Extracting Parallelism is key to performance.

Key goal of hardware, systems, and for more than a decade. The only way to get performance.
Old Slides from ~2017. But these are main ideas, we’ll see them at multiple scales.

Biased by my own work because I have slides and am lazy... not because I think it’s best.
Statistical algorithms have relaxed notions of correctness leads to new opportunities for:

- Algorithms,
- Systems, and
- Hardware.
Key Issue: Balance Statistical versus Hardware Efficiency.

- **Statistical efficiency** how many steps you take
- **Hardware efficiency** how efficiently you take each of those steps
Three driving trends in hardware

(1) Lots of smaller cores,
(2) Non-Uniform Memory (NUMA), and
(3) Single/Instruction Multiple Data (SIMD) (and SIMT)

Approximation allows major performance improvements.
Trend 1: Many different Cores

Steve Wright

Ben Recht

Feng Niu
Single Cores are not getting faster.

Chips now contain many cores, so throughput is increasing... but need to rewrite algos!
Statistical Analytics Crash Course

Staggering amount of machine learning/stats can be written as:

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

\(N\) (number of \(y_i\)s, data) typically in the billions

Ex: Classification, Recommendation, Deep Learning.

De facto iteration to solve large-scale problems: **SGD**.

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

Billions of tiny iterations.

Select one term, \(j\), and estimate gradient.
Multicore: Independent Case

Jobs with little communication, 2 cores executes twice as faster!
Protocol for “whose turn,” called locking, takes 100 cycles.
Communication scales quadratically

Suppose it takes 1 second to communicate with 2 cores.

- 4 cores takes 4 seconds
- 8 cores takes 16 seconds.
- $k$ cores takes $(k/2)^2$ seconds.

Server may have 100+ cores
The key algorithm in machine learning consists of BILLIONS of tiny jobs!

The core algorithm of modern learning called is **Stochastic Gradient Descent (SGD)**.

Implemented in a classical way (locking) SGD actually gets slower with more cores.

So what can we do?
Multicore: Hogwild! Case

Job 1

Job 2

Job 3

Job 4

Ignore the locks!

Is it my turn? Yes!
How do we run SGD in Parallel?

Just ignore the locking protocol…
As we say, go Hogwild!

This is computer science heresy!

Theorem (roughly, NIPS11): If we do no locking, SGD converges to correct answer—at essentially the same rate!

Hogwild! [Niu, Recht, Ré, Wright NIPS11]
AsySCD [Liu, Wright et al. ICML14, JMLR14]
Buckwild! [De Sa, Olukotun, Ré NIPS15]
Cortana: Microsoft’s Digital Assistant

**W I R E D**

AI breakthrough: Microsoft’s ‘Project Adam’ identifies dog breeds, points to future of machine learning

All web companies have similar: image rec, voice, mobile, search, etc.

“...using a technology called, of all things, Hogwild!”

http://www.wired.com/2014/07/microsoft-adam/
A larger trend?

Relaxing **consistency** to be **architecturally aware** can be a big performance win.
A regularizer is a (sane) statistical penalty...
Bugs in your implementation are not helpful
Trend 2: NUMA
Non-Uniform Memory Access

Steve Wright
Ji Liu
Ce Zhang
Krishna Sridhar
Modern version: Thousands of cores with close by memory
(Called high-bandwidth memory, called HBM)
One Example: Quadratic Programming with Orthant Constraints (on cpu, same tradeoffs)
One Example.

The graph shows the performance (Speed up) of parallel sum operations (Parallel Sum) and spinlock (Spinlock) compared to hogwild (Hogwild!) across different numbers of threads (# of Threads).

At higher numbers of threads, the parallel sum operation exhibits thrashing across sockets, indicating potential inefficiencies or resource contention.

The hogwild approach maintains a relatively stable performance, suggesting it may handle parallelism more effectively under these conditions.
What about multiple sockets?

![Chart showing speed up vs. number of threads for different parallel sum methods: Spinlock, Dimm Witted, Parallel Sum, and Hogwild!]

- **Spinlock** remains at a constant speed up of 0.
- **Dimm Witted** shows an increasing trend from 0 to 25.
- **Parallel Sum** starts at 0 and increases rapidly, reaching a peak of 25 at 36 threads.
- **Hogwild!** starts at 0 and shows a slight increase before plateauing.

The chart illustrates the performance differences across these methods as the number of threads increases.
Model Replication

PerMachine (Hogwild!)

RAM

Just
Data

Node 1
Core 1
Core 2

Cache coherence

Node 2
Core 3
Core 4

Stalls!

RAM

Model

PerCore

RAM

Model
Model

Node 1
Core 1
Core 2

Infrequent Communication

Node 2
Core 3
Core 4

Model
Model
Model Replication

<table>
<thead>
<tr>
<th>Statistical Efficiency</th>
<th>Hardware Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hogwild!</td>
<td>High</td>
</tr>
<tr>
<td>PerCore</td>
<td>Low</td>
</tr>
</tbody>
</table>

In between both **Hogwild!** and **PerCore**?

PerNode

- **Node 1**
  - Core 1
  - Core 2

- **Node 2**
  - Core 3
  - Core 4

Infrequent communication
Relaxing consistency results in new tradeoffs.

1. Access methods
   - {Row, Column, Row-col}

2. Model Replication
   - {Core, Node, Machine}

3. Data Replication
   - {Full, Importance, Shard}

Can be 100x faster than classical choices

*DimmWitted: A Study of Main-Memory Statistical Analytics. VLDB14.*
Trend 3: Single Instruction Multiple Data (SIMD)

Modern processors offer fine-grained parallelism. [NIPS15]
**SIMD Processing: Fine-grained parallelism**

**Single instruction multiple data (SIMD)**

<table>
<thead>
<tr>
<th>Standard Addition (Two registers)</th>
<th>SIMD Addition (4 way)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁ 4</td>
<td>R₁ 1 2 3 4</td>
</tr>
<tr>
<td>R₂ 4</td>
<td>R₂ 2 4 6 8</td>
</tr>
<tr>
<td>R₁ + R₂ = 8</td>
<td>R₁ + R₂ = 3 6 9 12</td>
</tr>
</tbody>
</table>

*Same operation on multiple data points* in parallel
SIMD: Doubling again!

SIMD bandwidth has doubled each of the last four generations.

Good old days of Moore’s Law! … If we can take advantage of fine-grained parallelism
Precision vs. Parallelism

SIMD Precision

- 32-bit float vector
- 16-bit int vector
- 8-bit int vector

SIMD Parallelism

- 8x ops/cycle (vmulps instruction)
- 16x ops/cycle (vpmaddwd instruction)
- 32 ops/cycle (vpmaddubsw instruction)

Tradeoff between precision & parallelism
A hardware model for precision
[ISCA17]

Chris De Sa
Kunle Olukotun
Four Classes of Numbers

- **Dataset numbers**
  - used to store the immutable input data

- **Model numbers**
  - used to represent the vector we are updating

- **Gradient numbers**
  - used as intermediates in gradient computations

- **Communication numbers**
  - used to communicate among parallel workers
Quantize classes independently

- Using low-precision for different number classes has different effects on performance.
  - e.g. quantizing the gradient numbers improves compute throughput, but has little effect on memory.

- Existing work often quantizes some classes, but doesn’t consider the others.
The **DMGC** Model

- **Idea**: associate each implementation with a **DMGC signature** that displays its precision for all four number classes
- Let's classify previous work and future systems

\[ D^8 M^{16} G^{32f} C^{16} \]

- The algorithm uses 8-bit numbers to store the dataset.
- It uses 16-bit numbers for the model.
- It computes gradients as 32-bit floats.
- It communicates among workers with 16-bit numbers.
Be warned: Your learning parameters depend on the hardware and those numbers. (e.g. momentum and delay are connected)
What shook my belief in progress through optimization...

• Turns out Optimization is a **leaky** abstraction for deep learning.
  • There are approaches that cause the loss to go down more slowly (worse optimization) but generalizing better (better test performance).

• Happy to give examples if you ask, so many out there it’s bizarre....

• This is so much more interesting than it should be!