Astra: Exploiting Predictability to Optimize Deep Learning

Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, Lidong Zhou
Microsoft Research

Abstract

We present *Astra*, a compilation and execution framework that optimizes execution of a deep learning training job. Instead of treating the computation as a generic data flow graph, *Astra* exploits domain knowledge about deep learning to adopt a custom approach to compiler optimization.

The key insight in *Astra* is to exploit the unique repetitiveness and predictability of a deep learning job, to perform online exploration of the optimization state space in a work-conserving manner while making progress on the training job. This dynamic state space exploration in *Astra* uses light-weight profiling and indexing of profile data, coupled with several techniques to prune the exploration state space. Effectively, the execution layer custom-wires the infrastructure end-to-end for each job and hardware, while keeping the compiler simple and maintainable.

We have implemented *Astra* in two popular deep learning frameworks, PyTorch and Tensorflow. On state-of-the-art deep learning models, we show that *Astra* improves end-to-end performance of deep learning training by up to 3x, while approaching the performance of hand-optimized implementations such as cuDNN where available. *Astra* also significantly outperforms static compilation frameworks such as Tensorflow XLA both in performance and robustness.

CCS Concepts • **Computer systems organization** → **Other Architectures**; • **Software and its engineering** → *Software Infrastructure*; Software Notation and Tools.

Keywords domain-specific compiler, deep learning, adaptation

ACM Reference Format:
Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, Lidong Zhou. 2019. Astra: Exploiting Predictability to Optimize Deep Learning. In *ASPLOS '19: ACM Symposium on Architectural Support for Programming Languages and Operating Systems, April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304072

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304072

1 Introduction

Test a single grain of rice, to see if the whole pot is cooked.

-Ancient Proverb

Deep Neural Networks (DNNs) already power several widely-used products today, ranging across domains such as speech understanding, image recognition and language translation. While the theory behind neural networks has been around for decades [12, 28], the recent success of DNNs was catalyzed in large part due to systems advances: first, the ability to scale to large clusters [9], and then the compute power of hardware accelerators such as GPUs.

Despite their popularity, there is a glaring lack of conceptual understanding to reason about DNNs - *e.g.* which model structure would be a good fit to a particular problem, or what *hyper-parameters* to use for a given structure. As a result, innovation in DNNs happens primarily through trial-and-error; the common way to find out whether a particular model would work is to run the model and see what accuracy it converges to. For large models, running even one such iteration can take several days.

As a result of the trial-and-error methodology of research in DNNs, advancement in AI is gated on systems advances to speed up DNN training. While hardware advances such as more powerful GPUs [32], and even ASICs [14] help bridge this gap, the software layers come in the way; for example, even with current GPUs, several state-of-the-art models such as text classification only utilize a small fraction of the GPU when run with modern frameworks such as Tensorflow and PyTorch. Software accelerators such as *cuDNN* speed up specific types of DNN layers by hand-optimizing the specific computation [4]. However, given the engineering effort required, such accelerators only cater to the most popular primitives (*e.g.* standard convolution or LSTM layers).

Unfortunately, by definition, novel model architectures that AI researchers invent are *long tailed*, *i.e.* they typically do not fit into the "popular" primitives that are addressed by hand-coded accelerators. For example, researchers have recently suggested several variants of recurrent neural network such as MI-LSTM [36], LSTM with Attention [35], SC-RNN [22], RHN [39], *etc.*, none of which are currently accelerated by cuDNN. However, it is precisely these "new" models that need to be fine-tuned by repeated trial and error.

Therefore, the efficiency of these long-tail models is crucial to AI innovation.

In this paper, we present *Astra*, a system that automatically optimizes such long-tail models to achieve performance similar to hand-coded accelerators such as cuDNN, thus significantly improving efficiency of DNN research. To achieve this, *Astra* employs a novel approach to optimization exploiting the unique characteristics of the DNN workload.

Our key observation driving the design of *Astra* is that a DNN training job is a unique workload from a systems perspective. Each job comprises of millions of *mini-batches*, where each mini-batch looks at a small number of training inputs. Because each mini-batch typically goes through exactly the same computation graph, mini-batches are identical to each other from a resource-usage and performance viewpoint. As a result, perhaps for the first time in systems research, we have a predictable, long-running workload that *declares* its resource usage and behavior ahead of time.

Astra takes advantage of such remarkable predictability, by adopting an aggressive form of multi-version compilation [17, 33, 38]. In contrast to traditional compilers that produce one version of the code using a cost model, multi-version compilers produce different versions of code for "hot" functions, with each version using a different choice of optimizations; the runtime runs the different versions and picks the version that runs fastest. Such division of functionality has two vital benefits. First, the compiler can be aggressive and go after long-tail optimizations (*e.g.* that only benefit 10% of the workloads) because it need not evaluate the optimizations, whereas a traditional compiler would be resource-limited to focus only on broadly useful optimizations. Second, the compiler is much simpler as it does not need to build (and keep up-to-date) complex performance models of GPU/hardware performance, DNN operators, *etc.*, a particularly challenging task given the frenetic pace of change in this space - both in hardware [14, 15, 21] and workload.

While multi-version compilation for *generic* programs has significant practical challenges because of variability in computation across input instances, often requiring statistical techniques that run each version thousands of times [17], the repeatability of the deep learning workload makes it an ideal fit, because a particular choice of optimizations needs to be measured only once. This empowers *Astra* to amplify the impact of multi-versioning by significantly expanding the scope of online adaptation. Instead of operating at the granularity of individual "hot" functions like generic multi-version compilers, *Astra* reasons about the entire data flow graph and explores interactions between dependent optimization choices at various parts of the graph, resulting in end-to-end improvement in training time across a variety of models.

A key challenge in this approach of end-to-end online adaptation is managing the size of the exploration state space; at one extreme, if the compiler enumerated every possibility, the state space could get too large. We use three key techniques to keep the state space under control. First, we use simple coarse-grained static information at the compiler to be intelligent in the enumeration phase, so that only potentially interesting candidates are enumerated. Second, we perform profiling at a fine-grained level to allow parallelism in state-space exploration. In contrast to techniques such as Halide OpenTuner [3] which only measure end-to-end runtime and hence rely on random mutations one at a time, *Astra* can simultaneously adapt several (hundreds of) independent configuration parameters by exploiting fine-grained measurements. This makes the size of the state space additive rather than multiplicative in the number of dimensions. Third, we use the initial profile measurements during the online exploration as signals to prune the dynamic state space in an intelligent manner.

We have implemented *Astra* in two popular DL frameworks - PyTorch [25] and Tensorflow [1], and have evaluated the system on several state-of-the-art deep learning models. We demonstrate that *Astra* significantly speeds up ad hoc models, up to 3x in cases where the model does not fit an existing cuDNN implementation. We also show that in cases where cuDNN does apply, *Astra* achieves performance comparable to cuDNN for those models. We also show that *Astra* manages its exploration state space quite effectively, and that it outperforms static optimization approaches such as XLA by up to 72% besides being significantly more robust.

The key contributions of the paper are as follows.

- We identify and leverage the unique characteristics of the deep learning workload to do extreme tailoring or custom-wiring of the infrastructure for a specific job, resulting in significant efficiency gains;
- We propose and evaluate a new architectural framework for optimization of such custom workloads, with aggressive multi-version compilation at a whole-program level that performs parallel exploration of independent choices by using fine-grained profiling.
- We demonstrate with a detailed evaluation that the state space of online exploration is manageable with our pruning strategies, and that end-to-end models get significant speedups over native PyTorch and Tensorflow, and even over static optimizers such as XLA.
- We identify some simple functionality that new hardware for deep learning needs to conform to in order to enable our adaptation aproach.

The rest of the paper is structured as follows. In § 2, we provide a background of deep learning workloads . In § 3, we outline the various dimensions of the optimization state space for DNNs. We present the design of *Astra* in § 4, describe the prototype implementation in § 5 and present a detailed evaluation in § 6. In § 7, we outline the basic features needed from new DNN hardware, to support *Astra*. We discuss related work in § 8, and conclude in § 9.

2 Background

In this section, we provide a brief background on the core primitives and operations of deep neural networks (DNNs), and the current state of optimizing neural networks.

2.1 Deep Neural networks

DNNs comprise of several *layers* of computation. Each layer comprises of several *neurons*; each neuron takes the outputs of neurons in the layer above as their inputs, and produce an output as a function of the inputs - *e.g.*, a weighted sum of the inputs followed by a non-linear function such as Sigmoid or ReLU [37]. The weights to apply to each of the inputs is one of the key parameters that the network learns during training.

The computation flow during a DL training job consists of a *feed-forward* pass and a *backward propagation* pass. During the feed-forward pass, the network takes a batch of training input (*e.g.*, a set of images for an image classification task, or a set of sentences for a language translation task), typically referred to as a *mini-batch*, and runs those inputs through the layers. The result produced by the last layer is the prediction of the network; for an already trained model that is used for prediction, this is the only computation it does. For a model that is being trained, however, this prediction is then compared to the actual *ground truth* that is provided as part of the input batch. The difference between the ground truth and the prediction is the error or loss of the network. The error values are then propagated back through the network until each neuron has an associated error value that reflects its contribution to the original output, and then uses these error values to calculate the gradient of the loss function. This gradient is fed to an optimization method such as such as stochastic gradient descent [27], which uses it to update the weights, in an attempt to minimize the loss function.

Computationally, both the forward and backward passes translate into a series of matrix or *tensor* operations, predominantly matrix multiplication, and other operations such as SoftMax, BatchNorm, L2Norm, etc. Because of the significant data parallelism available in matrix operations, GPUs are quite effective and hence have become a de facto standard for running deep neural network training.

2.2 Execution model

The operations in the forward and backward pass of a DNN training model naturally fit into a data flow graph (DFG) representation [1], where the nodes are the operators (*e.g.*, GPU kernels), and the edges are tensors/matrices. The set of primitive operators (such as matrix multiplication) is quite small, and several libraries such as cuBLAS [24] implement GPU kernels for those common operators. Prior work by several people has looked at the problem of how to automatically generate optimized code for these low-level primitive kernels [6, 26, 30], but that is not the focus of this paper.

Frameworks such as Tensorflow [1] treat these operators as building blocks and manage the end-to-end execution flow. They parse a model written in a high-level language such as Python, and construct a data flow graph from it, where the computation nodes typically map to an existing kernel implementation in say cuBLAS. They then dispatch the operations in the DFG asynchronously to the GPU driver through a cuda_launch API. Efficient execution depends heavily on this end-to-end orchestration of the DFG, which is the primary focus of the optimizations explored in this paper.

2.3 GPU characteristics

This subsection describes some salient features of the GPU architecture and execution model. GPUs have much higher parallelism than CPUs; a P100 GPU for example has about 3500 cores, and has a compute throughput of about 9 teraflops/sec [13]. Newer versions such as the Tesla V100 provide significantly higher compute of 120 teraops/sec for half-precision (16-bit) operations [32]. GPUs adopt a SIMD execution model where a group of cores (called a *warp*) execute lock-step on the same instruction stream but multiple parallel data streams, say different tiles of a matrix multiplication. In order to extract the full compute throughput of the GPU, the workload needs to have a high degree of parallelism – this can happen in two ways; either the matrices being operated are so large that the tiling of the matrices is enough to keep the cores busy, or multiple operations must be simultaneously scheduled on the GPU. To enable the latter, GPUs expose a *stream* abstraction to the higher level. A stream is akin to a thread of execution; all operations scheduled in a single stream are executed in FIFO order sequentially by the GPU, but operations scheduled in different streams run in parallel. When scheduling operations on multiple streams, the application needs to enforce dependencies across operations by adding appropriate synchronization events. Because of the complexity of doing this while keeping stream utilization balanced, most frameworks such as Tensorflow today use just a single stream.

GPUs have a memory hierarchy comprised of registers, on-chip shared memory, and external HBM (high bandwidth memory); GPU kernel libraries such as cuBLAS carefully manage these shared resources for locality to maximize the degree of parallelism realized on the GPU.

Another challenge to realizing the full throughput of a GPU is the fixed cost of about 5-10 usec to launch a kernel on the GPU; in order for this cost to be amortized, the granularity of individual operations scheduled must be large enough that the compute time for the kernel is higher than launch overhead. One common optimization that is employed towards this goal is to *fuse* multiple small operations so that it can be launched as one kernel.

2.4 Software accelerators

Given the complexity of fully utilizing the parallelism of GPU with just low-level kernels, popular layer structures in DNNs have hand-optimized "compound kernels", *i.e.*, accelerator implementations in software. cuDNN [7] is a popular example, that supports standard convolution layers and recurrent network layers such as long-short term memory (LSTM). In recurrent layers where the granularity of individual operations is quite small, cuDNN provides significant speedup - up to 6x - compared to default implementation in Tensorflow or PyTorch.

There are two shortcomings with this approach of hand-optimized accelerators. First, they can only cater to the most popular structures, and hence do not help with deep learning experimentation where researchers come up with novel structures; for example, they do not help with structures such as MI-LSTM [36], LSTM with Attention [35], SC-RNN [22], RHN [39], *etc.*. Second, these APIs work at the abstraction of a single layer, and hence cannot go after full-graph optimization opportunities. With *Astra*, we automate this process of generating an accelerated implementation for a long-tail model structure, and do so in a way that has visibility into the full graph to balance conflicting alternatives, *e.g.*, between fusion in the forward pass vs. in the backward pass, which may each require different allocation strategies for tensor objects.

3 State space of optimizations

In this section, we describe the various dimensions of the optimization state space for DNN training jobs, and illustrate the complexity of statically figuring out the best optimizations. We hint at how a measurement-driven adaptive approach during training can help with several of these dimensions.

3.1 Low-level Kernels

Kernels for basic operations such as GEMM (General matrix multiplication) balance several parameters such as the thread block size, the data tiling size, the amount of shared memory to use per block, etc. The optimal choice of these parameters depends on several factors such as the sizes of the operands and the GPU architecture, especially considering the *performance cliff* behavior exhibited by GPUs along some of these dimensions [31]. OpenAI GEMM libraries [11] provide minimal parametrization of the kernel parameters to allow for exploration at higher levels.

Several libraries exist for the low-level kernels such as GEMMs, such as cuBLAS [24], OpenAI [11] and Neon [29]. Interestingly, the best performing library varies depending on the input parameters and sometimes the GPU generation. Table 1 shows the varied performance of cuBlas and OpenAI for different sizes of GEMMs, and indicates that it is hard to statically identify the best choice.

Size	cuBlas	OAI_1	OAI_2
64x1024x4096	0.156	0.125	0.938
64x4096x1024	0.138	0.172	0.141

Table 1. This table shows time in ms for two GEMM operations during a run of a LSTM model on a P100 GPU. The first row is from a forward pass fused GEMM, and the second from a backward pass.

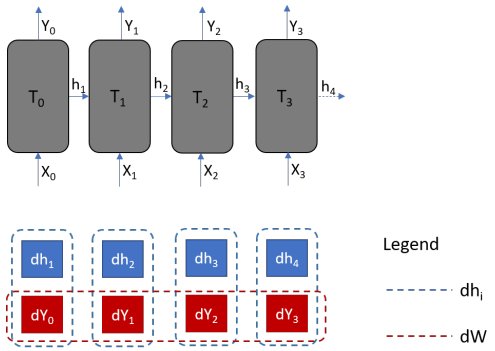

Figure 1. An example of conflicting choice in fusion of GEMMs. *This picture shows two sets of GEMM operations that occur in the SC-RNN model [22] during its backward pass of propagating gradients. As fusion requires contiguous memory allocation for the tensors, one needs to select the most beneficial fusion/allocation strategy.*

3.2 Fusion of kernels

A technique commonly used by accelerators such as cuDNN is to fuse multiple smaller operations into one large operation. This is particularly valuable in recurrent layers or LSTM layers, where the individual operations are too small to keep the GPU occupied. The most straightforward form of fusion is GEMM fusion, where multiple independent GEMMs with their own operand tensors are multiplied together as one large GEMM; performing a single GEMM of two 512x512 tensors is faster than performing four sequential GEMMs of 128x512 tensors with a 512x512 tensor. Ideally, this requires that operand tensors to be fused are allocated contiguously in GPU memory to avoid the cost of copying them into contiguous space. In a large data flow graph, picking the right allocation strategy to maximize fusion is challenging, because the choices of fusion groups sometimes require conflicting allocations (see Figure 1).

Further, while larger fusion groups are usually better for performance, they need not always be: for example, on a P100 GPU and CUDA v9.2, performing two GEMMs of size (256 x 1024) x (1024 x 1024) in parallel on two GPU streams takes 172 us, while the fused version, *i.e.*, a single (512 x 1024) x (1024 x 1024) GEMM runs *slower* at 211 us. Hence statically

figuring out fusion groups is challenging, necessitating the measurement-driven approach that *Astra* employs.

3.3 Using multiple streams

The GPU computation model provides an abstraction of streams to allow parallel execution of independent kernels, which is another way to exploint parallelism when fusion doesn't apply. While scheduling operations on multiple streams, one needs to balance utilization of streams, while avoiding dependent operations in the DFG from being scheduled on different streams (to reduce synchronization cost). In general, if operation B is dependent on operation A's output, scheduling B in a different stream and forcing it to wait for A would prevent other possibly independent kernels that could have run in parallel without waiting for A. Because of the asynchronous execution model of GPUs where the kernels are launched long before they actually execute on the GPU, the fine-grained completion time of the kernels cannot be estimated statically; further, the completion time of a kernel can be affected by interference across streams. Hence, balanced utilization of multiple streams is hard to achieve while dispatching the operations asynchronously. Not surprisingly, Tensorflow and pyTorch do not take advantage of streams, and use a single sequential stream. With *Astra*, the dynamic adaptation allows picking the right schedule on multiple streams without complex reasoning about operation latencies.

3.4 Other optimizations

While software accelerators such as cuDNN operate at the level of individual DNN layers, there are opportunities for cross layer or whole-graph optimizations that use context from the whole model to apply transformations. An example is to dynamically trade off computation for memory; saving part of the memory used for forward-pass activations by redoing the computation, thus accommodating a bigger model [34]. This idea can be used for optimization as well, if the cost of recomputation of some layers of the forward pass is lower than the parallelism benefit from supporting say a 2x larger mini-batch size, again a complex dynamic that needs measurement.

Distributed or multi-GPU training is another important dimension of the optimization state space. Depending on the communication cost of the model and the physical characteristics of the network, the choice of ideal degree of parallelism from a cost-benefit perspective, could be taken in an automated manner with runtime measurement and adaptation.

In this paper, we only focus on the dimensions listed in sub-sections 3.1, 3.2, and 3.3, but the approach is applicable to other dimensions as well.

3.5 Limitations of static optimization

In each dimension of the choice space listed above, reasoning about which set of options would be most efficient by statically modelling the interactions is too complex and expensive if not infeasible. Adding to the complexity is the rapid pace at which the field of deep learning evolves, both in terms of hardware (rapid new generations of GPUs, FPGAs, ASICs *etc.*), and the compute model and APIs (*e.g.*, new layer structures, novel operators *etc.*). Therefore, even if one had a sophisticated static cost model that captured the various aspects of the interaction, such a model would have to keep up with the frenetic pace at which the field evolves, making in an economically unviable approach. As an example, XLA is a static optimization layer for Tensorflow that benefits some models and hurts others [18]; because it is hard to figure out *a priori* where it hurts, XLA is still an experimental feature that is turned off by default; we present a comparison of *Astra* and XLA in Section 6.

Astra provides a new alternative to navigate the state space of optimizations, enabling aggressive, long-tail optimizations, without complex cost models of the end-to-end stack.

4 Design of *Astra*

The key insight in the design of *Astra* is that the unique predictability of mini-batch-level computation in a deep learning training job can be exploited to adopt a fundamentally different approach to optimizing DNNs, that gets around the challenges of static optimization listed in Section 3.

4.1 Mini-batch predictability

A DNN training job runs millions of mini-batches, each operating on a different set of training inputs. The execution time of a mini-batch can be anywhere from tens of milliseconds to a few seconds depending on the model and hardware. Importantly, independent of the actual inputs in the mini-batch, the DNN performs *exactly the same* computation on every mini-batch, [1] as it goes through the same layers and executes the same tensor operations; the cost of the operations (*e.g.*, matrix multiplication) depends only on the shapes of the tensors (which is constant across mini-batches), rather than the actual values. Hence, if a mini-batch is profiled for a certain choice of optimization and it speeds up the mini-batch by say 2x, it is guaranteed to also speed up every other mini-batch, and hence the whole training job, by the same factor.

4.2 A New Compiler-Runtime Interface

At a high level, *Astra* is a compiler – it takes existing model code and generates an optimized execution schedule. However, unlike a traditional compiler, *Astra* performs an amplified variant of *multi-version compilation* [17, 33, 38], changing the traditional division of functionality between the compiler and runtime.

A traditional compiler performs two tasks: (a) it enumerates the state space of relevant optimizations, and (b) it ranks

[1] One exception to this observation is dynamic graphs in frameworks such as PyTorch; we handle them with bucketed profiling discussed in § 5.5

those optimizations using a performance model, and produces a single version of the code. The runtime simply executes the code produced by the compiler.

In contrast, the optimizer in *Astra* is split between two parts: an *enumerator* and a *custom-wirer*. The compiler performs only one of the two tasks listed above: enumerating the state space of possible optimizations, using static knowledge to perform minimal pruning of the state space. The output of the compiler is not one version of the code, but conceptually N versions, each pertaining to one instance of its enumeration. The runtime in *Astra* performs the task of *custom-wiring* by ranking the optimizations to pick the best set of optimizations, but it does not need a sophisticated cost model. Instead, it simply runs each configuration and measures it; each mini-batch is run with a different option from the state space enumerated by the compiler, and the measurements are used to prune the state space. Unlike previous work on multi-version compilers [17, 33, 38] that focus on adapting individual functions that are "hot", the adaptation in *Astra* works at the whole program level and can reason about the interactions between operations across the whole data flow graph, such as conflicting memory allocations for different fusion choices (§ 3.2), assignment of kernels to multiple streams in a history-aware manner, (§ 3.3), and so on. The custom-wirer performs this exploration in a *work conserving* manner: a small number (*e.g.*, a few thousand out of millions) of mini-batches is used for exploration while still making useful training progress.

From a practical viewpoint, this new division of functionality has two key implications for the compiler:

Simplicity: The compiler does not need to implement a detailed cost model of the DNN operators or the hardware (*e.g.*, multiple generations of GPUs). This is significant in the DNN context where both evolve at a rapid pace. The compiler codebase remains simple and maintainable.

Long-tail Coverage: In a traditional compiler, the engineering effort needed for an optimization is high because its end-to-end impact and interactions need to be modelled. Hence only widely applicable optimizations make the cut. In contrast, with the enumeration approach in *Astra*, the compiler can be aggressive and go after even optimizations that only benefit 5% of models, because it does not need to reason about its performance. This is particularly useful in cases where researchers try out esoteric model structures: these are precisely the models that matter for AI innovation, yet a traditional compiler cannot afford to optimize.

4.3 Fine-grained Profiling

The key mechanism that enables *Astra* to manage the large state space during exploration, is fine-grained profiling. Autotuning systems such Halide OpenTuner [3] and Tensor Comprehensions [30] view the whole code as a black box and measure only end-to-end latency; they then use a genetic algorithm to explore the state space; in such a model, the state space exploration can only happen one *mutation* at a time because multiple changes will confound the measurement. In contrast, *Astra* performs hierarchical fine-grained profiling in a lightweight manner; for example, the time for each GEMM is measured, and simultaneously the time for the overall group of GEMMs within a layer is also measured. This allows *Astra* to change multiple exploration parameters in the same iteration, significantly reducing the number of iterations for exploration. In general, the fine-grained profiling is crucial to most forms of exploration described below.

The right profiling metric to use is another key aspect. While elapsed time is the most natural metric that works in simple explorations such as GEMM kernels and fusion, it fails to capture the dynamics when multiple kernels run in parallel on different streams. We therefore choose custom metrics; for streams we use the time for completion of all kernels scheduled across streams.

4.4 Enumerator

The enumerator generates *templated schedules* of execution, using static knowledge and general graph optimizations.

4.4.1 Static Analysis

One key part of the enumeration is selection of GEMMs for fusion. To select candidate nodes for fused GEMMs, the enumerator uses simple graph pattern matching. GEMMs which have a common argument, and no dependency relation among are chosen as candidates for fusion. Consider the following example from the Pytorch graph trace (the % nodes are the operand tensors, and mm is the operation for matrix multiplication)

```
%10 = mm (%1, %5)
%11 = mm (%1, %6)
```

In this example, both the operations can be replaced with a single operation provided there is no dependence between %5 and %6. Multiple such fusion sets are also chosen for fusion along the other dimension to generate 2-D fusion sets. At a high level, this is a graph colouring problem. To reduce the state space, we only consider nodes which have the same provenance (wrt GEMM nodes only).

Fusion Ladders are another pattern that the static policy exploits. A commonly observed pattern in the graph is a GEMM-accumulator ladder. Consider the following:

```
%10 = mm (%1, %5)
%11 = mm (%2, %6)
%12 = add (%10, %11)
```

All these nodes can be replaced by a single node (if %10 and %11 are not used elsewhere). This is extended to longer ladders and multiple such ladders are also chosen to be fused. Note that while the enumerator identifies *maximal* fusion groups, it is up to the custom-wirer to figure out the actual granularity of fusion by chunking the fusion group.

4.4.2 Configuration generation

The information extracted during the static analysis is organised as a set of *adaptive variables*. An adaptive variable is the basic unit of adaptation used by the custom-wirer, and has the following interface:

- `initialize`: Reset to default choice
- `iterate`: Change local choice to next option.
- `get_profile_value`: Get the profile metric

These adaptive variables are organised into an update tree. The update tree has different modes of exploration, which are annotated by the enumerator:
Parallel: All child nodes can be explored and profiled independently (*e.g.*, used for adaptation of fusion groups)
Exhaustive: Exhaustive (brute-force) exploration of the subtree; exponential in number of choices
Prefix-based: The search follows a specified update order. The first child is iterated while the others are kept constant. When the search for this child finishes, its best value is recorded and the exploration for the next child begins.

The role of the enumerator is thus to build this update tree of adaptive variables, with the appropriate mode annotations, which is then used for exploration by the run-time. The full list of exploration modes is described in the next subsection.

4.5 Pruning exploration state space

We now describe in detail the exploration modes stated above. Figure 2 shows the exploration graph for part of a model.

4.5.1 Parallel exploration

With fine-grained profiling, the exploration of state space becomes parallel across independent choices in the optimization hierarchy. Let's consider a a model that has 5 groups of fusion choices (*e.g.*, pertaining to 5 DNN layers), where each fusion group has 12 GEMM kernels. Let's assume the 12 kernels within a fusion group have three choices for fusion; they can be fused in chunks of size 1, 2 or 4 (*i.e.*, either 12, 6 or 3 fused GEMMs respectively), and each GEMM operation has 2 choices of kernels (*e.g.*, cublas, openAI) to try out. With a random mutation based exploration, each iteration can only vary one point in the state space, so we need $(3 * 2)^5 = 7776$ trials. However, with fine-grained profiling and parallel exploration, it needs a much smaller number of trials because the exploration for each group can be performed in parallel; we thus only need $3 * 2 = 6$ trials to complete this state space because all the choice dimensions are independent.

4.5.2 Hierarchical exploration

Independent exploration is not always feasible. For example, to be fusion-friendly, the tensors that participate in the GEMM fusion must be allocated contiguously in GPU memory. A given memory allocation strategy may thus permit only a subset of fusion choices; often, the forward pass and backward pass give rise to different fusion choices, perhaps requiring conflicting memory allocations. A large number of conflicts can be resolved statically. For example, if the conflict between two fusion groups is because of a single tensor, we just remove the offending node from the fusion groups; thus, both fusion groups can be simultaneously supported. However, some conflicts are non-trivial (*e.g.*Figure 1). Simply selecting the larger fusion group is inefficient given diminishing marginal returns from larger fusion groups.

Astra uses a measurement driven approach to select among conflicing memory allocation strategies. We introduce a high level fork in the exploration space pertaining to the allocation strategy. While exploring an allocation choice, we restrict the adaptation for fusion groups whose tensors are not contiguously allocated. The profiling results also record the allocation choice. After each allocation is explored, we build the best configuration for each allocation and then compare their end-to-end times.

4.5.3 Barrier exploration

When adapting the state space for stream scheduling, *Astra* needs to vary the assignment of individual kernels to different streams, and also vary the dispatch order. Interestingly, the mapping of different kernels to streams is no longer independent, as it is history-sensitive; the previous set of kernels scheduled on each of the streams affects the choice of which stream is the best fit for the present kernel. In the extreme case, this can remove the parallelism from the entire exploration of the streaming state space as the history builds up over the complete graph.

Astra addresses this by introducing the notion of *barrier exploration*. The enumerator statically partitions the data flow graph into super-epochs; a super-epoch is calibrated to be roughly a few milliseconds worth of computation time on the GPU (estimated based on the static flops calculation). At super-epoch boundaries, we introduce a forced barrier synchronization across all streams to *reset* their state. Thus, at the start of every super-epoch, the history of the stream is empty and hence exploration of multiple super-epochs can proceed in parallel. The coarse granularity of the super-epoch enables amortizing the cost of cross-stream barrier synchronization. Within a super-epoch we still need to perform exhaustive exploration as it is history sensitive.

4.5.4 Prefix exploration

To further control the state space within a super-epoch, the enumerator breaks up a super-epoch into multiple epochs, based on the dependency relationships. Operations within the epoch can be scheduled across multiple streams, with dependencies enforced through GPU synchronization events. In order to be history-aware of the effect of prior epochs within the super-epoch, we use a prefix-building approach for state-space exploration across epochs; the first epoch does its exploration, decides on the ideal stream mapping and then freezes the configuration chocies for the first epoch.

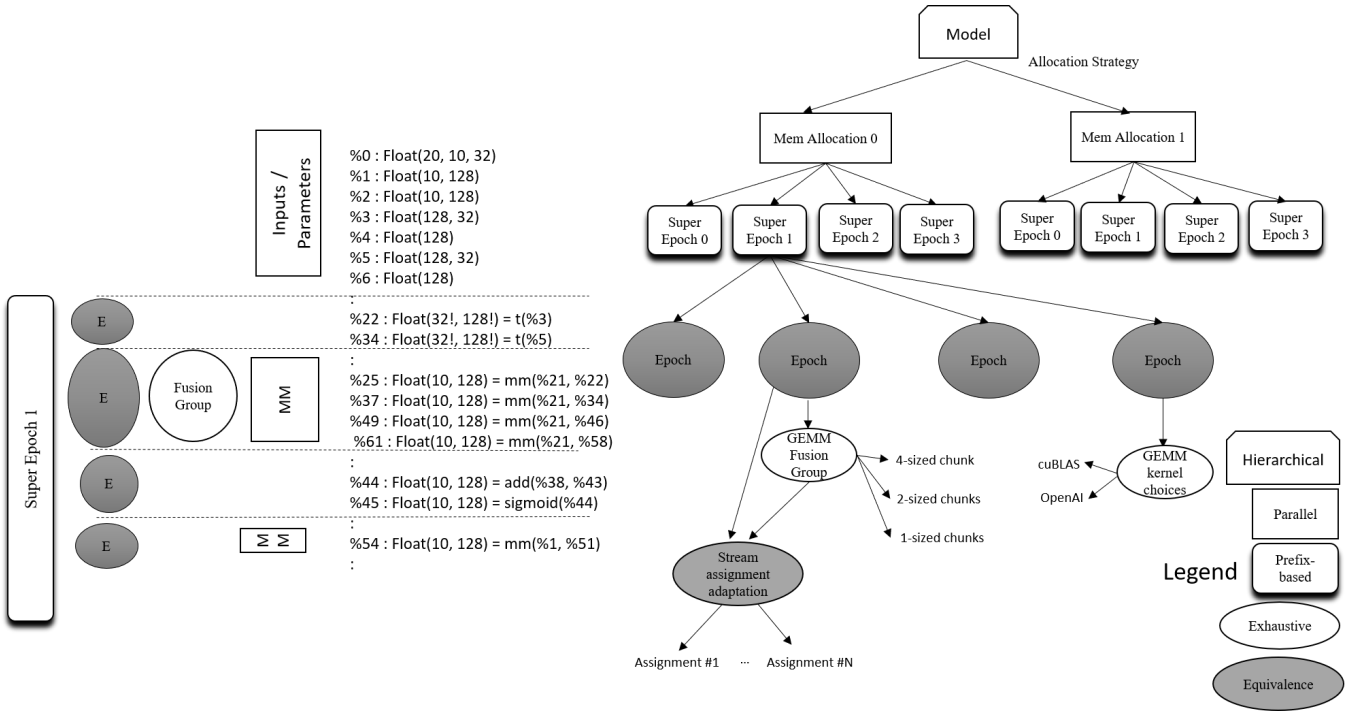

Figure 2. *Astra* Exploration. *This figure shows the different levels of exploration in Astra. At the left is a zoomed in version of Super Epoch 1 from the figure on the right. Legend at the bottom right shows what the various shapes mean.*

Next, the exploration for the second epoch happens, and so on. Therefore, we make the exploration additive in the number of epochs, thus significantly reducing the state space.

4.5.5 Equivalence exploration

While barriers and prefixes help cut down the state space significantly, the exploration within an epoch is still exhaustive. While most epochs are small (fewer than 5 kernels), there are epochs that are larger. To handle this, we introduce the notion of an equivalence class among kernels by making use of static knowledge of the dependency relationships of the operations, and the scope of the operations from the high level code for the model. Intuitively, if a group of GEMMs are of the same shape, and have similar inbound and outbound dependencies in the DFG, they can be treated as equivalent. Hence if there are 10 of those kernels in an epoch that need to be scheduled across 2 streams, we only need to adapt the *number* of operations to schedule in each stream (*e.g.*, 3 in first stream, 7 in second, etc. for a total of 5 choices) rather than *which exact* operations to schedule in each stream. In this example of 10 kernels, the equivalence exploration thus cuts down the state space from 2^{10} to just 5 within that epoch.

4.6 Profile indexing

The mechanism that *Astra* uses to manage different forms of exploration, is intelligent indexing of profile data, and mangling the key to this index helps dynamically control whether to re-run an instance of the exploration or not. For example, when performing parallel exploration, the key for the profiling of a GEMM will only contain the identifier for the GEMM. Hence if there are 3 choices for GEMM kernels, that GEMM will only need 3 iterations to pick its best value. If there are higher level dependencies such as the allocation strategy or stream mapping based on which the measurement for the GEMM has to be invalidated, those dependencies are added as prefixes to the indexing key of the corresponding profile, so that when the custom-wirer explores a different binding of the higher-level policy, there is a miss in the profile index, and it re-evaluates that instance.

4.7 Custom Wirer

The custom wirer takes the configurations generated by the enumerator, and builds the symbol tables for all nodes in the graph. The custom wirer then runs the forward and backward passes and updates the profile index. To update the profile index, the get_profile_value method is called on all the AdaptiveVariables. Each variable looks at all the profiled points and outputs one metric that the custom wirer tries to minimize. For GEMM nodes, the metric is simply the execution time. For fused GEMM ndes, it is the total time for all GEMM nodes. For epochs during stream adaptation, the metric is the time from the start of the super epoch to the end of all kernels dispatched in all streams till the end of present epoch. The stored configuration for a node includes the value

of that node, and of all its children. After updating the profile index, the custom wirer invokes the iterate() interface on the top-level adaptive variables to drive exploration according to the exploration mode annotated in the adaptive variable.

Some nodes in the update tree are constrained by the values of other nodes. For example, fusion groups can only be iterated through the fused tensors that are present in the current allocation strategy. The keys of the configuration which constrain a node forms the *context* of a node. While updating the profile index, each adaptive variable maintains a best configuration per context. Once the initial fine-grained profiling is done, the custom wirer builds the best configuration for each contexts and runs them; it selects the context (*e.g.*. the allocation strategy) that is fastest.

4.8 Using static knowledge

In addition to the pruning techniques described above, we also use coarse-grained static domain knowledge that does not require a detailed cost model. For example, when identifying candidates for fusion, static knowledge helps impose a range for the size of the GEMM fusion groups, as a fusion group of more than a certain size will give diminishing returns. Similarly, static knowledge is used to *constrain* the runtime exploration by giving it policies or *objective functions* to optimize – *e.g.*, in scheduling kernels across streams, roughly balance the amount of work (flops) scheduled on the different streams to ensure good utilization.

5 Implementation

We have implemented *Astra* in two popular deep learning frameworks - PyTorch [25] and Tensorflow [1]. In this section, we describe key aspects of the implementation.

5.1 Architecture

There are several high-level components in a DL framework such as PyTorch or Tensorflow. The input to the toolkit is a model file typically written in python. From the model, the toolkit builds a data flow graph. While the graph building happens by default in Tensorflow, PyTorch supports dynamic execution where the python interpreter handles control flow. We use an experimental feature called *tracing* in PyTorch that generates an execution graph like Tensorflow.

Another important component in a DL toolkit is the *automatic differentiation* module. To implement back-propagation for gradient descent, these toolkits automatically generate code for the differentiated versions of the forward pass computation. The user model simply specifies the forward pass computation, and the toolkit generates the backward pass code. From a training time perspective, roughly two-thirds of the computation happens during the backward pass.

Once the execution graph is generated, the toolkit invokes the *dispatcher* which iterates through the graph in data flow order, and uses the low-level libraries (for GEMM, etc) to asynchronously schedule the operations on the GPU. Once the GPU is done with execution of the kernels, the dispatcher is notified, and it proceeds to the next mini-batch.

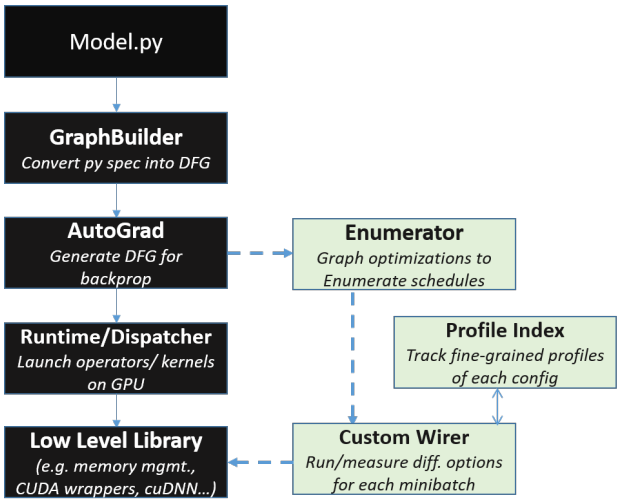

Figure 3. *Astra* interposes at the dispatcher of existing DL toolkits. Light-colored boxes are components that *Astra* adds.

Astra is built in such a way that most of the above functionality provided by the DL toolkit is reused. It interposes at the dispatcher layer of the toolkit as shown in Figure 3 and substitutes its own dispatch module. The modules implemented by *Astra* are an *enumeration* module which is the offline compilation/enumeration phase, and a *custom wiring* module which picks different instances of the enumerated space and executes that incarnation of the DFG by dispatching operations. Because *Astra* fully controls the dispatcher, it is not limited by the inability of the framework to use multiple streams; *Astra* performs its own stream management and interacts with the GPU directly, by leveraging the low-level libraries that the framework already has.

5.2 Profiling

Lightweight fine-grained profiling is a key requirement for the online exploration to work efficiently. Traditional approaches such as CUDA callbacks that are enabled by the Nvidia CUPTI layer, are too fine-grained because every kernel run on the GPU generates a callback in the critical path, whereas we need to control the granularity of profiling to only include regions of interest in order to amortize the profiling overhead. In *Astra*, we wrap the regions of interest or kernels between a pair of *cudaEvents*. During the offline stage, the profiler registers event pairs for such nodes and adds them to the profile index with the appropriate key. In the critical path, the runtime only needs to mark the events for the node. Around a super-epoch boundary, these events are global events, that synchroize across streams. For epochs and fusion sets, these events are stream-local events.

Mini-batch	PyT	$Astra_F$	$Astra_{FK}$	$Astra_{FKS}$	$Astra_{all}$
8	1	1.65	1.65	2.13	2.27
16	1	1.65	1.69	2.11	2.22
32	1	1.49	1.48	1.72	1.81
64	1	1.20	1.23	1.42	1.49
128	1	1.03	1.05	1.19	1.2
256	1	0.98	1.01	1.1	1.12

Table 2. Performance of SCRNN Model Factor speedup relative to native PyTorch for *Astra*

Mini-batch	PyT	$Astra_F$	$Astra_{FK}$	$Astra_{FKS}$	$Astra_{all}$
8	1	2.25	2.15	2.43	2.43
16	1	1.93	1.93	2.15	2.13
32	1	1.65	1.68	1.85	1.85
64	1	1.29	1.31	1.46	1.46
128	1	1.13	1.15	1.23	1.23
256	1	1.2	1.21	1.28	1.28

Table 3. Performance of Hutter MI-LSTM Model Factor speedup relative to native PyTorch for *Astra*

Mini-batch	PyT	$Astra_F$	$Astra_{FK}$	$Astra_{FKS}$	$Astra_{all}$
8	1	2.33	2.37	2.79	3
16	1	2.18	2.19	2.65	2.75
32	1	2	1.98	2.3	2.4
64	1	1.64	1.71	1.85	1.95
128	1	1.34	1.35	1.51	1.54
256	1	1.18	1.17	1.29	1.29

Table 4. Performance of subLSTM Model Factor speedup relative to native PyTorch for *Astra*

5.3 Element-wise Fusion

In addition to GEMM fusion, *Astra* also fuses element-wise operations. In PyTorch, *Astra* picks the set of operations to fuse based on data dependencies, and uses the experimental JIT support to compile the fused operations. In Tensorflow, the XLA framework is used for fusion and compilation.

5.4 Limitations of the Tensorflow prototype

Our Tensorflow prototype performs adaptation only for GEMM fusion and GEMM kernel selection. Because the low-level runtime of Tensorflow expects contiguous tensors (*i.e.*, does not handle strided access of tensors), GEMM fusion incurs additional tensor copies, which complicates stream adaptation as it causes super-epoch and epoch definitions to vary. Addressing this issue in tensorflow is ongoing work.

5.5 Handling dynamic graphs

PyTorch supports dynamic graphs: depending on the length of input (*e.g.*, the number of words in input sentences), it generates a different graph that has the right shape to accommodate that mini-batch. With dynamic graphs, mini-batches are no longer identical as the computation depends on the size of the inputs within a mini-batch. *Astra* handles this by bucketed profiling; it bucketizes the input sizes into a small number of buckets (currently 5) and performs the state space exploration independently within each bucket. The profile index key is also prefixed by the bucket size, increasing the state space by 5x. To avoid memory reallocation costs as the exploration switches buckets, *Astra* allocates memory corresponding to the largest bucket size and uses a cache indexed by tensor shape to reuse slices of the respective tensor buffers when running for lower-sized buckets.

6 Evaluation

We answer four key questions in the evaluation of *Astra*:

- How much does *Astra* speed up end-to-end models?
- How close does *Astra* get to hand-optimization (cuDNN)?
- What is the size of exploration state space in *Astra*?
- How does *Astra* compare to static optimization (XLA)?

6.1 Experimental Setup

All experiments were run on a single Tesla P100 GPU with a peak compute bandwidth of 9 teraflops/sec. The five models we evaluate in this section are (a) MI-LSTM [36] on the Hutter Challenge dataset, (b) SC-RNN [22] on the Penn Tree Bank dataset, subLSTM [8] on the Penn Tree Bank dataset, (d)Stacked LSTM for language modelling on the Penn Tree Bank dataset, (e) Google Neural Machine Translator [35]. Unless otherwise specified, the numbers pertain to our PyTorch implementation of *Astra*. We evaluate these models under various mini-batch sizes, although mini-batch sizes above 32 are rarely used in experimental long-tail models that *Astra* targets, as they typically affect accuracy [19, 20]. To break-up the benefits of the various parts of *Astra*, we report four numbers for *Astra*: $Astra_F$ has only GEMM fusion adaptation, $Astra_{FK}$ has GEMM fusion and kernel adaptation enabled, $Astra_{FKS}$ also has streams enabled, and finally $Astra_{all}$ shows the performance with adaptation of memory allocation as well. All our baseline numbers are with standard and efficient libraries for tensor operations (such as cuBlas). PyTorch 0.4 (with CUDA v8.0/cuDNN v6.0) and Tensorflow version 1.8 (with CUDA v9.2/cuDNN v7.0) were used for the baseline measurements.

6.2 Speedup in end-to-end models

For each model, we compare (a) native implementation in pyTorch (b) Pytorch+cuDNN where applicable (c) *Astra*.

Table 2 shows the speedups for *Astra* under various batch sizes on the SC-RNN model [22] normalized to the performance of the native PyTorch. As can be seen, *Astra* provides

Mini-batch	PyT	cuDNN	$Astra_F$	$Astra_{FK}$	$Astra_{all}$
8	0.43	1	0.87	0.89	1.09
16	0.59	1	1.1	1.11	1.32
32	0.86	1	1.43	1.46	1.64
64	0.69	1	0.94	0.95	1.05
128	0.76	1	0.92	0.93	1
256	0.79	1	0.94	0.94	1.02

Table 5. Performance of PTB Stacked LSTM Model Performance relative to cuDNN for *Astra*

Mini-batch	PyT	cuDNN	$Astra_F$	$Astra_{FK}$	$Astra_{all}$
8	0.19	1	0.58	0.55	0.65
16	0.23	1	0.66	0.59	0.75
32	0.3	1	1.54	1.37	1.71
64	0.23	1	0.95	1.01	1.17
128	0.26	1	0.9	0.94	1
256	0.31	1	0.87	0.91	1.02

Table 6. Performance of GNMT model Performance relative to cuDNN for *Astra*

Model	No. of configs	
	$Astra_{FKS}$	$Astra_{All}$
PTB SCRNN	303	1672
PTB Stacked LSTM	1219	1219
MI-LSTM	1191	1191
PTB SubLSTM	3207	5439
GNMT	2280	9303

Table 7. Size of exploration state space post-pruning. Each configuration is explored in one mini-batch of training

Model	Dynamic Graph	*Astra* + bucketing
SCRNN-16	1	1.61
SCRNN-32	1	1.43
subLSTM-16	1	2.47
subLSTM-32	1	2.13
StackedLSTM-16	1	2.44
StackedLSTM-32	1	2.22

Table 8. Speedup from *Astra* bucketed adaptation compared to dynamic graphs in Native PyTorch.

a speedup of up to 1.86x. Also, while for smaller batch sizes the kernel library selection doesn't yield a benefit, it gives a significant benefit at batch-size 64. Finally, stream adaptation provides 15-23% improvement on top of GEMM fusion and kernel selection. Table 3 and 4 shows the performance on the MI-LSTM model on Hutter [36] and sub-LSTM [8], with speedups up to 3x.

6.3 Comparison with cuDNN

Table 5 shows the performance of *Astra* on the PTB Stacked LSTM model under the "large" configuration (input size of 1500). This model is fully covered by cuDNN accelerator, so it gives a sense of how close to the hand-optimized version *Astra* performs. Interestingly, *Astra* is able to outperform cuDNN with the streaming configuration enabled, while native PyTorch is worse. Table 6 shows the Google Neural Machine Translator [35], which is mostly covered by cuDNN except the Attention module. *Astra* performs close to cuDNN performance, outperforming it in some configurations.

6.4 Size of exploration state space

Table 7 lists the number of configurations explored in the various models. As can be seen, the state space is a few thousand. Interestingly, as a result of techniques such as barrier exploration (detailed in Section 4.5), the state space for the GNMT model is similar to the other models, despite GNMT having about 8x more layers, pointing to the scalability of the exploration approach. A related metric is the overhead of profiling, as the profiling in *Astra* runs as part of regular training. The overhead of our profiling is $< 0.5\%$ for all the models evaluated. Hence it can be always on.

6.5 Dynamic graphs

As discussed in § 5.5, variable shapes of the input tensor violate the predictability assumption of *Astra*. *Astra* handles such models by bucketing the input shapes into a fixed number of buckets, and then performing adaptation separately for each bucket configuration. Table 8 compares the performance of *Astra* with bucketing, with a purely dynamic graph implementation in native PyTorch. We used 5 buckets in this experiment, calibrated on the distribution of input sentence lengths in the PTB dataset, which resulted in buckets of input lengths 13,18, 24, 30, and 83. As can be seen, *Astra* benefits even such input-variable models by a significant margin, despite performing a small amount of extra computation as a result of mapping to the nearest larger bucket.

6.6 Comparison with XLA

In this sub-section, we evaluate the tensorflow implementation of *Astra*. As described in § 5.4, the TF implementation does not support stream adaptation, so we only compare $Astra_{FK}$. For each model, we compare with both native Tensorflow v1.8 performance and XLA-optimized performance. Surprisingly, we noticed that the XLA implementation was *worse* than native tensorflow for many of the models (*e.g.*, 3x worse for SCRNN, 1.5x worse for subLSTM), which turned out to be because XLA handles *embeddings* poorly, resulting in multiple transitions between CPU and GPU for lookups;

Model	TF	TF + XLA	$Astra_{FK}$	cuDNN
SCRNN (16)	1	1.28	**1.58 (1.23)**	-
SCRNN (32)	1	1.11	**1.66 (1.49)**	-
MI-LSTM (16)	1	0.98	**1.69 (1.72)**	-
MI-LSTM (32)	1	1.31	**1.51 (1.15)**	-
SubLSTM (16)	1	1.42	**1.92 (1.35)**	-
SubLSTM (32)	1	1.39	**1.71 (1.22)**	-
Stack. LSTM (16)	1	1.45	**1.45 (1.0)**	1.38
Stack. LSTM (32)	1	1.41	**1.32 (0.95)**	0.88
GNMT (16)	1	1.19	**2.0 (1.68)**	2.35
GNMT (32)	1	1.17	**1.49 (1.27)**	2.23

Table 9. Comparison of *Astra* in Tensorflow with XLA. Factor speedups relative to native TF. Relative improvement from XLA shown in parantheses

many of our models use embeddings. This suggests a fundamental robustness concern with static approaches such as XLA in handling long-tail scenarios, which is perhaps why XLA is still experimental even after 2 years of launch. *Astra*'s measurement-based dynamic adaptation approach enables turning off any optimization such as fusion at a fine-grained level (*e.g.*, within a super-epoch). Nonetheless, for a more insightful comparison with XLA, we evaluate a slight variant of the models with the embedding operation alone removed. On native TF, the per-minibatch times with and without embedding were within 10% of each other for many of the above models, so these numbers are meaningful from a performance viewpoint.

Table 9 shows the results. For space reasons, we only report numbers with two mini-batch sizes:16 & 32. As can be seen, $Astra_{FK}$ achieves a speedup of up to 70% on top of XLA and an average speedup of about 25-30%, despite not supporting multi-stream adaptation.

6.7 Discussion

We have not reported accuracy numbers in our evaluation, as all optimizations explored in this paper are *value-preserving* optimizations and thus do not affect accuracy.

Our evaluation has focused on recurrent neural networks that are widely used in speech recognition and language understanding tasks, and according to Google, account for a much larger fraction of production workload than CNN models [14]. These are also the models that have the biggest gap in performance as the individual operations are small. However, with faster hardware, (*e.g.*, the Volta GPUs run at 120 teraflops of 16-bit precision ops [32]), even operations such as convolution become "cheap" and hence would benefit from techniques such as cross-layer fusion and using multiple streams. This points to a general strength of the exploration approach that *Astra* takes. As model structures and GPU architectures evolve, all one needs to do is add to the library of exploration, and models get automatic robust speedup without any need for hand-optimization or parameter tuning.

The limited number of exploration dimensions in the *Astra* prototype is only illustrative; several other dimensions listed in Section 3.4 naturally fit into the *Astra* approach. For example, although the present adaptation dimensions in *Astra* deal with only a single GPU, they will also benefit multi-GPU jobs by running each instance faster. The deterministic adaptation aspect of *Astra* can be extended to explore dimensions such as specifics of model-partitioning and data partitioning in multi-GPU jobs, but a detailed discussion of those techniques is beyond the scope of this paper.

Our key contribution is the generic approach to navigate a large state space online. ,This approach is complementary to the performance benefits from kernel synthesis approaches [6, 30].

7 Implications for hardware design

While the present prototype of *Astra* works on GPU, the key benefit of the *Astra* approach is that it can be easily made to run on top of new hardware [14, 15, 21] without building intricate cost models for it. However, there are a few basic properties (that GPUs already support) needed from the hardware in order to enable *Astra*; we list these below:

Predictable execution: Fine-grained profiling is the key mechanism that *Astra* uses to perform parallel exploration which is crucial to pruning the exploration state space (§ 4.5). For this to be feasible, the measurements of individual operations and kernels need to be repeatable at a fine-grained level. Given the simple in-order cores of GPUs, such repeatability existed even at the granularity of a single GEMM operation. Autoboost of clock speed in GPU violates this assumption and causes variance, so we set the frequency to the base clock value in our experiments, via `nvidia-smi`. Autoboost did not provide a measurable benefit to our workloads anyway compared to the static clock setting, but the static clock was key to enabling the wins from *Astra*. Supporting predictable execution is thus a key reqiurement from the hardware.

Lightweight profiling events: Fine-grained profiling needs to be done with low overhead so that it can be always on during job execution. GPUs have a simple and lightweight abstraction of cuda events and a simple API to get elapsed time between two events. A similar fine-grained profiling primitive that is low overhead, is another capability that hardware for DNNs need to support in order to enable *Astra*.

8 Related Work

Runtime adaptation across multiple choices produced by the compiler, has been explored for generic programs, with limited success. The body of work on multi-version compilers [17, 33, 38] explores the idea of generating multiple versions of "hot" functions, from which the runtime picks

the best-performing version. With generic programs, the main challenge of this approach that has limited its adoption/practicality, is the dependence of execution on the actual input; for instance, a function to sort may have widely different runtimes depending on the size of the input list. To counter this, such systems need to take multiple (often thousands of) measurements for each optimization choice [17], and hence can afford this approach only for the most critical functions. Adaptation of query execution has also been explored [2, 5]. Again, the variance due to data distribution skews, selectivity variance, *etc.*, means that they can only perform coarse-grained adaptation in a very small state space. *Astra*, on the other hand, can afford to be a lot more aggressive with whole program-level adaptation and reasoning about interacting optimizations across the data-flow graph, because of the unique mini-batch-level predictability of DNN jobs. We believe that the ability to exploit such predictability for this particular domain, significantly amplifies the benefit from the multi-version approach, besides making it pragmatic.

There is a large body of work on domain-specific compilers similar to how *Astra* targets code generation for deep learning. One of the early influential works in this space is FFTW [10], where the authors build a framework for generating highly optimized kernels for computing fast-fourier-transform, a popular operation in signal processing. Halide [26] is another compiler built for generating optimized implementations of image processing kernels. More recently, Tensor comprehensions [30] built on the Halide framework and combined it with Polyhedral compilation [16] to achieve optimized kernel generation for deep learning kernels. XLA [18] from Tensorflow is another example of a domain-specific compiler that auto-optimizes code for deep learning with a static approach. *Astra* differs from this body of work in two dimensions: (a) it achieves optimization as a co-operative task between the compiler and the runtime, instead of the traditional model of the compiler generating one optimized code; (b) *Astra* takes an end-to-end view of the whole (large) program as opposed to specific small kernels, and hence needs a more scalable approach to space exploration and reasoning about conflicting optimization choices at the whole graph granularity.

The approach of adaptation to drive optimization has been tried in the form of autotuning algorithms in the area of scientific computation. The Halide OpenTuner [3] uses a genetic algorithm to perform mutations on the generated code and uses measurement to decide the best variant of the code. Unlike OpenTuner which deals with a small kernel and hence can hope to converge to the optimal code with such a randomized approach, *Astra* deals with a much larger state space. The key novelty in the adaptation approach of *Astra* is using fine-grained profiling and systems reasoning coupled with static knowledge, to achieve aggressive pruning, and consequently a more systematic and comprehensive exploration of the state space. This in turns allows *Astra* to scale to much larger code such as a complex full training job.

TVM [6] is a recent system that targets optimizing DNNs using a combination of kernel synthesis and a machine-learning approach to autotuning. While TVM is focused on autotuning kernels, the adaptation in *Astra* is end-to-end (*e.g.*, dealing with model-wide memory allocation strategies, stream assignment *etc.*). Further, TVM is targeted at inference and doesn't handle training of DNNs, which *Astra* targets. Another system that uses machine-learning to adapt configurations is Google's data placement model [23]; we believe the systems-approach of *Astra* to manage the state space is simpler to reason about. *Astra* can transparently speed up execution of any unmodified training job that runs on Tensorflow and PyTorch.

9 Conclusion

Astra addresses a pressing need in machine learning experimentation to iterate fast on new model architectures in order to make advances. While accelerators such as cuDNN significantly speed up training of deep learning jobs, they are hand-optimized and hence only cater to popular models. *Astra* bridges this gap by bringing the power of optimization to long tail models, by adopting a novel division of functionality between the compiler and runtime, where the runtime adaptively explores the state space of optimizations by leveraging the unique repetitiveness and predictability of a deep learning training job. With fine-grained profiling and several techniques to perform the exploration in parallel, *Astra* effectively prunes the state space, unlike probabilistic or learning-based approaches to adaptation. The *Astra* approach is particularly attractive given the frantic pace of new custom hardware being built for DNNs, which make static optimization expensive. We believe *Astra* is an example of how tight integration of the systems layer such as compiler to a specific large workload can drive fundamental efficiencies with unconventional yet effective architectures.

Acknowledgements

We thank the anonymous reviewers for their valuable comments and suggestions. We thank Ramachandran Ramjee and Nipun Kwatra for their feedback on earlier drafts of this paper. We also thank Chandu Thekkath, Subir Sidhu, and Daniel Li from the Microsoft AI Platform team for their valuable feedback and inputs on the project, besides providing access to the GPU clusters and Azure GPU VMs.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 207–218. ACM, 2013.

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, pages 303–315. IEEE, 2014.

[4] Jeremy Appleyard, Tomás Kociský, and Phil Blunsom. Optimizing performance of recurrent neural networks on gpus. *arXiv Preprint*, abs/1604.01946, 2016.

[5] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2004.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[8] Rui Costa, Ioannis Alexandros Assael, Brendan Shillingford, Nando de Freitas, and TIm Vogels. Cortical microcircuits as gated-recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 272–283, 2017.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[10] Matteo Frigo. A fast fourier transform compiler. In *Acm sigplan notices*, volume 34, pages 169–180. ACM, 1999.

[11] Scott Gray. Open single and half precision gemm implementations, 2017.

[12] Geoffrey E Hinton. Learning distributed representations of concepts. In *Proceedings of the eighth annual conference of the cognitive science society*, volume 1, page 12. Amherst, MA, 1986.

[13] Nvidia Inc. Nvidia tesla p100 gpu accelerator, 2016.

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

[15] Simon Knowles. Graphcore: Scaling throughput processors for machine intelligence. *URL https://www.matroid.com/scaledml/2018/simon.pdf*.

[16] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. When polyhedral transformations meet simd code generation. In *ACM Sigplan Notices*, volume 48, pages 127–138. ACM, 2013.

[17] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online performance auditing: using hot optimizations without getting burned. In *ACM SIGPLAN Notices*, volume 41, pages 239–251. ACM, 2006.

[18] Chris Leary and Todd Wang. Xla: Tensorflow, compiled. *TensorFlow Dev Summit*, 2017.

[19] Yann Lecunn. Training with large minibatches is bad for your health. more importantly, it's bad for your test error. friends dont let friends use minibatches larger than 32. *URL https://twitter.com/ylecun/status/989610208497360896?lang=en*, 2018.

[20] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *CoRR*, abs/1804.07612, 2018.

[21] Microsoft. Real-time ai: Microsoft announces preview of project brainwave. *URL https://blogs.microsoft.com/ai/build-2018-project-brainwave/*.

[22] Tomas Mikolov, Armand Joulin, Sumit Chopra, Michael Mathieu, and Marc'Aurelio Ranzato. Learning longer memory in recurrent neural networks. *arXiv preprint arXiv:1412.7753*, 2014.

[23] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. *CoRR*, abs/1706.04972, 2017.

[24] CUDA Nvidia. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15(27):31, 2008.

[25] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration, 2017.

[26] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[27] Herbert Robbins and S Monro. ªa stochastic approximation method, º annals math. *Statistics*, 22:400–407, 1951.

[28] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[29] Anil Thomas Scott Leishmann, Alex Park. Intel nervana reference deep learning framework committed to best performance on all hardware, 2017.

[30] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[31] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu. Zorua: A holistic approach to resource virtualization in gpus. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, Oct 2016.

[32] Inside Volta. The worldâĂŹs most advanced data center gpu. *URL https://devblogs. nvidia. com/parallelforall/inside-volta*.

[33] Michael J Voss and Rudolf Eigemann. High-level adaptive program optimization with adapt. In *ACM SIGPLAN Notices*, volume 36, pages 93–102. ACM, 2001.

[34] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. *arXiv preprint arXiv:1801.04380*, 2018.

[35] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system:

Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[36] Yuhuai Wu, Saizheng Zhang, Ying Zhang, Yoshua Bengio, and Ruslan R Salakhutdinov. On multiplicative integration with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 2856–2864, 2016.

[37] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[38] Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, and Graham Yiu. Space-efficient multi-versioning for input-adaptive feedback-driven program optimizations. In *ACM SIGPLAN Notices*, volume 49, pages 763–776. ACM, 2014.

[39] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. *arXiv preprint arXiv:1607.03474*, 2016.