Methods for Accelerating Machine Learning in High Performance Computing

Robert Lim
roblim1@cs.uoregon.edu
University of Oregon

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.

KEYWORDS

High-performance computing, GPU programming, distributed systems

1 INTRODUCTION

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 (HPC) architectures and the models that learn the latent representations from the training sets. 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 purposes [30]. In addition, advances in neural machine translation (NMT) [38] have lowered communication barriers and is transforming interactive conversations with computer-aided translation, which has sociolological and economical impacts in society. However, the existing approaches in model training are not sufficient for maximizing performance. Considering a compute cluster with a number of nodes, accelerator architectures, and its network topology, an ideal compiler would automatically create an execution plan that maximizes high-performance and utilization of compute resources, eliminating pain points, such as system configuration issues or low level assembly syntax. This report surveys performance improvements made in the landscape of deep learning (DL) training enabled by HPC in single node and multi-node settings, with a focus on image recognition and natural language processing domains.

1.1 Evolution of Neural Networks

The design of neural networks is a black-box procedure that requires domain expertise and can vary from the number of layers, learning rates, input sizes, non-linear activation units, and feature maps. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [33] is an annual competition organized by the computer vision field that analyzes over 14 million hand-annotated images. Challenge objectives include image classification, single-object localization and object detection.

Table 1 compares selected neural network models from the ImageNet competition, along with the number of layers, parameters, size, operations and Top-5 accuracy. The table was adapted from [80] and the numbers were generated from Torchvision model zoo [84]. Note that LeNet [47] was one of the first convolutional neural networks (CNN) developed for digit classification, but did not participate in ILSVRC. AlexNet [44] is an 8-layer CNN that first won ILSVRC, where the filter shapes varied from layer to layer, resulting in significantly more weights than LeNet. VGG [72] is a 16-layer model similar to AlexNet, but decomposes large kernel-sized filters into smaller ones, leading to a 92.3% accuracy. GoogLeNet [81] introduced an inception module that consists of parallel connections of different sized filters to resemble processing at multiple scales, resulting in a smaller and faster model than VGG and more accurate than AlexNet (89.6%). Inception [82] is similar to GoogLeNet, but applies batch normalization, which achieved 93.3% accuracy. ResNet [28] proposes an even deeper model with 152 layers, using shortcut modules to address the vanishing gradient problem, achieving an accuracy of 96.4%. SqueezeNet [32] is a light-weight CNN model providing AlexNet accuracy with 50x fewer parameters. DenseNet [31] is similar to ResNet, but connects each layer to every other layer and concatenates the feature maps, providing stronger gradient flow with 3x fewer parameters, but a larger sized model. MobileNet [70] is a lightweight network for mobile devices that applies a

Table 1: Selected neural networks for image classification throughout the years.

Year	Model	Layers	Parameters	Size	Operations	Accuracy
1998	LeNet5 [47]	5	1 M	4 MB	60 K	-
2012	AlexNet [44]	8	60 M	240 MB	3 G	84.7%
2013	VGG [72]	16	140 M	500 MB	23 G	92.3%
2015	GoogLeNet [81]	22	6 M	30 MB	3 G	89.6%
2016	Inception [82]	19	23 M	35 MB	5.6 G	93.3%
2015	ResNet [28]	152	60 M	240 MB	23 G	95.5%
2016	SqueezeNet [32]	18	24 M	0.5 MB	23 G	80.6%
2017	DenseNet [31]	161	28 M	268 MB	15 G	93.8%
2018	MobileNet [70]	24	3 M	74 MB	627 M	90.3%
2018	ShuffleNet [102]	164	2 M	20 MB	297 M	88.3%

depthwise separable convolution approach, resulting in a computation reduction and a small reduction in accuracy. ShuffleNet [102] shuffles the order of channels in grouped convolution to alleviate bottlenecks from pointwise convolution, with significant reduction in parameters.

Table 2: Evaluation matrix for best performing machine translation system (BLEU) for WMT 2018. Table source [1]

	Output Language							
Input Language	CZH	X	33.9	X	X	X	X	X
	X	GER	48.4	X	X	X	X	X
	26.0	48.3	ENG	25.2	18.2	34.8	20.0	43.8
	X	X	30.9	EST	X	X	X	X
	X	X	25.6	X	FIN	X	X	X
	X	X	34.9	X	X	RUS	X	X
	X	X	28.0	X	X	X	TUR	X
	X	X	29.3	X	X	X	X	CHN

The impact of neural networks can also be witnessed in machine translation, where the dramatic shift from statistical- to neural-based approaches occurred in 2016 [42]. Similar to ILSVRC, the Conference on Machine Translation (WMT) is an annual event that hosts competitions for machine translation. In addition to translation tasks, competition categories include automatic post-editing, quality estimation and parallel corpus filtering. NMTs typically include a word embedding matrix, a context vector, and attention mechanisms [88]. An encoding and decoding phase takes in a source sentence and aligns words for a target language, whereas bi-directional RNNs [6] simultaneously reads the source sentence in forward and reverse order to alleviate bottlenecks from the disjoint phases and learn features from co-occuring words. Preprocessing of the sentences include tokenization, true casing, and byte-precision encoding. The quality of the translation systems are evaluated based on the Bilingual Evaluation Understudy (BLEU) score, where a score of 100.0 results in a perfect match, and 0.0 a mismatch. Table 2 displays the evaluation matrix of machine translation for the best performing system for WMT 2018. The types of translation systems vary and can include encoder-decoder with attention, deep models with layer normalization, weight tying, back translation, ensembles, and reranking in both directions. Table 2 also indicates that current NMTs still have strides to overcome before achieving a perfect BLEU score.

Efforts to automate the design of neural networks include neural architecture search and hyperparameter optimization. Neural architecture search uses reinforcement learning [105] and evolutionary algorithms [67] toward constructing a deep neural network from scratch, but the search space remains vast and can take up to 2000 GPU hours [105] (3150 GPU hours [67]). Hyperparameter optimization is a procedure that searches for a good range of options for tuning weights of neural networks [7, 74], where efforts include bandit-based [49] and Bayesian [22] approaches. As indicated by the number of parameters and layers in Table 10, neural architecture search and hyperparameter optimization presents a complex multi-modal exploration space to navigate.

1.2 HPC Architectures

This section briefly covers the architectures of CPU, many-integrated core (MIC), and graphic processing units (GPU), displayed in Table 3. Numbers were taken directly from [34, 96]. Tensor processing units (TPUs) and field-programmable gate arrays (FPGAs) are omitted due to the proprietariness of TPUs, the irregular compilation process of FPGAs, as well as the inavailability of both architectures in production systems. Jouppi, et. al. compare performances of CPU, GPU and TPU, as well as describe in-depth the operation of the matrix processor [37], whereas Nurvitadhi, et. al. discuss different FPGA fabrication designs for accelerating neural networks [59]. The number of cores reported for GPUs are

[1] http://matrix.statmt.org

Table 3: Comparison of selected CPU, MIC and GPU architectures.

Year	Name	Architecture	GFL (SP)	GFL (DP)	Cores	Mem (BW)	TDP	Freq (MHz)
2011	Xeon 5690	Westmere	166	83	6	32	130	3470
2012	Xeon E5-2690	Sandy Bridge	372	186	8	51	135	2900
2014	Xeon E5-2699 v3	Haswell	1324	662	18	68	145	2300
2017	Xeon Platinum 8180	Skylake	4480	2240	28	120	205	2500
2018	Xeon Platinum 9282	Cascade Lake	9320	4660	56	175	400	2600
2013	Xeon Phi 7120	Knight's Corner (KNC)	2416	1208	61	220	300	1238
2016	Xeon Phi 7290	Knight's Landing (KNL)	6912	3456	72	600	245	1500
2017	Xeon Phi 7295	Knight's Mill (KNM)	–	–	72	115	320	1500
2011	Tesla M2090	GF100 (Fermi)	1331	665	16	177	250	1300
2015	Tesla K40	GK180 (Kepler)	5040	1680	15	288	235	745
2015	Tesla M40	GM200 (Maxwell)	6844	214	24	336	250	1000
2016	Tesla P100	GP100 (Pascal)	9340	4670	56	720	250	1328
2017	Tesla V100	GV100 (Volta)	14899	7450	80	900	300	1328

Table 4: CPU memory hierarchy.

		Sandy Bridge	Haswell	Skylake
Size (bytes)	L1 (I)	32K, 8-w	32K, 8-w	32K, 8-w
	L1 (D)	32K, 8-w	32K, 8-w	32K, 8-w
	L2	256K, 8-w	256K, 8-w	1M, 16-w
	L3	8M, 12-w	8M, 12-16-w	57M, 11-w
Latency (cycles)	L1	4-5	4-5	4-5
	L2	11	12	12
	L3	30	34-65	34-45

Table 5: Comparison of MCDRAM and HBM.

Hardware	Type	Notes
P100 GPU	3D-stacked SDRAM	16 GB, 732 GB/s (HBM2), 32 GB/s (PCIe)
KNL MIC	MCDRAM + DDR	16 GB on-package, 8 channels, 490 GB/s

streaming multiprocessors, and not the individual cores per multiprocessor, to simplify the comparison with CPUs. Keep in mind that the programming models of CPU, GPU and MIC are vastly different. GPUs program in the single-instruction, multiple threads (SIMT) model, where multiprocessors execute warps of the same instruction in lock-step. CPUs, on the other hand, program in the multiple instruction, multiple data (MIMD) format that enable multi-processing and context switching. MIC architectures are designed as a collection of CPU cores with high bandwidth memory, each with SIMT capabilities. Memory accesses also differ, where CPUs are concerned with aligning data with the memory hierarchy, whereas GPUs transfer data in a bulk-synchronous manner. Table 4 displays memory hierarchy parameters for CPUs with cache capacity, set associativity, and stall cycles, whereas Table 5 compares memory specifications for the Intel Xeon Phi and NVIDIA Pascal GPU.

The comparison of CPU, GPU and MIC architectures were adapted from [69]. Dual-point floating point capabilities and AVX were introduced in 2014, which pushed the CPU FLOP capabilities closer to GPU. The gap was widen shortly after in 2016 with the introduction of additional cores in Pascal P100 and tensor cores in Volta V100. The number of FLOPs were increased with the introduction of AXV512 vectorized units in the Skylake architectures, which somewhat closed the FLOP gap again. Single precision FLOPs for GPUs is about 5x the performance of CPUs. GPUs and MICs generally have higher thermal design power (TDP), when compared to CPUs. The introduction of high bandwidth memory [95], such as multi-channel DRAM (MCDRAM) for Xeon Phi and 3D high-bandwidth memory (HBM) for GPUs in Pascal in 2016, shown in Table 5, further pushed the performance of accelerators by 3x over CPUs. Thus, any improvements to memory bandwidth directly attributes to an increase in the floating point capabilities.

Despite the capabilties of the Intel MIC architecture, the product line has been discontinued [2] in favor of discrete processors, such as Nervana and Arctic Sound. [3] The Knights Landing architecture consists of 36 tiles connected by a 2D mesh, where each tile includes 2 Cores, 4 VPUs, and 1 MB L2 cache capable of executing six concurrent operations (2 VPUs, 2 memory, 2 integer) [36]. This can be challenging

[2]https://www.top500.org/news/intel-dumps-knights-hill-future-of-xeon-phi-product-line-uncertain

[3]https://www.digitaltrends.com/computing/intel-provides-peek-at-what-arctic-sound-gpu-could-look-like

```
for i = 1, N, 1
  for j = 1, M, 1
    S[i, j] = ..
```

(a) Double-nested loop in C.

```
[n] -> { S[i,j] : 1 <= i <= n and 1 <= j <= i }
```

(b) Symbolic constants and set variables.

Listing 1: `isl` set instance.

for certain applications to keep the two-issue-wide (decode, retire) execution pipeline saturated [29]. Another reason for discontinuing the Intel MIC can be attributed to NVIDIA's dominance on the GPU market and its impacts on the machine learning domain. Regardless, the trend seems to indicate that the march toward Exascale computing need not include complex processors like the Xeon Phis, but SPMD processors with multiple discrete accelerators.

1.3 Motivation

The breakthroughs and challenges both in the neural network field in achieving human-like accuracy and the high performance computing architectures in increased floating point capabilities motivate the current investigation in how to accelerate machine learning training. This report aims to bring awareness on the frontiers of accelerated learning of large datasets through the use of HPC. The problem space is introduced in Section 2.1, which includes iteration spaces, transformations, data operations and performance modeling. Intermediate code generation and deployment are discussed in Section 3, which covers HPC and machine learning compilers. Section 4 discusses parallel and distributed approaches toward machine learning training, including optimization and scalable approaches. Section 5 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 iteration spaces, code transformations, linear algebra, and performance modeling.

2.1 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. An *affine* function consists of one or more variables, $i_1, i_2, ..., i_n$ that can be expressed as sums of constant $c_0, c_1, ..., c_n$ multiples of the variables, such as $c_0 + c_1x_1 + c_2x_2 + ... + c_nx_n$. The polyhedral framework models the iteration space of a loop as affine accesses of variables, such as arrays and constants, and aids in assessing run time performance without executing the application. Dependency analysis can be carried out in order to verify that transformations result in performance gains and do not change the logic of the program. Non-affine accesses require the extra step of reconstructing the iteration spaces using indices and pivots for dealing with traversal of sparse matrices.

Integer Set Library. The integer set library (`isl`) represents the iteration space in a polyhedral model as bounded integer sets [90]. Each map, or binary relation, is a finite union of basic sets, and each mapping to a binary relation is on tuples of integer parameters, bounded by affine constraints. *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. `isl` describes iteration domains representative of the GPU execution model that organizes parallel loops into grids, tiles and thread blocks. Listing 1 demonstrates how a loop is defined in `isl` as a set instance. This representation allows exploration of transformations, such as interchanging the *i* and *j* indices, or generating schedules that alter the permutation order of the array accesses.

Polyhedral Representation. Polyhedral techniques can be used to optimize an intermediate representation (IR) for data locality and parallelism. Polyhedral extraction tool (`pet`) [92] is a library that extracts C code based on Clang/LLVM source and builds loops around the representation, allowing further optimizations to take place [92]. Polly [27], which incorporates `isl` and `pet`, parses the static control parts (ScOP) of a source code. A SCoP is represented as 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 that can be scheduled independently.

Definition 1. *A domain $\mathcal{D}$ is an integer set that describes how the statement is executed in different loop iterations.*

Definition 2. *A schedule $\mathcal{S}$ is an integer map that assigns a multi-dimensional point in time to each iteration, which defines the execution order of different statement instances.*

Definition 3. *An access relation $\mathcal{A}$, or body, is an integer map from a domain of statements to a multi-dimensional memory space.*

Vectorization. Modern CPU architectures are equipped with vector hardware units capable executing single instruction, multiple data (SIMD), where an issue of one instruction causes the same operation to be applied to a vector of register

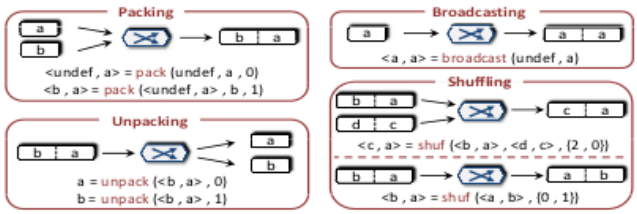

Figure 1: Data reorganization methods. Image source [103].

```
for i = 1, N, 1
  for j = 1, M, 1
    A[i, j] = A[i, j] + c
```

(a) Before

```
for j = 1, M, 1
  for i = 1, N, 1
    A[i, j] = A[i, j] + c
```

(b) After

Listing 2: Loop interchange.

```
for i = 1, N, 1
    A[i] = B[i] + d1

for j = 1, N, 1
    C[j] = A[j] + d2
```

(a) Before

```
for i = 1, N, 1
  A[i] = B[i] + d1
  C[i] = A[i] + d2
```

(b) After

Listing 3: Loop fusion.

units. Vector operations are performed in a pipeline manner, where elements in a vector are fetched serially and computations on different elements are overlapped. A vector length (*VL*) is defined as the instruction and floating point type, where an AVX512 with 32-bit will have a *VL* of 16. Figure 1 shows data reorganization methods when using vectorization, such as packing, unpacking, broadcasting, and shuffling, which introduces overhead. To reduce data reorganization overhead, a cost model can be used to determine whether to vectorize or interleave operations, based on *VL* statement groups that contain at least two loads or stores with consecutive memory accesses in the same iteration [103].

2.2 Code Transformations

A set of instructions is replaced with another set of instructions during code transformation that executes the program faster, with the goal of reducing loop overhead, increasing pipeline opportunities, and improving memory system performance. Examples include vectorization intrinsics, multiply-accumulate (MAC), variable type precision, loop unrolling and replication. Common loop transformations are briefly covered in this subsection.

Loop interchange (List 2) moves arrays from column-major to raw-major, and vice versa, which affects the spatial and temporal locality of the memory elements. *Loop fusion* (List 3)

```
for y = 1, N, 1
  for x = 1, M, 1
    B[y, x] = A[y, x-1] + A[y, x] + A[y, x+1]
for y = 2, N-1, 1
  for x = 1, M, 1
    C[y, x] = B[y-1, x] + B[y, x] + B[y+1, x]
```

(a) Before

```
for y = 2, N - 1, 1
  for x = 1, M, 1
    for i = -1, 1
      B[i] = A[y-1+i, x-1] + A[y-1+x, x]
                + A[y-1+i, x+1]
    C[y, x] = B[1] + B[2] + B[3]
```

(b) After

Listing 4: Interleave.

```
for i = 1, N, 1
  A[i] = B[i] + C[i]
```

(a) Before

```
for i = 1, N - mod(N, 4), i += 4
  A[i:i+3] = B[i:i+3] + C[i:i+3]
```

(b) After

Listing 5: Strip-mining.

```
for i = 1, N, 1
  for j = 1, N, 1
    C[i] = A[i,j] * B[i]
```

(a) Before

```
for i = 1, N, 2
  for j = 1, N, 2
    for ii = 1, min(i+2, N), 1
      for jj = 1, min(j+2, N), 1
        C[ii] = A[ii,jj] * B[ii]
```

(b) After

Listing 6: Tiling.

```
for i = 2, N + 1, 1
      = A[i-1] + 1
  A[i] =
```

(a) Before

```
t1 = A[1]
for i = 2, N + 1, 1
      = t1 + 1
  t1 =
  A[i] = t1
```

(b) After

Listing 7: Scalar replacement.

combines two loops into one, whereas *loop distribution* performs the opposite by breaking one loop into two. *Interleave* allows multiple occurrences of the same operation across consecutive iterations to be grouped together into single SIMD instructions (List. 4). *Strip-mining* (List. 5) fragments a large loop into small segments, which increases temporal and spatial locality for reuse in subsequent passes, and reduces the number of iterations by a factor of vector length performed per SIMD operation. *Tiling* (List. 6), which consists of strip-mining and loop interchanging, divides an iteration space into tiles and transforms the iteration of the nested

Table 6: Counts for GCC optimization flag levels.

-O, -O1	-O2	-O3	-O0	-Os	-Ofast	-Og
45	47	16	1	6	1	12

loops. *Scalar replacement* (List. 7) eliminates loads and stores for array references with use of a temporary variable.

Compiler flags enable code transformations during the compilation process. Table 6 shows counts for various types of GCC optimizations when turning on different levels. Refer to [4] for different variants included in each level. The number of transformation options increases the complexity of the search space when generating code variants. Exploring every single transformation possible would require m^n code variants, for m options and n transformations, which is NP-complete and not feasible for code generation.

2.3 Linear Algebra

The study of high-performance generalized matrix multiplication (GEMM) dates back to 1979 with the release of LINPACK, which measures the performance of a computer solving a system of linear equations [5]. LINPACK utilizes basic linear algebra subroutines (BLAS) for matrix and vector operations. To date, performance of the Top 500 Supercomputers are measured using LINPACK [6], which scales the problem size for a given system configuration.

The operations performed in machine learning include linear algebra and tensor decompositions, to name a few [58]. In machine learning, a high percentage of operations during training are variants of multiply-add accumulate (MAC) of matrices and vectors, as seen in Table 1. The ubiquitiousness of MAC operations in scientific applications have prompted architecture manufacturers to create dedicated hardware units, such as Intel's fused multiply-add (FMA), the tensor cores in NVIDIA V100 Volta, as well as FPGAs that provides customized fabrication of logic gates, as discussed in Sec. 1.2. High performance libraries for linear algebra include Intel MKL [35], openBLAS [62] and Eigen [19] for CPU, and cuBLAS [13], cuDNN [14] and MAGMA [54] for GPUs.

Definition 4. *The GEMM operation is defined as*

$$C = \alpha A * B + \beta C, \tag{1}$$

where A, B and C are matrices, A is $M \times K$, B is $K \times N$, and C is $M \times N$.

GEMM Abstractions. CUTLASS provides templates and abstractions to simplify GEMM operations in CUDA C++ that directly maps to the backend GPUs, based on user supplied input [15]. The result is partitioned into tiles that fit in on-chip memory, where the outer product is applied to each tile. Each thread block, partitioned by warps that are further partitioned by block tiles, computes a portion of the result by iteratively loading blocks of matrix data from the input matrices and computing the accumulated matrix product. Once data is written to shared memory, each warp computes a sequence of accumulated matrix products by iterating over the thread block tiles, loading submatrices from shared memory and computing the accumulated outer product.

Linear Algebra Language. BLAC, or basic linear algebra computation, provides a language for linear algebra (LL) [75, 76], where the output is a product, addition, transposition or a scalar. A fully tiled BLAC in LL is rewritten into Σ-LL, which captures the explicit gather and scatter operators on matrices. Given a matrix-vector multiplication, the input is transformed using the *SInfo* and *AInfo* dictionaries of matrices, where *SInfo* associates regions of a matrix to structures and *AInfo* provides information for matrix block accesses within a region. A set of CLooG (chunky loop generator) statements [7] is produced that defines the polyhedral domains of the loop. Similar to Polly [27], discussed in Sec. 2.1, a CLooG statement consists of a polyhedral set σ representing the iteration space, a polyhedral map ρ schedule that determines the traversal order of the domain and the body B, which is a Σ-LL expression. For the BLAS category, performance was 2.5x faster than MKL when data was able to fit in L1 cache and 1.6x for L2. For the BLAS-like category, improvements exceeded `icc`-compiled code by 7x and was 1.4x faster than MKL, although MKL performed better for bigger sized parameters.

GEMM Code Generator. A portable compiler approach, or POCA, is a GEMM kernel generator that consists of a μ-kernel optimizer that lowers to LLVM IR [79]. The μ-kernel optimizer includes a single-iteration scheduler, a two-iteration pipeline scheduler, and an unroll-based rotating register allocator. A data dependence graph is used in the *single-iteration scheduler* for revealing vectorization opportunities and available registers. The *two-iteration pipeline* attempts to maximize CPU utilization and increase register usage. Loops are unrolled by a factor of 2 to 8 with *unroll-based rotating register allocator*. For consecutive memory accesses, the layout strategy, whether column- or row-major, is to pack the accesses in iterations of a loop in contiguous buffers. The tile size is a multiple of vector length (*VL*), which represents the maximum number of data elements in a vector register. The first study showed that POCA outperformed `icc` and `gcc` for both Intel Sandy Bridge and ARM

[4] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[5] http://www.netlib.org/linpack/
[6] https://www.top500.org/project/linpack/
[7] https://www.cloog.org

Cortex-A57 processors. The results also showed that Intel MKL is unstable for smaller matrices, but exhibits the best performance for large matrix sizes. For instruction scheduling, combining single-iteration scheduling and two-iteration pipelining always yielded performance gains, when compared to each alone, outperforming the baseline by 12.5% on the Sandy Bridge and 16.7% on the Cortex-A57. Loop unrolling achieved superior performance for Cortex-A57 when unrolled by a factor of 8, and good performance for Intel with 4 as the unroll factor, which slightly worsened as the unroll factors increased due to cache misses.

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 linear algebra [43]. For instance, learning models such as hidden Markov models, latent Dirichlet allocation (LDA), and Gaussian mixture models can be interpreted as performing tensor decompositions [3]. Canonical polyadic decomposition (CPD) extends singular value decomposition (SVD) for high order tensors. Greedy and non-linear least squared methods are typical methods to solve CPD. The number of features are reduced in a $N \times M$ matrix, where $N < M$, for N observations and M features. Low-rank approximation can be obtained through factorizing a matrix $\mathbf{M}$, which results in a summation of F singular rank-one matrices. The problem of matricized tensor times Khatri-Rao product (MTTKRP) is often studied because it is the bottleneck when computing CPD [73]. Tensor decompositions can also be used to compress a trained network with minimal impact on accuracy [39, 46], where 4D tensor weights are compressed with low-rank approximation and trades off with degrading accuracy.

2.4 Performance Modeling

A model aides 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 cost models that augment the decision-making of the compiler. Instrumentation methods provide concrete results for measuring performance, though, at the cost of executing an application. Since analytical models are difficult to generalize and, in some cases, be limited in expressivity, collecting performance measurements, either via simulation or execution, may be the only option. Hybrid approaches, such as profile-guided optimization and feedback-directed optimization, make use of analytical models with profiles of an executed application, combining cost models with run time measurements to inform the compiler on whether code transformations yielded performance gains or drawbacks.

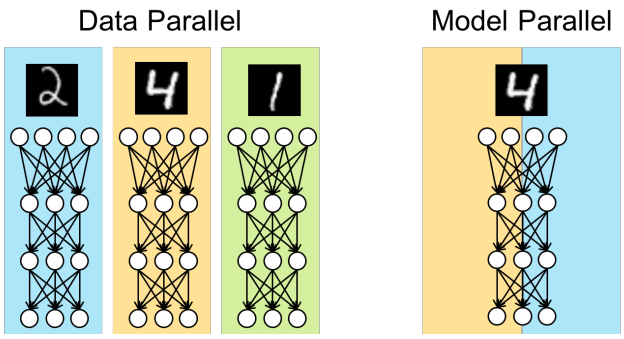

Figure 2: Parallelism strategies for distributed deep learning. Image source [8].

2.4.1 Neural Networks Primer. A neural network consists of an input layer, a hidden layer, and an output layer, each with neurons connected as weights that produce a classification result. A deep neural network is a neural network beyond three layers, with multiple hidden layers between the input and output layers. At each layer j in a neural network, each neuron computes $y_j = f(\sum_{i=1}^{n} W_{ij} \times x_i + b)$, where W_{ij} are the weights, x_i are the input activations, b is the bias term, and y_j are the output activations [80]. A non-linear function $f(\cdot)$ generates an output activation if the inputs crosses a threshold, and none otherwise.

Training a neural network involves updating the weights w_{ij} with an optimization algorithm, such as stochastic gradient descent. The gradient of the loss relative to each weight is the partial derivative of the loss with respect to the weight subtracted by the current weight, $w_{ij}^{t+1} = w_{ij}^{t} - \alpha \frac{\partial L}{\partial w_{ij}}$, where α is the learning rate. The gradient adjusts the weights in order to reduce the overall loss. Backpropagation computes the partial derivative of the gradients, which passes the values backwards through the network to compute how the loss is affected by each weight. Neural networks typically employed include fully-connected networks, convolutional neural networks (Sec. 3.3.1), recurrent neural networks (Sec. 3.3.2), and dynamic neural networks. CNN operations can be performed with either GEMM, FFT, direct convolution, or Winograd. For RNNs, gating operations include LSTM and GRU that are capable of tracking spatial-temporal dependencies.

2.4.2 Distributed Deep Learning. Parallelism strategies for deep neural networks include data parallelism and model parallelism. Data parallelism partitions the dataset across available compute nodes and each node trains on the dataset and maintains a local copy of the model. Model parallelism,

on the other hand, partitions the model across compute nodes and the dataset is passed across all nodes that define the model. Figure 2 compares the parallelism approaches.

2.4.3 Performance Models for Deep Learning. Paleo [65] models the behavior of a neural network by counting the number of operations with respect to the architecture of the network. They estimate a full pass of AlexNet and VGG-16 running on TensorFlow with an NVIDIA Titan X GPU. The authors also performed scalability studies, where they estimated AlexNet training with hybrid parallelism (8 GPUs with a weak scaling strategy) and were able to predict the execution time when scaling the number of workers, from 1 to 8. The study also showed that Paleo was able to simulate a completely different parallelism strategy, data parallelism, and estimate its training time, which could be useful in exploring alternatives for a given architecture.

The problem of resource allocation and task scheduling can be formulated as a minimization problem, where the objective is to find the right number of compute resources that maximizes execution performance, based on constraints of the problem [99]. Design decisions include data parallelism or model parallelism, the number of parameter servers for weight synchronization with other nodes, the number of workers, threads, tasks, type of communication, as well as the neural network operations. Scalability studies revealed that certain architectures benefited under certain parallelism strategies (e.g. ImageNet-22K achieves 7x speedup with model-parallelism), and that certain types of parallelism exhibited certain characteristics (e.g. data parallelism always linear, parameter server roughly linear). When estimating epoch training time, comparing the number of workers, replicas, and parameter servers, their models were also nearly accurate for ImageNet-22K. However, their results did not account for the total number of epochs for convergence, which may be non-trivial but more insightful. When comparing computation and communication, the estimated time for communication was generally overestimated, whereas computation was nearly exact, except for weight updates where the time was underestimated.

Scalability studies on AlexNet, GoogLeNet, and ResNet-50 neural network architectures were carried out by modeling GPU characteristics, such as I/O, host-to-device copies, forward time, backpropagation time, weight update, and iteration time [71]. For intra-node performance on multiple GPUs (2 Intel Xeon E5-2650v4 CPUs, 4 NVIDIA P100 GPUs), Caffe-MPI, MXNet and TensorFlow overlapped communication and computation when parallelizing gradient aggregation and backpropagation, whereas CNTK did not and ended up performing worse. For communication, CNTK uses NCCL [56], MXNet uses TCP socket communication with key-value pairs, TensorFlow uses `grpc` remote process call, and Caffe-MPI uses a decentralized method with NCCL that runs in parallel with backpropagation. Caffe-MPI processed more images-per-second than all other frameworks on GoogLeNet in a multi-node setting, but did slightly worse in single node settings, compared to CNTK and MXNet. Also, ResNet-50, which is significantly deeper with smaller kernel sizes, required more frequent gradient communication and less computation, which was difficult to overlap communication with backpropagation, whereas this overlap was better handled in AlexNet and GoogLeNet, which has larger kernel sizes and fewer layers.

[8] https://chainermn.readthedocs.io/en/stable/tutorial/overview.html

Table 7: LIFT-IL parallel patterns.

Type	Pattern
Algorithmic	map, reduce, id, iterate
Data layout	split, join, gather, scatter, slide
Hierarchical	global, local, private, map-, vec-

3 COMPILERS FOR HPC PLATFORMS

Deployed compilers and run times for machine learning code generation and execution are discussed in this section, which include abstract syntax trees, intermediate representations, and backend code generation.

3.1 Abstract Syntax Trees

The abstract syntax tree (AST) represents a program for a targeted backend that translates source code into machine code for execution. An intermediate representation (IR) decomposes a program into basic blocks that allow other transformations to take place. For neural network code generation, the AST abstracts the operations, such as parallel patterns and linear algebra, to simplify mapping between hand-written code to backend targets.

Intermediate Language. LIFT-IL [77] provides parallel patterns for the GPU thread and memory organization, where patterns are categorized as algorithmic, data layout and hierarchical. *Algorithmic* patterns are operations, such as `map`, `reduce`, `id`, and `iterate`. *Data layout* patterns include `split`, `join`, `gather`, `scatter`, and `slide`. *Hierarchical* patterns represent the memory hierarchy, such as `global`, `local` and `private` address spaces, in addition to map and vector versions, such as `mapGlobal`, `mapLocal`, and so forth. Table 7 displays the parallel patterns from LIFT-IL. Array accesses are simplified with data parallel patterns, with the proper array indices generated from the parallel patterns. For instance, a matrix transposition can be simplified with a `join` that flattens the 2D matrix, a `gather` that rearranges the indices with

a stride, and a `scatter` that splits the array. A tiled convolution operation can be overlapped with `slide` and `map` for 2D tiles, which consists of `split`, `join` and `gather` patterns. In addition, a barrier elimination procedure looks for `mapLcl` calls that do not call `split`, `join`, `gather` and `scatter`, since those patterns require that threads share data and read from the same memory location. With the array access simplification and barrier elimination, their approach improved performance, especially on the matrix multiplication and convolution benchmarks, where the relative performance improved by 20x.

Two-level IR. Glow is a compiler for PyTorch that takes the two-level IR approach, where a high-level IR layer represents the neural network and a low-level IR performs machine-specific optimizations [68]. The high-level IR is a dataflow graph that consists of operations, such as convolution, matrix multiply, and ReLU, that are connected with vertices to indicate the flow of directions and represent dependencies. Code transformations can be performed at the high-level IR level. Automatic differentiation is supported in Glow, which uses persistent tensors for phases that share intermediate data, such as forward and backpropagation. The node lowering phase consists of breaking up the high-level IR into linear algebra operations for the low-level IR phase, which gets offloaded to the LLVM backend.

3.2 Compilation Techniques

To enable code transformations, the compiler decomposes the basic blocks from an IR into individual instructions that map to the backend architecture. This provides an abstraction from machine code level, since a variety of architectures can be targeted. The dilemma is that the parameter options remain vast, when accounting for compiler flags, code transformation options, targeted hardware, thread and worker settings, and the amount of compute resources available. Exploring all possible options during compilation would be infeasible. Limitations to hardware resources, such as issue widths of an architecture, also factor into performance. Considering functional units can only issue specific instructions at a given clock cycle (integer, memory, floating point), a poorly selected set of instructions that fail to utilize the available functional units will severely hinder performance.

Parallelism and locality factor into the overall execution performance of a 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, since a cache miss requires fetching data from the memory hierarchy, and each level up the hierarchy incurs additional latency that degrades performance. The lack of high-level knowledge of an application may overlook any opportunities for performance enhancements during code generation.

3.2.1 Instruction Scheduling. As discussed in Sec. 2.1, an iteration space facilitates the compiler in reasoning about performance at the instruction level to generate efficient code. A *schedule* consists of choices of when and where instructions get executed, which is a compiler optimization to hide latency and increase instruction level parallelism. Since the architectures have complicated hardware units, knowing the number of cycles for each particular instruction can help predict performance. For instance, a 4-way superscalar might be limited to issuing at most 2 integer, 1 memory and 1 FP instruction, in addition to forcing stalls for a particular instruction (2 stalls per FPU instruction). Software pipelining, which overlap compute and memory operations, become an important performance improvement that can be made.

Polyhedral Representation for CUDA. A C++ to CUDA-OpenCL source-to-source transformation framework based on the polyhedral compilation is proposed with Polyhedral Parallel Code Generator (PPCG). A polyhedral model is extracted using `pet`, which consists of an iteration domain, access relations and a schedule, as defined in Section 2.1. Dependence analysis reveals parallel loops that can be tiled and mapped onto GPU blocks and threads. A schedule is generated, which maps data to several levels of the memory hierarchy, such as global, shared or register depending on its access needs, decides what parts of the program execute on the CPU and GPU, and maps tiles to blocks and threads. Once parallelism and tiling have been defined with an iteration domain, the final code is generated with `isl`. When comparing PPCG on a M2070 Fermi GPU with Pluto 0.7 (OpenMP), Pluto 0.6.2 (C-to-CUDA), and Par4All [64], speedups were achieved for linear algebra-related operations, such as correlation, covariance, 3D matrix multiplication, GEMM, and Jacobi. Since PPCG works in the polyhedral space, there were a handful of benchmarks where PPCG ended up hurting performance, such as Cholesky, Durbin and triangular solver, amongst others. Reasons included that certain applications were not suitable for GPU execution, excessive CPU-GPU interaction, and complicated generated code. There were no comparisons on how Par4All performed for the applications that PPCG performed worse, which would have been insightful, since Par4All does not rely on the polyhedral framework.

Image Processing Pipelines. A DSL compiler for image processing is proposed with Halide [66], which separates high-level image functions from low-level schedule that offloads to LLVM. Images are represented as pure functions over an infinite integer domain and the value of a function point represents its pixel color. Pipelines are specified as chains of functions, which are simple expressions or reductions over

a bounded domain. The pipeline schedules are discovered using stochastic search. The search space is vast (10^{720} schedules for local Laplacian), and depends on the machine architecture, image dimensions, and code generation, as well as transformation choices, such as sliding window, storage folding, vectorization, unroll, and GPU thread block dimensions. A genetic search algorithm generates a fixed population size of 128 individuals per generation, with mutation rules incorporating knowledge about imaging, such as transformations that performed poorly. When evaluated on an Intel Xeon W3520 CPU and NVIDIA Tesla C2070 GPU, speedups of 1.2x to 4.4x on the CPU and 2.38 to 9.9x on the GPU were observed, and the number of lines were significantly shorter compared to the expert version. When auto-tuned, the schedules generalized better from low to high resolution inputs, where slowdown was modest for low resolution inputs (0.97x-1.2x), compared to high-resolution inputs (1.4x-16x).

3.2.2 Sparse Matrices. Sparse matrices introduce non-affine array accesses because storing all zeros would be a waste of memory space. Instead, an iteration space and data space are used to track the *position* of non-zero locations and *indices* for accessing data. Converting from non-affine to affine accesses allow compilers to parallelize operations on bounded loops. Inspector-executor approaches facilitate in reconstructing the sparse matrix representation through indirect array accesses of the compressed matrices to enable loop and data layout transformations to be performed [78]. Storage formats for sparse matrices include compressed sparse row (CSR), compressed sparse column (CSC), diagonal (DIA) and Ellpack (ELL), which adds complexities that result from directional accesses, and the conjunctive or disjunctive merging of sparse tensor indices.

Tensor Compiler. The Tensor Algebra Compiler (TACO) [41] computes tensor algebra expressions on sparse and dense tensors for both CPU and GPU. An *iteration graph* describes how to iterate over non-zero values of a tensor expression, and 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 *merge lattice* comprises n lattice points and a meet operator, where each lattice point has a set of tensor dimensions to be merged consecutively and an expression to be evaluated. Extensions to TACO include format abstractions [11] and workspaces [40], used as temporary storage of intermediate results that avoids recomputing. *Concrete index notation*, similar to an ScOP (Sec. 2.1), is an intermediate language that describes the way an expression is computed. Applying workspace optimizations to sparse vector addition with a dense result enables partial results to be efficiently accumulated and increases locality and reuse.

Transformations for Sparse Matrices. The CUDA-CHiLL compiler applies transformations for sparse matrices, which includes *make-dense*, *compact*, and *compact-and-pad* [89]. *make-dense* converts a sparse matrix to a dense matrix by adding guard conditions that replace non-affine index expressions with affine accesses. *compact* and *compact-and-pad*, which convert matrices from dense to sparse, includes an inspector that gathers the iteration satisfying the guard and an executor that is the resulting transformed code. The *compact* transformation results in non-affine loop bounds and array index expressions, whereas *compact-and-pad* inserts zeros for padding to correspond to the optimized executor. Standard loop transformations (Sec. 3.2.2) can be applied in all three cases. When compared on a NVIDIA Tesla K20c Kepler with CUSP v4.0 on the Sparse Matrix Collection [9], their approach was on average within 5% the performance of CUSP for DIA and ELL. CUSP performed better in smaller stencil computations (3-7), whereas CHiLL performed better for larger stencils (27). CUDA-CHiLL requires a global memory read to initialize the result vector, which is noticeable in smaller stencils. DIA is on average 1.27x faster than CUSP, since CUSP takes two passes to initialize and copy values, whereas CUDA-CHiLL does it in one pass. For ELL, the performance gains were 0.52x to 1.26x over CUSP without transpose, but suffered between 0.26x to 0.40x with transpose, due to the additional overhead.

3.2.3 Code Optimizations. Code optimizations are improvements that can be made to the program either through rewriting instructions or with reordering of the code. Optimizing code can be achieved with compiler techniques, code transformations, and efficient memory management. 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 may overlook any performance enhancement opportunities and introduce unnecessary run time overhead. 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 and stored in registers as 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 execution time. Locality and prefetching, data alignment, replication, reordering of access patterns, are strategies that can maximize performance.

[9] https://www.cise.ufl.edu/research/sparse/matrices

Straight Line Scalar Optimizations. gpucc is a CUDA Clang compiler that originated from Google and has been mainlined in the LLVM branch[10] [97]. Several GPU transformations proposed include memory space inference, straight-line scalar optimizations, pointer arithmetic reassociation, straight-line strength reduction, and global reassociation. Memory spaces are inferred by noting all pointers declared in a program and reassociating previously declared ones. *Straight-line scalar optimization* (SLSO) exposes partially redundant computations typical in scientific applications that access multidimensional arrays using pointer and integer arithmetic. *Straight-line strength reduction* looks for partial redundancy in straight line code, whereas *global reassociation* rewrites expressions for better performance. Pointer arithmetic can be folded with *addressing mode* that adds or subtracts an integer constant from the expression of the pointer addresses. `reg + immOff` is an addressing mode by NVIDIA NVPTX, where `reg` is a register and `immOff` is a constant byte offset. Explicit variables for a memory location, such as shared load, can be emitted with NVPTX, which results in 10% faster executing instructions, compared to a standard load. On 5 end-to-end benchmarks (ML, image classification, NLP, mnist) evaluated on a NVIDIA Tesla K40c, gpucc achieved a 22.9% geometric mean speedup over nvcc, due to performing 64-bit divides in 32-bit, and better SLSO, but performed worse in others. SLSO optimization may help ML-related applications, as indicated by the 22.9.% geometric mean, but may perform worse, due to its aggressive approach in seeking partial redundancies and performing global reassociation. As described in the paper, a limitation of SLSR is that it cannot optimize instructions not dominating one another. A fallback mechanism that decides between gpucc and nvcc based on static code analysis would be useful.

Active Learning. The optimal set of compilation parameters for a program can be discovered with active learning that builds up a program-specific model to predict the run time from a given set of optimizations [60]. The algorithm constructs a model with training examples chosen from potential examples. At each iteration, the candidate combines random unobserved points and previously seen examples. The next training example is chosen based on a scoring function and the run time is measured to update the model. The static models used with the dynamic tree framework is a 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. Two heuristics include estimating variance of the maximized output relative to other candidates and selecting a candidate that reduces the predicted average variance across other candidates. Sample size is also accounted for, since not all samples may need to be evaluated in order to gain good choices, speeding up the learning process.

3.2.4 Compilation of ML Models. Techniques for ML compilation for improved performance include mapping iteration spaces that define schedule trees, learning to optimize the input size of tensor operations, mixed precision training, and dataflow analysis for distributed memory communication. With the proliferation of programming frameworks for deep learning [17], vendors have also released inference engines to allow exchangability amongst frameworks and portability across architecture backends. Inference engines allow neural network models, typically trained in data centers, to be deployed on mobile embedded devices that are constrained for performance and power consumption. The process takes as input a pretrained model in the form of a shared format, such as ONNX [61], that defines a computation graph model, as well as definitions for built-in operators and standard data types. The optimizer performs rewriting and code transformations for targeted backends, including precision calibration, layer fusion, kernel auto-tuning, dynamic tensor memory and multi-stream execution. The solution is deployed on a targeted device and real-time inferencing is performed, such as object detection in autonomous vehicles, image captioning, or speech translation.

Tensor comprehensions (TC) is a language for expressing element-wise computations of tensors, such that a JIT compiler can algorithmically search for an efficient execution schedule [87]. Affine maps are schedule trees that communicate properties of a high-level language (TC) to a downstream compiler with target-specific information. A schedule band is a tuple of functions that can interchange while preserving the semantics of a program. A context node provides additional information on variables and parameters, such as tensor extents or GPU grid/block sizes, which may also include local scopes and parameters within a subtree. The transformation engine is based on a polyhedral scheduler, which solves an integer linear program to compute piecewise affine functions that form schedule bands. The schedule is further tiled to facilitate mapping and temporal reuse on GPUs with PPCG [91], with added support for complex and imperfectly nested structures. A data-dependence graph is built, where nodes correspond to statements and edges express dependencies, annotated with a set of typed dependence relations. The experiments ran on 2-socket 8 GPU nodes with Caffe and ATen and compared transposed matrix multiplication, transposed batched matrix multiplication, grouped convolutions, and fused MLP. For all cases, the largest problem

[10] https://releases.llvm.org/3.9.1/docs/CompileCudaWithLLVM.html

Table 8: Selected compilers that support machine learning-related operations and whether polyhedral, GPU or LLVM-based.

Name	Description	Poly	GPU	LLVM	Transformations
PPCG [91]	CUDA/OpenCL source-to-source transformation framework, exploits GPU thread and memory hierarchy	✓	✓	✓	Tiling, strip-mining, loop fusion, fission, affine scheduling
TACO [41]	Compiler for dense and sparse tensor matrices		✓		Workspaces, CSR to CSC conversion
Halide [66]	DSL compiler for image processing pipelines, learns to optimize schedule	✓		✓	Interleave, fusion, tiling, sliding window
TVM [9]	ML compiler for CNN, MLP, others, learns to optimize schedule and tensor size, Halide IR	✓	✓	✓	Fusing, tiling, data layout, reduced precision, operator optimizer
Glow [68]	Compiler for PyTorch that has high-level IR (computation graph, conv, ReLU, etc.) and low-level IR (linear algebra operations)		✓	✓	Kernel fusion, automatic differentiation, graph optimizer, IR optimizer, quantization
gpucc [97]	Performs optimizations at NVPTX level, looks for partial redundancies typical in HPC, treats as constants, infer memory spaces		✓	✓	Loop unroll, straight-line scalar optimizations, pointer arithmetic reassociation, straight line strength reduction, global reassociation
Latte [85]	DSL for CNN based on Intel compiler, high level info gets propagated to backend				Kernel pattern matching, loop tiling, fusion, parallelization, vectorization
Tensor Comprehensions [87]	DSL for machine learning, polyhedral and Halide IR	✓	✓	✓	Fused, tiled, sunk, mapped, combination of all
CHiLL [89]	Inspector executor for sparse matrices	✓	✓		make dense, compact, compact and pad, permute, skew, shift, tile, unroll, scalar expand, coalesce
TensorFlow XLA [98]	Linear algebra compiler for TensorFlow, cuDNN support	✓	✓	✓	CSE, operator fusion, buffer analysis
NVIDIA TensorRT [83]	Performs auto-tuning when loading model, native cuDNN, CUDA support, optimized for GPU backends		✓		Batching, streaming, layer fusion, MLP fusion
Intel OpenVINO [63]	Supports ONNX, TensorFlow, Caffe, MNNet layers for Intel architectures		✓		Linear operation fusing, stride optimization, grouped convolution fusing

size was 4.2x on Maxwell (3.4x on Pascal) slower than Caffe2 with CUBLAS. TC was not implemented with register tiling, so performance was bounded by shared memory bandwidth, whereas CUBLAS operates at close to peak with larger input sizes.

Tensor virtual machine (TVM) is a ML compiler that takes as input a computation graph, lowers the nodes into a Halide-based IR for code generation [66], and targets multiple backends, such as GPU, TPU, and embedded devices [9]. A tensor expression consists of operations described in an index formula language. The loop structure and other information are tracked as schedule transformations get applied. Low-level code is generated for the final schedule, which uses the loop program AST and primitives for optimizing on GPUs and accelerators. The selection of operators for each layer in a neural network can also be automated with TVM [10], where the search space includes tiling size, loop unroll factor, loop order traversal, and overlapping compute and memory operations. The schedule optimizer utilizes XGBoost, a gradient boosted tree model that maps architecture-specific features to a low-level AST, such as memory access counts, data reuse ratio, vectorization, unrolling and thread binding. On the GPU, the performance speedup was between 1.6x to 3.8x, due to the graph and schedule optimizers that were able to fuse kernels and identify operators to optimize. For the convolution study on ResNet-18 and MobileNet, TVM outperformed Tensor Comprehensions (TC) [87], which generated 2000 trials per operator (10 generations x 100 population x 2 random

seeds), whereas in MobileNet, TC performed slightly better than TVM in the deeper layers.

Latte [85] is a domain-specific language (DSL) that takes in a computation graph and targets heterogeneous code generation, parallelization and optimization. *Dataflow analysis* informs the code generator to map shared variables to memory regions accessible to neurons that share dependencies. *Compute analysis* looks for ensembles with identical operations and performs transformations, such as converting array-of-structs to struct-of-arrays. Optimizations include kernel pattern matching, which invokes a high-performance implementation (e.g. Intel MKL), loop tiling, fusion and parallelization. Metadata, provided in loop transformations, can prevent illegal optimizations from taking place, whereas parallelization batches processing to run data independently. Julia AST is used in code generation and applies transformations as specified, which gets offloaded to the Intel C++ compiler. Julia's parallel accelerator consumes a node that indicates the explict parallel `for`-loop, which also contains information about the collapsed loop nests, schedules, chunk sizes, as well as pragmas to inform the compiler to ignore vector dependencies and aliasing. Speedups were achieved over Caffe (5-6x for AlexNet and VGG, 3.2x OverFeat) and Mocha (37.5x AlexNet, 16.2x OverFeat, 41x VGG) when applying individual optimizations, such as parallelization, tiling, and fusion to the neural network models. The numbers were significant for Mocha, which does not use parallelization and does not invoke Intel MKL. The results also showed that 50% additional throughput (img/s) can be achieved by adding a Xeon Phi co-processor.

Inference Engines. TensorFlow Accelerated Linear Algebra (XLA) [98] is a domain-specific compiler for linear algebra that optimizes computations, memory usage and portability on server and mobile platforms. The input language to XLA is a high level optimizer (HLO) IR, which gets compiled into primitive linear algebra operations for various architectures, including x64, ARM64 and NVIDIA GPU. Optimizations include common subexpression elimination, operation fusion and buffer analysis for run time allocation of memory. XLA sends HLO to the backend and performs further target-specific optimizations, such as fusing operators and partitioning computation into streams for GPUs.

NVIDIA TensorRT [83] is an inference engine that includes an inference optimizer and run time that delivers low latency and high-throughput. TensorRT supports ONNX formats, as well as accelerated TensorFlow RunTime and PyTorch, with a conversion to TensorRT format. INT8 and FP16 optimizations are also performed for production deployments of deep learning inference applications, where reduced precision inference significantly reduces application latency. TensorRT also includes an auto-tuning component for performance optimization.

Intel OpenVINO [63] is a lightweight inference engine and model optimizer for convolutional neural networks, with support for ONNX, Caffe, TensorFlow, and MXNet frameworks, that targets Intel architecture backends, such as FPGAs, GPUs and x86. The model optimizer facilitates in deploying trained models and includes static analysis to adjust the model for targeted backends. The inference engine is a C++ library that infers results from input data that is optimized for deployed devices. An API to allows users to read the IR, set the input and output formats and execute the model on targeted backends.

3.3 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.

3.3.1 Convolutional Neural Networks. Convolutional neural networks consist of a convolution layer, a pooling layer, and a softmax layer. 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. 2D input feature maps, or channels, get convolved, resulting in a 3D filter. *Pooling* consists of downsampling, or summarizing the neighboring features. The *softmax* layer is the classifier that finds the maximal possibility over previous layers, with all inputs shifted toward the maximum. Data layouts for CNNs typically include `NCHW` and `CHWN`, amongst others, where each letter represents the feature map or input dimension and its traversal order of the convolution operation. For instance, N is the outermost loop and W is the innermost loop in `NCHW`.

Memory Efficiency for CNNs on GPUs. Considering thread grids and blocks in the GPU execution paradigm, data layout strategies can significantly impact performance and memory efficiency for convolutional neural networks [48]. Caffe and cuDNN uses the `NCHW` layout, whereas cuda-convnet uses the `CHWN` layout. cuDNN 4 provides FFT as an option for the convolution operation, whereas cuda-convnet implements the direct convolution method. The benchmarks evaluated selected layers from LeNet on MNIST and CIFAR10 datasets, and AlexNet, ZFNet, and VGG on ImageNet datasets on an NVIDIA Titan Black and GTX Titan X GPUs. Table 9 summarizes the results of the two data layouts for CNN. When comparing speedups for data layouts, the first set of layers, which were inputs, had $1 \leq C \leq 64$, whereas the second set of layers, which were hidden, had $32 \leq N \leq 64$. When varying C and N, `CHWN` outperformed `NCHW` when $N > 64$. In

Table 9: Comparing memory layouts for CNNs.

	NCHW (Caffe, cuDNN)	CHWN (cuda-convnet)
Speedup	$32 \leq N \leq 64$ (hidden layers)	$C = \{1, 3\}$ (inputs)
Vary N	$N < 128$	$128 \leq N \leq 256$, warps process $N/warp$ images
Vary C	$32 \leq C \leq 512$, collapse HW (cuBLAS)	$C < 16$, fixed performance across
FFT	MM, F large, N large, or many C	Small C
Pooling	Poor, HW in low dimension, works on consecutive memory	Best performer (16.3x)

GPUs, warps of 32 threads were allocated to process 32 images, and increasing N to 128 enabled each thread to process four images, improving reuse and locality. On the other hand, cuBLAS requires a matrix collapse along H and W and exhibited overhead when $C < 32$, but performance gains were seen when $C > 32$. FFT-based implementations require significant memory for padding and storing the intermediate results and will not work if the GPU memory is insufficient. The FFT method performed well for the set of convolution layers that NCHW performed well in the previous study, and better than the matrix multiplication case, notably when F or N were large. For the pooling layers, CHWN outperformed NCHW significantly (16.3x speedup), since NCHW accesses memory in a strided and redundant manner inherent in the pooling operation. Kernel fusion was applied to the softmax layers, which consists of five separate GPU kernel launches for NCHW to ensure data dependencies, which yielded an average of 2.81x speedup.

FFT-based Convolutional Nets. Proposed as an alternate to the zero-padding requirement of cuFFT, fbFFT [86] implements the Cooley-Tukey FFT algorithm [93], which eliminates a full data transpose by returning the kernel in a form where the two innermost data dimensions are transposed. Complex twiddle factors, or roots of unity, are computed to perform butterfly operations by recursively combining smaller DFTs into one large DFT. All threads in a warp load one element and computes a complex twiddle factor, then exchanges data with another thread within the warp in parallel to produce a new value. Next, all threads within the warp exchange twiddle factors with another thread in parallel to produce a new value. These bulk synchronous exchanges can be written with one warp-wide instruction, where input $n \leq 32$ can be implemented with one warp shuffle. When $32 < n \leq 256$, the computation is distributed amongst threads within a warp that manage multiple registers. After $\log_2 n$ steps, the FFT is computed in a bit reversed manner within one register across a warp. The Hermitian symmetry performs half the computation [94], each twiddle factor is distributed by copying from register to register, and a bit reversal, where high-order bits represent the register and low-order bits represent the warp, is implemented in shared memory. Their experiments ran on a NVIDIA Tesla K40m GPU. Results show that fbFFT is 1.5x to 5x faster than cuFFT for various batch sizes. Speedups were generally seen for batch size of 128 and input sizes of 32 and 64. For the 2D case, performance gains were modest, but showed that smaller input sizes (8-32) and smaller batch sizes (32, 128) performed better. In general, the FFT method benefited from larger problem sizes. Performance suffered for small kernels (3x3) due to the overhead of zero-padding and multiple streams for memory management. For larger kernel sizes (13x13), maximum speedup of 23.54x can be attained.

Vectorizing Convolutional Neural Network on SIMD. A JIT-based (just in time) approach toward generating convolutional kernels applies vectorization, cache blocking, and prefetching schemes [24]. Register blocking was applied in the spatial domains of the output tensor, since points in the spatial tensor can be computed independently, whereas cache blocking was applied on the feature map and spatial domain operations, which alleviated memory accesses when activations and weight tensors did not fit in cache. A 2-layer prefetching strategy, with a tunable prefetch distance, consisted of a first layer that fetched data in L1 cache to be used by the same microkernel, and a second layer that fetched data in L2 cache to be used by a future kernel invocation. The feature map dimensions are vectorized with JIT representing a multiple length of VL. Small matrix-vectors are converted into a sequence of small GEMMs with blocking according to the width dimension, where `libxsmm` [29] and Intel MKL are invoked. In the backward pass, the gradient input tensor is computed by convolving the gradient output tensor with the weight tensor. The input gradients in back propagation can be rewritten to match the access patterns of the optimized forward propagation. In the update pass, the gradient weight tensor is computed by convolving the gradient output tensor with the input tensor, where each microkernel invocation computes a $VL \times VL$ sub-tensor of the weight gradient. The experiments ran on a 28-core Intel Xeon 8180 Skylake and a 72-core Xeon Phi 7295 Knights Mill (KNM) processor. For the Skylake architecture, forward propagation on ResNet-50 in GFLOPS nearly matched Intel MKL and speedups of 1.1x-1.2x were gained for selected layers. For spatial dimensions $R = 1$, $S = 1$, the peak operational intensity was 70%, and for $R = 3$, $S = 3$, the operational intensity was $\approx$ 80%. On the KNM architecture, $R = 1$, $S = 1$ achieved 55%, and $R = 3$, $S = 3$ achieved $\approx$ 70%. The differences were due to the hardware design, where the KNM has a higher core peak performance

(192 GFLOP) but lower L2 read and write bandwidth (54 GB read, 27 GB write), whereas the Skylake has lower core peak performance (147 GFLOP), but higher L2 read and write bandwidth (147 GB read, 74 GB write).

3.3.2 Recurrent Neural Networks. Recurrent neural networks (RNN) handle variable length sequences by capturing unbounded context dependencies typical in natural language comprehension and speech recognition systems.

Definition 5. *For inputs x_t and y_t, connection weight matrices $\mathbf{W}_{ih}$, $\mathbf{W}_{hh}$, $\mathbf{W}_{ho}$, indicating input-to-hidden, hidden-to-hidden and hidden-to-output, respectively, and activation function f, the recurrent neural network can be described as follows:*

$$h_t = f_H(\mathbf{W}_{ih}x_t + \mathbf{W}_{hh}h_{t-1})$$
$$y_t = f_O(\mathbf{W}_{ho}h_t)$$

RNNs learn a probability distribution over a sequence by being trained to predict the next symbol in a sequence. The output at each timestep t is the conditional distribution $p(x_t|x_{t-1}, ..., x_1)$. $\mathbf{W}_{hh}h_{t-1}$ is a weight matrix shared over all timesteps that can be unrolled into one two-dimensional matrix, whereas $\mathbf{W}_{ho}h_{t-1}$ involve outputs of each connected neuron to inputs of the current timestep. Released in cuDNN v5 [14], unrolling an RNN combines independent weight operations into one large step [5], which then invokes high-performance libraries, such as cuBLAS [13]. 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, so dependencies need to be considered in order for rolling to work.

Persistent RNN. Current neuron inputs from previous output timesteps, or h_t, is computationally expensive because of the output dependencies and explicit synchronization. Provided that weights fit in register files, efficient partitioning of the GPU memory hierarchy enables persistent access to weights and activations for RNNs [18]. Matrix multiplication is most efficient when the mini-batch size is large (N>64, per GPU), since recurrent weights loaded once from off-chip memory can be reused over each sample in 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 setting. Preemptive multitasking on the GPU was used, where threads attempted to synchronize using a global barrier and, if unsuccessful, eventually timed out and exited. DeepSpeech is a 5-layer RNN model, where the first three layers are non-recurrent and operate on the spectrogram frame, the fourth layer is a bi-directional RNN with two hidden units, and the fifth layer outputs to a softmax layer. Scalability studies of 1-128 GPUs compared cuBLAS and Nervana and varied mini-batch sizes, which showed near linear scaling of 250 TFLOPS on 128 GPUs.

Sparse Persistent RNNs. For cases where RNNs are sparse and cannot fit in registers, representing the RNNs as key-value pairs indicating the location of non-zero elements can further improve performance [104]. When dealing with sparsity, the *operate* phase assigns each thread to n nonzero weights ($n <$ `index, value` $>$ pairs) and executes $\delta+ = v_i * h\_shm[idx[i]]$ computations n times in series. *Wide memory loads* (`ld.shared.v4`) supports 4x the amount of threads loaded (128 vs. 32 threads) and incurs 8, instead of 32, bank conflicts in the worst case. A *bank-aware layout* transform assigns a color to each of n <`index, value`> pairs, with a colored bank indicating an occupied location. *Row-balanced pruning* assumed each row per layer had the same amount of nonzeros, and x% lowest magnitude amount of parameters were pruned. *Lamport timestamps* marked each output value to indicate whether or not the value had been computed, which is an alternative to preemptive multitasking and allows threads to progress individually. The experiments ran on an NVIDIA Volta V100 and performed the English to German translation tasks using the `WMT15` dataset for training and `newstest2013` for the test set using OpenNMT. When varying density from 1% to 30%, layer size from 1152 to 5632, batch size from 1 to 64, and timesteps from 16 to 256, the sparse persistent network outperformed dense GEMM, dense persistent [18], and sparse GEMM in all cases, except when layer size increased to 5632, due to increased pressure on shared memory and a global synchronization.

LightRNN. Performance improvements for RNNs can also be made at the word embedding layer, where a 2-component shared embedding for vocabularies in natural language processing [51] allows a simpler lookup, decreasing storage space by $2\sqrt{|V|}$, with $|V|$ vocabulary size. Large vocabulary embeddings directly impact training convergence, which also leads to low recall in inferencing. For instance, the ClueWeb dataset has a vocabulary of over 10M words, which easily occupies about 40 GB as a word embedding matrix, beyond the current GPU capability (16 GB). 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. Another improvement was a bootstrap word allocation procedure, which learned the best vocabulary embedding for all vocabularies. This procedure trained the input and output embedding vectors until a convergence criterion, by repeatedly fixing the embedding vectors and refining the

word allocation in the table to minimize the loss function over all words. 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.

4 PARALLEL AND DISTRIBUTED APPROACHES

The use of HPC in scaling up ML training provides compute power and dynamic flexibility of scheduling tasks as nodes become available. Performance factors include synchronization, fault tolerant and resiliency. Run time systems for ML typically include a client, a master, and worker(s). A client initiates a session that defines a computation graph to execute. The master receives a session object from the client, who schedules the job over one or more workers and coordinates the execution of the graph. Training at scale also requires modifications to the learning rate schedules, as will be discussed in Section 4.3, to offset the amount of communication frequency of gradient updates across nodes.

Distributed machine learning workflows include the parameter server approach, the dataflow approach, and dynamic run time systems [101]. The *parameter server* approach assigns computation to workers during training, where workers push and pull updates to and from parameter servers. The *dataflow* approach takes a functional programming approach toward state transformations, reducing complexity of model design with larger datasets. *Dynamic run time systems* allow mutable states, where the dataflow graph is changed for performance and flexibility purposes, and scheduled across available compute nodes.

4.1 Parameter Server

Parameter servers maintain weights on a collection of clusters, where workers are a collection of nodes that train on a local model replica that communicate weight updates to the parameter server. The parameter server maintains the weights as a distributed table with parameters within the cells. There is no communication amongst workers. FIgure 3, left, shows the parameter server architecture for downpour SGD [16].

DistBelief [16] proposes distributed model parallelism methods for parameter server training, based on downpour SGD and Sandblaster L-BFGS. Downpour SGD partitions the data into shards and individual workers train and keep a copy of the model, while updating the parameter server which synchronizes all models from the nodes. Sandblaster L-BFGS distributes the storage of parameters across nodes, which caches the gradients and coordinates with the parameter server via small messages, alleviating communication bottlenecks with the parameter server and enables training of larger and deeper models. The issues associated with downpour SGD include node failures, stale gradients, and inconsistent timestamps of gradients, since the gradients may be pushed in a different order. In practice, the authors found that relaxing these consistencies did not affect overall performance. Scalability studies show that Downpour SGD with 200 workers with Adagrad had the best accuracy, and Sandblaster L-BFGS with 2000 workers had similar performance. Both approaches were compared to a single GPU and a naive SGD implementation. When scaling the number of workers, Downpour SGD had the best performance but was limited to under 2000 cores, due to the inherent parameter server design, whereas Sandblaster L-BFGS was able to scale beyond 10000 cores, but took longer to achieve the same 16% accuracy.

Enhancements to the parameter server approach included key-value pairs, range-based push-pull updates, and vector clocks for efficient communication, scalability and fault tolerance [50]. The parameter server architecture uses key-value vectors as data structures, which induces sparsity and enables highly optimized linear algebra libraries to be invoked. Push-and-pull operations send data between nodes, whereas range-based push-pull groups updates, alleviating network bandwidth efficiency. Servers store parameters as key-value pairs using consistent hashing. Entries are replicated for fault tolerant and compressed on both data and range-based vector clocks. Each key-value pair is associated with a vector clock, which tracks aggregation status. Any modifications on the master node is copied with its timestamp to the slave nodes, where modifications are pushed synchronously to slaves. Sparse logistic regression was evaluated on 1000 machines, each with 16 cores, 192 GB DRAM, connected with 10 GB ethernet. L-BFGS and delayed block proximal gradient was compared with their approach, which was a parameter server running delayed block proximal gradient with KKT filter. The Krush-Kuhn Tucker (KKT) filter estimates the global gradient based on local information and enables a bounded delay, which requires more computation but minimal waiting, since the constraints are relaxed and updates can be made asynchronously. In general, the overall time spent computing and waiting for the logistic regression was decreased, when compared with the other two approaches. For evaluating LDA, which learned topics over a 500M dataset where each topic represents a user interest, 4x speedup in convergence was observed when increasing the number of machines from 1000 to 6000.

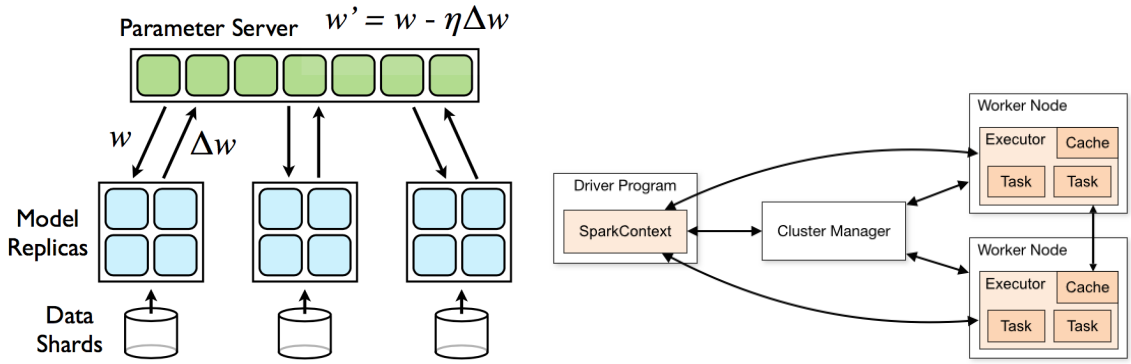

Figure 3: Left: Downpour SGD parameter server (image source [16]). Right: Spark cluster (image source [4])

4.2 Distributed Dataflow Programming

Apache Spark provides a framework that distributes a dataflow graph across all compute nodes, where frequently used data is cached in memory, with support for checkpointing and fault tolerance [4]. Resilient distributed datasets (RDD) are data structures of records, which are distributed across compute nodes for replication, checkpointing and fault tolerance. An RDD can be *transformed* with parallel primitives, such as map and join. A job, such as word count, gets triggered with an *action*. A client driver accepts requests from the master for submitted jobs, which is distributed amongst available workers. RDD employ lazy evaluation, where a transform is performed after the dataflow graph is constructed. Figure 3, right, shows the Spark execution environment [4]. The design of Spark with one driver becomes a bottleneck when scaling the problem to millions of parameters, where updates need to be communicated across workers.

SystemML is a declarative machine learning (DML) framework built on top of Spark that provides a MATLAB-like programming environment for linear algebra and numerical methods, and support for distributed training of machine learning [8]. The *optimizer integration* includes Spark-specific rewrites, memory budgets and constraints, operator selection, and extended parallel `for` (ParFor) loop. Spark-specific rewrites perform caching and checkpoint injection, which allows RDDs to persist at various storage levels, whereas checkpoints are inserted after persistent reads to prevent repeated lazy evaluation. Memory budgets and constraints perform an upper bound for memory requirements, based on the driver memory, executor memory, number of executors, data fraction, and shuffle fraction. Operator selection seeks patterns in linear algebra, where fusing physical operators would yield performance. ParFor was extended with three physical operators that consists of a local executor for multithreads, a remote executor for a Spark job, and a remote disjoint executor that partitioned the matrices. The *run time integration* includes distributed matrix representation, which partitions both dense and sparse matrices, based on its compressed storage format, and buffer pool integration, which manages all reads and writes of matrix objects and RDDs with lineage tracking. When evaluated on a 6 node cluster with 2x6 Intel E5-2440 CPU on various ML algorithms (K-means, alternating least squares, naïve Bayes), MapReduce performed better on smaller datasets, whereas SystemML outperformed for bigger sized problems (120 GB).

To fit large datasets in memory, SystemML compresses the column groups of matrices and performs linear algebra operations over the compressed representations [20, 21]. Various input formats, such as CSR, are supported and internally converted to tiles and blocks for matrix multiplication. The ratio of column cardinality (# distinct values) to the number of rows quantifies redundancy, independent of the actual values. Column co-coding partitions column groups, such that columns within each group are highly correlated and co-coded as a single unit. The column encoding formats include offset-list encoding (OLE) and run-length encoding (RLE). OLE divides the offset range into segments of fixed length (e.g. two bytes per offset), where each offset is mapped to its corresponding segment and encoded as the difference between the current and the beginning segment. In RLE, a sorted list of offsets are encoded as a sequence of runs for starting offset and run length. The compressed approach was compared with uncompressed linear algebra and Snappy on MNIST and ImageNet, and showed that the compressed approach outperformed both when data did not fit in memory, whereas the uncompressed approach performed well when data was able to fit in memory. This is attributed to the overhead of the column encoding format, which can be mitigated with larger problem sizes.

4.3 Dynamic Run Time Systems

Factors to consider when training on large scale systems include resource allocation, control flow and synchronization. This becomes challenging for dynamic neural networks, which allow individual cells and architectures to change as training progresses. Examples include variable-length neural networks, tree-structured neural networks, and graph neural networks. Frameworks that support static neural networks include CNTK, Theano and TensorFlow, whereas examples that support dynamic neural networks include Chainer [1], DyNet, and PyTorch. Static neural networks are based on the "define-and-run" scheme, where a network is fixed and mini-batches are fed in as input, whereas dynamic neural networks take the "define-by-run" scheme, where the network is defined dynamically as the forward computation takes place. Due to the overhead of creating dataflow graphs, dynamic neural networks are slower, but the flexibility of having networks that change makes learning certain types of tasks more achievable. Techniques to mitigate the overhead of dynamic graph creation include batching similar or independent operations and adding constructs for distributed model training.

Constructs. TensorFlow Fold [53] is a dynamic batching technique that supports variable length neural networks, where each graph is defined statically and generated dynamically as the program is being executed. A *depth-based* batching approach batches nodes with identical depth and signature, where nodes in the same depth have no dependencies and signatures identify nodes with similar or independent operations. Source nodes are constant tensors, whereas non-source nodes are operations, such as convolution and ReLU. `concat` and `gather` operations are inserted during the dataflow graph generation process as necessary for synchronization purposes. A combinator library, similar to LIFT-IL (Table 7), enables execution of parallel patterns on distributed systems. The dynamic batching technique was compared with manual batching on both an Intel Xeon 8-core CPU and a NVIDIA GTX 1080 GPU on the Tree-LSTM benchmark, where the tree size was 128 and LSTM state size was 1024. The dynamic batch size performed well for small batch sizes up to 128, but the manual technique worked better for larger batch sizes. Several reasons, which were not stated, could be that larger batch sizes may have induced compute bound behavior, e.g. convolution, and that dynamic batching may have adversely affected performance, due to its aggressiveness in grouping together operations.

Distributed Control Flow. To enable distributed execution of neural networks, the dataflow graph is partitioned across multiple nodes, and control flow primitives, such as `switch`, `merge`, `enter`, `exit`, and `nextIteration` are embedded in the dataflow graph. Edges are replaced with `send` and `receive` communication operations that share a key. Since there is no synchronization with devices, an `is_dead` signal communicates to other devices of an untaken branch at a receive node. A stack architecture overlaps compute and memory operations that pushes gradients and input operations, and popped off and copied to host and device for gradient computation and automatic differentiation. Their experiments ran on a cluster with NVIDIA K40 GPUs and a NVIDIA DGX cluster. Their memory management scheme, which evaluated a single-layer LSTM with 512 units, was able to train a sequence length beyond 1000 in the same amount of time as a sequence length of 100, whereas disabling memory swapping ran out of memory beyond a 500-length sequence. Dynamic RNN was also able to handle a single layer of LSTM model with batch size 256 and sequence length 256, whereas static unrolling ran out of memory at length 128.

Operation Batching. An *agenda-based* batching approach in the DyNet framework orders the dataflow graph execution, which adds a signature to nodes with the same operations and maintains a count of unresolved dependencies for available nodes [57]. Nodes with no incoming inputs are added during the initialization process. While nodes remain to be processed, available nodes with the same signature are grouped into a single batch operation. As nodes are removed from the agenda, the dependency counter of successors is decremented and nodes with zero dependencies are added. The process is repeated until no nodes are left. Padding and masking operations are required for variable length neural networks, which incurs overhead. Manual batching aggregates inputs that share the same timestep, whereas operation batching aggregates nodes that can be executed in batches, such as operations with the same names (e.g. tanh, softmax) or matrices within a dimension size and range. The experiments compared with TensorFlow Fold (depth-based batching) and evaluated a bi-directional LSTM sequence labeler on synthetic data on a Tesla K80 GPU and Intel Xeon 2.30GHz E5-2686v4 CPU. Results show that the agenda-based approach without manual batching resulted in a 11-fold increase and was faster than the depth-based approach by 15-30%. However, agenda-based with manual batching experienced slowdown, due to the overhead of dataflow graph construction and batch scheduling. When evaluated on Tree-LSTM and the sequence labeler, the agenda-based approach with operation batching was able to process significantly more sentences per second, 3.6x-9.2x on the CPU and 2.7x-8.6x on the GPU, compared to the manual batching approach.

Asynchronous Model-Parallel Training. AMPNet [23] distributes the computation graph across nodes on multicore CPUs for dynamic neural network training, where updates

are communicated via message passing. A dynamic controller executes the static IR graph by issuing operations and gathering intermediate results. Each message consists of a payload and a state, where a *payload* is a tensor and a *state* is model-specific that keeps track of algorithm and control flow information. The final loss layer initializes backpropagation through the IR graph. `max_active_keys` represents the maximum number of active instances in flight. A condition node 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 received from each of its ancestor nodes and records the origin of backpropagation to the correct origin. Aggregation and disaggregation combinators, similar to TensorFlow Fold, provide constructs, such as group, split and broadcast, that are carried out across incoming messages. The frequency of updates can be set with `min_update_frequency`, which trades off staleness of the gradient with frequency of sending messages. The authors compared their approach with TensorFlow Fold on a 4-layer perceptron on MNIST dataset and Tree-LSTM on the Stanford Sentiment Treebank dataset. The performance of AMPNet was slower for the MNIST experiment (44s, vs 34.5s) than TensorFlow. Asynchrony was tested on an 8-replica RNN model on the list reduction dataset and showed that `min_update_frequency` cannot be too large or too small, and that increasing `max_active_keys` increased performance up to the number of affinitized heavy operations (8 replicas), but suffered beyond that.

Scaled Up Approaches. When training neural networks at scale, the hyperparameters get altered due to delays from synchronization across all nodes. Updates to the learning rate schedules compensate for scalability and communication of gradients. Table 10 compares different approaches for scaling ImageNet, which was taken from [2] for reference.

Using 256 GPUs, the authors reduced the training time for ResNet-50 to 1 hour [26], while achieving near linear speedups when scaling from 8 to 256 GPUs (0.9x). A large mini-batch size of 8,192 images was used, which fully exploited the GPUs and made training run faster. Using large mini-batches introduces noise that may impact gradient updates, which slows convergence or may converge to a non-optimal solution. Additionally, multiple GPUs require synchronization of weights 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. 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. Their approach was evaluated on 8 NVIDIA Pascal P100 GPUs interconnected with NVLink, comparable to NVIDIA DGX machines. For communication, NCCL [56] was used for buffers of size 256 KB or more. 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.

The ResNet-50 training time was reduced to 31 minutes [100] with a layer-wise adaptive rate scaling (LARS) algorithm that calculated the learning rate for each layer, based on available compute resources. As discussed in [26], the challenges of large batch scaling can be addressed with linear scaling of the learning rate η and a warm up scheme for earlier phases of training. In LARS, different layers may have different η. LARS first gets the local η for each learnable parameter, then gets η for each layer. The gradients, acceleration term, and the weights are updated. Their approach was evaluated on Intel Xeon Platinum 8160 processors (Stampede2 supercomputer), which consists of 4,200 KNL nodes, where each KNL node is a 68-core processor. With LARS, along with the warmup technique, the authors were able to scale up the batch sizes to 32,000 as the baseline. 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, in addition to the KNLs ability to divide on-chip memory for increased memory bandwidth.

To follow up, the authors in [2] devised an RMSprop warm-up scheme, batch normalization without moving averages, and a slow-start learning rate schedule for ResNet-50. The warmup scheme first trained using RMSProp, then switched to SGD to address the difficulty of initial training. Their approach was evaluated on 1024 P100 GPUs with a mini-batch size of 32,768 using Chainer with NCCL and OpenMPI. Although their Top-1 accuracy was lower (74.9%), the training time for ResNet-50 was drastically reduced to 15 minutes.

5 SUMMARY AND DIRECTIONS

The behavior of learning algorithms in single node and multi-node settings, and inherent tradeoffs associated with model design and code transformation options, presents a complex, multi-modal landscape for exploring suboptimal state space solutions. The problem space was dissected, in terms of iteration spaces, code transformations, data operations, and performance modeling. Compilation techniques for improving execution performance, both at the machine-independent and architecture-specific level, were covered. Distributed

Table 10: ResNet-50 for ImageNet. Table source [2].

Team	Hardware	Software	Minibatch	Time	Top-1
He, et al. [28]	Tesla P100 x 8	Caffe	256	29 hr	75.3%
Goyal, et al. [26]	Tesla P100 x 256	Caffe2	8,192	1 hr	76.3%
You, et al. [100]	Xeon 8160 x 2048	Intel Caffe	32,000	20 min	75.4%
Akiba, et al. [2]	Tesla P100 x 1024	Chainer	32,768	15 min	74.9%

methods for scaling up model training were presented afterward.

Opportunities for improved model training exist at the algorithmic and systems level, where a guided approach that incorporates algorithmic meta-information can inform decision making during code generation. Latte [85] propagates properties of DL routines to the back end, but is currently tied to the Intel compiler. Abstractions, such as Lift [77], provide building blocks for generating efficient ML code, but a unified approach is needed that accounts for the computation requirements of model training. Variable-length precision operations are another way to increase instruction throughput, at the cost of accuracy degradation and data reorganization overhead. Performance measurement and analysis of DL applications is another direction, since current tools lack the ability to tie a model description with system-level execution bottlenecks that may surface. MLPerf [55] is a benchmark suite for measuring perfomance for ML software frameworks, but more proxy applications [12] will be needed to continue the hardware/software co-design process.

Given that similar patterns exist in scientific codes, pattern matching for code optimization [52] could possibly reduce the search space complexity. Prior knowledge on how transformations work on particular patterns could explain why certain transformation patterns lead to performance gains, or never succeeds. This can avoid any unnecessary experimentation and analysis, providing hints to the compiler in making informed decisions.

Quantifying the error bounds of model accuracy also needs further investigation, not only for evaluation purposes, but for identifying blind spots that may exist in classification. For instance, adversarial machine learning provides a secure and robustness measure against attacks [25], critical in areas such as self-driving cars, where a tainted stop sign can delude a model into making wrong, perhaps unsafe decisions. Scaled up approaches have been proposed [45], but as attacks evolve, the process of how attacks are constructed will need to be understood. Error propagation of fault injection attacks can also aid in understanding confidence intervals, as well as error regions where decisions may go haywire, which can reveal vulnerabilities and provide robustness for learning systems.

REFERENCES

[1] Takuya Akiba, Keisuke Fukuda, and Shuji Suzuki. 2017. ChainerMN: Scalable Distributed Deep Learning Framework. In *Proceedings of Workshop on ML Systems in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*.

[2] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. 2017. Extremely large minibatch SGD: training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325* (2017).

[3] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M Kakade, and Matus Telgarsky. 2014. Tensor Decompositions for Learning Latent Variable Models. *The Journal of Machine Learning Research* 15, 1 (2014), 2773–2832.

[4] Apache Spark 2019. Apache Spark. (2019). https://spark.apache.org.

[5] Jeremy Appleyard, Tomas Kocisky, and Phil Blunsom. 2016. Optimizing Performance of Recurrent Neural Networks on GPUs. In *GPU Technology Conference*. NVIDIA.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[7] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*. 2546–2554.

[8] Matthias Boehm, Michael W Dusenberry, Deron Eriksson, Alexandre V Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R Reiss, Prithviraj Sen, Arvind C Surve, et al. 2016. SystemML: Declarative Machine Learning on Spark. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1425–1436.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Megan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems (NIPS)*.

[11] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *ACM on Programming Languages* 2, OOPSLA (2018), 123:1–123:30.

[12] Coral-2 Benchmarks 2019. CORAL-2 Benchmarks. (2019). https://asc.llnl.gov/coral-2-benchmarks.

[13] cuBLAS 2019. NVIDIA cuBLAS. (2019). https://developer.nvidia.com/cublas.

[14] cuDNN 2019. NVIDIA cuDNN. (2019). https://developer.nvidia.com/cudnn.

[15] CUTLASS 2019. CUTLASS: Fast Linear Algebra in CUDA C++. (2019). https://github.com/NVIDIA/cutlass.

[16] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al.

2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems (NIPS)*. 1223–1231.

[17] Deep Learning 2019. Comparison of Deep Learning Software. (2019). https://en.wikipedia.org/wiki/Comparison_of_deep-learning_software.

[18] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent RNNs: Stashing Recurrent Weights On-Chip. In *International Conference on Machine Learning*. ICML, 2024–2033.

[19] Eigen 2019. Eigen. (2019). http://eigen.tuxfamily.org.

[20] Ahmed Elgohary, Matthias Boehm, Peter J Haas, Frederick R Reiss, and Berthold Reinwald. 2016. Compressed Linear Algebra for Large-Scale Machine Learning. *Proceedings of the VLDB Endowment* 9, 12 (2016), 960–971.

[21] Ahmed Elgohary, Matthias Boehm, Peter J Haas, Frederick R Reiss, and Berthold Reinwald. 2017. Scaling Machine Learning via Compressed Linear Algebra. *ACM SIGMOD Record* 46, 1 (2017), 42–49.

[22] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774* (2018).

[23] Alex Gaunt, Matthew Johnson, Maik Riechert, Daniel Tarlow, Ryota Tomioka, Dimitrios Vytiniotis, and Sam Webster. 2017. AMPNet: Asynchronous Model-Parallel Training for Dynamic Neural Networks. *arXiv preprint arXiv:1705.09786* (2017).

[24] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE.

[25] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[26] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677* (2017).

[27] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly: Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 04 (2012), 1250010.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[29] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 84.

[30] M Shahriar Hossain, Patrick Butler, Arnold P Boedihardjo, and Naren Ramakrishnan. 2012. Storytelling in Entity Networks to Support Intelligence Analysts. In *International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. ACM, 1375–1383.

[31] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[32] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[33] ImageNet Challenge 2012. ImageNet Large Scale Visual Recognition Challenge. (2012). http://image-net.org/challenges/LSVRC.

[34] Intel CPU 2019. WikiChip Intel Processors. (2019). https://en.wikichip.org/wiki/intel.

[35] Intel MKL 2019. Intel Math Kernel Library. (2019). https://software.intel.com/en-us/mkl.

[36] James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann.

[37] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. 2018. A Domain-Specific Architecture for Deep Neural Networks. *Commun. ACM* 61, 9 (Aug. 2018), 50–59. https://doi.org/10.1145/3154484

[38] Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Grundkiewicz, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, et al. 2018. Marian: Fast Neural Machine Translation in C++. *arXiv preprint arXiv:1804.00344* (2018).

[39] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2016. Compression of deep convolutional neural networks for fast and low power mobile applications. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[40] Fredrik Kjolstad, Peter Aherns, Shoabib Kamil, and Saman Amarsinghe. 2019. Sparse Tensor Algebra Optimizations with Workspaces. In *International Symposium on Code Generation and Optimization (CGO)*. ACM.

[41] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *ACM on Programing Languages* 1, OOPSLA (2017), 77:1–77:29.

[42] Philipp Koehn. 2017. Neural machine translation. *arXiv preprint arXiv:1709.07809* (2017).

[43] Tamara G Kolda and Brett W Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (2009), 455–500.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*. 1097–1105.

[45] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2017. Adversarial Machine Learning at Scale. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[46] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. 2015. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[47] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[48] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. 2016. Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 633–644.

[49] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560* (2016).

[50] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Berkeley, CA, USA, 583–598. http://dl.acm.org/citation.cfm?id=2685048.2685095

[51] Xiang Li, Tao Qin, Jian Yang, and Tie-Yan Liu. 2016. LightRNN: Memory and Computation-Efficient Recurrent Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*. 4385–4393.

[52] Robert Lim, Boyana Norris, and Allen Malony. 2018. A Similarity Measure for GPU Kernel Subgraph Matching. In *31st International Workshop on Languages and Compilers for Parallel Computing (LCPC)*.

[53] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. 2017. Deep Learning with Dynamic Computation Graphs. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[54] MAGMA 2019. MAGMA: Matrix Algebra on GPU and Multicore Architectures. (2019). https://icl.cs.utk.edu/magma.

[55] MLPerf 2019. MLPerf Benchmark Suite. (2019). https://mlperf.org.

[56] nccl 2019. NVIDIA Collective Communications Library (NCCL). (2019). https://developer.nvidia.com/nccl.

[57] Graham Neubig, Yoav Goldberg, and Chris Dyer. 2017. On-the-Fly Operation Batching in Dynamic Computation Graphs. In *Advances in Neural Information Processing Systems (NIPS)*. 3974–3984.

[58] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. 2016. A Review of Relational Machine Learning for Knowledge Graphs. *Proc. IEEE* 104, 1 (2016), 11–33.

[59] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. 2017. Can FPGAs beat GPUs in accelerating next-generation deep neural networks?. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 5–14.

[60] William F Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Minimizing the Cost of Iterative Compilation with Active Learning. In *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 245–256.

[61] ONNX 2019. ONNX: Open Neural Network Exchange Format. (2019). https://onnx.ai.

[62] OpenBLAS 2019. OpenBLAS: An optimized BLAS library. (2019). https://www.openblas.net.

[63] OpenVINO 2019. Intel OpenVINO Toolkit. (2019). https://software.intel.com/openvino-toolkit.

[64] Par4All 2019. Par4All An Automatic Parallelizing and Optimizing Compiler for C and Fortran Sequential Programs. (2019). http://par4all.github.io.

[65] Hang Qi, Evan R Sparks, and Ameet Talwalkar. 2017. Paleo: A Performance Model for Deep Neural Networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[66] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[67] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 4780–4789.

[68] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, et al. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *arXiv preprint arXiv:1805.00907* (2018).

[69] Rupp, Karl 2013. CPU, GPU, MIC Hardware Characteristics over Time. (2013). https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/.

[70] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4510–4520.

[71] Shaohuai Shi and Xiaowen Chu. 2018. Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs. In *16th International Conference on Dependable, Autonomic and Secure Computing*. IEEE, 949–957.

[72] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[73] Shaden Smith, Niranjay Ravindran, Nicholas D Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *International Conference on Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 61–70.

[74] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*. 2951–2959.

[75] Daniele G Spampinato and Markus Püschel. 2014. A Basic Linear Algebra Compiler. In *International Symposium on Code Generation and Optimization (CGO)*. ACM, 23.

[76] Daniele G Spampinato and Markus Püschel. 2016. A Basic Linear Algebra Compiler for Structured Matrices. In *International Symposium on Code Generation and Optimization (CGO)*. ACM, 117–127.

[77] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE Press, 74–85.

[78] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 99 (2018), 1–15.

[79] Xing Su, Xiangke Liao, and Jingling Xue. 2017. Automatic Generation of Fast BLAS3-GEMM: A Portable Compiler Approach. In *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 122–133.

[80] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[81] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[82] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[83] TensorRT 2019. NVIDIA TensorRT Programmable Inference Accelerator. (2019). https://developer.nvidia.com/tensorrt.

[84] Torchvision Models 2019. Torchvision Models. (2019). https://pytorch.org/docs/stable/torchvision/models.html.

[85] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. 2016. Latte: A Language, Compiler, and Runtime for Elegant and Efficient Deep Neural Networks. *ACM SIGPLAN Notices* 51, 6 (2016), 209–223.

[86] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2015. Fast Convolutional Nets with fbfft: A GPU Performance Evaluation. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[87] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[88] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[89] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 521–532.

[90] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *International Congress on Mathematical Software*. Springer, 299–302.

[91] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 54.

[92] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral Extraction Tool. In *International Workshop on Polyhedral Compilation Techniques*.

[93] Wikipedia 2019. Cooley-Tukey FFT Algorithm. (2019). https://en.wikipedia.org/wiki/CooleyâĂŞTukey_FFT_algorithm.

[94] Wikipedia 2019. Hermitian Function. (2019). https://en.wikipedia.org/wiki/Hermitian_function.

[95] Wikipedia 2019. High Bandwidth Memory. (2019). https://en.wikipedia.org/wiki/High_Bandwidth_Memory.

[96] Wikipedia 2019. List of NVIDIA Graphic Processing Units. (2019). https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units.

[97] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. 2016. gpucc: An Open-Source GPGPU Compiler. In *International Symposium on Code Generation and Optimization (CGO)*. ACM, 105–116.

[98] XLA 2018. TensorFlow Accelerated Linear Algebra (XLA). (2018). https://www.tensorflow.org/performance/xla.

[99] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. 2015. Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. ACM, 1355–1364.

[100] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. ImageNet Training in Minutes. In *International Conference on Parallel Processing (ICPP)*. ACM, 1.

[101] Kuo Zhang, Salem Alqahtani, and Murat Demirbas. 2017. A comparison of distributed machine learning platforms. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–9.

[102] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6848–6856.

[103] Hao Zhou and Jingling Xue. 2016. Exploiting Mixed SIMD Parallelism by Reducing Data Reorganization Overhead. In *International Symposium on Code Generation and Optimization (CGO)*. ACM, 59–69.

[104] Feiwen Zhu, Jeff Pool, Michael Andersch, Jeremy Appleyard, and Fung Xie. 2018. Sparse Persistent RNNs: Squeezing Large Recurrent Networks On-Chip. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[105] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).