Designing Computer Systems for Software 2.0

Kunle Olukotun
Stanford University
SambaNova Systems

NeurIPS Invited Lecture, December 6, 2018
Two Big Trends in Computing

- **Success of Machine Learning**
  - Incredible advances in image recognition, natural language processing, and knowledge base creation
  - Society-scale impact: autonomous vehicles, scientific discovery, and personalized medicine
  - Insatiable computing demands for training and inference

- **Moore’s Law is slowing down**
  - Dennard scaling is dead
  - Computation is now limited by power
  - Conventional computer systems (CPU) stagnate

Demands a new approach to designing computer systems for ML
The Rise of Machine Learning

Accuracy

Data size, model complexity

1980s

Today

More Compute

Neural networks

Conventional algorithms

Adapted from Jeff Dean
HotChips 2017
Software 1.0 vs Software 2.0

- Written in code (C++, ...)
- Requires domain expertise
  1. Decompose the problem
  2. Design algorithms
  3. Compose into a system
- Written in the weights of a neural network model by optimization

Andrej Karpathy
Scaled ML 2018 talk
Software 2.0 is Eating Software 1.0

Easier to build and deploy
- Build products faster
- Predictable runtimes and memory use: easier qualification

1000x Productivity: Google shrinks language translation code from 500k LoC to 500

Classical problems
- Data cleaning (Holoclean.io)
- Self-driving DBMS (Peloton)
- Self-driving networks (Pensieve)

Software 2.0: Programming is Changing

ML developers increasingly program Software 2.0 stacks by creating and engineering training data.

snorkel.stanford.edu
# Run mini-batch SGD
for epoch in range(n_epochs):
    for batch in range(0, n, batch_size):
        # Load training data from DB
        X_train, Y_train = load_data(
            offset=batch,
            limit=batch_size
        )

        # Augment training data
        X_train = augment(X_train)

        # Take *sparse* gradient step
        loss.backward()
**Sparsity** is becoming a design objective for neural networks of all types...

Sparsely connected network layers can maintain performance while reducing parameter number.


* Figure from Mocanu et al., 2018
Graph Neural Networks (GNNs) are increasingly popular for network-structured data.

Techniques like neural message passing algorithms leverage sparse graph structure and data access patterns.

* Figure from https://tkipf.github.io/graph-convolutional-networks/
Increasing Model Complexity

2015 - Microsoft ResNet
Superhuman Image Recognition

2016 - Baidu Deep Speech 2
Superhuman Voice Recognition

2017 - Google Neural Machine Translation
Near Human Language Translation

Source: Bill Dally, Scaled ML 2018
ML Training is Limited by Computation

From EE Times – September 27, 2016

“Today the job of training machine learning models is limited by compute, if we had faster processors we’d run bigger models...in practice we train on a reasonable subset of data that can finish in a matter of months. We could use improvements of several orders of magnitude – 100x or greater.”

Greg Diamos, Senior Researcher, SVAIL, Baidu
Microprocessor Trends

- Moores Law
- Multicore research
- Power wall

Graph showing trends in transistors (thousands), single-thread performance (SpecINT x 10^3), frequency (MHz), typical power (Watts), and number of logical cores over the years 1970 to 2020. Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. New plot and data collected for 2010-2017 by K. Rupp.
Power and Performance

\[ Power = \frac{Ops}{\text{second}} \times \frac{\text{Joules}}{\text{Op}} \]

FIXED

Specialization (fixed function) ⇒ better energy efficiency
Key Questions

- How do we speed up machine learning by 100x?
  - Moore’s law slow down and power wall
  - >100x improvement in performance/watt
  - Enable new ML applications and capabilities

- How do we balance performance and programmability?
  - Fixed-function ASIC-like performance/Watt
  - Processor-like flexibility

- Need a “full-stack” integrated solution
  1. ML Algorithms
  2. Domain Specific Languages and Compilers
  3. Hardware
ML Algorithms
Computational Models

- **Software 1.0 model**
  - Deterministic computations with algorithms
  - Computation must be correct for debugging

- **Software 2.0 model**
  - Probabilistic machine-learned models trained from data
  - Computation only has to be statistically correct

- Creates many opportunities for improved performance
SGD: The Key Algorithm in Machine Learning

Optimization Problem:
\[
\min_x \sum_{i=1}^{N} f(x, y_i)
\]

Solving large-scale problems:
Stochastic Gradient Descent (SGD)

\[
x^{k+1} = x^k - \alpha N \nabla f(x^k, y_j)
\]

Select one term, \( j \), and estimate gradient

Billions of tiny sequential iterations

E.g.: Classification, Recommendation, Deep Learning
**SGD: Two Kinds of Efficiency**

- **Statistical efficiency**: how many iterations do we need to get the desired accuracy level?
  - Depends on the problem and implementation

- **Hardware efficiency**: how long it takes to run each iteration?
  - Depends on the hardware and implementation

---

trade off hardware and statistical efficiency to maximize performance

Ce Zhang and Christopher Ré. DimmWitted: Proc. VLDB `14
Low Precision: The Pros

- Energy
- Memory
- Throughput

Google TPU
Intel CPU
Microsoft Brainwave (FPGA)
Low Precision: The Con

Low precision works for inference (e.g. TPU, Brainwave)

Training usually requires at least 16 bit floating point numbers
High Accuracy Low Precision (HALP) SGD

- The gradients get smaller as we approach the optimum
- Dynamically rescale the fixed-point representation (in higher precision)
- Get **less error** with the **same number of bits**

Chris De Sa | Chris Aberger | Megan Leszczynski | Jian Zhang | Alana Marzoev | Kunle Olukotun | Chris Ré
HALP Training

HALP provably converges at a linear rate
CNN: HALP versus Full-Precision Algorithms

HALP has better statistical efficiency than SGD!

14-layer ResNet on CIFAR10

- Training Loss
  - sgd (32)
  - halp (8)

- Validation Accuracy
  - sgd (32)
  - halp (8)
Relax, It’s Only Machine Learning

- Relax precision: small integers are better
  - HALP [De Sa, Aberger, et. al.]
- Relax synchronization: data races are better
  - HogWild! [De Sa, Olukotun, Ré: ICML 2016, ICML Best Paper]
- Relax cache coherence: incoherence is better
  - [De Sa, Feldman, Ré, Olukotun: ISCA 2017]
- Relax communication: sparse communication is better
  - [Lin, Han et. al.: ICLR 18]

Better hardware efficiency with negligible impact on statistical efficiency
Domain Specific Languages and Compilers
Domain Specific Languages

- Domain Specific Languages (DSLs)
  - Programming language with restricted expressiveness for a particular domain (operators and data types)
  - High-level, usually declarative, and deterministic
  - Focused on productivity not usually performance
  - High-performance DSLs (e.g. OptiML) ➔ performance and productivity

- OpenGL®
- MATLAB
- SQL
**K-means Clustering in OptiML**

```
untilConverged(kMeans, tol) { kMeans =>
  val clusters = samples.groupRowsBy { sample =>
    kMeans.mapRows(mean => dist(sample, mean)).minIndex
  }
  val newKmeans = clusters.map(e => e.sum / e.length)
  newKmeans
}
```

- No explicit parallelism
- No distributed data structures (e.g. RDDs)
- Efficient multicore, GPU and cluster execution

**K-means Clustering in TensorFlow**

```
points = tf.constant(np.random.uniform(0, 10, (points_n, 2)))
centroids = tf.Variable(tf.slice(tf.random_shuffle(points), [0, 0], [clusters_n, -1]))

points_expanded = tf.expand_dims(points, 0)
centroids_expanded = tf.expand_dims(centroids, 1)

distances = tf.reduce_sum(tf.square(tf.sub(points_expanded, centroids_expanded)), 2)
assignments = tf.argmin(distances, 0)

means = []
for c in xrange(clusters_n):
    means.append(tf.reduce_mean(tf.gather(points, tf.reshape(tf.where(tf.equal(assignments, c)), [1, -1])), reduction_indices=[1]))

new_centroids = tf.concat(0, means)

update_centroids = tf.assign(centroids, new_centroids)
```

- Calculate distances to current means
- Assign each sample to the closest mean
- Move each cluster centroid to the mean of the points assigned to it

Open, standard software for general machine learning

Great for Deep Learning in particular

First released Nov 2015
Compiler Architecture

- Build a full compiler stack to compile high level DSLs to accelerator hardware
Parallel Patterns

- Most data analytic computations including ML can be expressed as functional data parallel patterns on collections (e.g. sets, arrays, tables, n-d matrices)

- Looping abstractions with extra information about parallelism and access patterns

<table>
<thead>
<tr>
<th>Map</th>
<th>Zip</th>
<th>Reduce</th>
<th>FlatMap</th>
<th>GroupBy</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Map Diagram" /></td>
<td><img src="image2" alt="Zip Diagram" /></td>
<td><img src="image3" alt="Reduce Diagram" /></td>
<td><img src="image4" alt="FlatMap Diagram" /></td>
<td><img src="image5" alt="GroupBy Diagram" /></td>
</tr>
</tbody>
</table>

- **Map**
  - element-wise function $f$

- **Zip**
  - element-wise function $f$ (multi-collection)

- **Reduce**
  - combine all elements with $f$ ($f$ is associative)

- **FlatMap**
  - element-wise function $\geq 0$ values out per element

- **GroupBy**
  - group elements into buckets based on key

- $y = \text{vector} + 4$
- $y = \text{vector} \times 10$
- $y = \text{sigmoid} (\text{vector})$
- $y = \text{vecA} + \text{vecB}$
- $y = \text{vecA} / \text{vecB}$
- $y = \max (\text{vecA}, \text{vecB})$
- $y = \text{vector}.\text{sum}$
- $y = \text{vector}.\text{product}$
- $y = \max (\text{vector})$
- SELECT * FROM vector WHERE elem < 5
- vector.groupBy{e => e % 3}
Example application: \(k\)-means

A data-parallel language that supports nested parallel patterns \(\{\{\}\}\)

Hierarchical dataflow graph of parallel patterns

```scala
val clusters = samples GroupBy { sample =>
  val dists = kMeans Map { mean =>
    mean.Zip(sample) { (a, b) => sq(a - b) } Reduce { (a, b) => a + b }
  }
  Range(0, dists.length) Reduce { (i, j) =>
    if (dists(i) < dists(j)) i else j
  }
}
val newKmeans = clusters Map { e =>
  val sum = e Reduce { (v1, v2) => v1.Zip(v2) { (a, b) => a + b } }
  val count = e Map { v => 1 } Reduce { (a, b) => a + b }

  sum Map { a => a / count }
}
High-Level Compiler

- Optimizing locality
  - Tiling needed for finite on-chip memory and compute resources
  - Fuse loops to eliminate intermediate buffers
  - Existing methods for tiling and fusing (i.e. polyhedral analysis) can operate only on code sections with affine data access patterns

- Exploiting parallelism
  - Need to maximize utilization of all accelerator compute
  - Overlap compute with coarse-grain data dependencies (prefetching turns out to be a special case of metapipelining)
  - Hierarchical pipelining: Metapipelining
Markov State Models (MSMs)
MSMs are a powerful means of modeling the structure and dynamics of molecular systems, like proteins
Hardware
ML Accelerators Today

- **CPU**
  - Threads
  - SIMD

- **GPU**
  - Massive threads
  - SIMD
  - HBM

- **TPU**
  - MM unit
  - SW Cache

**What next?**
What to Accelerate? ML Arxiv Papers Per Year

Adapted from Jeff Dean
Scaled ML 2018
ML Accelerators for Tomorrow

The Future of ML Algorithms
Next-Gen ML Accelerators: Native Support for

- Hierarchical parallel pattern dataflow
  - Natural ML programming model
- Dynamic precision
  - HALP
- Sparsity
  - Graph based neural networks
- Data processing
  - SQL in inner loop of ML training
The Instruction Set Architecture (ISA) Bottleneck

- Programming model ⇒ Interface ⇒ Hardware

- Today
  - C++ ⇒ x86, ARM ⇒ CPU
  - CUDA ⇒ PTX ⇒ GPU

- ISA limitations
  - Fixed set of operations
  - Low level
  - Inefficient
Breaking the ISA Bottleneck

- Programming model ⇒ Interface ⇒ Hardware

Hierarchical parallel patterns

- GroupBy
  - Filter
- Weight
- Map
  - Reduce
- Output Data

↓

Hierarchical coarse-grain dataflow

↓

Hardware

ML Algorithms

ML Hardware
Spatial: Accelerator IR

- IR for hierarchical coarse-grain dataflow
  - Constructs to express:
    - Parallel patterns as parallel and pipelined datapaths
    - Explicit memory hierarchies
    - Hierarchical control
    - Explicit parameters

- Allows high-level compilers to focus on specifying parallelism and locality

spatial-lang.org

val output = vecA.Zip(vecB){(a, b) => a * b} Reduce{(a, b) => a + b}

val vecA = DRAM[Float](N)
val vecB = DRAM[Float](N)
val out = Reg[Float]

Reduce(N by B)(out) { i =>
val tA = SRAM[Float](B)
val tB = SRAM[Float](B)
val acc = Reg[Float]

  tA load vecA(i :: i+B)
  tB load vecB(i :: i+B)

  Reduce(B by 1)(acc){ j =>
    tA(j) * tB(j)
  }{a, b => a + b}
}
Tiled Dot Product

val vecA = DRAM[Float](N)
val vecB = DRAM[Float](N)
val out = Reg[Float]

Reduce(N by B)(out) { i =>
    val tA = SRAM[Float](B)
    val tB = SRAM[Float](B)
    val acc = Reg[Float]

    tA load vecA(i :: i+B)
    tB load vecB(i :: i+B)

    Reduce(B by 1)(acc){ j =>
        tA(j) * tB(j)
    }{a, b => a + b}
} {a, b => a + b}
Tiled Dot Product Design Parameters

val vecA = DRAM[Float](N)
val vecB = DRAM[Float](N)
val out = Reg[Float]

Reduce\(N \text{ by } B\)(out) { i =>
  val tA = SRAM[Float](B)
  val tB = SRAM[Float](B)
  val acc = Reg[Float]

tA load vecA\(i :: i+B\)
tB load vecB\(i :: i+B\)

Reduce\(B \text{ by } 1\)(acc){ j =>
  tA(j) * tB(j)
}{a, b => a + b}

Parallelism factor #1
Metapipelining toggle

Parallelism factor #2

Parallelism factor #3

Banking strategy

Tile Size (B)

Spatial compiler optimizes parameters
Compiler Architecture

- Start from productive, high level DSLs
- Use a common parallel pattern representation across DSLs
- Tile and metapipeline
- **Spatial**: captures memory hierarchy, design parameters, arbitrarily nested pipelines
- Map to accelerator hardware

---

**Dataflow graph of domain-specific operators**

**Hierarchical dataflow graph of parallel patterns**

**Hierarchical dataflow graph of tiled pipelines**

**Memory hierarchy**

---

**DSL application**

**IR Translation**

**Parallel Pattern IR**

**High-level Compiler**

**Spatial IR**

**Spatial Compiler**

**Accelerator Hardware**
Plasticine: A Reconfigurable Architecture for Parallel Patterns

High-level Parallel Patterns (Spatial)
- map
- filter
- reduce
- groupBy
- key1
- key2
- key3

Plasticine Architecture

Tiled architecture with reconfigurable SIMD pipelines, distributed scratchpads, and statically programmed switches

High Performance Energy Efficiency

Up to 95x Performance
Up to 77x Perf/W vs. Stratix V FPGA

Prabhakar, Zhang, et. al. ISCA 2017

Raghu Prabhakar
Yaqi Zhang
Compiler Architecture

- **IR Translation**
- **High-level Compiler**
- **Spatial Compiler**
- **Plasticine IR**
- **PIR Mapper**

- **Spatial IR**
- **Plasticine Configuration**

- **DSL application**

- **Dataflow graph of domain-specific operators**
- **Hierarchical dataflow graph of parallel patterns**
- **Hierarchical dataflow graph of tiled pipelines**

- **Memory hierarchy**
- **Memory and compute units**
- **Control information**

- **Weight**
  - **Input Data**
  - **Conv**
  - **Pool**
  - **Conv**
  - **Norm**
  - **Sum**

- **Weight**
  - **Input Data**
  - **Map**
  - **Reduce**

- **DRAM**
  - **Line Buffer**
  - **Reg File**
  - **Shift Reg**
  - **SRAM**

- **PMU**
  - **PCU**
  - **PMU**

- **PCU**

- **PMU**

- **PCU**

- **PMU**
Mapping Spatial to Plasticine

Load vecA(\(i : i+B\))

Load vecB(\(i : i+B\))

tA

tB

Dot Product
Energy Efficiency (MOPS/mW)

Reprogramming Time (seconds)

Software Defined Hardware (SDH)

- ASIC: Fixed-function (Not reprogrammable)
- SDH: Coarse-grain dataflow
- GPU: Instruction-based
- CPU: Instruction-based

Efficiency vs. Flexibility
We Can Have It All with Software 2.0!

- Productivity
- Power
- Performance
- Programmability
- Portability

ML Algorithms (e.g. Hogwild!, HALP)

High Performance DSLs (e.g. OptiML, TensorFlow, PyTorch)

High-Level Compiler

Accelerator IR (e.g. Spatial)

Low-Level Compiler

Hardware Architectures (e.g. SDH)
Thank You!

- Questions?