Methods for Accelerating Machine Learning in High Performance Computing

Robert Lim

Abstract

Driven by massive dataset corpuses and advances and programmability in accelerator architectures, such as GPUs and FPGAs, machine learning (ML) has delivered remarkable, human-like accuracy in tasks such as image recognition, machine translation and speech processing. Although ML has improved accuracy in selected human tasks, the time to train models can range from hours to weeks. Thus, accelerating model training is an important research challenge facing the ML field. This work reports on the current state in ML model training, both from an algorithmic and a systems perspective by investigating performance optimization techniques on heterogeneous computing systems. Opportunities in performance optimizations, based on parallelism and locality, are reported and sheds light on techniques to accelerate the learning process, with the goal of achieving on-the-fly learning in heterogeneous computing systems.

1 Introduction and Motivation

Machine learning has exceeded human capabilities in areas, such as image recognition, natural language processing and competitive gaming, which is attributed to the plethora of massive datasets, the advances in high performance computing architectures and the models that learn to extract latent features from the datasets and create knowledge representations and inferencing from those features. For instance, a relational network of entities consisting of people, places and organizations can be constructed that describes a sequence of events for intelligence gathering and storytelling [20]. In addition, advances in deep learning (DL) for neural machine translation [22] have lowered communication barriers and is transforming interactive conversations with computer-aided translation, which has sociological and economical impacts in society. However, the existing approaches in model training are not sufficient for maximizing performance. Programming a GPU accelerator that targets the available hardware resources is difficult, considering the number of multiprocessors, compute cores and wavefronts, in addition to effectively writing parallel programs in the single-instruction multiple thread (SIMT) model. An automated and guided approach towards generating high performing ML code in an intuitive manner for accelerator architectures is needed that accounts for the run time behavior of the underlying architecture.

Multiple unique types of challenges for GPU performance optimization exist. GPUs use the SIMT programming model, where control flow divergence may hurt parallelism, since serialization is required for threads that share the same program counter in different branch paths. Due to memory divergence caused by irregular or strided memory access patterns, GPU performance can be significantly degraded. Given the limited cache size on a GPU, a large number of threads can easily compete for cache resources without efficient cache management strategies. Optimizations at the level of cooperative thread arrays (CTA) cannot easily be conducted without clear guidance, especially when dynamic parallelism is involved.

Training is painfully slow because neural networks perform highly irregular, non-uniform computations that depend on individual instances. The cost of a matrix-vector product is dominated by
the cost of loading weights from DRAM, which is orders of magnitude slower than peak compute on both CPUs and GPUs. In any learning algorithm, a more detailed model improves accuracy up to a certain point, but will overfit and become a system that merely memorizes every item in the training set, whereas a small model may fail to capture interesting and relevant attributes. Thus, the optimization and convergence criteria directly affects the accuracy and robustness of the trained model, which factors into the overall time to train.

Forward and backward propagation are employed in training a deep neural network. Under forward propagation, input is fed through the network to produce output activations. Outputs activations are propagated backwards through the network to calculate the differences between the input and output values of each neuron, referred to as gradients, which are used to update the weights of the neuron. The network is trained on batches of input items, instead of a single input, to improve vectorization and parallelization. The solver coordinates the forward, backward and weight update phases of training. The objective is to learn the best parameters for a given dataset. Typical search methods used for model training include stochastic gradient descent and conjugate gradient descent.

Large networks can take days or weeks to converge, motivating distributed training algorithms that leverage parallelism opportunities. Model and data parallelism are two strategies for parallelizing training in a distributed memory environment. Data parallelism replicates the network across multiple workers, where each worker processes a different batch of inputs. Model parallelism shards a network across multiple workers, where each worker is responsible for a portion of a model and communicates updates with the other workers. Various schemes are devised for synchronizing network instances, including gradient summation, parameter averaging and parameter servers.

This report is organized as follows. The problem space is introduced in Section 2.2. Machine-independent and architecture-specific optimizations are discussed in Section 3. Intermediate code generation is presented in Section 4. Section 5 covers distributed approaches for model training. Section 6 summarizes and concludes with future work.

2 Problem Space

This section formulates the computation of a learning algorithm for the back end compiler through the use of operations, iteration spaces, performance modeling and an intermediate representation.

2.1 Data Operations

The operations performed in machine learning include linear algebra and tensor decompositions, to name a few [36].

2.1.1 Linear Algebra

GEMM Abstractions CUTLASS is a collection of CUDA C++ templates and abstractions for generalized matrix multiplication (GEMM) [9]. GEMM computes \( C = \alpha A \ast B + \beta C \), where \( A, B \) and \( C \) are matrices, \( A \) is \( M \times K \), \( B \) is \( K \times N \) and \( C \) is \( M \times N \). However, thrashing will occur as the dimension sizes increase. \( C \) can be partitioned into tiles of size \( M_{tile} \times N_{tile} \) that fit in on-chip memory, where the outer product is applied to each tile.

CUTLASS applies a hierarchical tiling structure by decomposing the computation into thread block tiles, warp tiles and thread tiles and applying multiply-accumulate operations. Each thread block computes its part of the output GEMM by iteratively loading blocks of matrix data from input matrices and computing the accumulated matrix product \( (C += A \ast B) \). The block tiles are further partitioned into warps, which are groups of threads that execute together in SIMT. Block items
Figure 1: An individual thread participates in a warp-level matrix product by computing an outer product of a fragment of $A$ and a fragment of $B$ held in registers (left). The CUDA WMMA API provides warp tile abstractions for loading matrix fragments and performing matrix multiply-accumulate (right).

{$X, Y, K$} are compile-time constants that the programmer specifies for a target processor and the data type. Once data is stored in shared memory, each warp computes a sequence of accumulated matrix products by iterating over $K$ dimensions of thread block tiles, loading submatrices from shared memory and computing the accumulated outer product.

In Figure 1, the upper-left quadrant, shaded in gray, corresponds to 32 threads in a warp. Multiple threads within the same row or same column fetch the same elements of $A$ and $B$ fragments. To maximize compute intensity, a basic structure is replicated to form a full warp-level accumulator tile, yielding $8 \times 8$ overall thread tiles, computed from outer products of $8 \times 1$ and $1 \times 8$ fragments.

Warp Matrix Multiply-Accumulate API (WMMA) is a warp tile structure released in CUDA 9 to target the Volta V100 tensor cores. Each tensor core provides a $4 \times 4 \times 4$ matrix processing array and performs the operation $D = A \ast B + C$, where $A$, $B$, $C$, and $D$ are $4 \times 4$ matrices. Matrix multiplication inputs $A$ and $B$ are FP16 matrices, whereas $C$ and $D$ may be FP16 or FP32. WMMA API is an alternative to the thread tile structure. Rather than decomposing the warp tile structure into scalar and vector elements owned by individual threads, WMMA API provides an abstraction to the programmer for warp-cooperative matrix fragment load/store and multiply accumulate math operations. Figure 1 (R) shows a warp tile in the CUDA WMMA API. CUTLASS implements GEMM based on WMMA API, where the warp tile must have dimensions that are multiples of matrix multiply accumulate shapes defined by `nvcuda::wmma` templates. For CUDA 9, the default WMMA size is $16 \times 16 \times 16$.

High Performance GEMM  BLAS3-GEMM is a code generator for matrix multiplication kernels, which makes use of vectorization and cache levels for blocking [50]. A general multiply of block $A$ and panel $B$ (GEBP) kernel with $M_c, N_c, K_c, M_r, N_r$ tiles is supplied to define an optimized GEMM on a target processor. The $\mu$kernel generator accepts inputs of size $M_r \times N_r$ for $\mu$kernel and the data type of matrix elements. Instruction scheduling involves vectorization, data prefetching and addressing mode optimization.

Listing 1: Structure of blocked DGEMM.

```plaintext
// layer 1
for jj = 0:Nc-1:
    A0 = A; B0 = B[:][jj:jj+Nc-1]; C0 = C[:][jj:jj+Nc-1];

// layer 2
for kk = 0; Kc:K-1:
    A1 = A0[:][kk:kk+Kc-1]; B1 = B0[kk:kk+Kc-1]; C1 = C0;

// layer 3, macro-kernel (GEBP)
```
For vectorization, two loops that perform a rank-1 update are fully unrolled. \( C \) resides in registers as live-ins, where two vectors \( A \) and \( B \) are loaded from memory into registers across iterations of loop \( k \). For consecutive memory accesses, \( A'[:k] \) and \( B'[:k] \) are packed in contiguous buffers. Tile size \( M_r \times N_r \) is selected so that \( M_r \) is a multiple of vector length (VL), which is the maximum number of data elements in a vector register.

Since \( A_1 \) varies but \( B_3 \) stays unchanged in Listing 1, Layer 5, \( B_3 \) rows are prefetched several iterations in advance. \( A_1 \) is also prefetched and used in the next \( \mu \text{kernel} \) call, which represents the \( M_r \times K_r \) row block of \( A_3 \), with \( A_4 := A_3[i + M_r : i2M_r - 1][:] \). For each iteration of loop \( k \) in \( \mu \text{kernel} \), a column of \( A_4 \) is a multiple of vector length (VL), which is the maximum number of elements that can be loaded from memory into registers across iterations of loop \( k \).

Listing 2: Vectorized double precision \( \mu \text{kernel} \) on Sandybridge (left) and Cortex-A57 (right).

Linear Algebra Language  BLAC, or basic linear algebra computation, provides a language for linear algebra [46], where an output is a product, addition, transposition or scalar, such as \( y = A' x + a z \) or \( C = a AB + C + D \). sBLAC extends BLAC with structured matrices [47], which considers lower/upper triangular, symmetric, all-zero and unstructured matrices, denoted as \( L, U, S, Z \) and \( G \), where the expand set of operators with the triangular solve is written as \( x = L \backslash y \). For a given BLAC, operands have fixed sizes and data types. Optimized C code is generated from LL (linear algebra language) as input, based on tiling, vectorization, precision and a search strategy. A fully tiled BLAC in LL is rewritten into \( \Sigma \)-LL, which captures the explicit gather and scatter operators on matrices. A gather \( g \) extracts a smaller matrix from a matrix, whereas a scatter \( s \) writes a smaller matrix into a larger, all-zero matrix. Formally,
\[ g = [i, j]^m,n_{k,l} : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{k \times l}, \]
\[ A \mapsto Ag = A[i : i+k-1, j : j+l-1] \]

Scatter \( s \) is the dual of gather:
\[ s = k,l_{m,n}[i, j] : \mathbb{R}^{k \times l} \rightarrow \mathbb{R}^{m \times n}, \]
\[ A \mapsto sA \]

Given
\[ A_{i,j} = L_{i,k}U_{k,j} + S_{i,j}, \tag{1} \]

the input sBLAC is transformed using the \( SInfo \) and \( AInfo \) of matrices. \( SInfo \) associates regions of a matrix to structures, whereas \( AInfo \) associates regions of a matrix to information on how to access blocks in that region. A set of CLooG statements is produced, \( \langle \sigma, \rho, B \rangle \), where the domain is a polyhedral set \( \sigma \) representing iteration space, the schedule is a polyhedral map \( \rho \) that determines the traversal order of the domain and the body is a \( \Sigma \)-LL expression \( B \).

\[ s = \langle \sigma = \{(i, k, j) \mid k = 0 \land 0 \leq i < 4 \land 0 \leq j \leq i \}, \rho = ((i, k, j), (k, i, j)), \]
\[ B = [i, j](L[i, k]U[k, j] + S[i, j]) \rangle \]

\( SInfo \) and \( AInfo \) dictionaries are expanded into prisms:
\[ L.SInfo = G : \{(i, k, j) \mid 0 \leq i < 4 \land 0 \leq k \leq i \}, \]
\[ Z : \{(i, k, j) \mid 0 \leq i < 4 \land i < k \leq 4 \}, \]
\[ U.SInfo = G : \{(i, k, j) \mid 0 \leq k < 4 \land k \leq j \leq 4 \}, \]
\[ Z : \{(i, k, j) \mid 0 \leq k < 4 \land 0 \leq j \leq k \}, \]

\[ A.SInfo = S.SInfo = \{G : \{(i, k, j) \mid 0 \leq i, j < 4 \} \} \]

\( AInfo \) is computed as follows.
\[ L.AInfo = \{(i, k, j) \mid 0 \leq i < 4 \land 0 \leq k \leq i \} : ([i, k]^4_{1,1}, id) \]
\[ U.AInfo = \{(i, k, j) \mid 0 \leq k < 4 \land k \leq j \leq 4 \} : ([k, j]^4_{1,1}, id) \]
\[ S.AInfo = \{(i, k, j) \mid 0 \leq i < 4, 0 \leq j \leq i \} : ([i, j]^4_{1,1}, id) \]
\[ A.AInfo = \{(i, k, j) \mid 0 \leq i, j < 4 \} : ([i, j]^4_{1,1}, id) \]

\( StmtGen \) builds a set of statements for every operator from the sBLAC input. The first operation is LU that determines the iteration space, which is discussed in Section 2.2.
\[ iterSpace_{LU} = L.SInfo[G] \cap U.SInfo[G] = \{(i, k, j) \mid 0 \leq k < 4 \land k \leq i, j < 4 \} \]

To generate domains and bodies for operations recursively, compute the iteration space
\[ iterSpace = \{(i, k, j) \in \sigma \mid (\sigma : M) \in A.SInfo, M \neq 2 \} \]
\[ = \{(i, k, j) \in A.SInfo[G] \} \]
\[ = \{(i, k, j) \mid 0 \leq i, j < 4 \} \]
CLooG statements are derived.

\[
\text{dom}_0 = \text{iterSpace} \cap \text{dom}^\text{init}_{LU} \cap \sigma_{s,0} = \{(i,0,j) \mid 0 \leq i < 4, 0 \leq j \leq i\},
\]

\[
\text{dom}_1 = \text{iterSpace} \cap \text{dom}^\text{init}_{LU} \cap \sigma_{s,1} = \{(i,0,j) \mid 0 \leq i < 4, i < j < 4\},
\]

where \(\text{dom}^\text{init}_{LU} = \text{iterSpace} \cap \text{dom}^\text{init}_{LU} = \{(i,0,j) \mid 0 \leq i < 4 \land 0 \leq j < 4\} \). Using \(s^\text{init}_{LU} = \langle \text{dom}^\text{init}_{LU}, \emptyset, B^\text{init}_{LU} \rangle\) and \(S,\text{AInfo}\), compute

\[
B_0 = [i,j] \left( B^\text{init}_{LU}[i,j] + s[i,j] \right)
\]

\[
= [i,j] \left( L[i,k]U[k,j] + S[i,j] \right)
\]

\[
B_1 = [i,j] \left( B^\text{init}_{LU}[i,j] + s[j,i] \right)
\]

\[
= [i,j] \left( L[i,k]U[k,j] + S[j,i] \right),
\]

where \(B^\text{init}_{LU} = [i,j](L[i,k]U[k,j])\). The final output is as follows:

\[
s_0 = \langle \text{dom}_0, \emptyset, B_0 \rangle, s_1 = \langle \text{dom}_1, \emptyset, B_1 \rangle, s_2 = s^\text{acc}_{LU}.
\]

Listing 3: Output C Code from Σ-LL generated for sBLAC, Eq. 1.

```c
for (int i = 0; i <= 2; i++) {
    for (int j = 0; j <= i; j++) {
    }
}

for (int j = 0; j <= 3; j++) {
}

for (int k = 1; k <= 3; k++) {
    for (int i = k; i <= 3; i++) {
        for (int j = k; j <= 3; j++) {
            A[4*i+j] += L[4*i+k] * U[4*k+j];
        }
    }
}
```

2.1.2 Tensors

Tensors are the generalization of matrices to \(n\) dimensions. Working with tensors, rather than matrices, result in deeper loop nests and presents optimization challenges beyond classical linear algebra. In recent years, tensor decompositions were used to design learning algorithms for estimating parameters of latent variable models. For instance, learning models such as Hidden Markov Models, Latent Dirichlet Allocation, Mixture of Gaussians and Independent Component Analysis can be interpreted as performing tensor decompositions [1]. The next few paragraphs largely follow [25], which provides an in-depth overview of tensor decomposition methods.

A tensor with \(n\) modes, or dimensions, is of order \(n\). The data structure for representing sparse tensors is the list of \((i,j,k,v)\) coordinates. Tensors are denoted as \(\mathbf{X}\) and matrices as \(\mathbf{A}\), whereas elements are written with coordinates as \(\mathbf{X}(i,j,k)\). A sparse tensor \(\mathbf{X}\) is of dimension \(I \times J \times K\) with \(m\) nonzeros. Fibers are the building blocks of tensors, which hold all but one index constant. The fibers of a matrix are its rows and its columns. In a 3rd-order tensor, a tube is referred to as the added fiber, whereas a slice results from holding all but two indices constant.
A tensor can be matricized, or unfolded, into a matrix along any of its modes. In mode-
$n$ matricization, mode-$n$ fibers are used to form the columns of the resulting matrix. Mode-
$n$ unfolding of $\mathbf{X}$ is denoted as $\mathbf{X}_{(n)}$. If $\mathbf{X}$ is of dimension $I \times J \times K$, then $\mathbf{X}_{(1)}$ is of dimension $I \times JK$.

Canonical polyadic decomposition (CPD) is an extension to singular value decomposition
(SVD) for tensors. In SVD, a matrix $\mathbf{M}$ is decomposed into a summation $F$ of rank-one
matrices, where $F$ is either a rank of $\mathbf{M}$ or some smaller integer if low-rank approximation is desired. SVD is written in terms of three matrices, $\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, where $\mathbf{U}$, $\mathbf{V}$ are unitary and $\mathbf{\Sigma}$ is a diagonal
matrix of scaling factors. The $i$-th rank-one matrix is the outer product of $u_i$ and $v_i$. $\mathbf{\Sigma}$ is often
absorbed by scaling $\mathbf{A}$ and $\mathbf{M}$, written as $\mathbf{M} = \mathbf{AB}^T$.

In CPD, two essential operations on matrices are the Hadamard product and the Khatri-Rao
(KR) product. The Hadamard product, denoted $\mathbf{A} \odot \mathbf{B}$, is an element wise multiplication of $\mathbf{A}$
and $\mathbf{B}$. Elements must match in their dimension for the Hadamard product to exist. Khatri-Rao,
denoted $\mathbf{A} \odot \mathbf{B}$, is defined in terms of the Kronecker product:

$$\mathbf{A} \odot \mathbf{B} = [a_1 \otimes b_1, a_2 \otimes b_2, ..., a_n \otimes b_n]$$

where $\mathbf{A}, \mathbf{B}$ must have matching column dimensions for their Khatri-Rao products to be defined. If $\mathbf{A}$ is $I \times J$ and $\mathbf{B}$ is $M \times J$, then $\mathbf{A} \odot \mathbf{B}$ is $IM \times J$.

CPD factors a tensor into the summation of $F$ rank-one tensors using the outer product of $n$
vectors. Determining the exact rank is NP-hard, since $F \ll \max(I, J, K)$ for sparse tensors. When computing rank-$F$ CPD for a third-order tensor, the factor matrices can be found as $\mathbf{A} \in \mathbb{R}^{I \times F}$, $\mathbf{B} \in \mathbb{R}^{J \times F}$, $\mathbf{C} \in \mathbb{R}^{K \times F}$, where $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are typically dense regardless of the sparsity of $\mathbf{X}$. The matricizations of $\mathbf{X}$ in terms of CPD is defined as follows:

$$\mathbf{X}_{(1)} \approx \mathbf{A}(\mathbf{C} \odot \mathbf{B})^T, \mathbf{X}_{(2)} \approx \mathbf{B}((\mathbf{C} \odot \mathbf{A})^T, \mathbf{X}_{(3)} \approx \mathbf{C}((\mathbf{B} \odot \mathbf{A})^T$$

Alternating least squares (ALS) is used to compute CPD, where each iteration fixes $\mathbf{B}$ and $\mathbf{C}$
and solves for $\hat{\mathbf{A}}$ via

$$\hat{\mathbf{A}} = \min_{\mathbf{A}} \| \mathbf{X}_{(1)} - \hat{\mathbf{A}}(\mathbf{C} \odot \mathbf{B})^T \|^2_F$$

The least squares problem can be minimized by

$$\hat{\mathbf{A}} = \mathbf{X}_{(1)}((\mathbf{C} \odot \mathbf{B})(\mathbf{C}^T \mathbf{C} + \mathbf{B}^T \mathbf{B}))^{\dagger}$$

where $\mathbf{M}^{\dagger}$ is the pseudo-inverse of $\mathbf{M}$ and $(\mathbf{C}^T \mathbf{C} + \mathbf{B}^T \mathbf{B})$ is a $F \times F$ matrix.

Matricized tensor times Khatri-Rao Product (MTTKRP) is defined as $\mathbf{M} = \mathbf{X}_{(1)}((\mathbf{C} \odot \mathbf{B})$, which is executed once per mode per iteration of ALS. $\mathbf{M}$ has $I$ rows, $\mathbf{B}, \mathbf{C}$ have $J$ and $K$ rows and $\mathbf{M}, \mathbf{B}, \mathbf{C}$
all have $F$ columns. MTTKRP is often the bottleneck when computing CPD. Although $\mathbf{M}$ is a $I \times F$
matrix, $\mathbf{C} \odot \mathbf{B}$ is a dense $JK \times F$ matrix, which occupies significantly more memory than $\mathbf{X}$.

**Tensor Optimizer** SPLATT is a novel data structure for efficient and parallel sparse
tensor-matrix multiplications [45]. The algorithm computes entire rows of $\mathbf{M}$ at a time and only requires a single
traversal of the sparse tensor structure. SPLATT is a C library for 3-mode tensors with shared-
memory parallelism.

Assume $\mathbf{X}$ is dense, so that each row of $\mathbf{X}_{(1)}$ has exactly $JK$ nonzeros.
\[ M(i, f) = \sum_{z=0}^{JK} X_{(1)}(i, z) B(z \% J, f) C(z/J, f) \]

\[ M(i, :) = \sum_{z=0}^{JK} X_{(1)}(i, z) (B(z \% J, :) \ast C(z/J, :)) \]

\[ = \sum_{k=0}^{K} \sum_{j=0}^{J} \mathcal{X}(i, j, k) (B(j, :) \ast C(k, :) \ast ) \]

\[ M(i, :) = \sum_{k=0}^{K} C(k, :) \ast \sum_{j=0}^{J} \mathcal{X}(i, j, k) B(j, :) \]

SPLATT represents sparse tensors in a hierarchical, fiber-centric fashion, where each mode is stored as a list of slices and each slice contains a list of fibers represented as sparse vectors. Slices are stored in CSR format. Each fiber is associated with an ID, which specifies the \( K \) coordinate index. \( P + 1 \) integers are required to store the start indices for fibers and one integer is used for each fiber ID. \( I + 1 \) integers are used to mark the start indices of each slice. The total memory footprint of \( \mathcal{X} \) is \( 2m + I + P + 2 \) words. Additional memory is used to update \( M(i, :) \), which uses \( 2F(m + P) \) FLOPs.

If \( \mathcal{X} \) is a \( n \)-mode tensor, then MTTKRP in the first mode becomes

\[ \mathbf{M} = \mathbf{X}_{(1)}(\mathbf{A}^{(n)} \cdot \mathbf{A}^{(n-1)} \cdot ... \cdot \mathbf{A}^{(2)}) \]

The block structure in KR is more pronounced as \( n \) increases. The block structure is exploited by factoring out a new set of multiplications per mode. Let \( \mathcal{X} \) be a \( n \)-mode tensor with dimensions \( I_1 \times I_2 \times ... \times I_n \). The algorithm in the first mode can be formulated as:

\[ M(i_1, :) = \sum_{i_n=0}^{I_n} A^{(n)}(i_n, :) \ast \sum_{i_{n-1}=0}^{I_{n-1}} A^{(n-1)}(i_{n-1}, :) \]

\[ ... \sum_{i_3=0}^{I_3} A^{(3)}(i_3, :) \ast \sum_{i_2=0}^{I_2} \mathcal{X}(i_1, i_2, ..., i_{n-1}, i_n) A^{(2)}(i_2, :) \]

The KR product operates on \( n - 1 \) modes, requiring \( F(n - 2) \) words of intermediate memory. The last mode does not need intermediate memory, since \( M \) is written to directly. Fibers of \( \mathcal{X} \) are used for inner products with \( A^{(2)} \) and scaled to the corresponding \( A^{(3)} \) row and so on. When forming each of the \( n \) representations of \( \mathcal{X} \), an ordering is chosen from the remaining \( n - 1 \) modes. Modes are arranged to minimize the number of fibers, or maximize the average fiber length, which impacts storage and computation. Modes can be sorted by dimension, such that the shortest modes correspond to the outer loops and the longest mode corresponds to the direction \( \mathcal{X} \) that stores its fibers.

**Mode-independent reordering** models the interactions between slices of each mode of \( \mathcal{X} \) with a partitioning graph. A bipartite graph model is used to reorder sparse matrices. Suppose \( \mathcal{X} \) is a \( n \)-mode tensor with dimensions \( I_1 \times I_2 \times ... \times I_n \). A \( n \)-partite graph can be constructed, where the vertex sets are of cardinalities \( I_1 \times I_2 \times ... \times I_n \) and a nonzero \( \mathcal{X}(i_1, i_2, ..., i_n) \) generates a clique that connects nodes \( i_1, i_2, ..., i_n \). Edge \( (i_a, i_b) \) represents a nonzero with indices \( i_a \) and \( i_b \) appearing together and the weight edges count the number of times each edge is generated.
Mode-dependent reordering partitions the hypergraph that models memory accesses to M, B and C. The hypergraph model is an extension of the column-net model used for parallel sparse matrix-vector multiplication. Fibers represent units of work, analogous to rows in a sparse matrix, that are mapped to vertices in a hypergraph, where each mode emits hyperedges by dimensions. The number of partitions where a hyperedge is found, or its connectivity, models the number of times its corresponding row in M, B, or C is fetched from memory. Minimizing the connectivity of hyperedges minimizes the total number of memory accesses. Partitioning a hypergraph induces a reordering of a tensor. Fibers, represented as vertices, are relabeled such that fibers in the same partition are given consecutive labels. Spatial locality is affected by fibers with similar sparsity pattern, where consecutive rows of a matrix can be processed nearby in time. The drawback of mode-dependent reordering is the need to construct and partition a hypergraph for each mode.

Compiler for Tensor Algebra The Tensor Algebra Compiler (TACO) [24] optimizes sparse tensor computations. Complexities for sparse tensor algebra kernels result from directional accesses of data structures and the merging of sparse tensor indices. Sparse data structures are accessed efficiently in one direction and depends on the ordering, where compressed sparse row (CSR) accesses from rows to columns and compressed sparse column (CSC) accesses from columns to rows. The second complexity is due to the merging of sparse tensor indices. Consider $a_i = b_i c_i + d_i$, where a component-wise product of two vectors is added to a third. The merge of $a$ with $b$ is different since an addition produces a value if either operand is non-zero, whereas a multiplication produces a value if both operands are non-zero. To support any order of tensors, formats are constructed from a bounded number of primitives. The number of different formats increases exponentially along with the tensor dimensionality ($d! \cdot 2^d$ for $d$-order tensor). Sparse storage dimensions are optimized for an iteration in a specific order, but do not allow efficient random accesses to indices and values.

Iteration graphs are a compiler intermediate representation that describes how to iterate over non-zero values of a tensor expression. An iteration graph is a directed graph $G = (V, P)$ with a set of index variables $V = \{v_1, ..., v_n\}$ and a set of tensor paths $P = \{p_1, ..., p_m\}$. A tensor path is a tuple of index variables.

Index variables that access dimensions of more than one tensor must iterate over merged indices of those dimensions. If tensors are multiplied, the merge is a conjunction ($\wedge$), whereas an addition is a disjunction ($\vee$) merge. The combination of conjunctions and disjunctions mirror the merged index expression, such as $(b_i + c_i) d_i \rightarrow (b_i \vee c_i) \wedge d_i$. Merge lattices are motivated by the cost of a disjunctive merge, where every loop iteration must check that each merge index has more values left to avoid out-of-bounds accesses. The two-way merge algorithm avoids expensive checks, where one loop iterates until either of the indices are exhausted, or runs out of values and the other two loops process the rest of the unexhausted index.

A merge lattice $L$ is a lattice comprising $n$ lattice points $\{L_1, ..., L_n\}$ and a meet operator, where each lattice point $L_p$ has a set of tensor dimensions $T_p = \{t_{p1}, ..., t_{pk}\}$ to be merged consecutively (i.e. $t_{p1} \wedge \ldots \wedge t_{pk}$) and an expression $expr_p$ to be evaluated. The meet of two lattice points $L_1$ and $L_2$ with associated tensor dimensions $T_1$ and $T_2$ is a lattice point with tensor dimensions $T_1 \cup T_2$. $L_1 \leq L_2$ if and only if $T_1 \subset T_2$.

To construct a merge lattice in index expression, the algorithm traverses an index expression tree, merging lattices at the leaves and combining the merged lattices at internal nodes. Let the conjunction of two lattice points $L_p$ and $L_q$ with operator $op$ be a new lattice point with associated tensor dimensions $T_p \cup T_q$ and expression $expr_p \land expr_q$. Let the conjunction merge of two lattices $L^1$ and $L^2$ with $op$, denoted $L^1 \land_{op} L^2$, be a new lattice with a pair of lattice points constructed by the Cartesian product $(L^1_1, ..., L^1_n) \times (L^2_1, ..., L^2_m)$. Let the disjunctive merge of two lattices $L^1$ or $L^2$ with operator $op$, denoted $L^1 \lor L^2$, be a new lattice containing all lattice points in $L^1 \lor_{op} L^2$, $L^1$ and $L^2$. 

9
Figure 2: Iteration graph for matrix addition (L). Dense merge lattice for $i$ (M). Sparse merge lattice for $j$ (R).

Workspaces  Extensions to TACO include format abstractions [7] and workspaces for optimizations [23]. Concrete index notation is proposed as an intermediate language for tensor operations by extending index notation with constructs that describe the way an expression is computed. The statement types include an assignment statement that assigns an expression result to a tensor element, a forall statement that executes a statement over some range inferred from tensor dimensions and a where statement that creates temporaries to store subexpressions.

For instance, let $A$, $B$ and $C$ be sparse matrices of dimension $I \times J$, $I \times K$ and $K \times J$, where $A$ and $B$ are row-major (CSR), $C$ is col-major (CSC) and $t$ is scalar. An inner-products matrix multiply described with concrete index notation is as follows:

$$\forall_{ijk} A_{ij} \leftarrow B_{ik} C_{kj}$$

$\iff$

$$\forall_{i} \left( \forall_{j} (A_{ij} = w_{j}) \text{ where } (\forall_{kj} w_{j} \leftarrow B_{ik} C_{kj} \right) \right) \iff \forall_{i} (A_{ij} = w_{j})$$

When matrices are sparse, a linear combination of rows matrix multiply is preferred over an inner products matrix multiply, since the linear combination of rows work on CSR and must simultaneously iterate over row/column pairs of non-zero values. The $k$ loop can be moved above the $j$ loop to express the linear combination of rows matrix multiply in concrete notation. If $A$ is sparse, $w$ can store intermediate computations to save from repeatedly adding rows.

$$\forall_{i} \left( \forall_{j} (A_{ij} = w_{j}) \text{ where } (\forall_{kj} w_{j} \leftarrow B_{ik} C_{kj} \right) \right)$$

For $i \in I$

$w = 0$

For $k \in K$

$$A_{ij} \leftarrow B_{ik} \ast C_{kj}$$

Let $(S, E, i...)$ be inputs to the optimization, where $S$ is a statement not containing sequences, $i...$ is the set of index variables, $E$ is an expression contained in an assignment or increment statement $S_A \subset S$. If $S_A$ is an increment statement, let $\oplus$ be an associated operator. The optimization rewrites statement $S_A$ to precompute $E$ in workspace.

Applying workspace optimizations to sparse vector addition with a dense result enables partial results to be efficiently accumulated:

$$\forall_{i} a_{i} = b_{i} + c_{i} \implies (\forall_{i} a_{i} = b_{i}; \forall_{i} a_{i} \leftarrow c_{i})$$

Listing 4: After optimization to pre-compute $B_{ik} C_{kj}$ in workspace $w$ at $j$ (left). After further optimization to pre-compute $w_{j} D_{kj}$ in workspace $v$ at $j$ (right).
2.2 Iteration Spaces

An iteration space is the set of dynamic execution instances in a computation, or the set of combinations of values taken on by the loop indexes. Often, the iteration space is rectangular, as in matrix multiplication, where each of the nested loops has a lower bound of 0 and an upper bound of \( n - 1 \).

A function of one or more variables, \( i_1, i_2, ..., i_n \), is affine if it can be expressed as a sum of a constant, plus constant multiples of the variables, i.e. \( c_0 + c_1 x_1 + c_2 x_2 + ... + c_n x_n \), where \( c_0, c_1, ..., c_n \) are constants. Affine functions are usually known as linear functions, although strictly speaking, linear functions do not have the \( C_0 \) term.

Affine partitioning and blocking are typical code transformations. Affine partitioning splits up the polyhedra of iterations into components, to be executed either on different machines or one-by-one sequentially. Blocking creates a hierarchy of iterations by subdividing an array into blocks and visiting all elements in a block before moving to the next.

2.2.1 Integer Set Library

isl [56] is an integer set library for the polyhedral model. Sets and binary relations over tuples of integers are bounded by affine constraints, called polyhedral sets and maps, where each map \( R \) is a finite union of basic maps \( R = \bigcup_i R_i \) and each mapping of a tuple of \( n \) integer parameters to a binary relation is on tuples of integers, e.g. \( R_i : \mathbb{Z}^n \rightarrow \mathbb{Z}^{d_1+d_2} : s \mapsto R_i(s) \), with

\[
R_i(s) = \{ x_1 \rightarrow x_2 \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \exists z \in \mathbb{Z}^e : A_1 x_1 + A_2 x_2 + B s + D z + c \geq 0 \},
\]

where \( A_i \in \mathbb{Z}^{m \times d_i} \), \( B \in \mathbb{Z}^{m \times n} \), \( D \in \mathbb{Z}^{m \times e} \) and \( c \in \mathbb{Z}^m \). Sets are defined similarly, but differ from maps in terms of usage. Maps have domains and ranges and can be composed with each other.
and applied to sets, whereas sets are projections of integer points in polyhedron and can include intersections of polyhedra and lattices as a special case.

Matrix multiplication will be used to describe an iteration domain for GPUs.

```c
void matmul(int M, int N, int K, float A[M][K], float B[K][N], float C[M][N])
{
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++) {
            S1: C[i][j] = 0;
            for (int k = 0; k < K; k++)
                S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
}
```

An example of an iteration domain is

\[(K, N, M) \rightarrow \{ S2(i, j, k) \mid 0 \leq i < M \land 0 \leq j < N \leq 0 \leq k < K \} \cup \{ S1(i, j) \mid 0 \leq i < M \land 0 \leq j < N \}, \]

where \( K, N \) and \( M \) are parameters. Access relations map statement instances to array elements accessed by those instances. Write accesses can be expressed as

\[ \{ S2(i, j, k) \rightarrow C(i, j) \} \cup \{ S1(i, j) \rightarrow C(i, j) \}, \]

whereas a read access relation can be expressed as

\[ \{ S2(i, j, k) \rightarrow A(i, k) \} \cup \{ S2(i, j, k) \rightarrow B(k, j) \} \cup \{ S2(i, j, k) \rightarrow C(i, j) \}. \]

The schedule specifies the order in which statement instances are executed. This schedule reflects the original execution order:

\[ \{ S2(i, j, k) \rightarrow (i, j, k, 1) \} \cup \{ S1(i, j) \rightarrow (i, j, 0, 0) \}. \]

Parallel loops need to be identified to tile and map to blocks and threads. To tile three loops, iterations of loops are executed in chunks, where each chunk processes its iteration completely before moving onto the next chunk. A tile size of 16 in each direction can be obtained by combining with the original schedule (Eq. 3):

\[ \{(i, j, k, s) \rightarrow ([i/16], [j/16], [k/16], i \% 16, j \% 16, k \% 16, s)) \}. \]

A grid of 16 \( \times \) 16 blocks, each with 8 \( \times \) 16 threads, can be described by intersecting the range of tiling map with the set

\[ (b_0, b_1, t_0, t_1) \rightarrow \{ (I, J, K, i, j, k, s) \mid \exists \alpha_0, \alpha_1, \beta_0, \beta_1 : I = 16\alpha_0 + b_0 \land J = 16\alpha_1 + b_1 \land i = 8\beta_0 + t_0 \land j = 16\beta_1 + t_1 \}, \]

where \( 0 \leq b - 0 \leq 16, 0 \leq b_1 < 16, 0 \leq t_0 < 8, 0 \leq t_1 < 16 \) represents block and thread identifiers.

### 2.2.2 Polyhedral Representation

Polly [19] is a framework that uses polyhedral techniques to optimize an IR program for data locality and parallelism that builds on the pet tool [58]. Static control parts (SCoPs) are parts of the program that the front end analyzes and translates to a polyhedral representation. An example of a SCoP is as follows.

```c
for (i = 0; i <= N; i++) {
    if (i <= N - 50)
        S1: A[5*i] = 1;
    else
        S2: A[3*i] = 2;
    for (j = 0; j <= N; j++)
        S3: B[i][2*j] = 3;
}
```
An SCoP is a subgraph of the control flow graph (CFG) that forms a single-entry-single-exit region semantically equivalent to a classical SCoP. To represent SCoP, Polly uses a polyhedral description based on integer sets and maps provided by the *isl* library. A SCoP is a pair, (context, statements), where the context is an integer set that describes constraints on the parameters of the SCoP and the statement is a quadruple (name, domain, schedule, accesses) that corresponds to a basic block in SCoP and is the smallest unit that can be scheduled independently. The domain \( \mathcal{D} \) of the statement is an integer set that describes the set of different loop iterations in which the statement is executed. A schedule \( \mathcal{S} \) is an integer map that assigns to each iteration vector a multidimensional point in time, which defines the execution order of different statement instances in the final target code. Similar to *isl*, the access relation \( \mathcal{A} \) is an integer map that maps from a domain of statements to named, possibly multi-dimensional memory space.

The polyhedral representation of SCoP above can be described as follows:

\[
\begin{align*}
D_{S1} &= \{s1[i] : \text{i} \geq 0 \land \text{i} \leq N \land i \leq N - 50 \} \quad D_{S2} = \{s2[i] : \text{i} \geq 0 \land \text{i} \leq N \land i \leq N - 50 \} \\
S_{S1} &= \{s1[i] \rightarrow [0,\text{i},0,0] \} \quad S_{S2} = \{s2[i] \rightarrow [0,\text{i},1,0] \} \\
A_{S1} &= \{s1[i] \rightarrow A[5i] \} \quad A_{S2} = \{s2[i] \rightarrow A\[3i \] \}
\end{align*}
\]

\[
\begin{align*}
D_{S3} &= \{s3[i,j] : \text{i} \geq 0 \land \text{i} \leq N \land \text{j} \geq 0 \land \text{j} \leq N \} \\
S_{S3} &= \{s3[i,j] \rightarrow [0,\text{i},2,\text{j},0] \} \\
A_{S3} &= \{s3[i,j] \rightarrow B[i][2j] \}
\end{align*}
\]

For polyhedral transformations, SCoPs can be optimized by changing the execution order of statement instances and the locations of memory accesses. Loop transformations, such as interchange, tiling, fusion and fission, change the order of execution. Schedules can be modified either by replacing schedules recalculated from scratch or by applying a set of transformations, which maps the integer map of the original schedule to a new execution time.

An example of loop blocking transformation can be applied for the following SCoP:

```plaintext
1 for (i = 0; i < M; i++)
2 for (j = 0; j < M; j++)
3 Stmt(i,j);
4
\( D_{Stmt} = \{Stmt[i,j] : 0 \leq i < N \land 0 < j < M \} \)
5 \( S_{Stmt} = \{Stmt[i,j] \rightarrow \Theta[i,j] \} \)
```

Loop blocking is the combination of transformations \( T_{StripMineOut} \), \( T_{StripMineIn} \) and \( T_{Interchange} \)

\[
\begin{align*}
T_{StripMineOut} &= \{\Theta[s0,s1] \rightarrow \Theta[t0,s0,s1] : t \% 4 = 0 \land t \leq s0 < t + 4 \} \\
T_{StripMineIn} &= \{\Theta[s0,s1,s2] \rightarrow \Theta[s0,s1,t,s2] : t \% 4 = 0 \land t \leq s2 < t + 4 \} \\
T_{Interchange} &= \{\Theta[s0,s1,s2,s3] \rightarrow \Theta[s0,s2,s1,s3] \}
\end{align*}
\]

\[
\begin{align*}
T_{Block} &= T_{Interchange} \circ T_{StripMineIn} \circ T_{StripMineOut} \\
&= \{\Theta[s0,s1] \rightarrow \Theta[t0,t1,s0,s1] : t0 \% 4 = 0 \land t0 \leq s0 < t0 + 4 \\
&\land t1 \% 4 = 0 \land t1 \leq s1 < t1 + 4 \}
\end{align*}
\]

\[
\begin{align*}
S'_{Stmt} &= T_{Block} \circ S_{Stmt} \\
&= \{Stmt[i,j] \rightarrow \Theta[t1,i,j] : t1 \% 4 = 0 \land t1 \leq i < t1 + 4 \\
&\land t1 \% 4 = 0 \land t1 \leq j < t1 + 4 \}
\end{align*}
\]

Code generated for the statement with its domain \( \mathcal{D}_{Stmt} \), together with the new schedule \( S'_{Stmt} \) yields the following blocked loops:

```plaintext
1 for (ti = 0; ti < M; ti++)
2 for (tj = 0; tj < N; tj++)
3 for (i = ti; i < min(M, ti-4); i++)
4 for (j = tj; j < min(N, tj+4); j++)
5 Stmt(i, j);
```
2.3 Modeling and Tools

A model aids the compiler in inferring the performance of an application, either analytically, through run time measurements, or with hybrid approaches. The performance of an application refers to the amount of work accomplished, such as instructions-per-cycle, which is estimated in terms of efficiency, effectiveness and speed. Analytical models can describe certain aspects of an application, such as computation and communication, which provide back-of-the-envelope cost models that augment the decision-making of the compiler. Instrumentation methods provide concrete results for discerning performance, though, at the cost of executing an application. Since analytical models are difficult to generalize and, in some cases, limited in expressivity (e.g. asynchronous run time behavior), collecting performance measurements, either via simulation or execution, may be the only option. Hybrid approaches make use of analytical models with profiles of an executed application, combining cost models with run time measurements.

2.3.1 Models for Deep Learning

Performance Model for Deep Neural Networks  Paleo formally models a fully-connected neural network [40]. The computation model on a single machine is defined as a directed graph \( N = \{\{u^{(i)}\}_{i=1}^n, \{w^{(j)}\}\}, \) where each node \( u^{(i)} \) is associated with an operation \( f^{(i)} \) on a device \( d^{(i)} \) and each directed edge \( (u^{(i)}, w^{(j)}) \) represents the dependency that the operation \( f^{(j)} \) cannot be executed until \( f^{(i)} \) is finished. \( Pa(u^{(j)}) \) represents the set of immediate parent nodes of \( u^{(j)} \), where each layer in a neural network is a node with connections between layers as edges.

To model the run time of layer \( u \), consider operation \( f \) and decompose the execution time of an operation into the time to fetch the input produced by parent layers \( R(P(a)) \), the time to perform a computation of \( f \) on a designated device \( d \) \( C(f, d) \) and the time to write outputs to local memory \( W(f, d) \). Assuming sequential execution, the run time for node \( u \) is written as a simple summation.

\[
T(u) = R(Pa(u)) + C(f, d) + W(f, d)
\]

where \( C(f, d) = \text{FLOPs}(f)/\text{speed}(d) \), or FLOP counts of operation divided by computation speed. I/O times \( R(Pa(u)) \) and \( W(u) \) are calculated as the size of memory footprints involved in computation divided by I/O bandwidth of the device.

The total execution time is calculated as the sum of execution time of all layers \( T(N) = \sum_{i=1}^n T(u^{(i)}) \). A supernode \( U = \{G^{(i)}\}_{i=1}^k \) represents a set of disjoint subgraphs sandwiched by synchronization barriers in the computation graph. When substituting subgraphs with the supernode, the network is reduced to a sequential structure as described above. For a supernode, the execution time is \( T(U) \) within range \( [\max, \sum T(g^{(i)})] \), where the lower bound corresponds to perfect parallelization and is calculated recursively.

Two implementations of convolutional operations include matrix multiplication and FFT. During forward propagation, a 2D convolutional layer takes as input feature map \( D_{N \times C \times H \times W} \) (batch \( N \) input feature maps with shape \( H \times W \), \( C \) channels) and a set of convolutional filters \( F_{K \times C \times R \times S} \) \((K \text{ filters, shape } R \times S \text{ and } C \text{ channels})\), and produces \( N \times K \) feature maps of each of shape \( P \times Q \) calculated from shapes of inputs and filters together with additional striding and padding parameters. The total FLOP count is \( 2KCRSNPQ \). In FFT, both input feature maps and filters are transformed into the frequency domain and element-wise multiplications are performed followed by the inverse Fourier transform. Computational complexity is reduced to \( O(NCKHW + (NC + CK + N)KHW\log(HW)) \). Convolutional layers with large filters or large problem sizes benefit from FFT. FLOP counts are approximated as \( 5n\log_2 n \) for complex-valued transformations of size \( n \) (Cooley and Tukey). To model communication, let \( |D| \) be the size of data communicated between two workers, \( B \) represent the bandwidth of communication channel and total communication time is \( T_{\text{comm}} = |D|/B \).
Performance and Scalability Optimization  Performance, in terms of workers, threads, nodes and scalability, can be aided with models that describe the phases of a deep neural network [61]. In model parallelism, a DNN has \( L \) layers, where each layer \( l \in [1, \ldots, L] \) is partitioned into \( P(l) \) segments. Each segment is denoted by \( p \), where \( p \in \{1, \ldots, P(l)\} \) and segments of layers are processed in parallel. The layer execution time is decided by the slowest segment. To reduce training time and improve system utilization, segments of the same layer can be evenly partitioned, where each segment has roughly the same number of neurons and connections. The time spent on layer \( l \) is estimated using any of segment \( p \). The total time in each layer is time spent on feed-forward evaluation, back-propagation and weight updates, further divided into computation and communication time. For each segment \( p \) at layer \( l \), the feed-forward evaluation time \( T_f(l, p) \) is equal to computing the outcome activations of neurons in a segment, \( U_f(l, p) \) and communicating activations from connected segments in layer \( l-1 \), \( M_f(l, p) \), resulting in \( T_f(l, p) = U_f(l, p) + M_f(l, p) \).

Feed-forward evaluation computes output activations of neurons in each layer. The output activation of neuron \( i \) in layer \( l \) is the result of a nonlinear function \( f(x) \), where the \( x \) input is a dot product of input activations of \( i \) from \( l-1 \) layer and weight values of connections. Thus, the computation time per neuron is \( C_{muladd} \times |S_i| + C_{act} \), where \( S_i \) is the set of neurons at layer \( l-1 \) connected to neuron \( i \) of layer \( l \), \( C_{muladd} \) is the time for one multiply-add operation and \( C_{act} \) is the time to compute \( f(x) \). The computation time of segment \( p \) is defined as follows:

\[
U_f(l, p) = C_{muladd} \times W(l, p) + C_{act} \times N_{neuron}(l, p),
\]

where \( N_{neuron} \) is the number of neurons in segment \( p \) and \( W(l, p) \) is the number of weights connected from layer \( l-1 \) to all neurons in segment \( p \).

The communication time of feed-forward evaluation is dominated by the delay in receiving cross-machine activations from the previous layer. Communication time \( M_f(l, p) \) is the function of data size received by \( p \) and the network performance

\[
M_f(l, p) = C_{ncost} + \frac{A(l, p) \times C_{bits}}{C_{nbw}},
\]

where \( C_{ncost} \) is the network latency of transmitting one data bit between two workers, \( C_{nbw} \) is the bandwidth of a machine’s NIC, \( A(l, p) \) are the remote activations that segment \( p \) receives from layer \( l-1 \) and \( C_{bits} \) is the size of each activation. Thus, \( A(l, p \cdot C_{bits}) \) is the number of data bits received by \( p \).

Backpropagation computes the error terms of neurons, where the error term of neuron \( i \) in segment \( p \) of layer \( l \) is computed from the input error terms of \( i \) in layer \( l+1 \), the connection weights and the error function \( f'(x) \). The computation time of segment \( p \) is estimated as

\[
U_b(l, p) = C_{muladd} + W'(l, p) + N_{neuron}(l, p) \times C_{err},
\]

where \( C_{err} \) is the basic cost of error function and \( W'(l, p) \) is the number of connections from layer \( l+1 \) to segment \( p \) of layer \( l \). For communication, the backpropagation time of segment \( p \) in layer \( l \) is \( M_b(l, p) \) and is delayed in receiving the remote error terms \( E(l, p) \) from layer \( l+1 \).

\[
M_b(l, p) = C_{ncost} + \frac{E(l, p) \times C_{bits}}{C_{nbw}}
\]

Error terms are propagated throughout the network to update the weight values in each layer. The weight connection \( \delta w_{ij} \) between neuron \( i \) in layer \( l+1 \) and neuron \( j \) in layer \( l \) is computed from the error term \( \delta_i \) and the activation \( a_j \) and the weight value \( w_{ij} \) is updated using \( \delta w_{ij} \). Thus, the computation time for weight updates is estimated as
\[ U_w(l, p) = C_{\text{muladd}} \times W(l, p). \]

Since weights are not communicated in model parallelism, \( M_w(l, p) = 0 \).

For data parallelism using CMP, one or more dimensions \( h \in [1, H(l)] \) are added to the base model, where \( H(l) \) is the number of parallel threads training in layer \( l \) and each index is a triple \((l, p, h)\) representing layer, partition and thread ID. For computation, contention may occur during concurrent training of multiple samples, competing for memory bandwidth and affecting per-sample computation time. Interference factor \( C_{\text{interf}}(H(l)) \) models the interferences among \( H(l) \) threads, estimated as the ratio of \( H(l) \)-thread execution time and a single-thread execution time. Thus, the computation time of the segment using CMP of \( H(l) \) threads is

\[ U_i = \{F, B, W\}(l, p, h) = C_{\text{interf}}(H(l)) \times U_i(l, p), \]

where \( U_i(l, p) \) is the computation time of having one thread per layer. This shows that data parallelism may not reduce the computation time of an example, but may instead increase time due to potential conflicts among threads. Running multiple samples concurrently reduces \( T_{\text{epoch}} \), training the entire sample set once, for all \( N_{\text{sample}} \). The data parallelism degree \( Q(l) \) is defined as the concurrency of training multiple samples in parallel at layer \( l \). When \( H(l) \) concurrent threads run a sample, \( Q(l) = H(l) \) at layer \( l \). Per-sample execution time and data parallelism degree represents the epoch time and system throughput. For communication, each thread gets \( C_{\text{nbw}}(H(l)) = 1/h(l) \) bandwidth. Network latency \( C_{\text{ncost}}(H(l)) \) is modeled as a function of \( H(l) \), since latency may increase when both sender and receiver establishes \( H(l) \) concurrent connections. Communication time is estimated as

\[ M_f(l, p, h) = C_{\text{ncost}}(H(l)) + \frac{A(l, p, h)}{C_{\text{nbw}}(H(l))} \times \frac{C_{\text{bits}}}{\text{nbw}(H(l))} \]
\[ M_b(l, p, h) = C_{\text{ncost}}(H(l)) + \frac{E(l, p, h)}{C_{\text{nbw}}(H(l))} \times \frac{C_{\text{bits}}}{\text{nbw}(H(l))} \]

Replicating layers across worker machines can reduce network overheads of cross-machine activations. Since a fully connected layer can be expensive to partition, layers are replicated among several machines where each machine processes a subset of training data. The data parallelism degree is extended to \( Q(l) = H(l) \times R(l) \), where \( R(l) \) denotes the number of replications for layer \( l \).

Data is partitioned to train the model replicas in parallel, where replicas share weights through a parameter server. To improve throughput, weights are partitioned across multiple servers for more network bandwidth. The load for weight communication scales with number of model replicas, whereas computation time per sample remains unchanged. The data parallelism degree is extended to \( Q(l) = H(l) \times R(l) \times N_{\text{mr}} \), where \( N_{\text{mr}} \) represents the number of model replicas. For communication, weights are written to parameter servers asynchronously and does not affect training time on replicas, whereas reading weights is synchronous and can stall training until reads are completed. \( N_{\text{read}} \) represents the read frequency, where a replica reads weights from parameter servers after training \( N_{\text{read}} \) samples.

The communication time between replicas and parameter servers may vary, depending on the communication patterns of the actual implementation. The worst case is when all replicas read weights from the same parameter server simultaneously, where the bandwidth of the parameter server to each replica is limited. The time to read weights from parameter servers is:

\[ M_w^{\max} = \sum_{i=1}^{L} \left( C_{\text{ncost}}(H(l)) + \frac{N_{\text{mr}} \times W(l)}{C_{\text{nbw}}(H(l))} \right), \]
where \( W(l) = \sum_{1 \leq p \leq P(l)} W(l, p) \) is the data size of all weights for layer \( l \). The best case occurs when a single replica uses all its \( N_{wr} \) workers and reads from all parameter servers in parallel using accumulated bandwidth without overlaps among multiple replicas while reading the weights.

\[
M_{\text{min}}^w = \sum_{i=1}^{L} (C_{\text{cost}}(H(l))) + \frac{W(l)}{C_{\text{bw}}(H(l)) \times \min(N_{ps}, N_{wr})}
\]

Optimization is when read time for different replicas do not overlap with communication to the parameter servers

\[
T_{\text{epoch}} = \frac{M_{\text{w}} \times N_{\text{sample}}}{N_{\text{read}}} + \sum_{i=1}^{L} \left\{ (U_f(l, p, h, r) + M_f(l, p, h, r) + U_w(l, p, h, r)) \times N_{\text{sample}}/Q(l) \right\}
\]

**Algorithm 1 Problem Formulation**

**Variables** to define a configuration \( \Phi \)
- \( N_{ps} \): number of parameter servers
- \( N_{mr} \): number of model replicas
- \( N_{wr} \): number of workers per model replica; \( SW = \{1, ..., N_{wr}\} \)

**for each layer** \( 1 \leq l \leq L \) **do**
- \( P_l \): number of partitions; \( SP_l = \{1, ..., P_l\} \)
- \( H_l \): number of threads; \( SH_l = \{1, ..., H_l\} \)
- \( R_l \): number of replicas; \( SR_l = \{1, ..., R_l\} \)
- Mapping function \( f_l : SP_l \times SH_l \times SR_l \rightarrow SW \)

**Objective:** minimize \( T_{\text{epoch}}(\Phi) \)

**Subject to:**
- \( C1: N_{ps} + N_{mr} \times N_{wr} \leq N \)
- \( C2: H_l \leq K, \) for all \( 1 \leq l \leq L \) (\( K \) is total number of cores in a machine)
- \( C3: f_l(p, h, r) \neq f_l(p', h', r'), \) if \( p \neq p' \) or \( r \neq r' \), for all \( 1 \leq l \leq L, \)
- \( p, p' \in SP_l, r, r' \in SR_l \) and \( h, h' \in SH_l \)
- \( C4: f_l(p, h, r) = f_l(p, h', r), \) for all \( 1 \leq l \leq L, \)
- \( p \in SP_l, r \in SR_l \) and \( h, h' \in SH_l \)

For a DNN, its training data and a cluster of \( N \) machines, the objective is to find an optimal system configuration \( \Phi \) such that \( T_{\text{epoch}} \) is minimized. Considerations include the parameters that define the resources, such as the number of parameter servers, model replicas and workers per model replica \( (N_{ps}, N_{mr}, N_{wr}) \), the parameters that define the number of partitions, thread and replications at each layer of the DNN and the mapping function \( f \) between resources and segments of the DNN, in particular between workers of each model replica and its partitioned and replicated segments.

Constraints include \( C1 \), which bounds the total number of machines used by the DNN task, \( C2 \), which bounds the total number of concurrent threads per machine by the total number of cores per machine, \( C3 \), which ensures that at each layer two segments of different partitions or replications are not mapped to the same worker and \( C4 \), which segments chip-level multiprocessing where the same partition and replication IDs are mapped to the same worker. Given a system configuration \( \Phi \), the training epoch time of DNN can be calculated using the performance model.

Considering different selections of resource parameters \( (N_{ps}, N_{mr}, N_{wr}) \), there are at most \( N^3 \) combinations. The assignment of segments to workers in each layer becomes complex as different layers require a different number of segments and workers. The permutation of segments and
workers results in $O(N!)$ possible combinations. DNN consists of multiple layers, where each layer makes a different decision. At each layer, there are $N^2$ combinations on the selection of the number of partitions and replications and $K$ choices on number of threads per worker. The possible mapping of workers to segments is $K \times N^2 \times (N!)$ combinations for each layer. For DNN with $L$ layers, the entire search cost is $O(N^3 [K \times N^2 \times (N!)^L])$, which is exponential with respect to the number of machines and layers. Optimal approaches can be discovered, either by solving recursively, assuming the optimal solution for $l - 1$ is given, or with dynamic programming, where complexity can be reduced to $O(L \times K \times N^5 \log^2 N)$.

Performance Modeling on GPUs  

Factors that entail model training on GPUs, such as compute and memory overlaps, can also be described formally [44]. The iteration time $t_{iter}$ is represented as

$$t_{iter} = t_{io} + t_{h2d} + t_f + t_b + t_u + t_{comm}$$

Let $T_{gpu} = t_{h2d} + t_f + t_b + t_u$. Then,

$$t_{iter} = t_{io} + T_{gpu} + t_{comm},$$

where $t_{comm} = 0$ in a single GPU environment. Some terms in Eq. 5 can be eliminated with pipelining.

I/O is slow because of low bandwidth. $t_{io} = \frac{D_{mn}}{R_{io}}$, where $t_{io}$ should be hidden to achieve better utilization of GPUs. Calculate the average iteration time of pipelined SGD as

$$\overline{t}_{iter} = \max(t_{gpu} + t_{comm}, t_{io})$$

Gradient computation has no dependency with updating the next layers, so the gradient computation in layer $l_{i-1}$ can be parallelized with gradient aggregation, which reduces overhead of data communication and iteration time.

$$t_{iter} = t_{io} + t_{h2d} + t_f + t_b + t_{comm} - \tau_{comm}s + t_u$$

$\tau_{comm}s$, $\tau_{comm}e$ denote start and end times, $t_{b}^{(L)}$ is the gradient computation time of the last learnable layer. For parameter server methods, updating gradients are done in the parameter server, so communication is hidden by pipelining. If gradient communication is hidden by backward propagation, $t_{comm}^{(i)} \leq t_{b}^{(i-1)}$, $2 \leq i \leq L$. There are also layers where communication times are longer than the time of backward computation of the previous layers. Formulate as $t_{comm}^{(i)} \leq t_{b}^{(i-1)}$ for $i = 2, ..., C - 1$ and $t_{comm}^{(i)} > t_{b}^{(i-1)}$ for $i = C, C + 1, ..., L$.

For the scenario where communication time is hidden by computation, $t_{iter}$ can be updated as

$$t_{iter} = t_{io} + t_{h2d} + t_f + t_b + t_{comm} + t_{comm} + t_u$$

In the case where some layers of communication overheads are not hidden, $t_{iter}$ can be estimated as

$$t_{iter} = t_{io} + t_{h2d} + t_f + \sum_{i=C}^{L} t_{comm} + \sum_{i=1}^{C-1} t_{b}^{(i)} + t_{comm} + t_{comm} + t_u$$
where $L$ is the number of learnable layers of DNN. More $C$ means more communication needs to be hidden. Two types of overhead for network communication include latency and transmission time. Gradients from different $c$ layers can be merged into a big data packet and transmitted once, which reduces the impact of latency.

Let $t_{\text{iter,}N_g}$ and $t_{\text{io,}N_g}$ denote iteration time and I/O time of mini-batch $M$ with $N_g$ GPUs across $N$ machines, where each machine has $n_g$ GPUs. Speedup is formulated as

\begin{align}
S &= \frac{MN_g / t_{\text{iter,}N_g}}{M / t_{\text{iter,}1}} = N_g \frac{T_{\text{iter,}1}}{t_{\text{iter,}N_g}}
\small{= N_g \frac{T_{\text{io,}1} + t_{\text{gpu}}}{t_{\text{io,}N_g} + t_{\text{gpu}} + t_{\text{comm}}},}
\end{align}

where speedup is achieved if I/O and communication are completely hidden.

### 2.3.2 Instrumentation

CUDAAdvisor [43] instruments CUDA programs at the IR level for both CPU and GPU code. The workflow includes an instrumentation engine, a profiler and an analyzer. The instrumentation engine accepts source code of a CUDA program, performs code transformations and invokes the CUDA compiler to produce binary code. The profiler collects data that represent the behavior of the binary code during its execution on real GPU hardware. The profiling data is then analyzed and optimization advice is generated with attributions to source code.

CUDAAdvisor’s engine inserts probes to enable the reconstruction of the call path and data flow in a profiling component. Calls and returns are instrumented from CPU functions, as well as GPU kernels. Functions that are instrumented include memory allocation calls in CPU code (e.g. malloc, calloc, realloc), in GPU code (cudaMalloc), and CPU-GPU data transfer functions (e.g. cudaMemcpy). At each instrumentation site, the engine inserts function and passes information as arguments of each function. For memory allocations, the arguments include the starting addresses of memory ranges on both CPU and GPU, as well as the amount of bytes transferred. CUDAAdvisor also provides optional instrumentation for memory operations, arithmetic operations and control flow operations. The CUDAAdvisor profiler includes two stages, where data is collected during CUDA kernel execution and attributed at the end of each CUDA kernel instance. The combination of the two stages enable both code- and data-centric analyses.

For code-centric profiling, CUDAAdvisor maintains a shadow stack to mirror the execution stack of each thread when the kernel runs on the GPU. The profiler pushes a call site onto the shadow stack in the instrumented function at every call instruction and pops the call site from the shadow stack in instrumented function at every return instruction. All GPU threads share the same encoding map from the number to the function name and source code, and each GPU thread maintains its own shadow stack. Both the encoding map and shadow stack reside in GPU global memory. Upon kernel return, CUDAAdvisor concatenates the CPU call path with GPU kernel instances to give a complete execution path.

For data-centric profiling, the data flow is reconstructed from CPU to GPU that reveal access patterns of data objects of CPU and GPU, and across different GPU kernels. Data flow of one data object starts at the beginning of its lifetime and ends in memory accesses in GPU kernels that reference this data object. The profiler interprets malloc family functions for dynamic allocation and reads the symbol table for static allocation. The profiler maintains a map that records the allocation call paths for dynamic data packets and names for static data objects, and their allocated memory ranges. Allocations from the GPU side are also collected in a separate map. By correlating the two maps, CUDAAdvisor overloads memcpy and captures the two memory ranges involved in memory copy.
Figure 3: The composition of data and iteration reorderings applied to a COO version of SpMV. The original code is in the top left rectangle. To the right of that rectangle is the SPF set and relation representations of the data spaces, iteration space, and data access functions. The transformation specifications are shown next to two large arrows.

2.3.3 Hybrid Approaches

Inspector-executor transformations enable loop and data layout transformations that require a run time component due to indirect array accesses that cannot be resolved until run time [49]. To enable compiler support for inspector-executor transformations, the compiler inserts calls to handwritten inspectors, generating inspectors for specific strategies and transforming key loop nests into executors.

Reorganizing data layout and computation to improve locality and parallelism, as well as the inner loop of a computation, which has an unknown number of iterations, can only be determined at run time. Thus, to parallelize computation at the outer loop level, work per thread may be imbalanced. The inner loop poses additional challenges for parallelizing compilers, which mostly operate on computations in affine domain, where subscript expressions and loop bounds are linear functions of surrounding loops. Code transformations, such as make-dense, compact and compact-and-pad, discussed in Section 3.1.1, facilitate composing inspector generation and executor optimization with standard loop transformations. Loop transformation frameworks enable composing of transformations, such as loop fusion, fission, skewing, permutation and tiling.

Sparse polyhedral framework (SPF) extends the polyhedral framework with the use of uninterpreted functions to represent nonaffine array accesses, nonaffine loop bounds, and inspector-executor transformations. An uninterpreted function is where a mapping of input to outputs is unknown. The advantage of representing transformations on code is that some information is known about them. A domain describes the bounds on indices into an index array and the range indicates the possible values being stored in index arrays. SPF uses uninterpreted functions at compile time to represent memory accesses, loop bounds and reordering functions not made explicit until run time. The extensions to compiler abstractions include nonaffine iteration spaces for functions unknown at compile time and determined at run time, nonaffine data accesses to specify data access information for sequences of loops represented in a loop chain abstraction, and nonaffine transfor-
Figure 4: Algorithmic patterns, data layout patterns, parallel patterns, address space patterns, vectorize patterns.

mations by specify information that will be passed from inspector to executor using uninterpreted functions.

SPF enables composition of multiple inspector-executor transformations. Inspector composition is the connection of outputs from one inspector to inputs of another. Figure 3 shows how changes made by one inspector can become input to another inspector. The top-left shows original code, which is SpMV written for coordinate sparse matrix format. To the right are SPF descriptions of data space index domains, iteration space for loop and data access functions.

2.4 Intermediate Representation

When translating a program from source language into code for a given target machine, a compiler may construct a sequence of intermediate representations (IR). Syntax trees are high level that depict the natural hierarchical structure of the source program and are well suited for tasks, like type checking and operator overloading, whereas machine-dependent tasks, such as register allocation and instruction selection, make use of a low-level representation. An IR is either a language, or a collection of internal data structures shared by phases of the compiler.

2.4.1 Intermediate Language

The LIFT-IL [48] expresses a program as compositions and nesting of functions, which operate on arrays. The foundation is Lambda calculus. Four algorithmic patterns are supported, which include mapSeq, reduceSeq, id and iterate. The data layout patterns supported include split, join, gather, scatter, zip and slide. The parallel patterns draw from the OpenCL hierarchical organization of parallelism and consists of threads that are grouped into work groups of local threads, or a flat organization where threads are simply global. A hierarchy is represented with three patterns, mapGlb\(^{\{0,1,2\}}\), mapWrg\(^{\{0,1,2\}}\), mapLcl\(^{\{0,1,2\}}\), where a mapLcl must be nested inside of mapWrg and the superscripts \(\{0,1,2\}\) represent thread dimensions. Semantics are similar to mapSeq, but \(f\) is applied in parallel.

The address space patterns include three primitives, which wrap functions and influence the OpenCL address space used to store the output: toGlobal, toLocal and toPrivate. This decouples the decision of where to store data (i.e. address space) from the decision of how data is produced (i.e. sequentially or parallel). Vectorize patterns transform data between scalar and vector types and patterns that apply a function to a vector. asVector, asScalar, mapVec. During code generation, \(f\) is transformed into a vectorized form, where \(f\) is applied to each scalar in the vector.

In the LIFT intermediate representation, classes organize programs as graphs and nodes as objects. Expressions are values and type associated and include literals, parameters and function calls. Function declarations are either lambda, a predefined pattern, or a user function. Anonymous functions are user-defined built-in patterns. A dependent type system keeps track of the length and shapes of nested arrays, which include array types, scalar types, vector types and tuple types (as
structs). Length information are arithmetic expressions of operations on natural numbers larger than zero, where named variables are unknown at compiler time. Memory allocation, FunCall node, is only performed for functions that actually modify data, where the functions contain User-Fun node (UserFun(add)). The compiler uses array length information to compute the required memory buffer. An internal view data structure is created, which remembers how memory should be accessed by subsequent functions.

Memory is allocated in one of three OpenCL address spaces. The subclasses of expressions include literals that reside in private memory, where parameters have their address space set when their function is called, FunCalls determine the address space of arguments and investigate which function was called, whereas UserFun takes the address space from the writeTo argument or is inferred from the address space of its arguments. toPrivate, toLocal, toGlobal change writeTo before recursing to produce output in the address space.

Multi-dimensional array accesses are implicit, where patterns determine which thread accesses which element in memory. The design simplifies lowering of high-level programs to Lift IR and guarantees data races are avoided since no arbitrary memory accesses are permitted. Two challenges include avoiding unnecessary intermediate results from functions which only change the data layout and generating efficient accesses to multi-dimensional arrays in memory from a flat representation. A view is an internal data structure that stores information for generating array accesses for functions that only change the data layout of an array. Barrier elimination generates a synchronization primitive after each occurrence of a parallel map pattern.

OpenCL code generation is the final stage, where low level optimizations are performed. The Lift IR graph is traversed following the data flow, where matching OpenCL code snippets are generated for every pattern. No OpenCL code is generated for split and toLocal, where the effect is recorded in view. For different map patterns, for loop generated, parallel variations are executed in parallel. For reduceSeq, a loop with an accumulation variable is generated by calling its function in every iteration.

### 2.4.2 Graph Lowering

The high-level IR of Glow [42] structures graphs as modules that contains multiple functions, which, in turn, consists of multiple nodes with variables. Persistent tensors are variables that are either public or private, which enables sharing between functions within a scope. An example of a compute graph for the $A/B$ expression is shown in Fig 5, where $A$ is auto-differentiated and updated with the gradient of the expression.

Instead of compiling high-level operators directly, Glow performs node lowering, where the compiler breaks high-level operator nodes into low-level linear algebra operator nodes. IRGen (IR generation) is the low-level IR and provides one-to-many translations, where each high-level node is translated into one or more instructions. Instruction-based representation that operates on tensors are referenced by addresses, which allows in-place transformations of buffers. This separation of levels allows additional graph optimizations to be performed at its respective level, since a new graph structure may affect decisions of the low-level instruction scheduler, as well as enable additional target-specific optimizations on the lowered graph in the backend.

A function in IR form contains a declare and program section. The first IR section declares the number of memory regions that are live throughout the lifetime of the program. The second part consists of a list of instructions, where each variable is annotated with the kind of initialization that the program should do. Global memory regions are found in declare, whereas locally allocated regions are found in program.
2.4.3 Computation Graphs

For building blocks in machine learning with individual cells and architectures with blocks that change rapidly, frameworks need to provide general facilities for dynamic control flow. Existing frameworks offer an in-graph approach (e.g. TensorFlow), where control-flow decisions are encoded as operations in the dataflow graph, and an out-of-graph approach, where control-flow decisions are executed in the client process, using control-flow primitives of a host language.

**Constructs**  TensorFlow Fold [32] provides dynamic batching of computation graphs, where a directed acyclic computation graph is received as input and batches of multiple input graphs are treated as a single disconnected graph. Source nodes are constant tensors, whereas non-source nodes are operations. Edges connect one output to the input of another node.

Each dynamic operation is instantiated once in the static data-flow graph. `tf.gather` receives input operations and outputs are fed into `tf.concat` and placed in a `tf.while_loop`. Each iteration evaluates all operations at a particular depth. The loop maintains state variables for each tensor type $t$ and feeds output of `concat` for tensor type $t$ at iteration $d + 1$. Indices for `gather` at iteration $d$ are drawn from edge labels $i$ for depth $d$ in the schedule. Initial values for state variables at depth 0 are constants in the input graph.

Combinators simplify the task of constructing neural networks for DGCs. The basic computation unit in `tf` is a block. In a typical DCG model, the input is a graph or a tree, and the output is a vector that can be attached to a loss for training. For example, consider a model where the inputs are sequences of words with varying lengths and the output is a sentence vector. Given a simpler block $f$ that operates on elements of the sequence, or $g$ on pairs of elements, the following combinators can be defined:
Figure 6: The static data-flow graph created by dynamic batching for a binary TreeRNN over parse trees (left), and input graph corresponding to the parse tree ((word₁, word₃), word₅) (right).

- **Map(f)**: yields \([f(x₁), f(x₂), ..., f(xₙ)]\). Applies \(f\) to each element of the sequence, e.g. embedding each of the words of a sentence into \(\mathbb{R}^N\).
- **Fold(g, z)**: yields \(g(...g(g(z, x₁), x₂),..., xₙ)\). Applies \(g\) sequentially in a leftward chain, e.g. running an RNN over a sequence. By default, \(z = 0\).
- **Reduce(g)**: yields \(g(\text{Reduce}([x₁,...,x_{⌊n/2}⌋]), \text{Reduce}([x_{⌊n/2⌋+1},...,xₙ]))\). Applies \(g\) in a balanced tree, e.g. max or sum-pooling over the elements.

Blocks are statically typed, where each block has an input and output type and inferred where possible.

- **Input** - objects in host language (Python), such as trees and dictionaries
- **Tensor dtype, shape** - tensors of dtype, shape
- **Tuple(t₁, ..., tₙ)** - tuple of values of types \(t₁, ..., tₙ\).
- **Sequence(t)** - sequence of type \(t\), any length
- **Void** - unit type

Blocks are composed hierarchically, where each block expression is always a tree. Non-terminals are combinators (Map, Fold), which take simpler blocks as arguments. The leaves of trees are atomic blocks.

- **Scalar**: \(\text{Input} \rightarrow \text{Tensor}\) - Python scalar to tensor
- **Tensor**: \(\text{Input} \rightarrow \text{Tensor}\) - Numpy array to tensor
- **Function(h)**: \([\text{Tensor or Tuple(Tensor,...)}] \rightarrow [\text{Tensor or Tuple(Tensor,...)}]\) - defines an operation \(h\) over tensors, operations with multiple inputs and outputs use tuples of tensors
- **InputTransform(h)**: \(\text{Input} \rightarrow \text{Input}\) - applies user-defined Python function \(h\) to pre-process input

Other important combinators include:

- \(b₁ \gg b₂\) - Function composition; the output of \(b₁\) is fed to the input of \(b₂\).
Figure 7: Block architectures for a pipeline, feed-forward attention, binary Tree-LSTMs, and weave module for molecule graphs.

- **Record** \( \{ l_1 : b_1, ..., l_n : b_n \} \) : Input \( \rightarrow \) Tuple\( (t_1, ..., t_n) \) - Take Python dictionary or tuple as input, applies each \( b_i \) block to field labeled \( l_i \) to yield an object of type \( t_i \)

- **OneOf** \( (b_1, ..., b_n) \) : Input \( \rightarrow \) \( t \) - Conditionally dispatches on its input to one of the blocks \( b_1, ..., b_n \)

- **Optional** \( b \) : Input \( \rightarrow \) \( t \) - Applies \( b \) if input not None, otherwise returns zeros (special case of OneOf)

- **AllOf** \( (b_1, ..., b_n) \) : \( t_0 \) \( \rightarrow \) Tuple\( (t_1, ..., t_n) \) - Passes input of type \( t_0 \) to each of blocks \( b_1, ..., b_n \) returning a tuple of results

**Control Flow** A set of primitives for large-scale machine learning, particularly recurrent neural networks (RNN) which have variable-length sequences, enable dynamic control flow [63]. Most traditional neural networks, including multi-layer perceptrons and convolutional neural networks (CNN), are static and composed of a fixed number of layers, each with fixed operators. For learning over sequences, such as RNNs, fixed-length iterations are typically unrolled statically. However, dynamic control flow is needed in RNNs, tree neural networks and reinforcement learning. For instance, RNNs calculate an adaptive computation time with nested while-loops to learn how many computational steps to take at each time step of the outer loop. The amount of available memory in an accelerator limits sequence lengths, since memory usage often grows linearly along with sequence length.

The TensorFlow system architecture runtime is responsible for execution of the dataflow graph. Optimizations, such as common subexpression elimination and constant propagation, can be performed. Partitioning a graph cuts edges between two devices, which automatically replaces an edge with a pair of communication operations \( \text{Send}(t, k) \) and \( \text{Recv}(k) \) and shares a rendezvous key. Each subgraph is executed by a local executor, which communicates directly amongst themselves using \( \text{Send} \) and \( \text{Recv} \) operations with no involvement from the central coordinator. \( \text{Send}(t, k) \) publishes tensor \( t \) to devices and \( \text{Recv}(k) \) pulls \( t \) under key \( k \) from the sender’s device, blocking if necessary.

Control-flow primitives, displayed in Figure 8 (left), include **Switch**, **Merge**, **Enter**, **NextIteration**, and **Exit**. Every execution of an operation takes place within a “frame,” which serves as dynamically allocated execution contexts associated with each iteration of a loop. To compile control-flow constructs, \( \text{cond}(p, \text{true}_f, \text{false}_f) \) invokes \( \text{true}_f \) and \( \text{false}_f \) respectively to construct two branches of a condition. A guard ensures that any operation in a branch will be executed only when that branch is taken. External tensors become available at any time, where a switch for each external tensor is used in order to maximize parallelism. Figure 8 (right) sketches a dataflow graph for a while-loop with a single loop variable.

A local executor starts from source nodes and repeatedly executes nodes that become ready. A node is ready when all inputs are available. All \( \text{Recv} \) nodes are regarded as source nodes. The executor must manage possible concurrent execution of multiple instances of the same operation and determine the completion of the entire execution. A redesigned executor is represented as a
tuple \((\text{value}, \text{is}_\text{dead}, \text{tag})\), where \text{value} is the tensor value, \text{is}_\text{dead} is a boolean that indicates whether a tensor is on an untaken branch of a \text{Switch} and \text{tag} is a globally unique identifier that defines dynamic execution of a context, such as a frame.

The executor implements evaluation rules, where each evaluation rule \(\text{Eval}(e, c) = r\) describes how to evaluate an expression \(e\) in frame \(c\) to yield output \(r\). The last rule applies to all non-control-flow operations, such as \text{dead} input, which skips computation and instead propagates the \text{dead} signal downstream. More parallelism typically leads to more memory consumption. A limit of 32 parallel loops worked well, as discovered. The challenge for distributed, dynamic control flow occurs when the subgraph of a conditional branch or a loop body is partitioned across devices. No synchronization is necessary after the loop iteration, which severely limits parallelism. The partitioned subgraphs can make progress independently without a centralized coordinator.

The propagation scheme handles distributed execution of nested conditionals and interacts with the distributed execution of loops. However, when many \text{Send-Recv} pairs across devices are on a rarely taken conditional branch, a large number of \text{is}_\text{dead} signals, used for unblocking system resources, can cause performance issues. In order to know whether to proceed or to exit at each iteration, the graph is automatically rewritten with simple control-loop state machines. Figure 8 illustrates a distributed execution of a while-loop, which makes use of \text{Send-Recv} structures for communicating messages. The overhead of distributed execution of loop is that every participating device needs to receive a boolean at each iteration from the device that produces the loop predicate.

### Listing 5: Evaluation rules for control-flow operators (left). Computing the gradient of a loop by unrolling (right).

```
1 \text{Eval}(\text{Switch}(p, d), c) = (r_1, r_2), \text{where}
2 r_1 = (\text{value}(d), p || \text{is}_\text{dead}(d), \text{tag}(d))
3 r_2 = (\text{value}(d), !p || \text{is}_\text{dead}(d), \text{tag}(d))
4 \text{Eval}(\text{Merge}(d_1, d_2), c) = r, \text{where}
5 r = \text{if} \text{is}_\text{dead}(d_1) \text{then} d_2 \text{else} d_1
6 \text{Eval}(\text{Enter}(d, \text{name}), c) = r, \text{where}
7 r = (\text{value}(d), \text{is}_\text{dead}(d), \text{tag}(d)/\text{name}/0)
8 \text{Eval}(\text{Exit}(d), c) = r, \text{where}
9 r = (\text{value}(d), \text{is}_\text{dead}(d), c.\text{parent}.\text{tag})
10 \text{Eval}(\text{NextIteration}(d, c)) = r, \text{where}
11 \text{tag}(d) = \text{tag}/\text{name}/n
12 r = (\text{value}(d), \text{is}_\text{dead}(d), \text{tag}/\text{name}/(n+1))
13 \text{Eval}(\text{Op}(d_1, \ldots, d_m), c) = (r_1, \ldots, r_n), \text{where}
14 \text{value}(r_1), \ldots, \text{value}(r_n) = \text{Op}(\text{value}(d_1), \ldots, \text{value}(d_m))
15 \text{is}_\text{dead}(r_i) = \text{is}_\text{dead}(d_1) || \ldots || \text{is}_\text{dead}(d_m), \text{for all} i
16 \text{tag}(r_i) = \text{tag}(d_i), \text{for all} i
```

```python
1 w = tf.Variable(tf.random_uniform((10, 10)))
2 x = tf.placeholder(tf.float32, (10, 10)))
3 a_0 = x
4 a_1 = tf.matmul(a_0, w)
5 a_2 = tf.matmul(a_1, w)
6 a_3 = tf.matmul(a_2, w)
7 y = tf.reduce_sum(a_3)
8 g_y = 1.0, g_w = 1.0
9 # the gradient of reduce_sum is broadcast
10 g_a_3 = tf.fill(tf.shape(a_3), g_y)
11 # Apply MatMulGrad() 3x, acc g_w.
12 g_a_2 = tf.matmul(g_a_3, tf.transpose(w))
13 g_w += tf.matmul(tf.transpose(a_2), g_a_2)
14 g_a_1 = tf.matmul(g_a_2, tf.transpose(w))
15 g_w += tf.matmul(tf.transpose(a_1), g_a_2)
16 g_a_0 = tf.matmul(g_a_1, tf.transpose(w))
17 g_w += tf.matmul(tf.transpose(a_0), g_a_1)
```

Figure 8: Control-flow primitives (left), distributed execution of a while-loop (right).
Gradient computations usually take more than half the compute time when training models. Automatic differentiation and memory management alleviates communication overheads. For backpropagation with control flow, the automatic differentiation algorithm is a vector chain rule, described as follows:

1. Identify subgraph $G$ of operations between symbolic tensor representing $y = f(x_1, x_2, ...)$ and each parameter tensor $x_1, x_2, ...$

2. For each edge in $G$, which represents an intermediate value $t$ in $f$, set $\text{Grads}[t] := 0$, $\text{Grads}[y] := 1$.

3. Traverse vertices of $G$ in reverse topological order. For each vertex representing an intermediate operation, $o_1, ..., o_n = \text{Op}(i_1, ..., i_m)$ invokes corresponding gradient function $\partial i_1, \partial i_2, ..., \partial i_m = \text{OpGrad}(i_1, ..., i_m, \text{Grads}[o_1], ..., \text{Grads}[o_n])$, where each $\partial i_k$ is added to $\text{Grads}[i_k]$.

4. After traversing every vertex, the gradient is $\frac{\partial y}{\partial x_k}$.

Asynchronous Model-Parallel Training  AMPNet supports asynchronous model-parallel training on dynamic neural networks [14], which is similar to model parallelism where sequential operations are computed in a pipelined-parallel fashion. During training, a computation graph is distributed across compute nodes, while nodes update one another with activations. During training, nodes from the computation graph exchange forward or backward messages. Once the number of accumulated gradients exceed a threshold $\text{min_update_frequency}$, a local update is applied and the accumulator gets cleared. The staleness of a gradient is measured by the number of updates between forward and backward computations that produce the gradient. A small $\text{min_update_frequency}$ may increase gradient staleness, whereas a large update reduces variance, but slows down convergence. $\text{max_active_keys}$ controls the maximum number of active instances in-flight at any point in time. Setting $\text{max_active_keys}$ to 1 restricts to single-instance processing, whereas more in-flight messages generally increases hardware utilization at the cost of increased gradient staleness.

AMPNet defines a static IR graph and a dynamic controller loop that supplies instances and other data while throttling asynchrony accordingly, as shown in Fig 9 (L). Each message consists of a payload and a state, where a payload is a tensor and a state is model specific and keeps track of algorithm and control flow information. The final loss layer initializes backpropagation through the IR graph. $\text{max_active_keys}$ represents the number of messages in flight.

A condition node is parameterized by a function $f$ that queries the state of an incoming message and, based on the response, routes input to one of the successor nodes. Phi nodes join propagated messages it receives from each of its ancestor nodes and records the origin of backpropagation to
the correct origin. A Phi node, like payload nodes, must be parameterized over the keying function on the state of the incoming message. Invertible state updates are parameterized by $f$ and $f^{-1}$ that operate on the state of the message to satisfy $f^{-1}(f(x)) = x$.

To enable interaction with data parallelism and replicas, the approach is to replicate the linear layer and place replicas inside Cond and Phi nodes, as shown in Figure 9 (R). Different instances, or messages from the same instance but with different positions in sequence, can be processed in a pipelined-parallel fashion using three replicas in this case. To enable parameters to be shared among the replicas, use infrequent end-of-epoch replicas for synchronization.

**Operation Batching** Since the grouping of operations into parallel batches is crucial for making the most of available hardware resources, on-the-fly batching for dynamic computation graphs [35] separates graph construction from execution using operator overloading and lazy evaluation. Given a large enough computation that makes batching possible, such as summing the losses of several instances, the framework handles the low-level details of operation batching using JIT compilers.

For on-the-fly batching, the proposed method identifies and aggregates computation graph nodes that can be executed in a batched fashion for a given graph. This reduces the need for padding and masking, and allows seamless efficient execution, as well as identifying batching opportunities not apparent in input-centric view. Lazy evaluation performs forward evaluation only when the resulting value is requested by the user. A computation graph (Fig 10, L) is converted where operations with no dependencies are batched together (Fig 10, R).

Compatibility groups partition nodes into groups, where nodes in the same group have the potential for batching. Each node is associated with a signature, such that nodes that share the same signature can be executed in a single operation if inputs are ready. Signatures vary depending on the operation the node represents. For instance, element-wise operations would imply that all nodes with the same operation can be batched together, so the signature is simply the operation name ($tanh$, log, ...), or a slice of the input matrix. An execution order ensures that each node is executed after its dependencies, and nodes with the same signature and no dependencies are scheduled for execution on the same step, which is a NP hard problem.

**Depth-based batching**, used in TensorFlow Fold, calculates the depth of each node in the original computation graph, batching nodes with identical depth and signature. By construction, nodes in the same depth are not dependent on each other. **Agenda-based batching**, the proposed method, tracks “available” nodes that have no unresolved dependencies. For each node, a count of unresolved dependencies is maintained and initialized as the number of inputs to the node. The agenda
is initialized by adding nodes that have no incoming inputs. At each iteration, nodes are selected from the agenda together with all available nodes in the same signature and grouped into a single batch operation. When nodes are removed from the agenda, the dependency counter of successors is decremented and new zero-dependency nodes are added to the agenda. The process is repeated until all nodes are processed.

To avoid prematurely executing nodes that can be potentially batched, a heuristic is introduced based on the average depth of all nodes with their signatures such that nodes with a lower average depth are executed earlier. This prioritizes nodes that occur in the earlier parts of graph, resulting in successor nodes executing later and hopefully batched together, as seen in the loss calculations. The tradeoff is that execution needs to occur quickly so that overhead due to batch scheduling calculations do not cancel out efficiency gains from operation batching.

To execute and update graphs in both forward and backward directions, single nodes are converted to batched nodes, which requires modification of the underlying operations such as converting multiple matrix-vector ops $W h_i$ to a single matrix-matrix operation $W H$. Ensuring contiguous memory, in some cases, require rearrangements of inputs ($h^{(i)}$) via memory copies ($H_i$).

## 3 Optimizations

Improvements to execution performance of an application can be enabled, either at the machine-independent level or at the hardware level.

### 3.1 Machine-Independent Optimizations

The task of the code optimizer is to replace a set of instructions with a faster sequence of instructions, since high-level constructs that are naively translated into machine code can introduce unnecessary run time overhead. Data-flow analyses gather information about a program, which include properties that hold each time an instruction is executed. For instance, constant-propagation analysis computes, for each point in the program and for each variable used by the program, whether that variable has a unique constant value at that point, which may be used to replace variable references with constant values. Since programs spend most of the time executing loops, optimizations that improve the performance of loops can have a significant impact in reducing overall executing time.

#### 3.1.1 Parallelism and Locality

Parallelism and locality factor into the overall execution performance of a computationally intensive program. Parallelism describes the amount of work that can be completed concurrently, based on the available hardware resources and the total number of work to complete. Data locality for an instruction issue is achieved when data is readily available in register or L1 cache, which attributes to high performance, since a cache miss requires fetching data from the memory hierarchy, a non-trivial task for programmers. Ideally, a compiler automatically translates ordinary sequential programs into efficient parallel programs and optimizes the locality of the programs. The lack of high-level knowledge of an application that only preserves the semantics of the original algorithm may overlook any opportunities for parallelism.

**Straight Line Scalar Optimizations** Data accesses can be rewritten with fewer operations and stored in registers with straight-line scalar optimizations (SLSO) [59], which eliminates redundant operations while increasing data locality for multi-dimensional arrays typical in scientific computing. For instance, $(b + 1) \times n$ in SLSO is $b \times n + n$ if $b \times n$ was computed by its dominators. Unrolled loops can walk through an array with a fixed access pattern. Expressions that compute indices or
pointer addresses of these accesses are often partially redundant. Pointer arithmetic reassociation (PAR) folds more computation into addressing modes, which may not be in a form that NVPTX address folding matches. \[ \text{reg} + \text{immOff} \] is the only supported nontrivial addressing mode, where \text{reg} is a register and \text{immOff} is a constant byte offset. Addressing mode is used to fold pointer arithmetic that adds or subtracts an integer constant, which extracts additive integer constants from the expression of pointer addresses. Straight-line strength reduction (SLSR) reduces the complexity of operations along dominator paths. Strength reduction candidates are identified in certain forms and replacement for candidates are made in the same form.

Local reassociation transforms each individual expression by rankings, which may imprecisely reflect how values are associated in other expressions. Global reassociation transforms expressions according to how operands are actually associated in its dominators and selects forms that exposes more CSE opportunities by promoting redundancy elimination, but is more expensive since global information is used for decisions.

A limitation of SLSR is that it cannot optimize instructions not dominating one another. Speculative execution hoists side-effect free instructions from conditional basic blocks, which dominate more instructions and are consequently more likely to be optimized by SLSO. Speculative execution promotes predication, which is another benefit. Predicated instructions are executed only when true at run time, since jumps are expensive in SIMD, which facilitates in instruction scheduling.

Listing 6: Original (left), speculative execution (middle) and straight-line optimizations (right).

```plaintext
1. if (b)
  2. u = a[i];
  3. if (c)
  4. v = a[i+j]

1. p = &a[i];
  2. if (b)
    3. u = *p;
    4. if (c)
      5. v = *q;
```

Loop and Data Transformations  Transformations for sparse matrix codes for loop and data optimizations were added to the CUDA-CHiLL compiler [55]. make-dense takes as input a set of non-affine array index expressions and introduces guard conditions and needed dense loops to replace non-affine index expressions with affine accesses, which enables loop transformations, such as tiling. Below shows before and after code templates for make-dense. make-dense identifies the statement and index expressions to which the transformation should be applied to. Each input expression is replaced with a dense loop iterator \( I_1, \ldots, I_m \), which iterates over a range of corresponding index expressions and placed immediately outside some loop \( I_k \). \( I_k \) is the innermost loop upon which any of \( m \) index expressions depend on. A guard surrounds a loop body to compare new dense iterators to the associated index functions. Non-affine index expressions are replaced by references to the new loop iterator.

Listing 7: make-dense(\( s_0, [idx_1, idx_2, \ldots, idx_m] \)), before and after.

```plaintext
1. for(I_1)
2. ...
3. for(I_k)
4. ...
5. for(I_n)
6. s0: ...A1[idx_1(I1,I2,...,Ik)]...
7. ...A2[idx_2(I1,I2,...,Ik)]...
8. ...A3[idx_m(I1,I2,...,Ik)]...

1. for(IX_1)
2. ...
3. for(IX_2)
4. ...
5. for(IX_M)
6. for(IX_k)
7. if(IX1 == idx1(I1,I2,...,Ik))
8. && IX2 == idx2(I1,I2,...,Ik)
9. && ...
10. && IXm == idxm(I1,I2,...,Ik)
11. for(In)
12. s0: ...A1[IX1]...
13. ...A2[IX2]...
14. ...A3[IX3]...
```

Compact (also compact-and-pad) inspector-executor transformations use an automatically-
generated inspector that gathers iterations of a dense loop that are executed. The optimized executor visits those iterations and the transformed code (executor) is a compacted loop that can be further optimized. The compact transform replaces the dense loop containing conditional guarding execution with a sparse loop that only visits iterations when the condition is true.

Each of the outer loops $I_k : I_1, ..., I_{k-1}$ are represented by dimensions in offset_index. For iterations that satisfy the guard, an explicit index records the original value of $I_k$ and uses, in place, references to $I_k$ in the executor. The inspector needs to ensure it only stores the specific value of $I_k$ once. A compacted loop can be marked with a flag, which is counted only once. After compact transformation, the resulting code has more non-affine loop bounds and array index expressions.

Listing 8: compact($s_0, I_k$), before and after.

```plaintext
for(I1)...
for(Ik-1)
for(Ik)
for(Ik+1)
...for(Ik+d)
if (cond(Ik))
s0: ...X[Ik]...
```

```plaintext
for(I1)...
for(Ik-1)
for(Ik') = offset_idx[1][...][Ik'-1];
Ik < offset_idx[1][...][Ik'-1+1];
Ik'++
for(Ik'+1)
...for(Ik'+d)
if (cond(explicit_idx[Ik']))
s0: ...X[explicit_idx[Ik']]...
```

Additionally, the compact-and-pad inspector performs data transformation, inserting zeros to correspond to the optimized executor. The arguments to the statement of the loop body are $s_0, I_k$ is where the loop level of iterations is to be evaluated against the guard. $A$ is the data transformation to be applied to.

Conceptually, the inspector copies the footprint of compacted iterations associated with a specified array $A$ into transformed $A_{\text{prime}}$, which will be referenced in the optimized executor code. The count of compacted loop’s iterations that satisfy the guard assumes a leading dimension of the newly reorganized array $A_{\text{prime}}$. When the compacted loop level is not the innermost, the inner loops ($I_{k+1}, ..., I_{k+d}$) iteration space are used to derive the size of $A_{\text{prime}}$. For each loop level $j$ nested inside $I_k$, the size of dimension corresponding to that level is computed as $ub_j - lb_j + 1$, $lb_j \leq I_j \leq ub_j$, which represents the lower and upper bounds in the set of constraints from Project($S, I_k$),..., Project($S, I_{j-1}$).

Listing 9: compact-and-pad($s_0, I_k, A$), before and after.

```plaintext
for(I1)...
for(Ik-1)
for(Ik)
for(Ik+1)
...for(Ik+d)
if (cond(Ik))
s0: ...+=...A[...]
* X[Ik]...
```

```plaintext
for(I1)...
for(Ik-1)
for(Ik') = offset_idx[1][...][Ik'-1];
Ik < offset_idx[1][...][Ik'-1+1];
Ik'++
for(Ik'+1)
...for(Ik'+d)
s0: ...+=...A_{\text{prime}}[Ik'][Ik'+1][...][Ik'+d]
* X[explicit_idx[Ik']]...
```

In optimizing the inspector and executor, dynamic memory allocation and reduced traversals of the inspector are needed. OSKI requires two passes over the input, including initialization of the data structure and traversing and counting the number of nonzeros. Dynamic memory allocation is used to create a linked-list of nonzero elements or blocks when discovered by the inspector or use static memory allocation when the size known apriori, (e.g. ELL). This allows copy data associated with nonzeros while counting them in single effective pass over the input matrix. There is overhead in copies of linked list to array, but is faster compared to two separate passes.

Listing 10: CSR SpMV after make-dense, skew and permute (left). Executor for DIA SpMV (right).
The example describes a parallel GPU implementation of DIA. To uncover diagonals implicit in SpMV CSR format, make-dense convert iteration space of sparse computation to its corresponding dense one, corresponding transformation relations (above, left)

\[ T = \{ [i, k, j] \rightarrow [i, k-1, j] \} \]

\[ T = \{ [i, k, j] \rightarrow \{k, i, j\} \} \]

The CUDA-CHiLL script for DIA is shown after make-dense and compact-and-pad commands, where the inner \( i \) loop is parallelized for threaded execution on a GPU.

Active Learning The optimal set of compilation parameters for a program can be discovered with active learning [37], including loop unrolling, cache tiling and register tiling factors to build a program-specific model, which can be used to predict the run time from a given set of optimizations. Current approaches for creating machine learning based heuristics do not consider sample size as a parameter for optimization. Knowledge can be gained by building up the algorithm over time to adaptively select a more appropriate sample size per example, which speeds up training overall.

At each iteration of a learning loop, the algorithm considers not only a new example, but also whether trying the old example is more profitable, similar to a multi-armed bandit problem. A scoring function quantifies the uncertainty of a particular point in the space, given known knowledge. As knowledge is gained, noisy examples or examples in a complex space begin to stick out and more likely visited because more information can be extracted.

The algorithm constructs a model \( M \) with \( n_{\text{init}} \) training examples chosen from \( F \) potential examples and \( n_{\text{obs}} \) is a fixed number of observations. At each iteration, the candidate \( C \) combines \( n_c \) random unobserved points, and examples previously seen but less than \( n_{\text{obs}} \) times. The next training example \( x \) is chosen based on predicted usefulness and the run time \( y \) is measured once more to update the model.

Dynamic trees are needed to estimate the uncertainty of prediction. Advantages of dynamic trees include the ability to evolve over time as new data comes in without reconstructing the model from the ground up, the estimation of uncertainty at any given point in the space and the avoidance of overfitting training data.

The static models used with the dynamic tree framework is a traditional decision tree with regression. A set of rules recursively partitions the search space into a set of hyper-rectangles, such that the training examples with the same or similar output values are contained within the
same leaf node. Dynamic trees change over time as new information is introduced by a stochastic process, avoiding the need for pruning at the end. At time $t$, a tree $T_t$ is derived from the training data $(x, y)^t$. When new data $(x_{t+1}, y_{t+1})$ arrives, an updated $T_{t+1}$ is created.

Estimating which training example from a pool of candidates $C$ that yields the most profit is the crucial part of active learning. Two heuristics include estimated variance of the maximized output relative to other candidates and selecting a candidate that reduces the predicted average variance across other candidates.

### 3.1.2 Optimizing Neural Networks

This subsection covers how neural networks, particularly CNNs and RNNs, can be optimized for high performance. Techniques include memory layout strategies, computing the convolution in the Fourier domain and the unrolling and fusing of computations.

#### Memory Efficiency for CNNs on GPUs

Two aspects that pose non-trivial impact on memory efficiency and overall CNN performance include data layouts and redundant off-chip memory access [28]. Data layouts are greatly affected by GPU thread organization, which determines memory access pattern and has critical performance impact. For CNNs, data is organized in multi-dimensional 4D arrays. Depending on the placement, there can be up to 24 ways to store data in memory. Redundant off-chip memory access also affects the memory efficiency of memory-bounded pooling layers and classifier (i.e. softmax) layers, which is far from optimal. CNN requires multiple steps to complete and also has sequential data dependencies across steps.

The working principle of CNN is to extract and combine local features from high-resolution feature maps into more abstract low-resolution feature maps. Two alternating types of layers include convolutional and pooling layers. The last few layers are fully-connected classifiers that combine all local features together to produce an abstracted classification result.

A convolutional layer extracts various features, such as oriented edges, corners and crossings from input feature maps via convolutional filters and combines them into more abstract output feature maps, where each feature map is a 3D volume (width, height, depth). Convolution is computed as follows:

$$
Out_{co}[Ni][Co][Hi][Wi] = \sum_{ci=0}^{C} \sum_{fh=0}^{Fh} \sum_{fw=0}^{Fw} \sum_{ln_{co}[Ni][Ci][Hi+f_h][Wi+f_w]} \times \text{filter}[Co][Ci][f_h][f_w],
$$

where $Ni$ is the batch size, $Ci$ is the depth or number of input feature maps, $Hi$ and $Wi$ are height and width of feature map and $Fh$ and $Fw$ are convolutional filters.

A pooling layer, or downsampling layer, summarizes features from its neighbors by performing average or max operations. $X$ and $Y$ defines the size of the pooling window, whereas the stride is the distance between successive pooling windows. If the stride is smaller than the window size, pooling is performed in an overlapped manner.

![Figure 11](image-url)
\[
Out_{po}[Ni][Ci][Hi][Wi] = \left( \sum_{x=0}^{X} \sum_{y=0}^{Y} In_{po}[Ni][Ci][Hi \ast strde + y][Wi \ast strde + x] \right) / Y / X
\]

Pooling layers are usually paired with convolutional layers in CNNs. Compared to convolutional layers, pooling layers have low arithmetic intensity \(O(N \times C \times H \times W)\) and are mainly memory bounded by bandwidth and latency.

A classifier, or softmax layer is the final layer of CNN for classification, which computes distances over different labels. Before the softmax layer, fully-connected layers, flatten 4D feature maps to a 2D matrix. Matrix multiplication implements the fully-connected layer, where the output is fed to the softmax layer, which finds the maximal possibility over each batched image and shifts each possibility by the maximum. The exponential operation is performed on each batched image and summation normalizes all the possibilities.

\[
\begin{align*}
Maxv[Nx] &= \sum_{x=0}^{X} \sum_{y=0}^{Y} \max(ln[Nx][Cy]) \\
Midv1[Nx][Cy] &= \sum_{x=0}^{X} \sum_{y=0}^{Y} (ln[Nx][Cy] - Maxv[Nx]) \\
Midv2[Nx][Cy] &= \sum_{x=0}^{X} \sum_{y=0}^{Y} \exp(Midv1[Nx][Cy]) \\
Sumv[Nx] &= \sum_{x=0}^{X} \sum_{y=0}^{Y} \sum(Midv2[Nx][Cy]) \\
Out[Nx][Cy] &= \sum_{x=0}^{X} \sum_{y=0}^{Y} (Midv2[Nx][Cy] / Sumv[Nx])
\end{align*}
\]

Each step in the softmax layer involves element-wise matrix or matrix-vector computation. Matrix vector operations are low in arithmetic intensity, but are memory bounded because of the intermediate data communication across different steps.

The memory issues in CNNs include data layout and off-chip memory accesses. The data layouts considered are NCHW and CHWN, where \(N\) is the batch size, \(C\) is channel, \(H\) is height and \(W\) is width. Caffe and cuDNN uses NCHW layout. The two convolution implementations used in cuDNN 4 include MM and FFT. FFT has significant overhead for intermediate data and works for certain types of convolution layers due to several limitations. cuda-convnet uses CHWN and the direct convolution method.

CHWN and HCWN uses \(N\) as its lowest dimension. CHWN is used in cuda-convnet, where each convolution filter is applied on the \(H\) and \(W\) dimensions to generate one output feature map. HCWN has the same performance as CHWN, since memory coalescing does not change for \(N\) and retains the same data reuse within the batch. 2D convolution operators are applied on the \(H\) and \(W\) dimensions, where a 4D tensor is mapped to a 2D array and computation is performed with matrix multiplication. NCHW, used in Caffe and cuDNN, applies this dimension reduction. Special FFT in cuDNN also inherits the NCHW layout.

Performance studies show that cuda-convnet (CHWN) is more sensitive as \(N\) changes, whereas cuDNN (NCHW) is more sensitive as \(C\) changes. For CHWN, the \(N\) dimension is used for both memory coalescing and data reuse, thus the sensitivity. cuda-convnet allocates warps of 32 threads
to process 32 images. In cases where the batch size is 128, each thread handles 4 images. Memory reuse performance is reduced when $N < 128$ and degrades as the number of images are decreased.

cuDNN and Caffe (NCHW) invokes cuBLAS. A matrix unroll step along $H$ and $W$ merges the multiple dimensions into two dimensions. When $C < 32$, cuda-convnet (CHWN) performs much better, since there is no overhead from the matrix expansion. cuDNN is better when $C > 32$ where matrix expansion exhibits better data reuse and results in higher parallelism due to the merging of dimensions.

For a given convolution configuration, CHWN is preferred if $C < C_t$ and $N \geq N_t$ thresholds as the cost of memory transformation overhead used in NCHW is high and $N$ is large enough to achieve both memory coalescing and data reuse. For the rest of the configurations, NCHW is the better choice. Due to memory design differences, such as cache capacity and bandwidth, $C_t$ and $N_t$ thresholds may vary. For instance, the Titan black $(C_t, N_t)$ is $(32, 128)$, whereas for GTX Titan X, $(128, 64)$. Profiling is needed to determine the thresholds.

When considering data layouts for FFT-based methods, additional memory is needed to pad the filter kernel to match the size of the feature map, which adds overhead for small filters. Tiling facilitates in reducing overhead. cuDNN-4 has FFT and FFT-tiling, where NCHW is employed as the layout. The FFT method performs better than cuDNN-MM when the filter kernels are large, batch sizes are large or consists of many channels. For small channels, FFT performs much worse. The overhead of multiple steps, e.g. multiple kernel launches, streaming memory, zero padding and FFT outweigh the algorithmic advantages.

Pooling layers are memory-intensive and represented as 4D. cuda-convnet (CHWN) outperforms others (NCHW). CHWN works through each slice of feature maps in 4D and memory coalesced accesses are achieved along the lowest $N$ dimension. For NCHW, the memory access is different, where pooling is directly applied to pixels consecutively stored in memory $(H, W)$. Consecutive threads in warps are uncoalesced and can overfetch, resulting in poor memory efficiency. Thus, CHWN is preferred over NCHW for pooling. For CHWN, direct convolution performs better than MM or FFT, whereas MM and FFT performs better with NCHW layouts. One-time profiling is typically used to determine the overhead transformation over improvements obtained from a suitable data layout.

Fast Convolutional Nets  
Fast convolutional nets with fbFFT [53] evaluates discrete convolution and cross-correlations used in CNNs, a known approach in the signal processing field. Forward propagation inputs are a set $f$ of input feature planes $x_i$, cross-correlated with $f' \times f$ different filter kernel weights $w(j,i)$, producing output feature planes $y_j$, where each input and output feature is part of a mini-batch $S$ that has $x_{(s,i)}$, $y_{(s,j)}$, with $i \in f$, $j \in f'$ and $s \in S$.

$$y_{(s,j)} = \sum_{i \in f} x_{(s,i)} \star w_{(j,i)}$$

where feature planes $f$ are reduced, or summed, pointwise. For back-propagation, the gradient of loss with respect to the outputs are convolved with kernels

$$\frac{\partial L}{\partial x_{(s,i)}} = \sum_{j \in f'} \frac{\partial L}{\partial y_{(s,j)}} \star x_{(s,i)}$$

with reduction over $f'$. Finally, the kernel weights are updated using the gradient of loss with respect to weights:

$$\frac{\partial L}{\partial w_{(j,i)}} = \sum_{s \in S} \frac{\partial L}{\partial y_{(s,j)}} \star x_{(s,i)}$$
with reduction over \( S \). Each input plane is size \( h \times w \), each filter kernel is \( k_h \times k_w \), and the output planes \( y(s,i) \) are size \((h - k_h + 1) \times (w - k_w + 1)\).

The convolution of two discrete signals can be performed with lower asymptotic complexity by multiplying in the frequency domain, instead of the time domain via direct computation. When applied to the forward pass:

\[
y(s,j) = \sum_{i \in f} x(s,i) * w(j,i) = \sum_{i \in f} \mathcal{F}^{-1}(\mathcal{F}(x(s,i)) \circ \mathcal{F}(w(j,i))^*)
\]

where \( * \) denotes complex conjugation and \( \circ \) is pointwise product. Applying FFT yields \( O(Sf f'n^2 + (Sf + f f' + Sf') n^2 \log(n)) \) procedure instead of \( O(Sf f'n^2 k^2) \), with \( n = h = w \) and \( k = k_h = k_w \).

Since the exact tensor dimensions are given, the Hermitian symmetry property of 2D DFT for \( \mathbb{R} \)-valued inputs require storage of only half the complex entries, where the rest can be obtained by complex conjugation. This results in array sizes of \( \lceil \frac{n+1}{2} \rceil \) + 1. Interpolation is performed by zero-padding for boundary conditions and interpolating over the same Fourier basis.

Transpositions, which convert BDHW layout into HWBD, prepare tensors for \texttt{cgemm} matrix multiplication library calls. The \texttt{cgemm} library calls are performed on transposed tensors in the frequency domain. A \texttt{cgemm} casted call invokes the cuBLAS routine. The tensors are transposed back to BDHW format and the 2D inverse FFT is performed. The resulting tensor \((h+p_h) \times (w+p_w)\) is clipped to the appropriate final size: \((h-k_h+1) \times (w-k_w+1)\) for \texttt{fprop}, \( h \times w \) for \texttt{bprop} and \( k_h \times k_w \) for \texttt{accGrad}. The appendix of [53] provides the derivation of the Fourier transform that recursively decomposes the radix-2 Cooley-Tukey algorithm into an odd and even part, representing the roots of unity.

\texttt{fbFFT} designs a set of leaf kernels with well-tuned in-register performance and reduces the larger problem to the combination of these kernels by data and loop tiling and recursive decompositions. For common dimensions, convolutional layers consisting of many batched small 2D convolutions can be used. Some limitations of \texttt{cuFFT} include the need for zero padding embedded in input and output arrays, which may require a duplicate allocation of a larger memory region and data copies from non-padded tensors to padded tensors, since memory consumption and spurious copies affect latency significantly. Instead, batched 1D-FFT and 2D-FFT of sizes 2-256 can be used, which achieves up to 78% efficiency and 97.5% occupancy.

For 1-D FFT and 2-D FFT, \( n \leq 32 \), each warp thread loads one real element of input vector and locally computes one complex twiddle factor, or root of unity. At each step, all warp threads exchange data and twiddle factor with another thread in parallel to produce a new value. Two bulk synchronous exchanges are written, each with one warp-wide instruction. After \( \log_2 n \) steps, FFT is computed and stored in distributed and bit reversed manner within 1 register across warp. The twiddle factors can be loaded from device memory or computed with the \texttt{sincosf} function and subsequently swapped with registers. Memory bandwidth is reduced at the expense of additional registers. The bottleneck is with sizes 16 and 32, where loading from memory exhibits better performance.

For 1-D FFT and 2-D FFT, \( 32 < n \leq 256 \), computation is distributed across multiple registers and across threads in a warp, which consists of \( \lceil n/32 \rceil \) Fourier coefficients and twiddle factors in registers per thread. The best performance for 1-D FFT for \( n \leq 256 \) is at the register limit of \( n = 512 \), or 128 and 256 for 2-D FFT.

**Optimizing for cuDNN 5** For recurrent neural networks, independent steps, particularly weight operations, can be combined into one large step [2]. The following equations get carried out during forward propagation through an LSTM unit.
\[ i_t = \sigma(W_i x_t + R_i h_{t-1} + b_i) \]
\[ f_t = \sigma(W_f x_t + R_f h_{t-1} + b_f) \]
\[ o_t = \sigma(W_o x_t + R_o h_{t-1} + b_o) \]
\[ c'_t = \tanh(W_c x_t + R_c h_{t-1} + b_c) \]
\[ c_t = f_t \odot c_{t-1} + i_t \odot c'_t \]
\[ h_t = o_t \odot \tanh(c_t) \]

The pseudocode for cuBLAS GEMM is as follows:

```plaintext
1 for layer in layers:
2   for iteration in iterations:
3       perform 4 SGEMMs on input from last layer
4       perform 4 SGEMMs on input from last iteration
5       perform point-wise operations
```

The four weight matrices on the recurrent step, as well as the weights from the input, can be combined, which results in 2 matrix multiplications instead of 8, each 4× size and 4× parallelism, or 16 blocks per GEMM, a 2× increase. Streaming GEMMs can also be used, where 16 blocks are streamed, instead of 4, with the target at 96. The two remaining GEMMs are independent and can be computed with streams.

Performing a transpose on the weight matrix may slightly speed up execution per step, although at the cost of a transpose, which is amortized if used in more than a few iterations. Combining input GEMMs give more parallelism in that operation, but also prevents overlap with recurrent GEMMs. Propagation of recurrent GEMMs depend on the completion of input GEMMs. Combining the two input GEMMs work best in this case. The pseudocode shows the transposed weight matrix for back propagation, as well as point-wise operator fusing per stream.

```plaintext
1 for layer in layers:
2   transpose weight matrices
3 for iteration in iterations / combination size:
4   perform sgemm on combined input from last layer in stream A
5   wait for stream A
6   perform sgemm on input from last iteration in stream B
7   wait for stream B
8 for sub-iteration in combination size:
9   perform pointwise operations in one kernel
```

Since the iteration of \( n \) for a given layer only depends on iteration \( n - 1 \) of that layer and iteration \( n \) of the previous layer, the current layer could proceed before finishing with the previous layer, where each layer processed is an added level of parallelism.

**Persistent RNNs** Efficient partitioning of the GPU memory hierarchy can enable weights and activations of recurrent neural networks to be stored in registers [11]. A single input sequence \( x \) and corresponding output sequence \( y \) is sampled from a training set \( \mathcal{X} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots\} \). Each input sequence, \( x^{(i)} \), is a time-series of length \( T^{(i)} \) where each time slice is a vector of application-specific features, \( x_t^{(i)}, t = 0, \ldots, T^{(i)} - 1 \). The forward in time \( h^{(i)} \) recurrent layer activations are

\[
\begin{align*}
h_t &= f(h_{t-1}^{(i)}, h_{t-1}^{(i)}) \\
    &= \sigma(W_t^{(i)}h_{t-1}^{(i)} + U_t^{(i)}h_{t-1}^{(i)} + b_t^{(i)}),
\end{align*}
\]

where \( W_t^{(i)} \) is the hidden input weight matrix, \( U_t^{(i)} \) is the recurrent weight matrix and \( b_t^{(i)} \) is a bias term. The implementations are usually separated into two stages. The first stage, \( W_t^{(i)}h_{t-1}^{(i)} \), is a weight matrix that is shared over all timesteps, which can be unrolled into one two-dimensional matrix,
as discussed in [2]. The second stage, $U_l^l h_{t-1}^l$, involve outputs of each connected neuron to inputs of the current timestep, which is most computationally expensive. The sequential dependence between timesteps require explicit synchronization, whereas the recurrent weight matrix require reloading at each timestep.

Matrix multiplication is most efficient when the mini-batch size is large ($N > 64$, per GPU), since recurrent weights can be loaded once from off-chip memory and reused over each sample in the mini-batch. The drawbacks of a large mini-batch approach include the increase in memory required to train a network, the decrease in data parallelism within a single GPU and complications associated with deploying trained models in resource-constrained environments, such as mobile phones. Increasing the mini-batch size directly increases the memory required to train an RNN, since activations require more memory space when training over multiple timesteps. In speech recognition models, a 30-second utterance, or 3000 timesteps, can be processed with an RNN with 1760 hidden units and a mini-batch size of 64, which generates 1.3 GB of storage for activations per layer, much more than the 12.3 MB required to store layer weights. GPUs with 12 GB memory will constrain networks to about 9 layers.

The mini-batch size was fixed at 512 and distributed amongst GPUs. For instance, a mini-batch size of 512 can be carried out with 128 GPUs at a mini-batch-per-GPU size of 4, or 16 GPUs at a mini-batch-per-GPU size of 64. This sidesteps the issue of activations increasing as timesteps grow, since each GPU is responsible for a smaller portion of the mini-batch. When working on a subset of the network weights, GPU threads require communication and synchronization between each timestep for aggregated partial results and updated activations, which adds additional overhead, especially in a scaled up setting. A form of preemptive multitasking on the GPU can be used as a workaround, where threads attempt to synchronize directly using a global barrier and, if unsuccessful, eventually time out and otherwise exit.

**Sparse Persistent RNNs** For cases where RNNs are sparse and cannot fit in registers, representing the RNNs as $<\text{index}, \text{value}>$ pairs indicating the location of nonzero elements, as well as pruning the RNN, can further improve performance [65]. For this discussion, each time step in a recurrent network is expressed as follows:

$$h_t = g(U_r h_{t-1} + W x_t + b),$$

where $U_r$ is the recurrent weight matrix, $W$ is the input-to-hidden weight matrix, $b$ is a bias term and $g$ is an elementwise activation function. The input-to-hidden weight matrix $W x_t$ calculation has no dependencies and can be processed in parallel.

$$h_t = g(U_r h_{t-1} + b')$$

In a persistent RNN implementation, weights $U_r$ are stored in on-chip register files so each thread keeps $x \times y$ (rows $\times$ columns) weights. The number of thread blocks is set to the number of streaming multiprocessors in the system and blocks work together to process all rows from (12). Figure 12 shows the work distribution, where recurrent weights are loaded cooperatively per block. Each CTA consists of warps that, in turn, consist of a group of threads, where each thread manages one row of the recurrent weight with registers.

The data format for sparse persistent RNNs are $<\text{index}, \text{value}>$ pairs, which represents the location and value of one nonzero weight and each thread keeps a fraction of the total nonzero weights. All nonzero elements on one thread must belong in the same row and the number of thread blocks depend on the hidden layer size and degree of sparsity, rather than being fixed to the number of SMs available. Several optimizations were made. Wide memory loads (LDS.128) were used to eliminate bank conflicts, where threads access the same address space, which supports 4x the amount of threads load (128 vs. 32 threads) and incurs 8, instead of 32, bank conflicts in the worst case. A bank-aware layout transform was applied to improve shared memory access,
where each thread assigned a color to each of \( n <\text{index}, \text{value}> \) pairs, indicating a bank location. A pruning approach was applied, which significantly reduced the RNN parameters, in some cases up to 90% in a speech recognition LSTM, while maintaining accuracy. The approach started with a model that was too large to fit in memory and applied heuristics to prune down the model to size. Lamport timestamps were also used as an alternative to global synchronization, which added a flag for each output value used to indicate whether or not the value has been computed. This addresses the issue with threads needing to communicate with each other, as described in [11], which allows threads to make progress individually.

**LightRNN** A 2-component shared embedding can be used for vocabularies in natural language processing [30], decreasing storage space by \( 2\sqrt{|V|} \), where \(|V|\) represents the vocabulary size. Large vocabulary embeddings directly impact training convergence, which also leads to low recall in inferencing. Every word in the vocabulary is allocated in a table, where each row is associated with a vector and each column is associated with another vector. Depending on the position in the table, a word is jointly represented by two components via a row vector and a column vector. This arrangement significantly reduces the model size and speeds up training and inference processing for large vocabulary corpora.

Let \( n \) and \( m \) denote the dimensions of a row and column for an input vector and a hidden state vector, respectively. The probability distribution of \( w_t \) requires the column vector \( x_c^{t-1} \in \mathbb{R}^n \), the row vector \( x_r^{t} \in \mathbb{R}^n \) and the hidden state vector \( h_r^{t-1} \in \mathbb{R}^m \), where the column and row vectors are input-embedded matrices \( X_c, X_r \in \mathbb{R}^{n \times \sqrt{|V|}} \). The two hidden state vectors \( h_c^{t-1}, h_r^{t} \in \mathbb{R}^m \) are produced by applying the following recursive operations:

\[
h_c^{t-1} = f(W x_{t-1}^c + U h_c^{t-1} + U h_r^{t-1} + b), \ h_r^{t} = f(W x_{t}^r + U h_c^{t-1} + b)
\]

where \( W \in \mathbb{R}^{m \times n}, U \in \mathbb{R}^{m \times m}, b \in \mathbb{R}^m \) are parameters of affine transformations and \( f \) is a nonlinear activation, such as sigmoid.

The objective in language modeling is to minimize the negative log-likelihood of the next word in a sequence, which is equivalent to optimizing the cross-entropy between the target probability distribution and the prediction. Given a context with \( T \) words, the overall negative log-likelihood can be expressed as

\[
NLL = \sum_{t=1}^{T} -\log P(w_t) = \sum_{t=1}^{T} -\log P_r(w_t) - \log P_c(w_t),
\]

where \( NLL_w \) is the negative log-likelihood for a word \( w \). The probability \( P(w_t) \) of a word \( w \) at
Figure 13: Common data reorganization operations.

position $t$ is determined by the row probability $P_r(w_t)$ and column probability $P_c(w_t)$:

$$P_r(w_t) = \frac{\exp(h^r_{i-1} \cdot y^r_{c(w)})}{\sum_{i \in S_r} \exp(h^r_{i-1} \cdot y^r_i)}, \quad P_c(w_t) = \frac{\exp(h^c_t \cdot y^c_{c(w)})}{\sum_{i \in S_c} \exp(h^c_i \cdot y^c_i)},$$

$$P(w_t) = P_r(w_t) \cdot P_c(w_t), \quad (13)$$

where $r(w)$ is the row index of word $w$, $c(w)$ is the column index, $y^r_i \in \mathbb{R}^m$ is the $i$-th vector of $Y^r \in \mathbb{R}^{m \times \sqrt{|V|}}$ and $y^c_i \in \mathbb{R}^m$ is the $i$-th vector of $Y^c \in \mathbb{R}^{m \times \sqrt{|V|}}$.

The 2-Component embedding approach reduced perplexity, $PLL = \exp(NLL/T)$, for large sized datasets, where a lower $PLL$ lower is better, while significantly requiring less parameters for word embeddings. For instance, the model size of the BillionW dataset, which consists of 799M tokens and 793K vocabulary, reduced by a factor of 40 over the previous approach. Combined with Kneser-Ney (KN) 5-gram smoothing, their approach achieved a perplexity of 43, a 36% improvement over a stand-alone KN approach.

### 3.2 Architecture-Specific Optimizations

Although the discussion so far has focused on GPU accelerators, vectorized processing units are another way of achieving instruction-level parallelism for a program, where an issue of one instruction causes the same operation to be applied to a vector of data. Vector operations are performed in a pipelined manner, where elements in a vector are fetched serially and computations on different elements are overlapped. SIMD instructions carry out the same operation on contiguous memory locations by loading, storing and computing data from memory using parallel hardware.

#### 3.2.1 Vectorization

Modern CPU architectures, such as Intel Skylake, include vector hardware units that allow efficient throughput of instructions that can be vectorized. Figure 13 displays common data reorganization vector operations. Although vectorization provides pipelining capabilities that lead to an increase in performance, overhead in execution performance can occur if reorganization methods are not accounted for.

**SIMD Mixed** Vector data in registers may not be in a required format, unless data reorganization is performed. LOOP-MIX is proposed to harness both intra- and inter-iteration parallelism simultaneously [64]. LOOP-VEC represents a traditional loop-level vectorizer that does packing and unpacking for data reorganization. LOOP-ILV provides interleaved data access patterns. LOOP-AWARESLP exploits both intra-iteration parallelism (SLP) and inter-iteration (loop-level) parallelism, but does not support mixed-parallelism-inhibiting dependencies. The objective is to maximize SIMD resource utilization while keeping data reorganization overhead low.
Loops that are vectorizable are strip-mined into vector length (VL) sized chunks so that all statements can be vectorized. Scalar variables are at the top-level in SSA form, implying that each has a unique definition. Address taken variables are array subscripted and appear in load and stores only. A group of VL statements or operations is an ordered list. A statement group is a VL-sized group containing VL statements that are isomorphic that share the same operator and are independent with no statement-wide dependencies. Both s and s' statement groups that are equal component-wise imply that they are identical. s overlaps partially with s', s ⊿◁ s', if s and s' are not equal, but share some statements in common. Reorganization is required in this case to make s values available for s'.

H-LOOPLEVEL looks for statement groups across VL consecutive loop iterations at the loop level. H-CMA forms statement groups for loads or stores that make consecutive memory accesses. H-UD traces a use-def chain of an existing statement group to form more statement groups, whereas H-DU traces a def-use chain of an existing statement group to form more statement groups. LOOP-AWARESLP traces use-def chains. LOOP-MIX traces use-def and def-use chains to exploit mixed parallelism more effectively. Only scalar variables in SSA are traced.

To reduce data reorganization overhead, a cost model determines whether to switch between intra- and inter-iteration parallelism. Given a loop L, a set of VL-sized statement groups are formed, where loads or stores are applied with H-CMA. Every group must contain at least two
Convolutional Neural Network on SIMD  

Factors to consider when performing the computations of CNNs on an Intel Skylake architecture include FMA latencies, as well as applying blocking and vectorization strategies [15]. FMA latencies is 4 in Skylake, compared to 6 in Knight’s Landing (KNL), which means that more instructions need to be pipelined in a KNL in order to achieve peak performance. For the purposes of this discussion, a neural network consists of layers of multiple neurons connected by weights, where values assigned to neuron are called activations. Both activations and weights are represented as tensors. An activation is either an input or an output consisting of 4 dimensions: mini-batch size $N$, feature maps $C$, spatial dimensions $H$ and $W$. Inputs and outputs are represented as $N \times K$ output feature maps and $P \times Q$ output spatial dimensions. A weight tensor is also 4D, with feature map dimensions $C \times K$ and spatial dimensions $R \times S$.

Register blocking, used to improve data reuse in registers, is applied in the spatial domains of the output tensor, since points in the spatial tensor iteration space can be computed independently. Register blocking $RB_P$ and $RB_Q$ are chosen based on the architectural target. For cache blocking and loop ordering, unless activations and weight tensors fit in cache, convolutional loops are bandwidth bound. To maximize data reuse from cache, cache blocking can be applied in the feature map and spatial domains. Also, loop ordering determines the way tensors are accessed and impacts the reuse of data. For large weights ($R = 3$, $S = 3$), the output tensor can be reused from registers, whereas small weights ($R = 1$, $S = 1$) do not have the same degree of reuse.

Listing 11: Naive forward propagation loop (left), with microkernel calls and fused layer $L()$ (right).

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>for n = 0...N-1 do</td>
<td>for loop over mini-batch size</td>
</tr>
<tr>
<td>2</td>
<td>for k = 0...K-1 do</td>
<td>for loop over weight dimension</td>
</tr>
<tr>
<td>3</td>
<td>for c = 0...C-1 do</td>
<td>for loop over feature map dimension</td>
</tr>
<tr>
<td>4</td>
<td>for oj = 0...Q-1 do</td>
<td>for loop over spatial dimension</td>
</tr>
<tr>
<td>5</td>
<td>for oi = 0...P-1 do</td>
<td>for loop over spatial dimension</td>
</tr>
<tr>
<td>6</td>
<td>ii = stride * oi</td>
<td>calculate index in feature map</td>
</tr>
<tr>
<td>7</td>
<td>ij = stride * oj</td>
<td>calculate index in spatial domain</td>
</tr>
<tr>
<td>8</td>
<td>for s = 0...S-1 do</td>
<td>for loop over output dimension</td>
</tr>
<tr>
<td>9</td>
<td>O[n][k][oj][oi] +=</td>
<td>add to output tensor</td>
</tr>
<tr>
<td>10</td>
<td>W[k][c][r][s] *</td>
<td>multiply weight tensor</td>
</tr>
<tr>
<td>11</td>
<td>1[n][n][ij+r][ii+s] *</td>
<td>multiply input tensor</td>
</tr>
<tr>
<td>12</td>
<td>W[k][c][r][s];</td>
<td>end multiplication</td>
</tr>
</tbody>
</table>

Listing 11 (R) shows forward propagation as a convolution microkernel. The inner-most computation is a small matrix-vector product of partial weight tensor with partial input tensor: $O'[k] += W'[c][k] \ast I'[c]$, where $c$ and $k$ are dimension multiples of an architecture’s vector length $VLEN$ (e.g. 16). Although matrix-vector is not compute intense, there is reuse in the output or weight tensor when accounting for the outer loops, as well as the input feature map (line 3 in Listing 11, R), which can be pulled to improve register reuse by $C_b$. Small matrix-vectors are converted into a sequence of small matrix multiplications (GEMM) with blocking $RB_Q$. JIT is used to optimize $M = k$, with $K = c$ being multiples of a machine’s vector length. For Intel AVX512, a small GEMM loads a full vector register with output channel weights from $W$ at position $0 \leq x < c$ and loops over $RB_Q$ pixels of input activation, broadcasting the output and multiplying with the loaded weight. The JIT-compiled GEMM code is as follows: for $0 \leq x < c$ and $0 \leq y < RB_Q$ do $O'[y][k] += W'[x][k] \ast I'[y][x]$.

Loops at lines 1, 2, 4 and 5 in Listing 11 (R) are independent output tensor slices that can be processed in parallel, with $N \times K_b \times P_b \times Q_b$ independent microkernel invocations (or “work items”) assigned to threads. Work can be divided, based on mini-batch iteration, where threads share entire weight tensors reused from shared caches. Work items can be further extracted from the output feature map domain. Examples of operators that can be fused with $L()$ include ReLU, Pooling, LRN, Normalization and Bias, which are typically low in operational intensity, but bandwidth
The backpropagation algorithm is described by the loop structure of Listing 12 (L). In the backward pass, the gradient input tensor $dI$ is computed by convolving the gradient output tensor $dO$ with weight tensor $W$. This approach is different from forward propagation, since accumulation happens in the gradients of the inputs and an update. Thus, backpropagation (Lis. 12 (L))

$$dI[n][c][ij + r][ii + s] += dO[n][k][oj][oi] \ast W[k][c][r][s]$$

has a different access pattern than the update of forward propagation

$$O[n][k][oj][oi] += I[n][c][ij + r][ii + s] \ast W[k][c][r][s].$$

Weight tensors can be transformed to enable reuse of the fully optimized forward propagation. Refer to [15] for how rewrites of the input gradients in back propagation can match the access patterns of the optimized forward propagation.

Listing 12: Naive backpropagation (left). Backward propagation with small GEMM calls (right).

```python
1 for n = 0...N-1 do
2   for k = 0...K-1 do
3     for c = 0...C-1 do
4       for oj = 0...P-1 do
5         for ii = 0...Q-1 do
6           for r = 0...R-1 do
7             dI[n][c][ij+r][ii+s] +=
8               dO[n][k][oj][oi] \ast W[k][c][r][s]
```

For all other cases, small GEMMs are used to update the input gradient tensor (see Listing 12 (R)). In GEMM calls, $\text{GEMM}(A,B,C)$, where $A = M \times K$, $B = K \times N$, $C = A \times B$. The small downside is that loops 2, 7 and 8 cannot be embedded in a small GEMM call, since the data reuse from registers is not properly exploited, resulting in redundant data movement.

In the update pass of weight gradients (Lis. 13, left), the weight gradient tensor $dW$ is computed by convolving the gradient output tensor $dO$ with the input tensor $I$. In Listing 13 (R), the last 4 loops (lines 13-21) are implemented as a JIT-ed microkernel. The main difference is that each microkernel invocation computes a $VLEN \times VLEN$ sub-tensor of the weight gradient. Register blocking is employed up to a factor of $VLEN$, or the amount exposed by $VLEN$ independent FMA instructions.


```python
1 for n = 0...N-1 do
2   for k = 0...K-1 do
3     for c = 0...C-1 do
4       for oj = 0...P-1 do
5         for ii = 0...Q-1 do
6           for r = 0...R-1 do
7             dW[k][c][r][s] +=
8               I[n][c][ij+r][ii+s] \ast W[k][c][r][s]
```

```python
1 Pb = P/Bp, Qb = Q/Bq
2 for n = 0...N-1 do
3   for kb = 0...Kb-1 do
4     for cb = 0...Cb-1 do
5       for ojb = 0...Pb-1 do
6         for oib = 0...Qb-1 do
7           for r = 0...R-1 do
8             for p = 0...Bp do
9               dW[kb][cb][r][s][c][k] +=
10              I[n][cb][ij+r][ii+s][c] \ast W[kb][cb][r][s][c][k]
```
3.2.2 GPU Performance Tuning

TwinKernels is proposed to allow warps execute different program counters in GPUs, a dramatic shift in the current approach. Code variant tuning provides a way to build up knowledge of an application through use of multi-task learning acquired from previous training runs.

**TwinKernels**

GPU wavefronts with different program counters execute in parallel with a round-robin wavefront scheduling policy [16]. Kernels are identical in terms of functionality, but differ significantly in terms of instruction schedules. Workload-kernel binding is carried out at the wavefront or work-group granularity, where each wavefront chooses one of twin kernels to execute. In traditional SIMT models, all wavefronts must execute the same kernel. TKMT relaxes this restriction and instead allows hybrid execution of kernels for wavefronts. Twin kernel binaries are generated in two steps. Preprocessing is handled by the finalizer, which filters, selects and merges the kernels. The assembler takes output from the finalizer and generates executables. In traditional SIMT models, all wavefronts must execute the same kernel.

Figure 14 shows how a twin kernel binary compares with two regular binaries, which is encapsulated into a single ELF-formatted binary as a regular binary. An additional conditional branch is inserted before instructions of the first twin kernel. Wavefronts or work-groups start with $PC = 4$, the size of an extra branch instruction, and executes the first twin kernel, while wavefronts with $PC = 0$ are redirected to the second twin kernel. PC initialization is the only additional requirement needed to support the TKMT model. Hardware design does not require bookkeeping and only occupies a small real estate. Multi2sim was used for experiments.

The execution engine consists of profiling, scheduling and execution. Regular kernels are profiled in SIMT. The potential for collaboration of kernels is statically evaluated based on memory distance and instruction interleaving. Profiling and evaluation information are used for an execution schedule. Trail execution executes three sequential or random mix ratios. The other twin kernel can be executed if no performance differences exist. Otherwise, execution resumes on the rest of the mix ratios. The last step performs trail execution and updates the record when a better combination is found.

**Code Variant Tuning**

Performance results from two or more GPU source architectures can be used as training inputs to create a variant selection model for a different target architecture [34]. The cross-architectural problem is posed as a multi-task learning problem, where each separate
Each separate architecture represents a task, where inter-task relationships are learned through MTL algorithms. Feature concatenation derives a code variant selection model for a target architecture. Due to code variant selection being application-specific, device feature selection (DFS) pinpoints a small number of relevant device features for a particular application. For instance, a cosine similarity matrix for the SpMV kernel can communicate that the Fermi GPU is very dissimilar to Maxwell.

The tuning process consisted of a two-phase DFS strategy. The feature concatenation strategy for MTL appends device features to input features and builds a support vector machine (SVM) model based on new training dataset. \( \{a_1, a_2, ..., a_M\} \) denotes device feature vectors for each of the \( M \) source architectures. For source architecture \( s \), the training set is represented as \( T_s = \{(x_1 \circ a_s, y_{s1}), ..., (x_N \circ a_s, y_{sN})\} \), where \( \{x_1, ..., x_N\} = N \) are the input feature vectors from training set, each \( y_{si} \) denotes a variant label for training input \( i \) on architecture \( s \), \( \circ \) is a vector concatenation and the full training set, \( T = \bigcup_{s=1}^{M} T_s \), is used as input to the SVM classifier. During testing, device features of a target architecture are concatenated with input features before querying the model.

The source architecture side profiles metrics \( C_{v} \) of each code variant \( v \) of application proxies at different intensities \( \{C_{\phi_j}, \phi\} \), which is used as inputs to training. In the target architecture side, device features selected from P-DFS are used, where \( \lambda_j \) is a proxy \( P_j \). \( \lambda_j \) is queried with profiling metrics of variant \( C_v \), which yields the intensity value corresponding to the \( P_j \) variant.

For Profile Device Feature Selection (P-DFS), application proxies are small programs that takes intensity \( \phi \) as input, ranging from 0 to 5 and produces a GPU kernel with \( \phi \times 20\% \) instructions. Each proxy \( P_j \), where \( j \) ranges from 1 to 5 (total number of proxies), is associated with a set of device features \( F_j \) representing a particular hardware component (e.g. ATOMIC proxy associated with shared_atomic and global_atomic). For each proxy \( P_j \), the system collects tuples of the form \( \{C_{\phi_j}, \phi\} \), where \( C_{\phi_j} \) represents profiling data of a single run of proxy \( P_j \) and \( \phi \) is the execution intensity.

Cross-Validation Device Feature Selection (CV-DFS) performs cross validation on a target architecture dataset only, which relies on the assumption that device features that yield good prediction performance on source architectures are likely to be good predictors on a target for the same computation.

### 3.2.3 Hardware Approaches

Approaches to improve performance at the hardware level are discussed, including a domain-specific architecture and an instruction set architecture (ISA) for neural networks. For deep neural
networks, a domain-specific architecture consists of stacks of processing units that perform a few tasks, but is very specialized at those tasks. An ISA provides an abstraction that enable common operations in deep learning to be translated into low-level assembly code.

### A Domain-Specific Architecture

The tensor processor unit (TPU) is an example of a domain-specific architecture deployed by Google in 2015, which runs DNNs 15 to 30 times faster and is 30 to 80 times more energy efficient, compared to GPUs and CPUs [21]. The TPU instructions are sent from the host over the peripheral component interconnect express (PCIe) Gen3 x16 bus into an instruction buffer, see Figure 15. The internal blocks are connected by 256 byte-wide paths. The matrix unit is capable of producing one 256-element partial sum per cycle. The main computation is the **Matrix Multiply Unit**, the inputs are the **Weight FIFO** and **Unified Buffer** and its output are the **Accumulators**. The TPU floor plan (right) consists of 35% memory, 35% computation, 10% I/O and 2% control. Figure 16 shows a comparison of computer architectures. Note that the TPU die is less than half the size of the Haswell die. Also, note the differences in tera operations between the architectures, with the TPU capable of 92 TOPS/s in 8-bit precision.

The matrix unit uses “systolic execution” to save energy by reducing reads and writes from the unified buffer, since reading a large static random access memory (SRAM) uses much more power than performing arithmetic. Data from different directions arrive at cells in an array at regular intervals, which are combined. A given 65,536-element vector-matrix multiply operation moves through the matrix as a diagonal wavefront. The weights are preloaded and the wavefront advances along the first data of a new block. Control and data are pipelined, where the 256 inputs are read at once and instantly updates one location of each of 256 accumulators. From a correctness perspective, the software is unaware of the systolic nature of the matrix unit, but, for performance, must account for the latency of the unit.

### Instruction Set Architecture

Cambricon is an instruction set architecture (ISA) for neural networks [31], since the ISA design is still an unresolved challenge that limits both the flexibility and efficiency of existing neural network accelerators. Data-level parallelism is enabled by vector and

![Figure 15: TPU block diagram and floor plan.](image)

![Figure 16: A comparison of computer architectures.](image)

<table>
<thead>
<tr>
<th>Model</th>
<th>mm$^2$</th>
<th>nm</th>
<th>MHz</th>
<th>TOPS/s</th>
<th>Bin</th>
<th>FP</th>
<th>Gb/s</th>
<th>Chips/Server</th>
<th>TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haswell CPU</td>
<td>992</td>
<td>22</td>
<td>2.00</td>
<td>145W</td>
<td>2.8</td>
<td>1.3</td>
<td>61</td>
<td>2</td>
<td>65W</td>
</tr>
<tr>
<td>Nvidia K50 GPU</td>
<td>661</td>
<td>28</td>
<td>660</td>
<td>135W</td>
<td>2.9</td>
<td>1.8</td>
<td>180</td>
<td>0</td>
<td>1.1GW</td>
</tr>
<tr>
<td>TPU</td>
<td>~33L</td>
<td>28</td>
<td>700</td>
<td>75W</td>
<td>92</td>
<td>2</td>
<td>34</td>
<td>4</td>
<td>881W</td>
</tr>
</tbody>
</table>

...
matrix instructions that are more efficient than instruction-level parallelism, which corresponds to higher code density, since most neural networks organize neurons and synapses as layers that are manipulated in a uniform and symmetric manner. A customized representative set of vector and matrix instructions for existing neural network techniques is presented. On-chip scratchpad memory is used in place of vector register files providing flexible widths for each data access, since neural network techniques often require intensive, contiguous, variable-length accesses to vector and matrix data.

The types of instructions in Cambricon include computational, logical, control and data transfer. The two control instructions are jumps and conditional branches. The data sizes for data transfer instructions are variable in order to support matrix and vector computational and logical instructions. Data transfers are performed to and from main memory or to and from on-chip scratchpad and scalar GPRs.

A multi-layer perceptron (MLP) has multiple layers, where each layer computes values of output neurons according to known input neurons. Specifically, an output neuron $y_i (i = 1, 2, 3)$ is computed as $y_i = f(\sum_{j=1}^{3} w_{ij} x_j + b_i)$, where $x_j$ is the $j$-th input neuron, $w_{ij}$ is the weight between the $i$-th output neuron and the $j$-th input neuron, $b_i$ is the bias of the $i$-th output neuron and $f$ is an activation function. The output neurons are computed as a vector $y = (y_1, y_2, y_3)$ with $y = f(Wx + b)$, where $x = (x_1, x_2, x_3)$ and $b = (b_1, b_2, b_3)$ are vectors of input neurons and biases, $W = (w_{ij})$ is the weight matrix and $f$ is an element-wise activation function.

The critical step is computing $Wx$, which is performed by the Matrix-Vector-Multiplication (MMV) instruction, as illustrated in Fig 18 (L). $Wx$ is computed with dedicated MMV instructions, instead of decomposed as multiple vector dot products to reuse the input vector $x$ among different row vectors of $M$. The vector matrix multiplication (VMM) instruction is used in the backforward training process and has the same fields as MMV, with a different opcode. An alternative for backpropagation, which is MMV + transpose, would be expensive in data movements. The weight matrix $W$ also needs to be incrementally updated with $W = W + \eta \triangle W$, where $\eta$ is the learning rate and $\triangle W$ is estimated as the outer product of two vectors. Cambricon provides the Outer-Product (OP) instruction, Matrix-Mult-Scalar (MMS) instruction and a Matrix-Add-Matrix (MAM) instruction for weight updating. Matrix-Subtract-Matrix (MSM) is also provided to support weight updates in Restricted Boltzmann Machines (RBM).

Vector-Add-Vector (VAV) instructions are used for vector additions, which require multiple instructions to support element-wise activation. For example, $f(\alpha) = e^{\alpha}/(1 + e^{\alpha})$ is an element-wise sigmoid activation performed on each element of the input vector, $\alpha$. The Vector-Exponential (VEXP) instruction computes the exponential $e^{\alpha_i}$ for each element ($\alpha_i, i = 1, \ldots, n$) from the input the vector $\alpha$. The Vector-Add-Scalar (VAS) instruction adds a constant 1 to each element of the vector ($e^{\alpha_1}, \ldots, e^{\alpha_n}$), provided. The Vector-Div-Vector (VDV) instruction is provided for element-wise
Figure 19

division between vectors. Other vector arithmetic instructions include Vector-Mult-Vector (VMV), Vector-Sub-Vector (VSV), Vector-Logarithm (VLOG) and Random-Vector (RV), which generates a vector of random numbers from the uniform distribution at interval \([0,1]\).

For logical instructions, neural network techniques consist of a few operations that incorporate comparisons or other logical manipulations. For instance, max-pooling computes the neuron with the largest output amongst neurons within a pooling window, which is repeated for different input feature maps. Cambricon supports the max-pooling operation with Vector-Greater-Than-Merge (VGTM), which designates each element of the output vector \(V_{out}\) by comparing the corresponding elements of the input vector-0 \((V_{in0})\) and input vector-1 \((V_{in1})\), e.g., \(V_{out}[i] = (V_{in0}[i] > V_{in1}[i]) \ ? V_{in0}[i] : V_{in1}[i]\). Other logical instructions include Vector-Greater-than (VGT), Vector-Equal (VE) and Vector AND/OR/NOT (VAND/VOR/VNOT).

Listing 15: MLP code (left), pooling code (right).

```c
1 // $0: input size, $1 output size, $2: matrix sz
2 // $3: input address, $4: weight address
3 // $5: bias address, $6: output address
4 // $7-$10: temp variable address
5
6 VLAD $3 $0, #100 // load input vector
7 MLOAD $4, $2, #300 // load weight matrix
8 MMV $7, $1, $4, $3, $0 // Wx
9 VAV $8, $1, $7, $5 // tmp=Wx+b
10 VEXP $9, $1, $8 // exp(tmp)
11 VAS $10, $1, $9, #1 // 1+exp(tmp)
12 VDV $6, $1, $9, $10 // y=exp(tmp)/(1+exp(tmp))
13 VSTORE $6, $1, #200 // store output vector to
```

Listing 16: BM code.

```c
1 // $0: feature map size, $1 input data size,
2 // $2: output data size, $3: pooling window
3 // $4: x-axis loop num, $5: y-axis loop num
4 // $6: input addr, $7: output addr
5 // $8: y-axis stride of input
6
7 VLAD $6, $1, #100 // load input neurons
8 SMOVE $5, $3 // init y
9 L0: SMOVE $4, $3 // init x
10 L1: VTMG $7, $0, $6, $7
11 // forall feature map m,
12 // output[m]=(input[x][y][m]>output[m]) ? input[x][y][m] : output[m]
13 // input[x][y][m]:output[m]
14 SADD $6, $0, $9 // update input address
15 SADD $4, $4, #1 // x--
16 CH $11, $4 // if (x>0) goto L1
17 SADD $6, $6, $8 // update input address
18 SADD $5, $5, #1 // y--
19 CH $10, $5 // if (y>0) goto L0
20 VSTORE $7, $2, #200 // store output neurons
```

Table I. An overview to Cambricon instructions.

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Examples</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>jump, conditional branch</td>
<td>register (scalar value), immediate</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>matrix load/store/move</td>
<td>register (matrix address/size, scalar value), immediate</td>
</tr>
<tr>
<td>Vector</td>
<td>vector load/store/move</td>
<td>register (vector address/size, scalar value), immediate</td>
</tr>
<tr>
<td>Scalar</td>
<td>scalar load/store/move</td>
<td>register (scalar value), immediate</td>
</tr>
<tr>
<td>Computational</td>
<td>matrix multiply vector, vector multiply matrix, matrix multiply scalar, outer product, matrix add matrix, matrix subtract matrix</td>
<td>register (matrix/vector address/size, scalar value)</td>
</tr>
<tr>
<td>Vector</td>
<td>vector elementary arithmetics (add, subtract, multiply, divide), vector transcendental functions (exponential, logarithmic), dot product, random vector generator, maximum/minimum of a vector</td>
<td>register (scalar value), immediate</td>
</tr>
<tr>
<td>Scalar</td>
<td>vector elementary arithmetics, scalar transcendental functions</td>
<td>register (scalar value), immediate</td>
</tr>
<tr>
<td>Logical</td>
<td>vector compare (greater than, equal), vector logical operations (and, or, invert), vector greater than or equal</td>
<td>register (vector address/size, scalar)</td>
</tr>
<tr>
<td>Scalar</td>
<td>vector compare, scalar logical operations</td>
<td>register (scalar), immediate</td>
</tr>
</tbody>
</table>
A prototype accelerator (Cambricon-ACC) was synthesized with Synopsys Design Compiler using TSMC 65 nm GP standard VT library, which places and routes the synthesized design with the Synopsys ICC compiler. The approach was simulated and verified with Synopsys VCS. Power consumption was estimated with Synopsys Prime-Time PX according to the simulated Value Change Dump (VCD).

The proposed accelerator implements both vector and matrix functional units. The vector unit contains 32 16-bit adders, 32 16-bit multipliers, equipped with 64 KB scratchpad memory. The matrix unit contains 1024 multipliers and 1024 adders, divided into 32 separate computational blocks. Each computational block is equipped with a separate 24 KB scratchpad. 32 computational blocks are connected through a h-tree bus that broadcasts input values to each block and collects output from each block.

4 Code Generation and Deployment

Once the problem space has been defined and improvements to our algorithm have been applied, the process of lowering takes place. Specifically, the code generator takes as input the intermediate representation (IR) produced by the front end of the compiler and produces as output a semantically equivalent target program. The target program must preserve the semantics of the source program and make effective use of available resources of the target machine. Moreover, the code generator itself must run efficiently. The problem of generating an optimal target program for a given source program is undecidable. This section covers approaches for lowering code to the compiler, several ML compilers and inference engines currently being deployed.

4.1 Lowering Techniques

4.1.1 Polyhedral Parallel Code Generator

A polyhedral parallel code generator (PPCG) for CUDA defines the kernel as a polyhedral representation and applies thread and block arrangements, based on the SIMT programming model [57]. Matrix multiplication is the illustrative example.

```c
void matmul(int M, int N, int K, float A[M][K], float B[K][N], float C[M][N])
{
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < K; k++)
        C[i][j] = C[i][j] + A[i][k] * B[k][j];
}
```

The iteration domain for the example program is

\[(K, N, M) \rightarrow \{S2(i, j, k) \mid 0 \leq i < M \land 0 \leq j < N \leq 0 \leq k < K\} \cup \{(K, N, M) \rightarrow \{S1(i, j) \mid 0 \leq i < M \land 0 \leq j < N\}\}
```
where \( K, N \) and \( M \) are parameters. Access relations map statement instances to array elements accessed by those instances. The write access relation can be expressed as
\[
\{ S_2(i, j, k) \rightarrow C(i, j) \} \cup \{ S_1(i, j) \rightarrow C(i, j) \},
\]
whereas the read access relation is
\[
\{ S_2(i, j, k) \rightarrow A(i, k) \} \cup \{ S_2(i, j, k) \rightarrow B(k, j) \} \cup \{ S_2(i, j, k) \rightarrow C(i, j) \}.
\]

Tiling a dimension consists of strip-mining and loop interchanging, which replaces the original dimension with two new dimensions, where the first identifies the strip and the second identifies the point inside the strip. Tile dimensions are mapped onto thread block identifiers and point dimensions are mapped onto thread identifiers.

Dependence analysis detects parallel and tilable loops using the polyhedral model. Desired properties are determined by dependencies in the program, represented with a dependence relation. A relation maps statement instances \( i \) to statement instances that need to be executed after \( i \), either because a read value was written by \( i \) or an overwrite value was read or written by \( i \).

An example of a schedule is as follows:
\[
\{ S_2(i, j, k) \rightarrow (i, j, k, 1) \} \cup \{ S_1(i, j) \rightarrow (i, j, 0, 0) \}
\]

An example of a dependence relation is as follows:
\[
\{ S_1(i, j) \rightarrow S_2(i, j, 0) \} \cup \{ S_2(i, j, k) \rightarrow S_2(i, j, 1 + k) \},
\]
which is simplified with respect to the domain constraints. Applying schedule (17) to both sides of the relation (18) and taking the difference between the image and domain elements results in dependence distances \( \{ (0, 0, 0, 1), (0, 0, 1, 0) \} \). The initial two zeros in the two tuples indicate that the first two schedule dimensions \((i, j)\) loops are parallel and have no dependencies. Since entries are nonnegative, loops can be permuted without changing semantics, which also allows for loops to be tiled.

Exposing parallelism and tiling opportunities is through affine scheduling, which transforms loop nests into a tilable, parallel form, resulting in single-valued multidimensional affine functions.
\[
S = (S_i)_{1 \leq i \leq d}
\]
Affine schedules map statement iterations from the iteration domain to logical dates in multidimensional time. More precisely, the schedules that are affine on each part of the iteration domain corresponds to iterations of the given statement. Affine functions \( S_i \) are constructed one-by-one in such a way that the corresponding dependence distances are non-negative.
\[
\Delta(S_i \circ F \circ S_i^{-1}) \geq 0,
\]
where \( F \) represents dependence relations and \( \Delta \) maps relation to differences between image and domain elements, which ensures validity of the schedule and the ability to freely permute loops enclosing \( S_i \).

Within each kernel, bands are tiled based on user-specified tile sizes. Tiling splits parallel loops into pairs of loops, with one “tile” loop iterating over tiles and one “point” loop iterating inside tiles. The outermost two parallel tile loops are mapped to blocks in the grid, while the innermost three parallel point loops are mapped to threads in a block. Block and grid sizes are specified by the user.
\[
\{(i, j, k, s) \rightarrow ([i/16], [j/16], [k/16], i \% 16, j \% 16, k \% 16, s)\}.
\]
Initial reference groups are connected components in graph nodes, where array references represents an edge between two nodes if access relations intersect and if at least one of the two references is a write. During scheduling, tiling is applied to a tilable band, resulting in tiled loops and point loops. These elements are represented as a relation between tile loop iterators and array indices.

\[
\{(I, J, K) \rightarrow A(a_0, a_1) \mid 16I \leq a_0 \leq 15 + 16I \land 16K \leq a_1 \leq 15 + 16K
\land 0 \leq J \leq 15 \land 0 \leq a - 1 \leq 255 \land 0 \leq a_0 \leq 255 \}\]

A relation can be obtained from the access relation (Eq. 16), applying schedule (Eq. 17) followed by tiling (Eq. 19) to its domain and subsequently projecting the domain onto its first three coordinates.

For mapping to registers, consider a relation between point loops, which are mapped to threads, and array elements, with outer loops as parameters. A relation for \(C\) references is defined as

\[
(I, J, K) \rightarrow \{(i, j) \rightarrow C(c_0, c_1) \mid c_0 = 16I + i \land c_1 = 16J + j\}.
\]

Each element can only be accessed by a single thread and the accessed element cannot depend on the inner loops. If so, loops can be permuted innermost and completely unrolled. To compute the register tile size, apply a mapping to threads by intersecting its domain and projecting out the parallel loops. This results in a simplified set

\[
(t_0, t_1, I, J, K) \rightarrow \{C(c, t_1 + 16J) \mid 16I \leq c \leq 16I + 15 \land (c - t_0)\}
\]

4.1.2 Halide

Halide is an end-to-end image processing pipeline, where mutable arrays are treated as functions from coordinates to values [41]. Images are represented as pure functions defined over an infinite integer domain. The value of a function point represents the color of a corresponding pixel. Pipelines are specified as chains of functions, where functions are either simple expressions or reductions over a bounded domain.

```
1 UniformImage in(UInt(8), 2)
2 Var x, y
3 Func blur(x, y) = in(x-1, y) + in(x, y) + in(x+1, y)
4 Func out(x, y) = blur(x, y-1) + blur(x, y) + blur(x, y+1)
```

Reduction functions are defined in two parts. The initial value function is the value at each point in the output domain. The reduction function executes recursively, which redefines values at points given by the output coordinate expression in terms of prior values of function. A reduction domain is specified, which depends on the order applied and is bounded by minimin and maximum expressions for each dimension.

```
1 UniformImage in(UInt(8), 2)
2 RDom r(0..in.width(), 0..in.height()), ri(0..255)
3 Var x, y, i
4 Func histogram(i) = 0; histogram(in(r.x, r.y))++
5 Func cdf(i) = 0; cdf(ri) = cdf(ri-1) + histogram(ri)
6 Func out(x, y) = cdf(in(x, y))
```

A pipeline’s schedule consists of a specific set of choices of when and where values should be computed. To schedule a two-stage pipeline, the first stage, \(\text{blurx}\), computes a horizontal blur of an input by averaging over a \(3 \times 1\) window:

```
1 blurx(x, y) = in(x-1, y) + in(x, y) + in(x+1, y)
```

The second stage, \(\text{out}\), computes a final isotropic blur by averaging a \(1 \times 3\) window of output from the first image:
\[ \text{out}(x, y) = \text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1) \]

**Listing 17: Naive.**

```
alloc blurx[2048][3072]
for each y in 0..2048:
  for each x in 0..3072:
    blurx[y][x] = in[y][x-1] + in[y][x] + in[y][x+1]
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    out[y][x] = blurx[y-1][x] + blurx[y][x] + blurx[y+1][x]
```

**Listing 18: Interleave.**

```
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    alloc blurx[-1..1]
    for each i in -1..1:
      blurx[i] = in[y-1+i][x-1] + in[y-1+x][x] + in[y-1+i][x+1]
```

**Listing 19: Loop fusion.**

```
alloc out[2046][3072]
alloc blurx[3][3072]
for each y in -1..2047:
  for each x in 0..3072:
    blurx[(y+1)%3][x] = in[y+1][x+1] + in[y+1][x] + in[y+1][x-1]
  if y < 1: continue
  out[y][x] = blurx[(y-1)%3][x] + blurx[y%3][x] + blurx[(y+1)%3][x]
```

Drawbacks for each include lost locality for naive, redundant work for interleave and limited parallelism for loop fusion. Tile interleaving trades off a small amount of redundant computation on tile boundaries for producer-consumer locality, leaving parallelism unconstrained both from within and across tiles.

**Listing 20: Interleaving with tiles.**

```
alloc out[2046][3072]
alloc blurx[-1..33][32]
for each ty in 0..2048/32:
  for each tx in 0..3072/32:
    alloc blurx[-1..33][32]
    for y in -1..33:
      for x in 0..32:
        blurx[y][x] = in[ty*32+y][tx*32+x-1] + in[ty*32+y][tx*32+x] + in[ty*32+y][tx*32+x+1]
    for y in 0..32:
      for x in 0..32:
        out[ty*32+y][tx*32+x] = blurx[y-1][x] + blurx[y][x] + blurx[y+1][x]
```

Sliding window over scanline, the best performer, sacrifices two scanlines of redundant work on overlapping tops and bottoms of independently-processed strips to reclaim fine-grained parallelism within each scanline and coarse-grained parallelism across scanline strips.

**Listing 21: Sliding window over scanline.**

```
alloc out[2046][3072]
alloc blurx[-1..1][3072]
for each ty in 0..2048/8:
  alloc blurx[-1..1][3072]
  for each y in -2..8:
    for x in 0..3072:
      blurx[(ty+1)%3][x] = in[ty*8+y+1][tx*32+x-1] + in[ty*8+y+1][tx*32+x] + in[ty*8+y+1][tx*32+x+1]
  if y < 0: continue
  for x in 0..3072:
    out[ty*8+y][x] = blurx[(y-1)%3][x] + blurx[(y%3)[x] + blurx[(y+1)%3][x]
```

52
The call schedule describes the granularity of interleaving the computation of a function with storage and computation for each function on which it depends. Breadth-first schedule stores and computes $\text{blurx}$ at coarsest granularity. Fused schedule stores and computes $\text{blurx}$ at finest granularity, where values are used and discarded in the same iteration, limiting data reuse. Sliding window stores at root granularity and computes at finest granularity, where $\text{blurx}$ values are computed in the same iteration and persist across iterations.

The lowering process synthesizes a complete set of loop nests and allocations, given a Halide pipeline and a fully specified schedule. For each function, bounds are symbolically evaluated for each dimension using interval analysis, based on the dimensions and symbolic indices. Sliding window optimization shrinks the interval at each iteration by excluding the region computed at previous iterations, whereas storage folding rewrites indices when accessing the region by reducing them modulo the maximum extent of the region used. Multi-dimensional loads, stores and allocations can be transformed to a single-dimension equivalent. Vectorization replaces loop of size $n$ with a single statement, whereas unrolling replaces a loop of size $n$ with $n$ sequential statements. The auto-tuning pipeline schedule (Fig. 20) applies stochastic search to find good schedules for Halide pipelines, where the optimal schedule depends on the machine architecture, image dimensions, the complexity in code generation and dependencies associated with transformations.

4.1.3 Pencil

PENCIL, a platform-neutral compute intermediate language for accelerator programming, provides directives for high-level program behavior to propagate to the back end code generator [3]. \_pencil\_assume($e$) is an assume predicate, where $e$ is a boolean and guaranteed to hold whenever control flow reaches the predicate. The \textit{independent} directive annotates loops that carry no dependencies, which can be executed in parallel. A summary function is associated with each function, where a signature is an identical function that derives the memory access patterns of the library functions.

Since source code is usually not available, \_pencil\_access(name) is used, where name is the summary function describing accesses to $\text{foo()}$. A summary can be used by the compiler to determine memory elements needed to be marshaled into and out of a function. \_pencil\_use and \_pencil\_def are used in summary functions to mark memory accesses, where \_pencil\_use($A[e]$) indicates reads from array $A$ at index $e$ may occur, \_pencil\_def($A[e]$) represents a write may occur and \_pencil\_maybe evaluates to a boolean value, which indicates writes may occur to array $A$ at index $e$. The code below shows a loop nest extracted from the Adaptive Beamformer (ABF) bench-
mark, which invokes \texttt{fft32}. Without a summary, the compiler would have estimated the entire array’s reads and writes, preventing parallelism.

\begin{verbatim}
__attribute__((pencil_access[summary_fft32]))
void fft32(int i, int j, int n,
float in[pencil_attributes n][n][n]);

int ABF(int n, float in[pencil_attributes n][n][n]) {
  // ...
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      fft32(i, j, n, in);
  // ...
}

void summary_fft32(int i, int j, int n,
float in[pencil_attributes n][n][n]) {
  for (int k = 0; k < 32; k++)
    __pencil_use(in[i][j][k]);
  for (int k = 0; k < 32; k++)
    __pencil_def(in[i][j][k]);
}
\end{verbatim}

\_pencil\_kill signals that an argument variable or array element is dead at a program point, which prevents the variable from flowing for any statement executed before a kill to any statement executed after. An example:

\begin{verbatim}
__pencil_kill(A);
for (int i = 0; i < n; i++) {
  if (B[i] > 0)
    A[i] = B[i];
}
\end{verbatim}

If the loop is mapped to a GPU kernel, array \textit{A} needs to be copied from GPU to host after the computation because some elements of \textit{A} may be written to by the loop. The data in \textit{A} is not expected to be preserved by the region and the copy-in may be omitted.

### 4.2 Machine Learning Compilers

#### 4.2.1 Tensor Comprehensions

A language for deep learning is proposed with Tensor Comprehensions [54] (TC), which specifies a network, such that a JIT compiler can algorithmically search for the most efficient execution plan. Code is generated in a language suited for a specific backend, e.g., CUDA for GPU. TC adapts the Einstein notation (a.k.a. summary notation) in expressing tensor operations, which simplifies operations such as reductions, indicated with “!.” For example, a move routine in TC is written as follows:

\begin{verbatim}
def mv(float(M,K) A, float(K) x) -> (C) {
  C(i) +=! A(i, k) * x(k)
}
\end{verbatim}

whereas a strided convolution is written as follows:

\begin{verbatim}
def sconv2d(int sh, int sw, float(N,C,H,W) I,
float(F,C,KH,KW) W, float(F) B) -> (O) {
  O(n,f,h,w) = B(f);
  O(n,f,h,w) += I(n,c,sh*h + kh, sw * w + kw)
  * W(f,c,kh,kw)
}
\end{verbatim}

Affine maps are schedule trees that communicate properties of a high-level language (TC) to a downstream compiler with target-specific information. Schedule trees consist of specific node types. A band node is a partial execution order through one or multiple piecewise affine functions over iteration domains. A permutable schedule band is a tuple of schedule functions that can freely interchange while preserving the semantics of a program. A band member or a schedule domain is an affine function in a band, where a collection of filter nodes partition an iteration space. A context
node provides additional information on variables and parameters, such as tensor extents or GPU grid/block sizes, which may also introduce local scopes and parameters that are constant within a subtree. Extension nodes are auxiliary computations that are not part of an iteration domain, such as copying data to and from shared memory. Elements that belong to $T = D.A$, but not to the set of tensor elements $T$, correspond to out-of-bounds accesses, or $(D.A) \setminus T = \emptyset$.

Listing 22: Canonical sgemm (left). Fused (right).

```plaintext
1 Domain \( \{S(i,j) \mid 0 \leq i < N \wedge 0 \leq j < K\} \)
2 \( \{T(i,j,k) \mid 0 \leq i < N \wedge 0 \leq j < K \wedge 0 \leq k < M\} \)
3 Sequence
4 \( S(i,j) \)
5 \( T(i,j,k) \rightarrow (i, j) \)
6 \( T(i,j,k) \rightarrow (i, j, k) \)
```

```plaintext
1 Domain \( \{S(i,j) \mid 0 \leq i < N \wedge 0 \leq j < K\} \)
2 \( \{T(i,j,k) \mid 0 \leq i < N \wedge 0 \leq j < K \wedge 0 \leq k < M\} \)
3 Band \( \{S(i,j) \rightarrow (i, j)\} \)
4 \( \{T(i,j,k) \rightarrow (i, j, k)\} \)
5 Sequence
6 \( S(i,j) \)
7 \( T(i,j,k) \rightarrow (i, j) \)
8 \( T(i,j,k) \rightarrow (i, j, k) \)
```

Listing 23: Fused and tiled (left). Fused, tiled and sunk (right).

```plaintext
1 Domain \( \{S(i,j) \mid 0 \leq i < N \wedge 0 \leq j < K\} \)
2 \( \{T(i,j,k) \mid 0 \leq i < N \wedge 0 \leq j < K \wedge 0 \leq k < M\} \)
3 Band \( \{S(i,j) \rightarrow (32[i/32], 32[j/32])\} \)
4 \( \{T(i,j,k) \rightarrow (32[i/32], 32[j/32], 32[k/32])\} \)
5 \( \{S(i,j) \rightarrow (i \mod 32, j \mod 32)\} \)
6 \( \{T(i,j,k) \rightarrow (i \mod 32, j \mod 32, k \mod 32)\} \)
7 Sequence
8 \( S(i,j) \)
9 \( T(i,j,k) \rightarrow (k) \)
```

```plaintext
1 Domain \( \{S(i,j) \mid 0 \leq i < N \wedge 0 \leq j < K\} \)
2 \( \{T(i,j,k) \mid 0 \leq i < N \wedge 0 \leq j < K \wedge 0 \leq k < M\} \)
3 Band \( \{S(i,j) \rightarrow (i, j)\} \)
4 \( \{T(i,j,k) \rightarrow (i, j, k)\} \)
5 Sequence
6 \( S(i,j) \)
7 \( T(i,j,k) \rightarrow (i \mod 32, j \mod 32) \)
8 \( T(i,j,k) \rightarrow (32[k/32]) \)
9 \( T(i,j,k) \rightarrow (i \mod 32, j \mod 32) \)
```

Listing 24: Fused, tiled, sunk and mapped.

```plaintext
1 Domain \( \{S(i,j) \mid 0 \leq i < N \wedge 0 \leq j < K\} \)
2 \( \{T(i,j,k) \mid 0 \leq i < N \wedge 0 \leq j < K \wedge 0 \leq k < M\} \)
3 Context \( \{0 < b_x, b_y < 32 \wedge 0 \leq b_x, b_y < 16\} \)
4 Band \( \{S(i,j) \mid i = 32b_x - 31 \leq 32 \times 16 \lfloor i/32/16 \rfloor \leq i - 32b_x \land \}
5 j = 32b_y - 31 \leq 32 \times 16 \lfloor j/32/16 \rfloor \leq j - 32b_y \land \}
6 \( (T(i,j,k) \rightarrow i \mod 32, j \mod 32, 32[k/32]) \}
7 Band \( \{S(i,j) \rightarrow (32[i/32], 32[j/32])\} \)
8 \( \{T(i,j,k) \rightarrow (32[i/32], 32[j/32], 32[k/32])\} \)
9 Sequence
10 \( S(i,j) \)
11 \( T(i,j,k) \rightarrow (i \mod 32, j \mod 32) \)
12 \( T(i,j,k) \rightarrow (i \mod 32, j \mod 32, k \mod 32) \)
```

The transformation engine is based on a core polyhedral scheduling provided by isl, which automatically optimizes for outer loop parallelism and locality. The schedule is further tiled to facilitate mapping and temporal reuse on deep parallelism and memory hierarchy of GPUs. The PPCG compiler maps to GPUs, with additional extensions to support more complex and imperfectly nested structures. Memory promotion deals with explicit data transfers to and from shared and private memory spaces.

The core part of the scheduler iteratively solves an integer linear programming problem to compute piecewise affine functions that form schedule bands. This ensures that bands are permutable and can be further tiled. A data-dependence graph (DDG) is internally built, where nodes correspond to statements, edges express dependencies and each edge is annotated with a set of “typed” dependence relations. The isl scheduler was extended for affine transformations, where users can supply additional arbitrary constraints to be inserted in a linear program. For clustering, a decision function can be supplied for pairwise dependence graph matching and scheduling strategies.
4.2.2 TVM

TVM [5] is an end-to-end ML compilation framework that provides capabilities to express DL algorithms as a computation graph. A Python-based DSL takes in a NN computation graph, which offloads the operations to targeted hardware backends, such as x86, ARM, GPU and FPGAs. TVM/NNVM lowers nodes into low-level Halide-based IR, where loop-based optimizations can be performed.

Optimizing computational graphs involve operator fusion and data layout transformations. Operator fusion combines multiple operators into a single kernel without saving intermediate results back into memory. Categories for graph operators include injective, such as a one-to-one map, reduction, complex-out-fusable and opaque, which cannot be fused. Multiple injective operators can be fused to produce another injective operator. A reduction operator can also be fused with input injective operators. Data layout transformations convert a computational graph to better arrange data for execution on a targeted hardware. A preferred data layout can be specified for each operator, based on memory hierarchy constraints. Generating tensor operations decouple descriptions from computational rules, or schedule optimizations and supports new optimizations, such as nested parallelism or tensorization.

A tensor expression language and a schedule space are used to support automatic code generation, where each operation is described in an index formula expression language. A schedule denotes a specific mapping from tensor expressions to low-level code, where a schedule is built by incrementally applying basic transformations, or schedule primitives that preserve the logical equivalence of a program. Internally, TVM uses a data structure to keep track of the loop structure and other information as schedule transformations get applied. Low-level code is generated for the final schedule, which reuses primitives and loop program ASTs from Halide and introduces new primitives to optimize for GPUs and accelerators.

Latency hiding overlaps memory operations with computation to maximize the utilization of memory and compute resources. CPUs use multithreading, whereas GPUs use context switching of warps and threads. TPUs have a decoupled access-execute (DAE) architecture, which offloads the fine-grained synchronization problem to software. To achieve high utilization of the DAE pipeline, the instruction stream is augmented with fine-grained synchronization operations, which compiles programming for DAEs. Virtual threading is a scheduling primitive that allows a programmer to specify a high-level data parallel program. TVM automatically lowers the program to a single instruction stream with low-level explicit synchronization. The operations of all virtual threads are interleaved into a single instruction stream and the hardware recovers pipeline parallelism required to hide memory access latency.
low-level synchronization in the instruction stream.

Automating optimization is to find the optimal operator implementations for each layer of a deep learning model. Schedule optimizations are chosen, such as modifying a loop order or optimizing for memory hierarchy. Schedules include specific parameters, such as tiling size and loop unroll factor and spans a large search space. An automated schedule optimizer was built that utilizes a machine learning cost model that predicts performance for a given configuration and a schedule explorer that proposes new promising configurations. The schedule space specification is domain-specific knowledge in specifying possible schedules.

This work, which is part of TVM, learns to optimize tensor programs [6]. Tensor operators are specified using index expressions, such as \( C_{ij} = \sum_k A_{ki} B_{kj} \), where \( E \) represents the space of index expressions. Multiple variants of low-level code are generated for a given \( e \in E \), where \( S_e \) is the space of possible transformations, or schedules, from \( e \) to low-level code. For \( s \in S_e \), let \( x = g(e, s) \) be a generated low-level code. \( f(x) \) needs to be minimized, but can be queried by running experiments on the hardware. For a given tuple \( (g, e, S_e, f) \), the problem is formalized by the objective function:

\[
\arg\min_{s \in S_e} f(g(e, s))
\]

The prerequisites include defining an exhaustive search space \( S_e \) covering all hardware-aware optimizations in hand tuned libraries and finding an optimal schedule in \( S_e \). Many DSLs for code generation have different definitions of \( E, S_e, g \). Polyhedral models are a popular choice for \( S_e \), where loop domains are modeled as integer linear constraints. A statistical cost model \( \hat{f}(x) \) estimate the cost of each low-level program \( x \), where run time statistics are collected in a database \( D = \{(e_i, s_i, c_i)\} \) and used to update \( \hat{f} \).

The statistical cost model used was XGBoost, a gradient boosted tree (GBT) which extracts domain-specific features from a given low-level AST \( x \) loop structure, such as memory access counts, data reuse ratio and generic annotations, such as vectorization, unrolling and thread binding. TreeGRU was the other statistical cost model evaluated, which recursively encodes a low-level AST into an embedding vector and is mapped to the final predicted cost using a linear layer.

Multiple training objective functions are given a collection of data \( D = \{(e_i, s_i, c_i)\} \). Loss functions include regression \( \sum_i (\hat{f}(x_i) - c_i)^2 \) and rank \( \sum_{i,j} \log(i + e^{-\text{sign}(c_i - c_j)}(\hat{f}(x_i) - \hat{f}(x_j))) \), which evaluates based on the relative order of a program run time. \( \hat{f}(x) \) is used to select the top-performing implementations. The exploration module, based on simulated annealing, picks a batch of candidate programs with the energy function \( \hat{f}(x) \), which is queried on real hardware. Batches of parallel Markov chains are executed to improve the prediction throughput of the statistical cost model. Assume that a schedule configuration \( s \) is decomposed to \( m \) components \( s = [s_1, s_2, ..., s_m] \). The objective function maximizes a select candidate set \( S \) from the top \( \lambda b \) candidates.

\[
L(S) = -\sum_{s \in S} \hat{f}(g(e, s)) + \alpha \sum_{j=1}^m \left| \bigcup_{s \in S_j} \{s_j\} \right|
\]

The first term encourages candidates that have a low run time cost, whereas the second term counts the number of different configuration components covered by \( S \). Bayesian optimization methods was the uncertainty estimator used, which provides acquisition functions, such as the expected improvement and upper confidence bound, in cases where the uncertainty for \( \hat{f} \) was not available.

A loop-level AST \( x \) (Fig. 22 (a)) is a shared representation of programs that is invariant to the search space. The cost model \( \hat{f}(x) \) takes a low-level loop AST \( x \) as input. \( x \) also needs to be converted to vector space to perform prediction. Context features can be defined at each loop level.
to represent loop characteristics before performing GBT (Fig. 22 (b)). To build context relation features, context vectors are treated as a bag of points and features are extracted that model relations between the feature axes. TreeGRU, displayed in Fig. 22 (c), encodes a program by learning the embedding vector for each identifier and summarizing the AST. Each loop variable is encoded using a context vector to summarize the AST (Fig. 22 (d)). Each loop level embedding $h_i$ is scattered into $m$ vectors with $\text{out}_i = \text{softmax}(W^T h_i) h_i$, where the scattered vectors of all loop levels are summed up for the final embedding.

### 4.2.3 Latte

Latte is a domain-specific language (DSL) for deep neural networks, which abstracts low-level details including parallelization, heterogeneous code generation, and optimizations [52]. Two approaches for implementing NNs include high-performance layer-specific libraries and computational graph engines. Layer-specific libraries expose a set of functions to perform computation for various layer types, which composes functions to construct various network architectures. Computational graph engines describe neural networks as computation graphs, where graphs are constructed using nodes that represent generic operations on $n$-dimensional arrays called tensors. Latte extends the computational graph engine approach by replacing generic operations on multi-dimensional arrays with abstractions specific to neural networks.

Listing 25: The final code after tiling, fusion and parallelization.

```plaintext
#pragma omp for collapse(2) schedule(static, 1)
for n in 1:length(batch)
  for y_tile in 1:TILE_SIZE:height
    gemm(‘T’, ‘N’, TILE_SIZE*width, n_filters,
         n_inputs, convinput[n], conv1weights,
         conv1[n])
    for c = 1:n_filters,
      for y = y_tile:y_tile+TILE_SIZE*2,
        for x = 1:width
          conv1[x, y, c] = max(conv1[x, y, c], 0.0)
        for y = y_tile:y_tile+TILE_SIZE,
          for x = 1:width
            maxval = -Inf
            for p = 1:2, q = 1:2
              maxval = max(poolinput[i, x, y, c], maxval)
            pool1[x, y, c] = maxval
```

A data-flow graph is represented with implicit adjacency lists, where neurons in an ensemble are retrieved with a mapping function. Shared variable analysis on data-flow graph is used to guide code synthesis and optimization, where the compiler determines the compute nodes in the data-flow graph that share data dependencies. Shared variables occur when neurons consume the same input values and neurons share local fields such as parameters. Analysis is performed by partitioning the data-flow graph into ensembles and performing a traversal that compares dependencies amongst compute nodes within a partition.

Users define neuron functions using references to neuron fields, which maps to an array-of-structures (AoS) representation, but prevents vectorization. Vectorization can be enabled by converting references to a struct-of-arrays (SoA) layout. A set of nested for-loops is synthesized by calling
the neuron function for each neuron in an ensemble. Gradient summation is used for synchronizing
model replicas when using distributed data parallelism. After synthesizing a section of the code
for backpropagation of one ensemble, call are inserted during run time to perform asynchronous
reduction of computed gradient between workers, where gradients are computed and communica-
ted asynchronously with other workers.

Latte uses a super-set of Julia internal AST as its IR. During parallelization, a node is introduced
that is consumed by Julia’s parallel accelerator package that indicates the explicit parallel for-loop,
which contains information about the collapsed loop nests, scheduling and the chunk size. During
tiling, an AST node is introduced, which carries metadata used during fusion and can prevent fu-
sion across blocks for ensembles that cannot be fused, such as normalization. After optimization,
the tiled loop nodes are lowered into normal for-loops. The Julia AST is transformed before low-
ering to preserve the loop based structure of the code. Latte also annotates for-loop nodes with
pragmas to guide the C++ compiler to ignore vector dependencies and aliasing.

4.3 Inference Engines

Inference is the stage at which a trained model is used to infer or predict the testing samples and
consists of a similar forward pass as training to predict the values. Unlike training, it does not
include a backward pass to compute the error and update weights and is usually a production
phase where models are deployed to predict real world data.

Training deep learning networks are typically performed in data centers or server farms and the
inference often take place on embedded platforms that are optimized for performance and power
consumption. These platforms are typically limited in memory consumption and power envelope,
which limits usage of the original training framework for inference. Inference engines optimized
for specific hardware platforms can be used in place.

4.3.1 ONNX

Open Neural Network Exchange Format [38] provides an open source format for AI models, de-
fines an extensible computation graph model, as well as definitions of built-in operators and stan-
dard data types, which enables trained models to be transferred between frameworks, such as
PyTorch, CNTK and TensorFlow.

4.3.2 TensorFlow XLA

TensorFlow Accelerated Linear Algebra (XLA) [60] is a domain-specific compiler for linear alge-
bra that optimizes TensorFlow computations, memory usage and portability on server and mobile
platforms. XLA lowers nodes into primitive linear algebra operations and invokes backend-specific
libraries, such as Eigen for CPUs or cuDNN for GPUs. XLA emits a vectorized IR, where the input
language to XLA is “HLO IR,” or high level optimizer. XLA takes computation graphs defined in
HLO and compiles them into machine instructions for various architectures, including x64, ARM64
and NVIDIA GPU.

XLA applies several optimizations and analysis passes, such as CSE, operation fusion and buffer
analysis for allocating run time memory for computation. XLA sends HLO to the backend and per-
foms further HLO-level optimizations, with target specific information, such as fusing operators
and partitioning computation into streams for GPUs.

4.3.3 NVIDIA TensorRT

NVIDIA TensorRT [51] is a platform for high-performance deep learning inference, which includes
a deep learning inference optimizer and run time that delivers low latency and high-throughput for
deep learning inference applications. With TensorRT, neural network models trained in all major frameworks can be optimized, calibrated for lower precision with high accuracy and deployed to hyperscale data centers, embedded, or automotive product platforms. TensorRT provides INT8 and FP16 optimizations for production deployments of deep learning inference applications, where reduced precision inference significantly reduces application latency, a requirement for many real-time services, auto and embedded applications.

4.3.4 Intel OpenVINO

Intel OpenVINO [39] is an inference engine that is similar to XLA. A summary of the steps for optimizing and deploying a trained model includes configuring the model optimizer for your framework, converting a trained model to produce an optimized IR of the model based on the trained network topology, weights, and biases values, testing the model in the IR format using the inference engine in the target environment by the validation or sample applications and integrating the inference engine in an application to deploy the model in the target environment. After the model optimizer is used to create the IR, the inference engine is used to infer input data. The inference engine is a C++ library with a set of C++ classes to infer input data and get a result. The C++ library provides an API to read the IR, set the input and output formats and execute the model on devices.

5 Distributed Methods for Model Training

Attempts have been made to scale the training of deep learning with commodity compute clusters. Training neural networks takes a long time, especially when the training set is large. The benefits of a scaled up approach are the amount of compute nodes and GPUs available in cluster computers. The drawback is in the communication amongst worker nodes and synchronization of gradients, which can severely limit performance, although the gains in a scaled up approach outweigh the drawbacks.

5.1 Parameter Server

5.1.1 DistBelief

DistBelief [10] is one of the earlier known frameworks for scaling up training of deep neural networks. DistBelief initially proposed the parameter server approach, which is still being used today, as well as model parallelism, which shards the network across multiple machines, where each machine is responsible for a portion of the model.

The model was sharded across 8 worker machines, where scalability suffered beyond 8 workers. Proper partitioning of the model to workers is critical for performance, since the models and workers interact as forward activation and backward gradient propagation gets carried out. Similar to distributed graph processing, the performance benefits of distributing the DNN model across multiple worker machines depend on the connectivity structure and computational needs of the model. Models with local connectivity structures are more amenable to extensive distribution than fully-connected structures, given their lower communication requirements. Since the parameter server itself is also distributed, each worker communicates with just a subset of parameter server shards that hold the model parameters relevant to its partition. For fetching the model from the parameter server, workers coordinate with each other in a somewhat synchronized manner before starting a new mini-batch.
Figure 23: (L) Steps required to perform distributed subgradient descent. (R) Each worker caches the working set of \( w \), rather than all the parameters. The architecture of a parameter server communication with several groups of workers.

5.1.2 Key-Value Storage

The scaling of distributed machine learning with a parameter server can be extended with use of key value pairs, which represent a feature ID and weight and is shared amongst a node [29]. The model that is shared amongst nodes can be represented as a set of key-value pairs, where a pair in a loss minimization problem can be a feature ID and its weight. Each entry of the model is read and written locally or remotely by its key.

Data sent between nodes use push-and-pull operations. Range-based push-and-pull optimizes the updates for programmer convenience as well as computational and network bandwidth efficiency. If \( R \) is a key range, \( w.push(R, \text{dest}) \) sends the existing entries of \( w \) in key range \( R \) to a destination, which can be either a particular node or a node group. Similarly, \( w.pull(R, \text{dest}) \) reads all existing entries of \( w \) in key range \( R \) from the destination. \( R \) can be set as either a whole vector or a single key entry.

Servers store parameters as key-value pairs using consistent hashing. Chain replication helps with fault tolerance, where entries are replicated and optimized for range-based communication with compression on both data and range-based vector clocks. Each key-value pair is associated with a vector clock, where time is recorded on each individual node on a key-value pair. Vector clocks track aggregation status, rejecting doubly sent data. A naive implementation requires \( O(nm) \) space for \( n \) nodes and \( m \) parameters, but can be compressed to a single range vector clock, since nodes that push parameters will likely share timestamps. Assume \( v_c(k) \) is the time of key \( k \) for node \( i \). Given a key range \( \mathcal{R} \), a ranged vector clock \( v_c(R) = t \) means that for any key \( k \in \mathcal{R}, v_c(k) = t \), let \( k \) be the total number of unique ranges communicated by the algorithm. Then, at most \( O(mk) \) vector clocks are used.

Nodes send messages to individual nodes or node groups. A message consists of a list of key-value pairs in key range \( R \) and an associated range vector clock.

\[
[v_c(R), (k_1, v_1), ..., (k_p, v_p)] k_j \in \mathcal{R} \text{ and } j \in \{1, ..., p\}
\]

Key and server node IDs are both inserted into a hash ring. Each server node manages a key range, starting with its insertion point to the next point by other nodes in a counter-clockwise direc-
The server node is referred to as the master node of the key range. Each server node stores a replica of \( k \) counterclockwise neighbor key ranges relative to its own. Worker nodes communicate with the master of the key range for both push and pull. Any modifications on the master node are copied with its timestamp to the slave nodes, where modifications are pushed synchronously to slaves.

Fetching data in range \( R \) from some node \( S \) proceeds in two stages. First \( S \) pre-copies all key-value pairs in a range together with its associated vector clocks. If a new node fails at this stage, \( S \) remains unchanged. At the second stage, \( S \) no longer accepts messages affecting key range \( R \) by dropping messages without executing and replying. Also, \( S \) sends all changes that occurred in \( R \) during the pre-copied stage.

When node \( N \) receives a change message, it first checks if also maintains the key range \( R \). If so and the key range is no longer to be maintained by \( N \), all associated (key, value) pairs and vector clocks in \( R \) are deleted. \( N \) scans all outgoing messages that did not receive replies yet. If the key range intersects with \( R \), then the message is split and resent. Departure of the server node is similar to a join, where the server manager tasks a new node by taking the key range of the departing node and detects node failure by a heartbeat signal.

### 5.2 Scaling Linear Algebra with Spark

#### 5.2.1 Compressed Linear Algebra

Compressed linear algebra for large-scale machine learning [12] is scaled with Spark, which uses resilient distributed datasets (RDD) that enable localized lazy evaluation, or compute and discard operations, as well as fault tolerance through replication of RDD. SystemML aims at declarative ML, where algorithms are expressed in a high level scripting language having R-like syntax and compiled to hybrid run time plans that combine both single-node, in-memory operations and distributed operations. This work was extended to scale up machine learning via compressed linear algebra [13].

SystemML supports various input formats and internally converts to binary block matrix format with fixed sized blocks, since tiles and chunks are widely used in large-scale machine learning. SystemML uses a modified CSR for sparse matrix blocks. For a single node, the entire matrix is represented in memory as a single block and block operations can be reused across backends.

Common data characteristics include tall and skinny matrices that consists of more rows (observations) than columns (features), non-uniform sparsity, where many features are created with pre-processing and low column cardinates, which have a few distinct values. The ratio of column cardinality (# distinct values) to the number of rows quantifies redundancy, independent of the actual values. Correlations between columns with respect to the number of distinct value-pairs also quantifies compression potential. For value-based offset lists, column \( i \) with \( d_i \) distinct values requires \( \approx 8d_i + 4nB \), where \( n \) is the number of rows and each value is encoded with \( 8B \) and a list of \( 4B \) row indexes. Co-coding two columns \( i \) and \( j \) as a single group of value-pairs and offsets
requires $16d_{ij} + 4nB$, where $d_{ij}$ is the number of distinct value-pairs. Larger correlation means larger size reduction by co-coding.

Column-wise compression utilizes fewer distinct values per column and high cross-column correlations. Column co-coding partitions column groups, such that columns within each group are highly correlated and columns within the same group are co-coded as a single unit. Column encoding formats use an offset list associated with each distinct tuple stored as compressed sequence of bytes. The efficiency of executing linear algebra operations over compressed matrices depends on how fast the iteration can occur over the compressed representation. The column encoding formats include offset-list encoding (OLE) and run-length encoding (RLE).

Offset-list encoding (OLE) divides the offset range into segments of fixed length $\Delta^a = 2^{16}$ (two bytes per offset), where each offset is mapped to its corresponding segment and encoded as the difference from the beginning of its segment. The size $S_{i}^{OLE}$ of column group $G_i$ calculated as

$$S_{i}^{OLE} = 4|G_i| + d_i(4 + \alpha|G_i|) + 2 \sum_{j=1}^{d_i} b_{ij} + 2z_i,$$

where $b_{ij}$ denotes the number of segments of tuple $t_{ij}$, $|O_{ij}|$ are offsets for $t_{ij}$, $z_i = \sum_{j=1}^{d_i} |O_{ij}|$ is the total offset in the column group and the common header has a size of $4|G_i| + d_i(4 + \alpha|G_i|)$.

In run-length encoding, a sorted list of offsets are encoded as a sequence of runs, where each run represents a consecutive sequence of offsets, via two bytes for starting offset and two bytes for run length. Iterating over RLE group entails scanning runs and enumerating offsets per run. The size $S_{i}^{RLE}$ of column group $G_i$ is computed by

$$S_{i}^{RLE} = 4|G_i| + d_i(4 + \alpha|G_i|) + 4 \sum_{j=1}^{d_i} r_{ij}$$

where $r_{ij}$ is the number of runs for tuple $t_{ij}$.

For matrix-vector multiplication, the product $q = Xv$ of $X$ and column vector $v$ is represented with respect to column groups $q = \sum_{i=1}^{|X|} \sum_{j=1}^{d_i} (t_{ij} \cdot v_{G_i}) 1_{O_{ij}}$, where $v_{G_i}$ is the subvector of $v$ corresponding to indexes $G_i$ and $1_{O_{ij}}$ is the 0/1-indicator vector of offset list $O_{ij}$. For vector-matrix multiplication, column-wise compression allows for efficient vector-matrix products $q = v^TX$ because individual column groups update disjoint entries of the output vector $q$. Each $q_i$ can be expressed over columns as $q_i = v^TX_i$, rewritten in terms of column group $G_i$ as scalar-vector multiplications $q_{G_i} = \sum_{j=1}^{d_i} \sum_{l \in O_{ij}} v_{G_i} l t_{ij}$.

### 5.2.2 Declarative Machine Learning

Declarative machine learning (DML) on Spark [4] was added to SystemML. The MLContext API enables Spark users to register resilient distributed datasets (RDDs) and data frames as input and output variables of the DML script.

```python
1 from SystemML import MLContext
2 X = csvReader.load("ratings.csv").drop("timestamp")
3 ml = MLContext(sc)
4 ml.registerInput("X", X)
5 ml.registerOutput("U", U)
6 ml.registerOutput("V", V)
7 outputs = ml.execute("ALS.dml", params)
8 U = outputs.getDF(sqlContext, "U") ... 
9 outputs1 = ml.execute("ALS_predict.dml", params)
10 predictIds = outputs1.getDF("OutP")
11 movies = csvReader.load("movies.csv")
12 prediction = movies.join(predictIds.C1==movies.movieId).select("title")
```
Spark-specific rewrites include caching, checkpoint injection and repartition injection. By default, storage level MEMORY_AND_DISK is used to exploit caching without deserialization and to prevent repeated lazy evaluation of expensive operations. Checkpoints are injected after every persistent read or reblock to prevent repeated reads, text parsing and shuffles from HDFS. Checkpoints are also injected before loops for all read-only variables in the loop body. For sparse matrices, matrix blocks can be converted into memory-efficient CSR representation. Operations like join or reduceByKey that cause shuffles are very expensive operations because shuffle operations dominate execution time, compared with reads from distributed cache. Unnecessary shuffling can be avoided if inputs to a join are partitioned with the same partitioning function.

Given a Spark configuration with driver memory $M_D$, executor memory $M_E$, the number of executors $|E|$, data fraction $\delta$ and shuffle fraction $\sigma$, the effective memory budget can be derived as follows. The control program memory budget $\bar{M}_{CP}$ for in-memory single-node operations is given by $\bar{M}_{CP} = \alpha M_D$ (default $\alpha = 0.7$). The broadcast memory budget $\bar{M}_B$ is derived with $\bar{M}_B = \beta \delta M_E$ (default $\beta = 0.3$) as broadcasts are managed in data space. Similarly, the total data memory is calculated as $|E| \delta M_E$. $M(X)$ denotes the memory estimate of a single-block matrix, whereas $M(X_P)$ denotes the memory estimate of a block partitioned matrix. The optimization objective $\phi$ is to minimize the total program execution time subject to memory constraints over all operations and execution contexts.

5.3 Large Scale Training

5.3.1 Training ImageNet in 1 Hour

Using 256 GPUs, the Facebook research team reduced the training time for the ResNet-50 model to 1 hour [18], while also achieving near linear speedups when scaling from 8 to 256 GPUs (0.9x). In comparison, AlexNet took 5-6 days to train for 90 epochs on two GPUs [26]. A large mini-batch size consisting of 8192 images was used, which fully exploited the GPUs and made training run faster. However, noise from large mini-batch sizes impact gradient updates, which slows convergence or may converge to a wrong, non-optimal solution. Additionally, multiple GPUs require weights to be synchronized after each mini-batch update, where smaller mini-batches require more communication, leading to overhead.

To compensate for noise, a linear scaling rule was applied, where the learning rate was multiplied by the mini-batch size, which enabled the accuracy between small and large mini-batches to match. According to the authors, the assumption that the two gradients are similar does not hold during the initial training, when the weights are rapidly changing and only for a large, but finite, range of mini-batch sizes, which was 8192 for ImageNet. The authors devised a “warmup” strategy to mitigate the problems with divergence during initial training, where the model used less aggressive learning rates and switched to the linear scaling rule after a few epochs.

Facebook’s Big Basin GPU servers were used, where each unit consisted of 8 NVIDIA Pas-
cal P100 GPUs interconnected with NVLink and is comparable to NVIDIA DGX machines. For communication, NVIDIA Collective Communication Library (NCCL) was used for buffers of size 256 KB or more. NCCL uses GPU kernels to accelerate intraserver collectives, so this approach dedicates more time on the GPU to backprop while using the CPU resources that would otherwise have been idle to improve throughput. Scaling up training to 256 GPUs and a large batch sizes contributed to the accelerated training. This work demonstrates that if the capabilities of learning features drastically increases, larger datasets can be injested, where the added observations could result in even higher accuracies.

5.3.2 ImageNet Training in Minutes

The AlexNet training time was reduced to 24 minutes [62], which made use of a layer-wise adaptive rate scaling (LARS) algorithm that calculated the learning rate for each layer, based on the compute resources. As discussed in [18], the challenges of large batch scaling can be addressed with linear scaling of the learning rate $\eta$ and a warm up scheme for earlier phases of training. The base learning rate (LR) rule is

$$\eta = l \times \gamma \times \frac{\|w\|_2}{\|\nabla w\|_2},$$

where $l$ is the scaling factor, set at 0.001 and $\gamma$ is a tunable parameter selected by linear scaling. Different layers may have different LRs. LARS first gets the local LR for each learnable parameter by $\alpha = l \times \frac{\|w\|_2}{\left(\|\nabla\|_2 + \beta \|\nabla w\|_2\right)}$, then gets the LR for each layer by $\eta = \gamma \times \alpha$. Gradients are updated by $\nabla w = \nabla w + \beta w$, the acceleration term is updated by $a = \mu a + \eta \nabla w$ and, finally, the weights are updated by $w = w - a$.

Using this approach, along with the warmup technique, the authors were able to scale up the batch sizes to 32k as the baseline. Their approach was evaluated with Intel Xeon Phi 7250 processors with eight NVIDIA P100 Pascal GPUs and Intel Xeon Platinum 8160 processors (Stampede2 supercomputer\footnote{https://portal.tacc.utexas.edu/user-guides/stampede2}), which consists of 4,200 KNL nodes, where each KNL node is a 68-core, stand alone self-booting processor. The software ran on two distributions of Caffe, one for GPUs and the other for Intel Machine Learning Scaling Library (MLSL). KNLs divide processors on chip with the memory, which makes memory bandwidth much wider, since memory is the bottleneck in training. This work demonstrates that throughput on large scale model training can be achieved by scaling the number of compute cores and the size of batches.

6 Summary and Directions

Accelerating model training requires in-depth knowledge of the moving parts that constitute a learning algorithm, as well as trade-offs associated with code transformation options that ultimately impacts overall program execution. The problem space was dissected, in terms of data operations, iteration spaces, modeling and an intermediate representation. Optimizations for improving execution performance, both at the machine-independent and architecture-specific level, were covered. Code generation was discussed that included lowering techniques and the deployment of models. Distributed methods for scaling up model training was presented afterward.

Several directions can be pursued. Opportunities for improved model training exist at both high and low levels, where a guided approach that incorporates algorithmic meta-information can inform decisions during the code generation phase. Latte [52] is an example that propagates properties of DL routines to the back end, but is currently tied to the Intel compiler. Abstractions, such as Lift [48] and Cambricon [31], provide building blocks for generating efficient ML code, but a
unified approach is needed that accounts for the computation requirements of model training. Performance measurement and analysis of DL applications is another direction, since current tools lack the ability to report pertinent information that ties a model description with low-level execution bottlenecks that may surface during training. MLPerf [33] is a benchmark suite for measuring performance for ML software frameworks, but more proxy applications [8] will be needed to continue the hardware/software co-design process.

Quantifying the accuracy of model training with error bars also needs further investigation, not only for evaluation purposes, but also for identifying blind spots that may exist in classification. For instance, adversarial machine learning provides a secure and robustness measure against attacks [17], critical in areas such as self-driving cars, where a tainted stop sign can coerce the model into making wrong, perhaps unsafe decisions. Scaled up approaches have been proposed [27], but as attacks evolve, the process of how attacks are constructed will need to be understood. Confidence intervals provide bounds on the accuracy of a model and the procedure of classifying an observation, as well as the locations of the error regions where decisions may go haywire, can reveal the vulnerabilities of ML and provide robustness for learning systems. In addition, certain scenarios can tolerate an error threshold, where sub-optimal solutions, such as the Top-$k$ accuracy, may be sufficient, leading to an overall reduction in training time.

References


67
tional Neural Networks. In Advances in Neural Information Processing Systems (NIPS), pages


Convolutional Neural Networks on GPUs. In International Conference for High Performance

and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In 11th USENIX
Conference on Operating Systems Design and Implementation (OSDI), pages 583–598,

4393, 2016.

Architecture for Neural Networks. In ACM SIGARCH Computer Architecture News, volume 44,

2017.


tation Graphs. In Advances in Neural Information Processing Systems (NIPS), pages 3974–3984,
2017.


[37] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather. Minimizing the Cost of Iterative Com-
pilation with Active Learning. In International Symposium on Code Generation and Optimization


Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image


