080246

[42] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* 596, 7873 (August 2021), 583–589. DOI: https://doi.org/10.1038/s41586-021-03819-2

[43] Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. (2009), 60. Retrieved from https://api.semanticscholar.org/CorpusID:18268744

[44] kuangliu and Perry Gibson. 2023. PyTorch Lightning CIFAR10. Retrieved March 15, 2023 from https://github.com/Wheest/pytorch-lightning-cifar

[45] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. Association for Computing Machinery, Williamsburg, VA, USA, 461–475. DOI: https://doi.org/10.1145/3173162.3173176

[46] Daniel Langr and Pavel Tvrdík. 2016. Evaluation criteria for sparse matrix storage formats. *IEEE Trans. Parallel Distrib. Syst.* 27, 2 (Feb. 2016), 428–440. DOI: https://doi.org/10.1109/TPDS.2015.2401575

[47] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-stage Optimization*. Ph. D. Dissertation. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.

[48] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 4013–4021. DOI: https://doi.org/10.1109/CVPR.2016.435

[49] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (May 2015), 436–444. DOI: https://doi.org/10.1038/nature14539

[50] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2021. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (March 2021), 708–727. DOI: https://doi.org/10.1109/TPDS.2020.3030548

[51] Renjie Liu. 2020. *Higher Accuracy on Vision Models with EfficientNet-Lite*. Retrieved March 14, 2023 from https://blog.tensorflow.org/2020/03/higher-accuracy-on-vision-models-with-efficientnet-lite.html

[52] Ilya Loshchilov and Frank Hutter. 2017. SGDR: Stochastic gradient descent with warm restarts. In *Proceedings of the International Conference on Learning Representations (ICLR'17)*.

[53] Manolis Loukadakis, José Cano, and Michael O'Boyle. 2018. Accelerating deep neural networks on low power heterogeneous architectures. In *Proceedings of the Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG'18)*. Retrievedfrom:http://eprints.gla.ac.uk/183819/

[54] Nikolaos Louloudakis, Perry Gibson, Jose Cano, and Ajitha Rajan. 2022. Assessing robustness of image recognition models to changes in the computational environment. In *NeurIPS ML Safety Workshop*.

[55] TorchVision maintainers and contributors. 2016. TorchVision: PyTorch's Computer Vision Library. Retrieved March 15, 2023 from https://github.com/pytorch/vision

[56] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. A hardware-software blueprint for flexible deep learning specialization. arXiv:1807.04188 [cs, stat] (April 2019).

[57] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue* 6, 2 (Mar. 2008), 40–53. DOI: https://doi.org/10.1145/1365490.1365500

[58] Intel. 2020. *oneDNN*. oneAPI. Retrieved June 19, 2020 from https://github.com/oneapi-src/oneDNN

[59] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, High-Performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Curran Associates Inc., Red Hook, NY, USA, 8026–8037.

[60] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2021. Carbon Emissions and Large Neural Network Training. DOI: https://doi.org/10.48550/arXiv.2104.10350

[61] Kaveena Persand, Andrew Anderson, and David Gregg. 2021. Taxonomy of saliency metrics for channel pruning. *IEEE Access* PP, (August 2021), 1–1. DOI: https://doi.org/10.1109/ACCESS.2021.3108545

[62] Valentin Radu, Kuba Kaszyk, Yuan Wen, Jack Turner, José Cano, Elliot J. Crowley, Björn Franke, Amos Storkey, and Michael O'Boyle. 2019. Performance aware convolutional neural network channel pruning for embedded GPUs. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 24–34. DOI: https://doi.org/10.1109/IISWC47752.2019.9042000

[63] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling algorithms from schedules for high-performance image processing. *Communications of the ACM* 61, 1 (December 2017), 106–115. DOI: https://doi.org/10.1145/3150211

[64] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10684–10695.

[65] Simon Rovder, José Cano, and Michael O'Boyle. 2019. Optimising convolutional neural networks inference on low-powered GPUs. In *Proceedings of the International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG'19)*. Retrieved from https://eprints.gla.ac.uk/183820/

[66] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen. 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, 4510–4520. DOI: https://doi.org/10.1109/CVPR.2018.00474

[67] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. 2022. Tensor program optimization with probabilistic programs. In *Advances in Neural Information Processing Systems*. Retrieved March 16, 2023 from https://openreview.net/forum?id=nyCr6-0hinG

[68] Laurent Sifre. 2014. *Rigid-motion Scattering for Image Classification*. Ph. D. Dissertation. Ecole Polytechnique, CMAP.

[69] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556* (Sept. 2014).

[70] Leslie N. Smith and Nicholay Topin. 2018. Super-convergence: Very Fast Training of Neural Networks Using Large Learning Rates. DOI: https://doi.org/10.48550/arXiv.1708.07120

[71] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng.* 12, 3 (May 2010), 66–73. DOI: https://doi.org/10.1109/MCSE.2010.69

[72] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2020. *Efficient Processing of Deep Neural Networks*. Springer International Publishing, Cham. DOI: https://doi.org/10.1007/978-3-031-01766-7

[73] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the International Conference on Machine Learning (ICML'19)*. PMLR, 6105–6114.

[74] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. 2013. Optimizing Google's warehouse scale computers: The NUMA experience. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 188–197. DOI: https://doi.org/10.1109/HPCA.2013.6522318

[75] Marvin Teichmann, Michael Weber, Marius Zöllner, Roberto Cipolla, and Raquel Urtasun. 2018. MultiNet: Real-time joint semantic reasoning for autonomous driving. In *Proceedings of the IEEE Intelligent Vehicles Symposium (IV'18)*. 1013–1020. DOI: https://doi.org/10.1109/IVS.2018.8500504

[76] The IREE Authors. 2019. IREE. Retrieved from https://github.com/openxla/iree

[77] Neil C. Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F. Manso. 2022. The Computational Limits of Deep Learning. DOI: https://doi.org/10.48550/arXiv.2007.05558

[78] Nicolas Tollenaere, Guillaume Iooss, Stéphane Pouget, Hugo Brunie, Christophe Guillon, Albert Cohen, P. Sadayappan, and Fabrice Rastello. 2023. Autotuning convolutions is easier than you think. *ACM Transactions on Architecture and Code Optimization* 20, 2 (March 2023), 20:1–20:24. DOI: https://doi.org/10.1145/3570641

[79] J. Turner, J. Cano, V. Radu, E. J. Crowley, M. O'Boyle, and A. Storkey. 2018. Characterising across-stack optimisations for deep convolutional neural networks. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'18)*. 101–110. DOI: https://doi.org/10.1109/IISWC.2018.8573503

[80] Jack Turner, Elliot J. Crowley, Valentin Radu, José Cano, Amos Storkey, and Michael O'Boyle. 2018. Distilling with performance enhanced students. *arXiv:1810.10460 [cs, stat]* (Oct. 2018).

[81] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 30 (2018), 5998–6008. DOI: https://doi.org/10.5555/3295222.3295349

[82] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the International Conference on Learning Representations*.

[83] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. 2020. High-throughput CNN inference on embedded ARM Big. *LITTLE Multicore Processors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (October 2020), 2254–2267. DOI: https://doi.org/10.1109/TCAD.2019.2944584

[84] Xiaofei Wang, Yiwen Han, Victor C. M. Leung, Dusit Niyato, Xueqiang Yan, and Xu Chen. 2020. Convergence of edge computing and deep learning: A comprehensive survey. *IEEE Communications Surveys Tutorials* (2020), 1–1. DOI: https://doi.org/10.1109/COMST.2020.2970550

[85] Yuan Wen, Andrew Anderson, Valentin Radu, Michael F. P. O'Boyle, and David Gregg. 2020. TASO: Time and space optimization for memory-constrained DNN inference. In *Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'20)* . 199–208. DOI: https://doi.org/10.1109/SBAC-PAD49847.2020.00036

[86] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, Washington, DC, USA, 331–344. DOI: https://doi.org/10.1109/HPCA.2019.00048

[87] Ikuya Yamada, Akari Asai, Hiroyuki Shindo, Hideaki Takeda, and Yuji Matsumoto. 2020. LUKE: Deep contextualized entity representations with entity-aware Self-attention. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Association for Computational Linguistics, Online, 6442–6454. DOI: https://doi.org/10.18653/v1/2020.emnlp-main.523

[88] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*. ACM, New York, NY, USA, 660–678. DOI: https://doi.org/10.1145/3582016.3582047

[89] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. 2022. DietCode: Automatic optimization for dynamic tensor programs. *Proceedings of Machine Learning and Systems* 4, (April 2022), 848–863. Retrieved July 21, 2022 from https://proceedings.mlsys.org/paper/2022/hash/fa7cdfad1a5aaf8370ebeda47a1ff1c3-Abstract.html

[90] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 863–879. Retrieved December 26, 2020 from https://www.usenix.org/conference/osdi20/presentation/zheng

Received 14 November 2023; revised 20 June 2024; accepted 19 July 2024