Designing Computer Systems for Software 2.0

Kunle Olukotun
Stanford University
SambaNova Systems

NeurIPS Invited Lecture, December 6, 2018

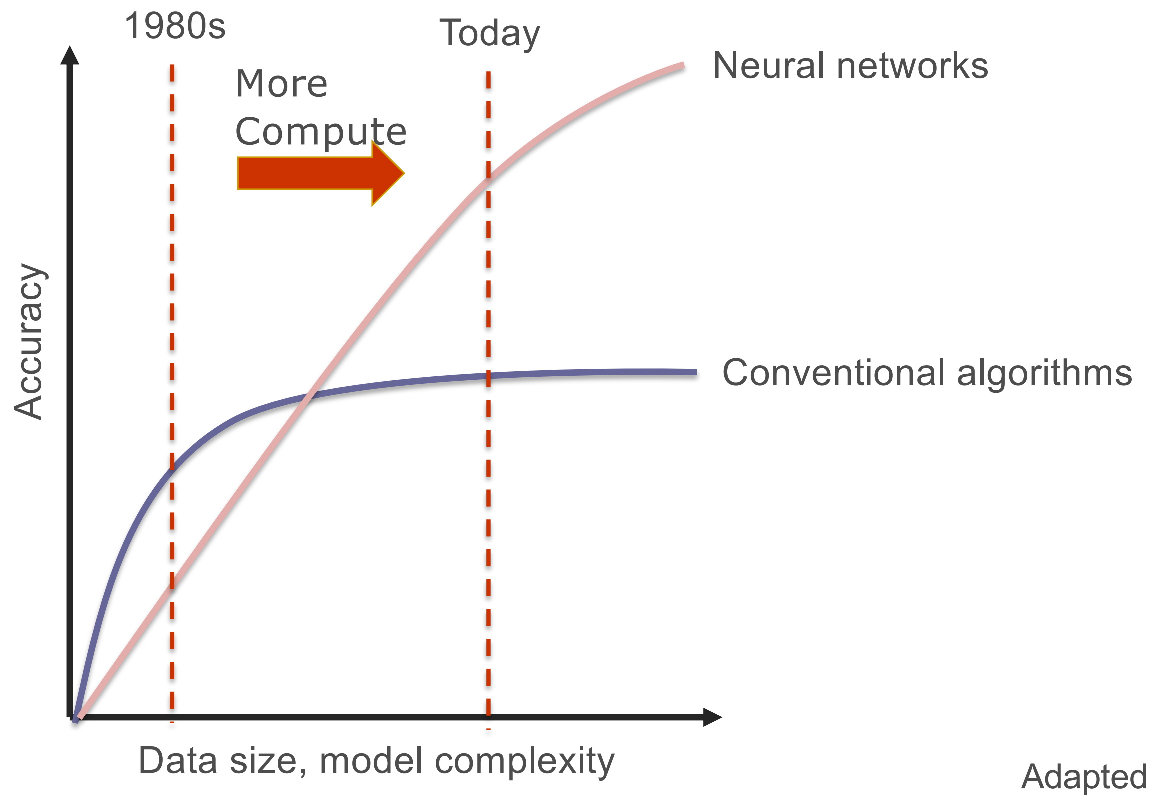
Two Big Trends in Computing

- 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



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

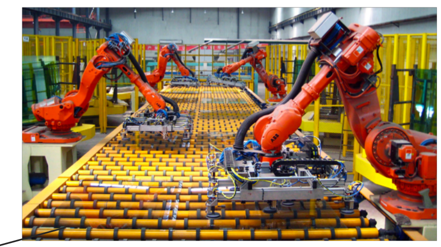
- Written in the weights of a neural network model by optimization

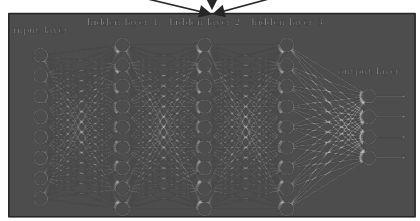
Andrej Karpathy
Scaled ML 2018 talk

Software 2.0 is Eating Software 1.0

Easier to build and deploy

- Build products faster
- Predictable runtimes and memory use: easier qualification





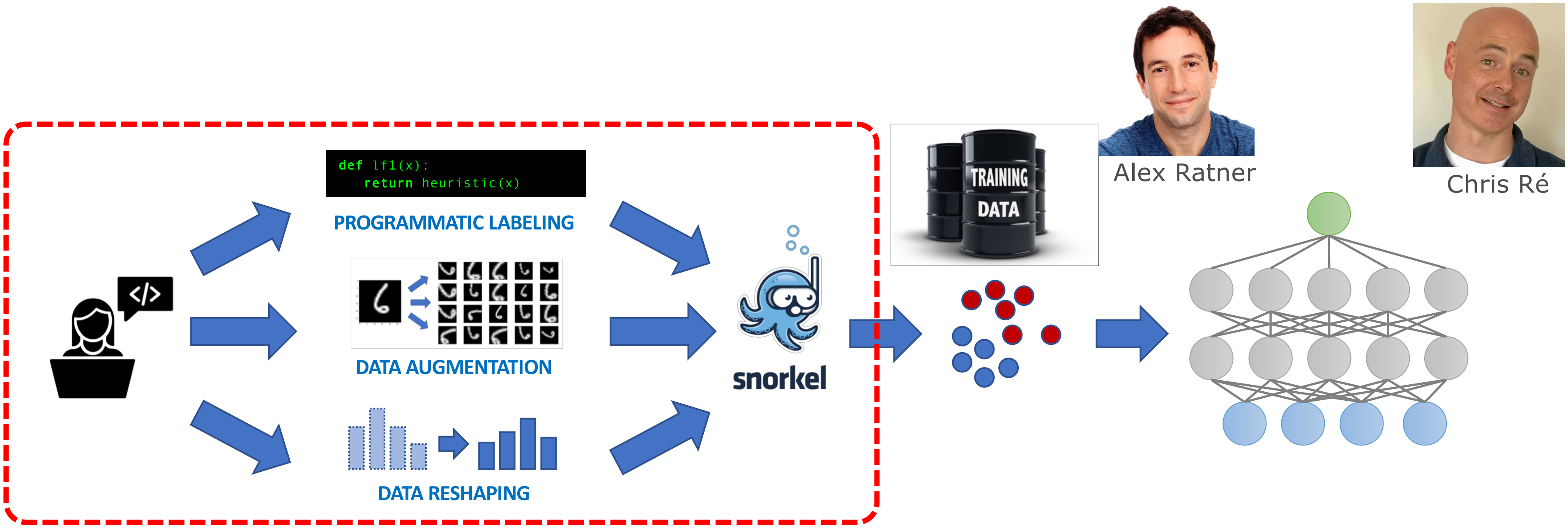
1000x Productivity: Google shrinks language translation code from 500k LoC to 500

Classical problems

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

https://jack-clark.net/2017/10/09/import-ai-63-google-shrinks-language-translation-code-from-500000-to-500-lines-with-ai-only-25-of-surveyed-people-believe-automationbetter-jobs

Software 2.0: Programming is Changing



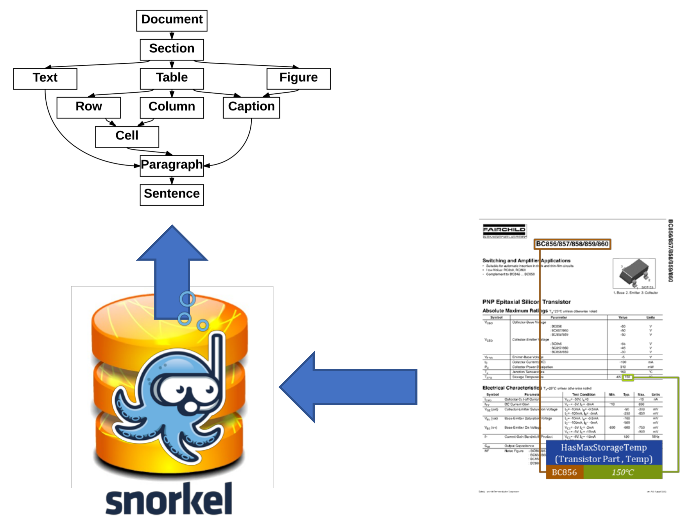
ML developers increasingly program Software 2.0 stacks by *creating and engineering training data*

snorkel.stanford.edu

SQL Queries in Inner ML Training Loops

```python
# 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()
        …
```

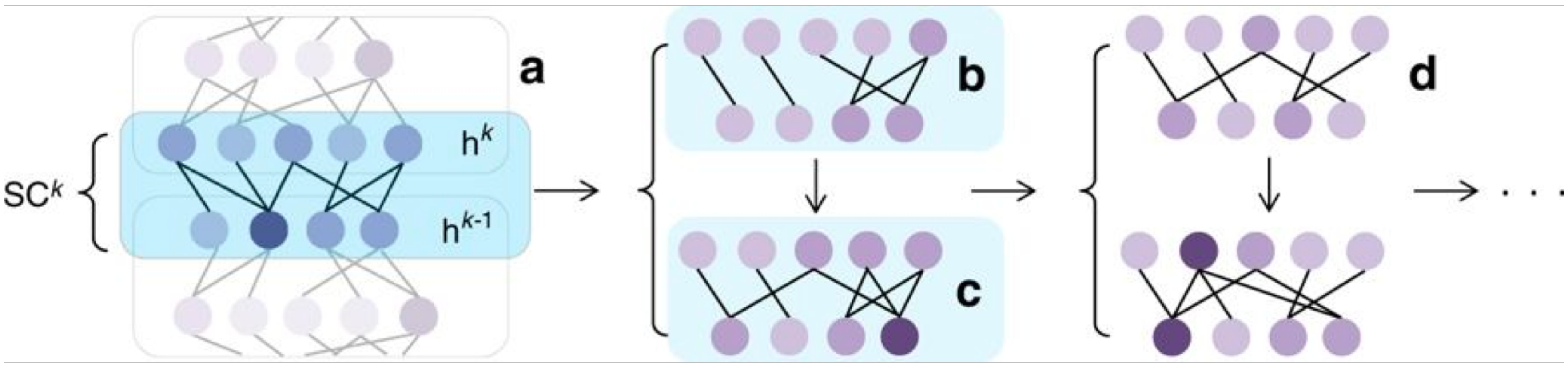
Complex structured data stored in RDBMS



Loaded dynamically during training

(Pulling training points from a database backend)

Sparsity is becoming a design objective for neural networks of all types…

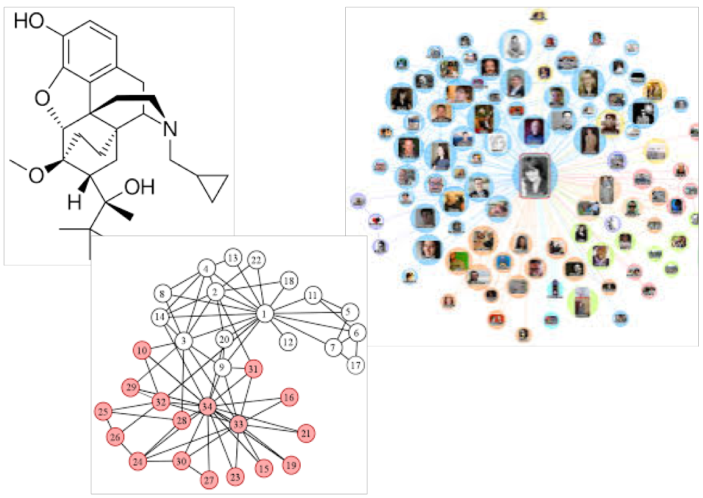


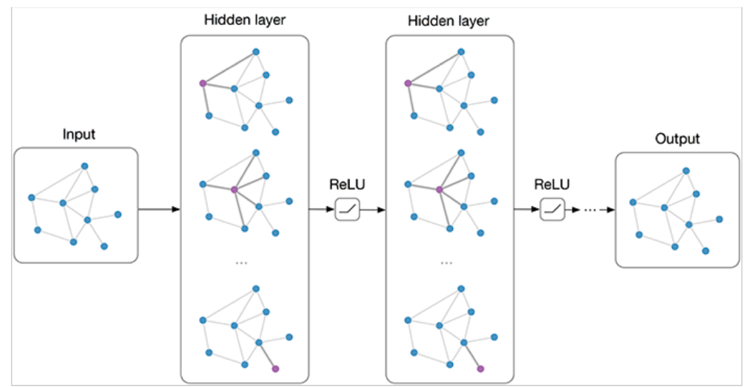
Sparsely connected network layers can maintain performance while reducing parameter number

Mocanu, D. Cet al. (2018). N*ature Communications*, *9*(1), 2383.; Left panel from https://tkipf.github.io/graph-convolutional-networks/

* Figure from Mocanu et al, 2018

Graph Neural Networks (GNNs) are increasingly popular for network-structured data

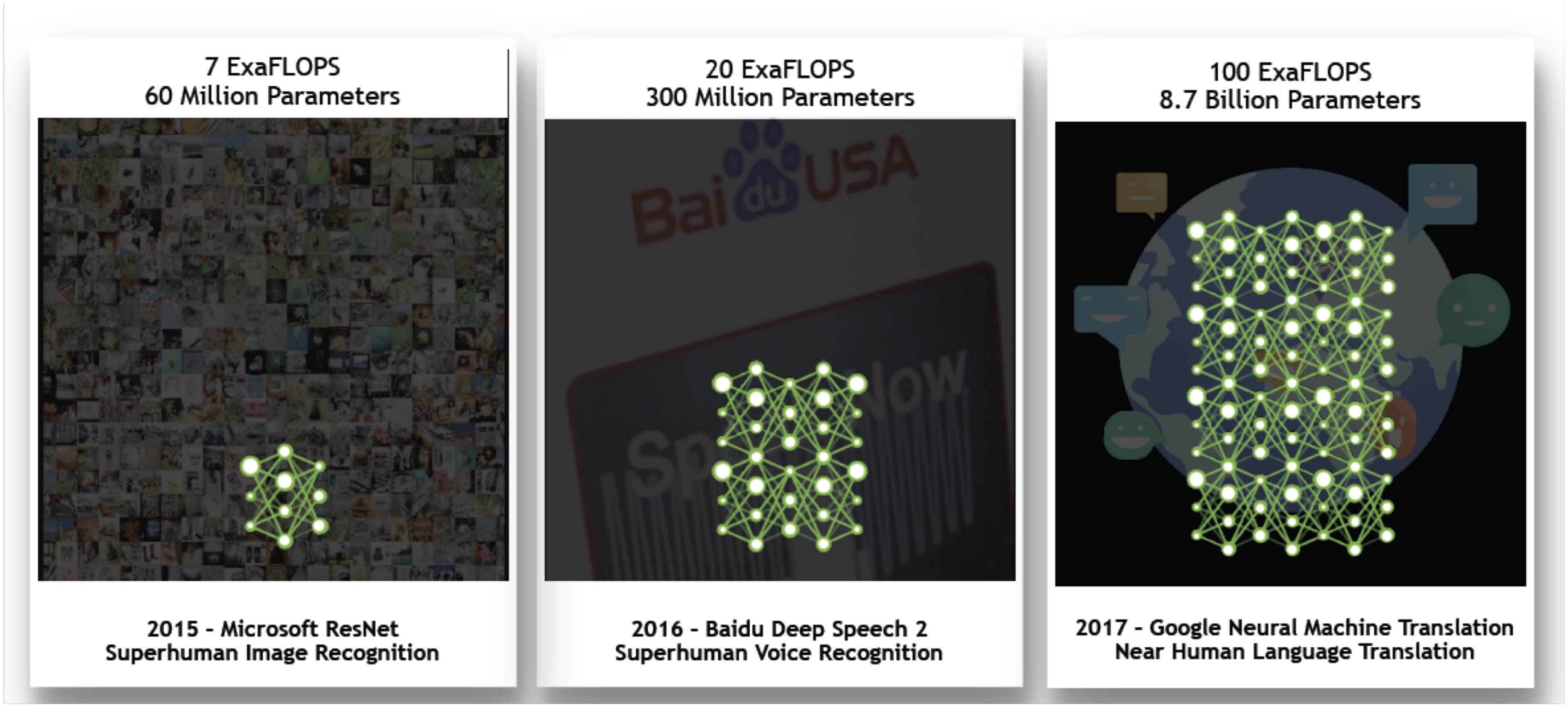




Techniques like *neural message passing algorithms* leverage sparse graph structure and data access patterns

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

Increasing Model Complexity



Source: Bill Dally, Scaled ML 2018

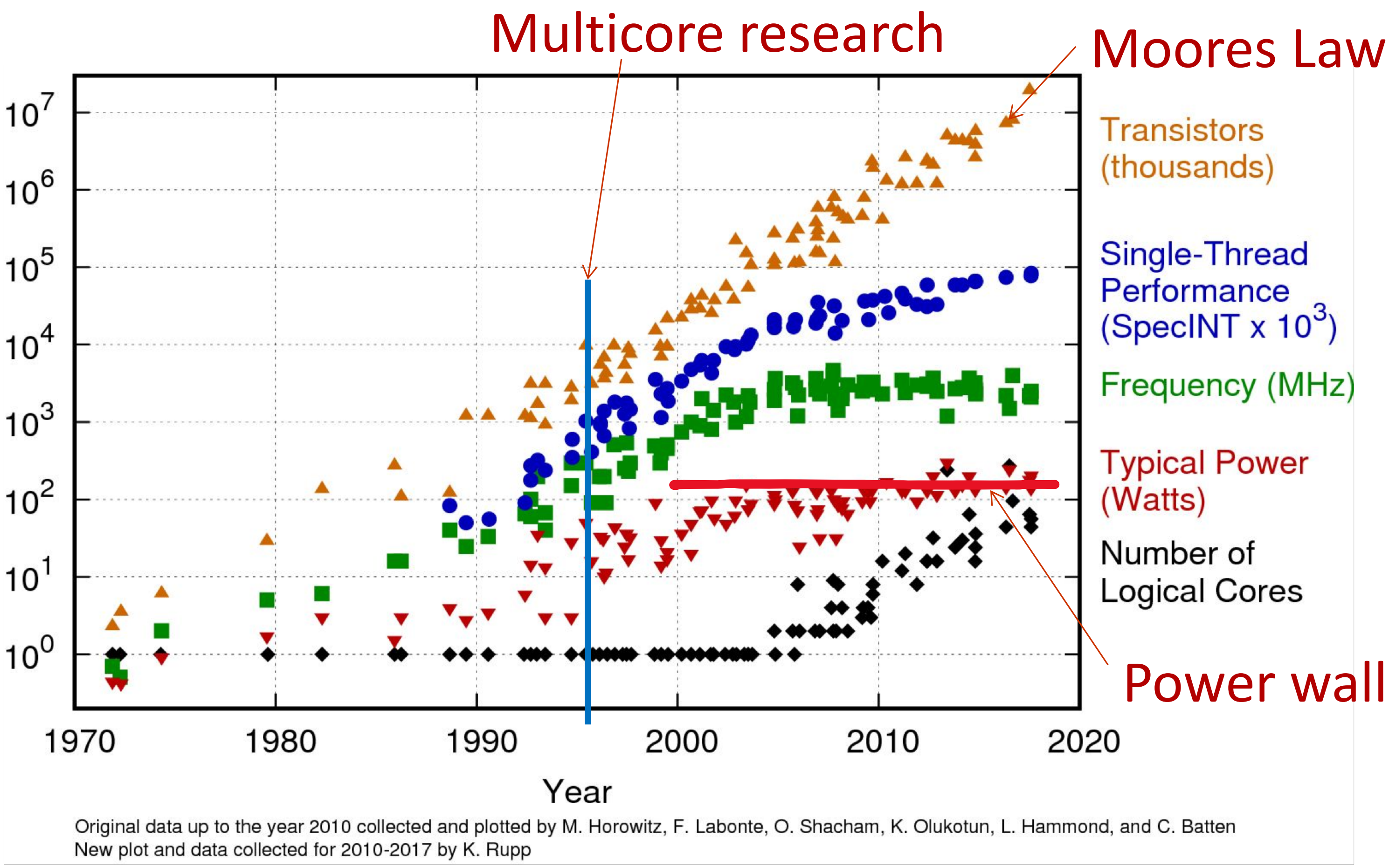
ML Training is Limited by Computation

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



Power and Performance

Performance

Energy efficiency

$$Power = \frac{Ops}{second} \times \frac{Joules}{Op}$$

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

Billions

Loss function

Optimization Problem:

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

Data

Model

E.g.: Classification, Recommendation, Deep Learning

Solving large-scale problems:

Stochastic Gradient Descent (SGD)

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

Select one term, j, and estimate gradient

Billions of tiny sequential iterations

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

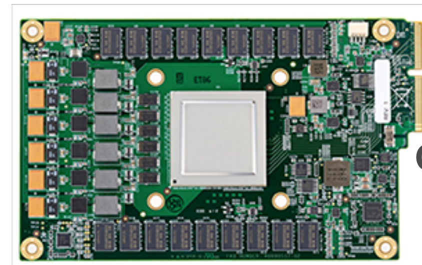
Ce Zhang and Christopher Ré.. DimmWitted: Proc. VLDB `14

Low Precision: The Pros

↓ **Energy**

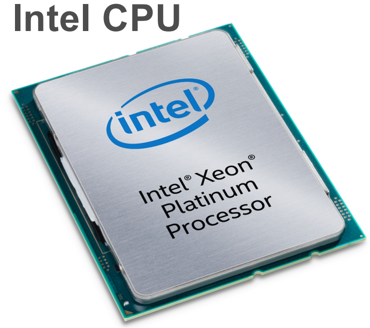
↓ **Memory**

↑ **Throughput**



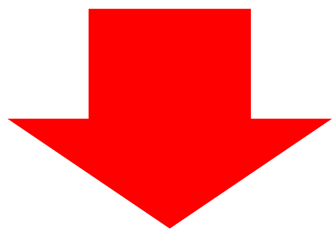
Google TPU

Intel CPU



Microsoft Brainwave (FPGA)

Low Precision: The Con



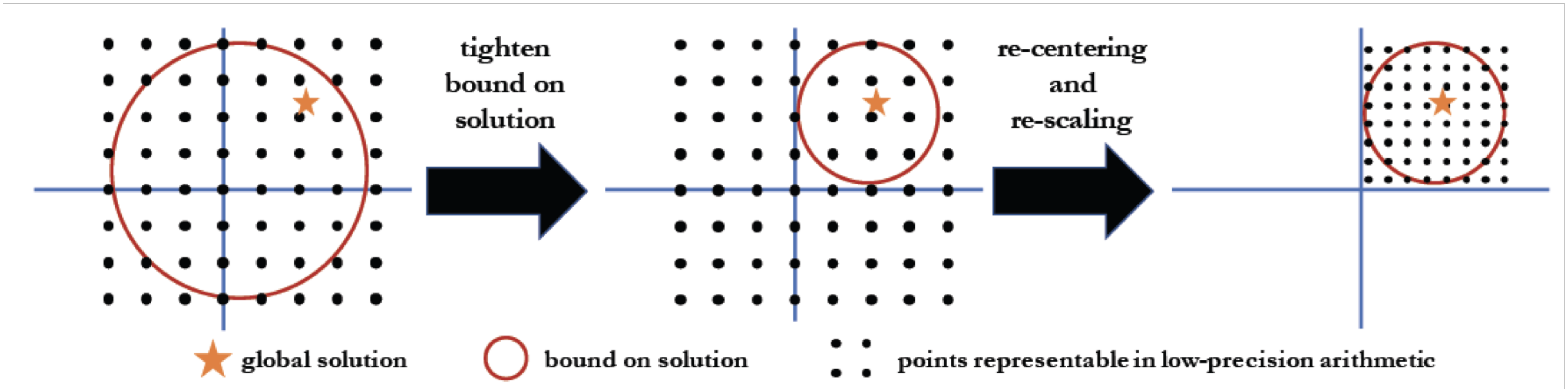
Accuracy

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

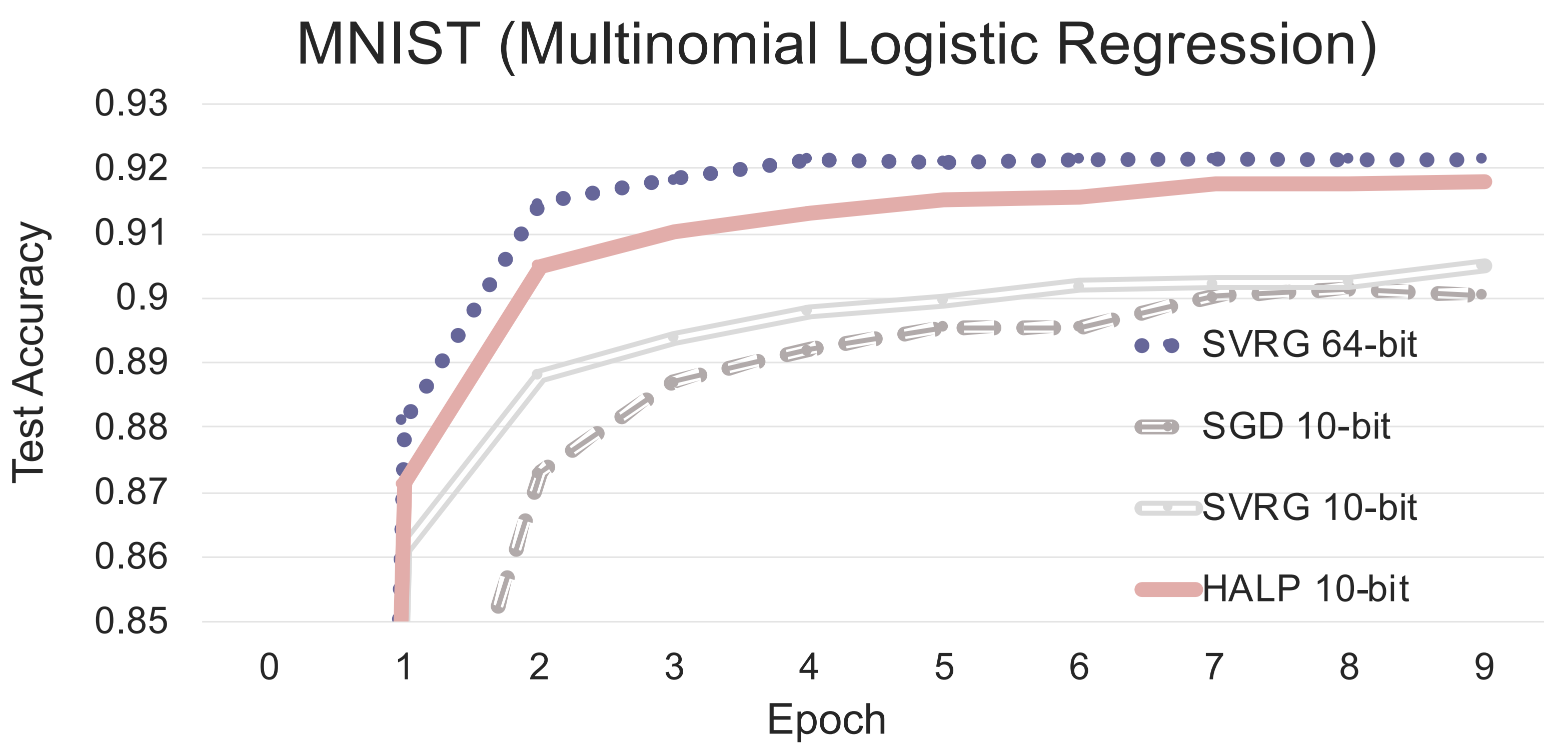
Bit Centering: bound, re-center, re-scale



- 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

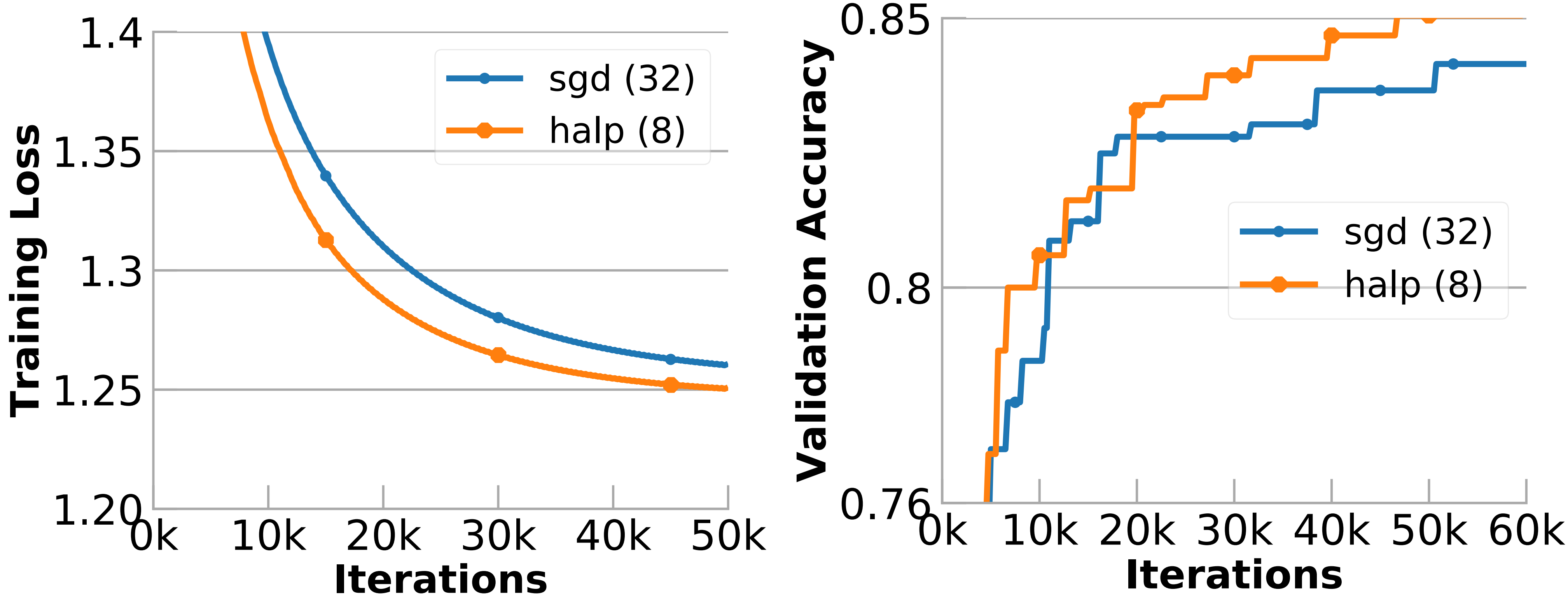
HALP Training



HALP provably converges at a linear rate

CNN: HALP versus Full-Precision Algorithms

14-layer ResNet on CIFAR10



- HALP has better statistical efficiency than SGD!

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


Chris De Sa


Song Han


Chris Aberger

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

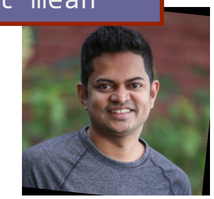
K-means Clustering in OptiML

```
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
}
```

assign each sample to the closest mean

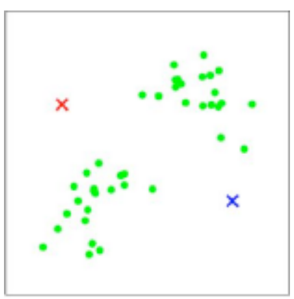
calculate distances to current means

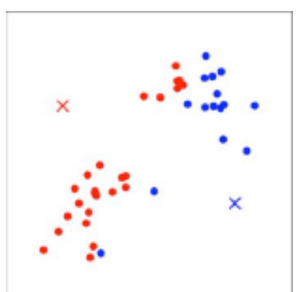
move each cluster centroid to the mean of the points assigned to it

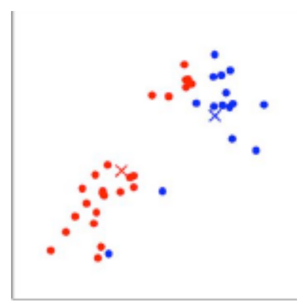


Arvind Sujeeth

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







A. Sujeeth et. al., "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning," *ICML, 2011*.

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

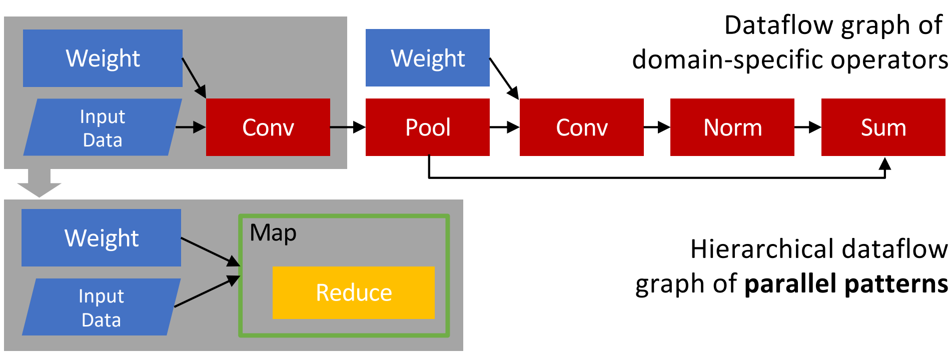
move each cluster centroid to the mean of the points assigned to it

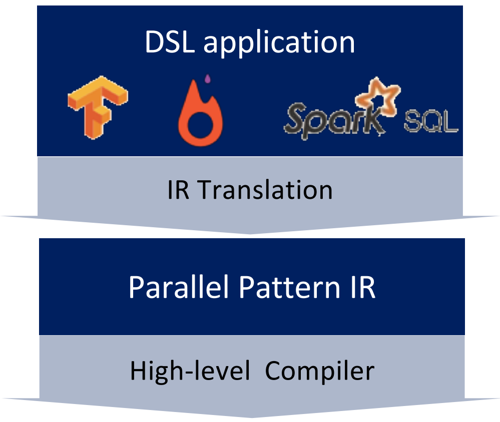
Open, standard software for general machine learning

Great for Deep Learning in particular

First released Nov 2015

Compiler Architecture

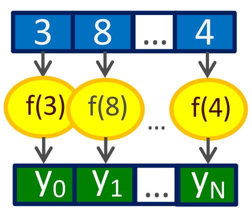




- Build a full compiler stack to compile high level DSLs to accelerator hardware

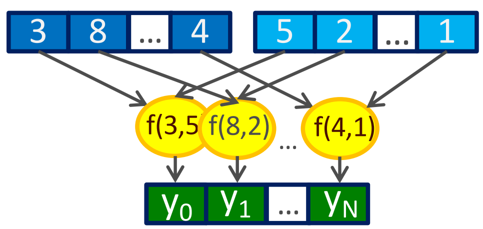
Parallel Patterns

- 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



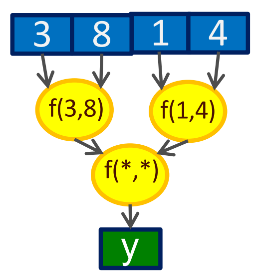
Map
element-wise function f

```
y = vector + 4
y = vector * 10
y = sigmoid(vector)
```



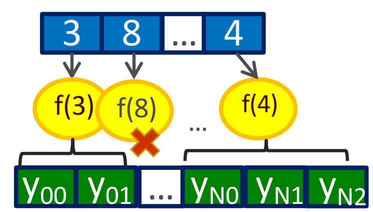
Zip
element-wise function f (multi-collection)

```
y = vecA + vecB
y = vecA / vecB
y = max(vecA,vecB)
```



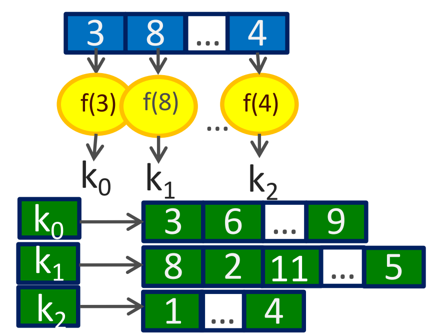
Reduce
combine all elements with f (f is associative)

```
y = vector.sum
y = vector.product
y = max(vector)
```



FlatMap
element-wise function ≥0 values out per element

```
SELECT * FROM vector
WHERE elem < 5
```



GroupBy
group elements into buckets based on key

```
vector.groupBy{e => e % 3}
```

Parallel Pattern Language ➔ High Level Parallel ISA

- Example application: *k*-means
- A data-parallel language that supports nested parallel patterns {{{}}}
- Hierarchical dataflow graph of parallel patterns

```
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 }
}
```

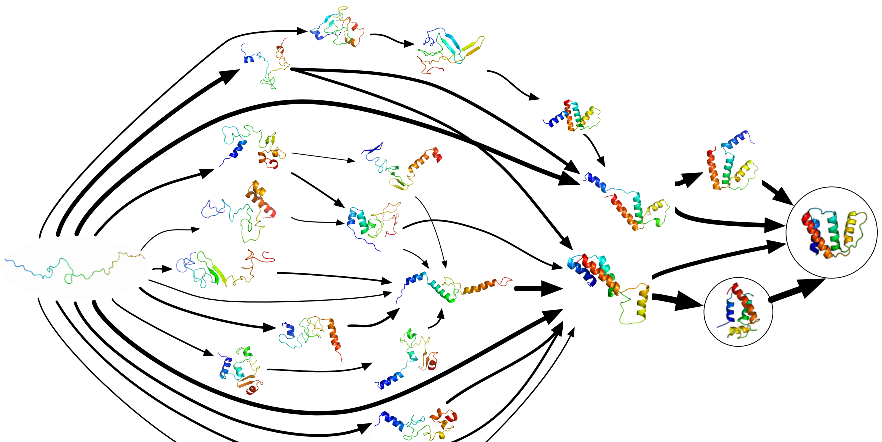
High-Level Compiler

- 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

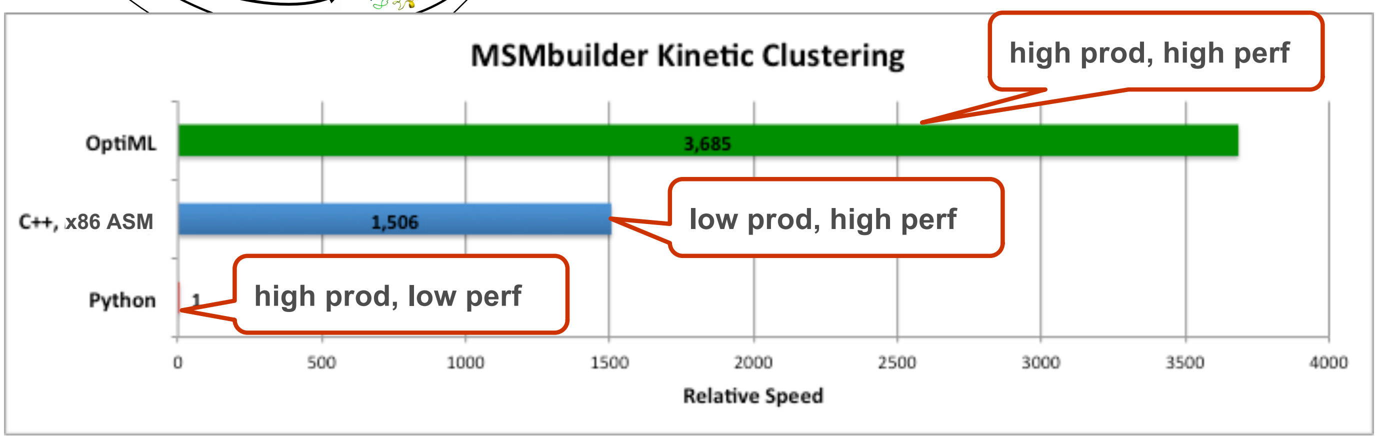
- Exploiting parallelism
  - Need to maximize utilization of all accelerator compute
  - Overlap compute with coarse-grain data dependencies (prefetching turns out to be a special case of metapipelining)
  - Hierarchical pipelining: Metapipelining

MSM Builder Using OptiML

with Vijay Pande



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

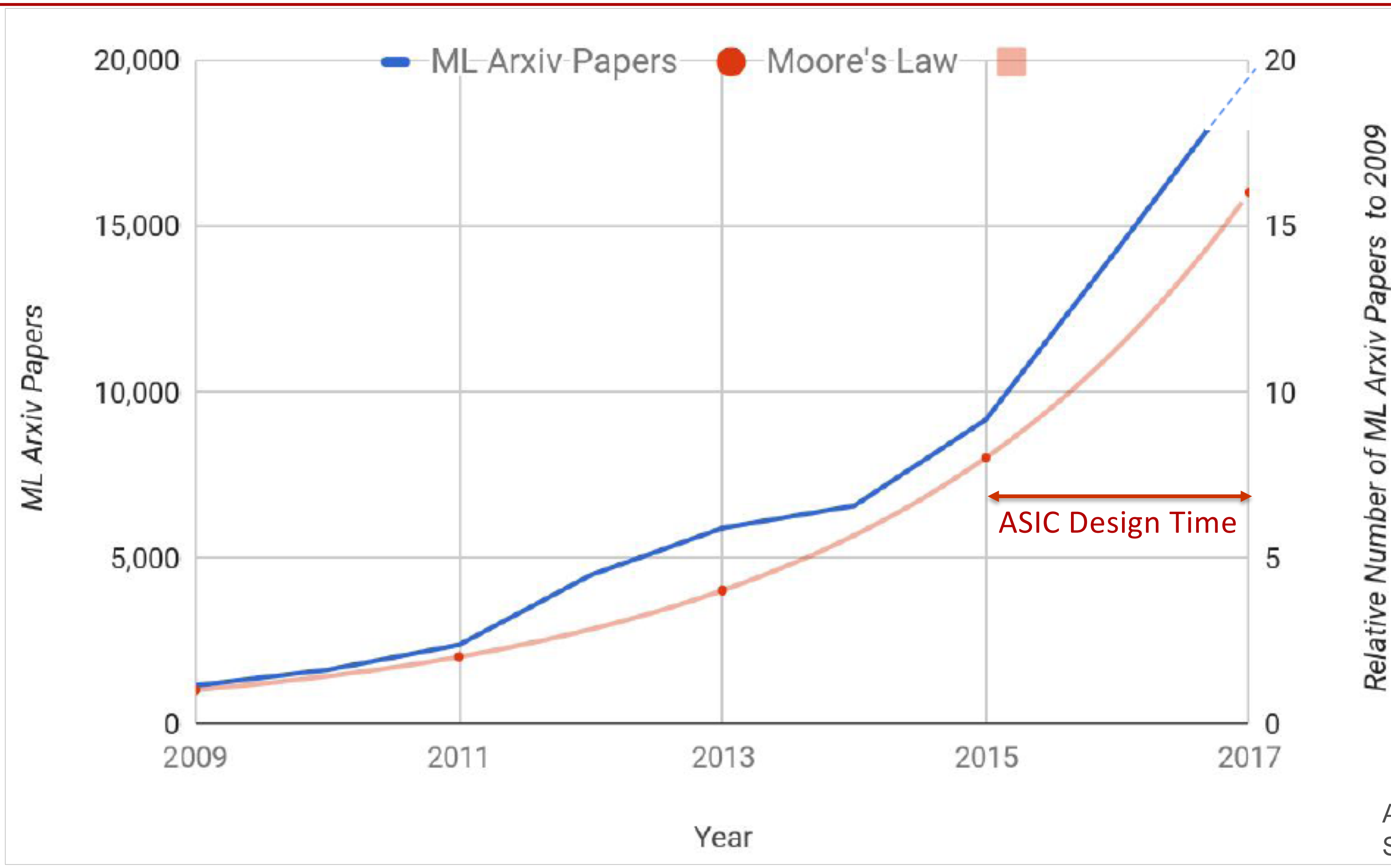
- Massive threads
- SIMD
- HBM

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

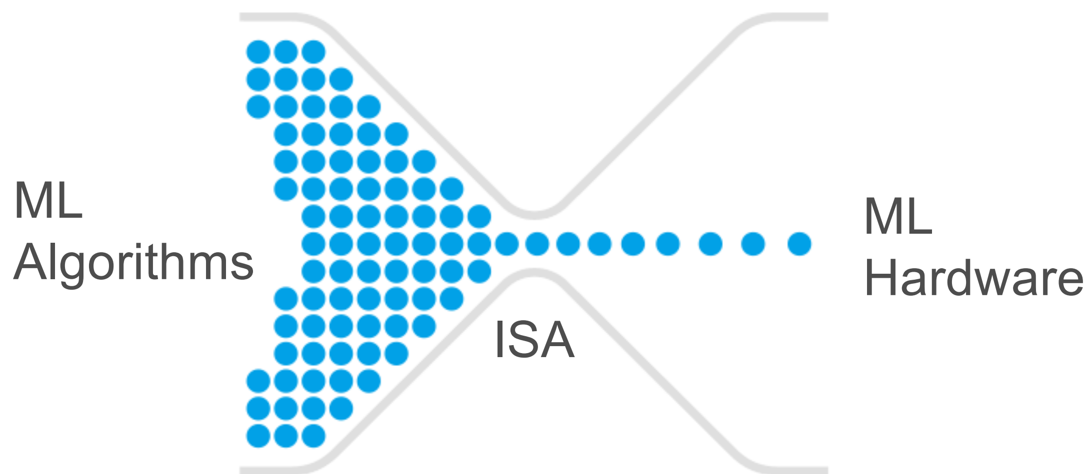
- Hierarchical parallel pattern dataflow
  - Natural ML programming model
- Dynamic precision
  - HALP
- Sparsity
  - Graph based neural networks
- Data processing
  - SQL in inner loop of ML training

The Instruction Set Architecture (ISA) Bottleneck

- Programming model ⇒ Interface ⇒ Hardware

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

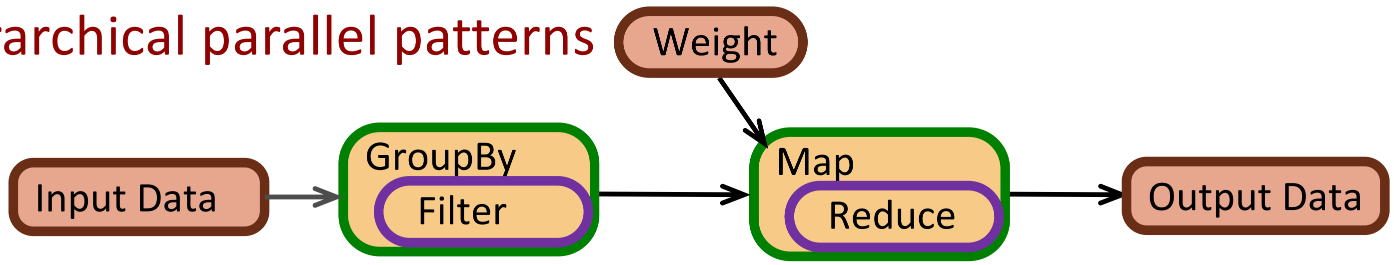
- ISA limitations
  - Fixed set of operations
  - Low level
  - Inefficient



Breaking the ISA Bottleneck

- Programming model ⇒ Interface ⇒ Hardware

Hierarchical parallel patterns

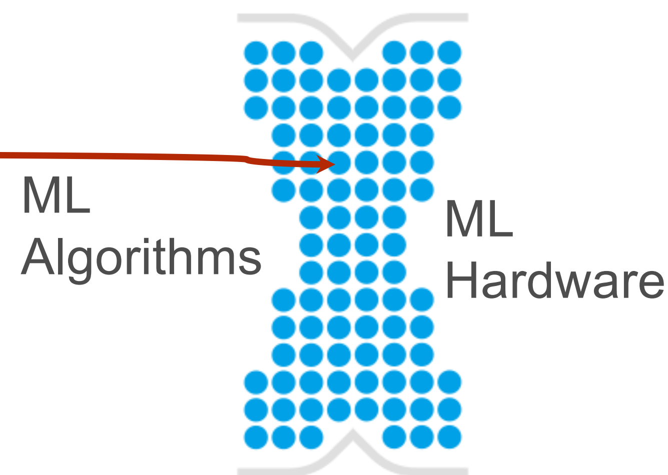


⇓

Hierarchical coarse-grain dataflow

⇓

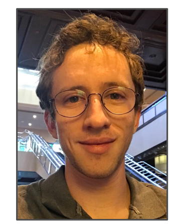
Hardware



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


David Koeplinger


Matt Feldman

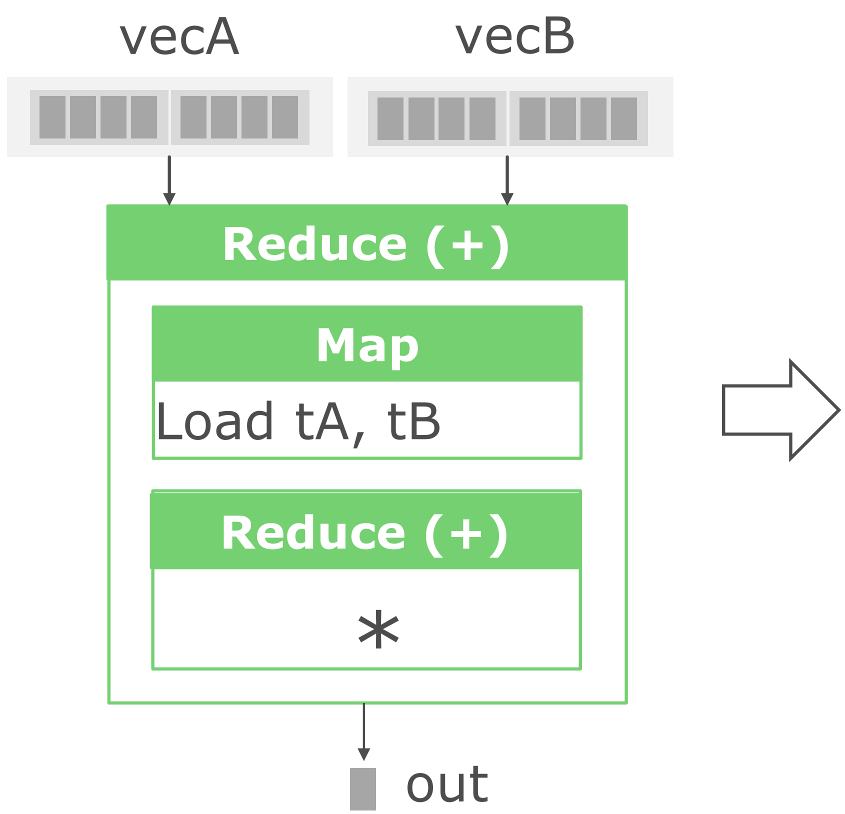
spatial-lang.org

D. Koeplinger et. al., "Spatial: A Language and Compiler for Application Accelerators" *PLDI 2018.*

12/5/18

Tiled Dot Product

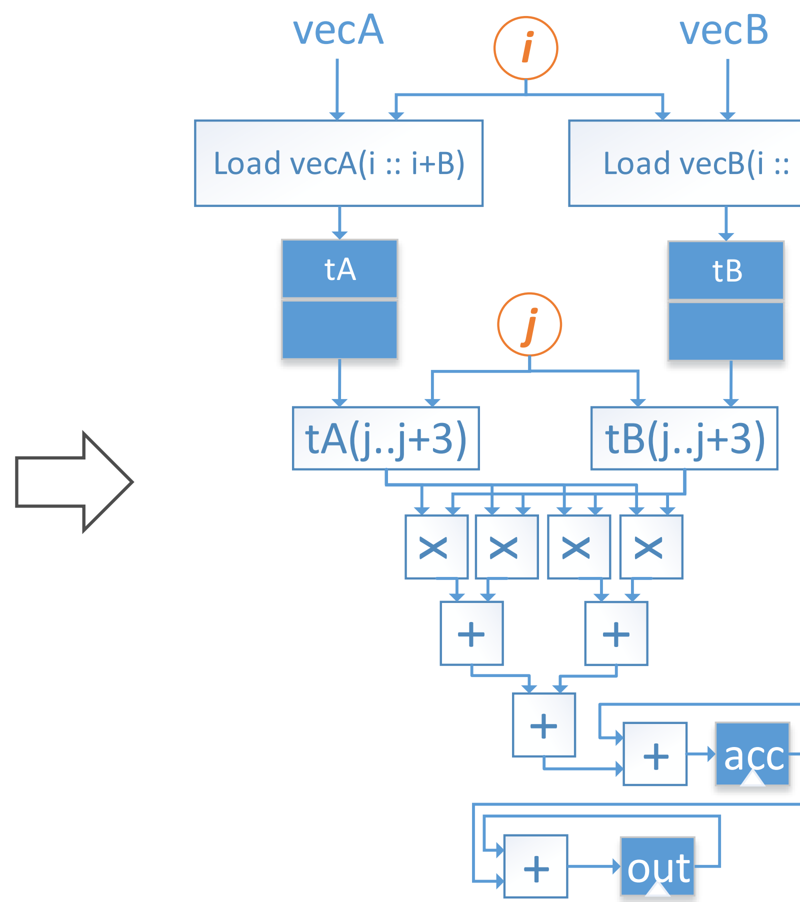
```
val output = vecA.Zip(vecB){(a,b) => a * b} Reduce{(a,b) => a + b}
```




```
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}
}{a, b => a + b}
```

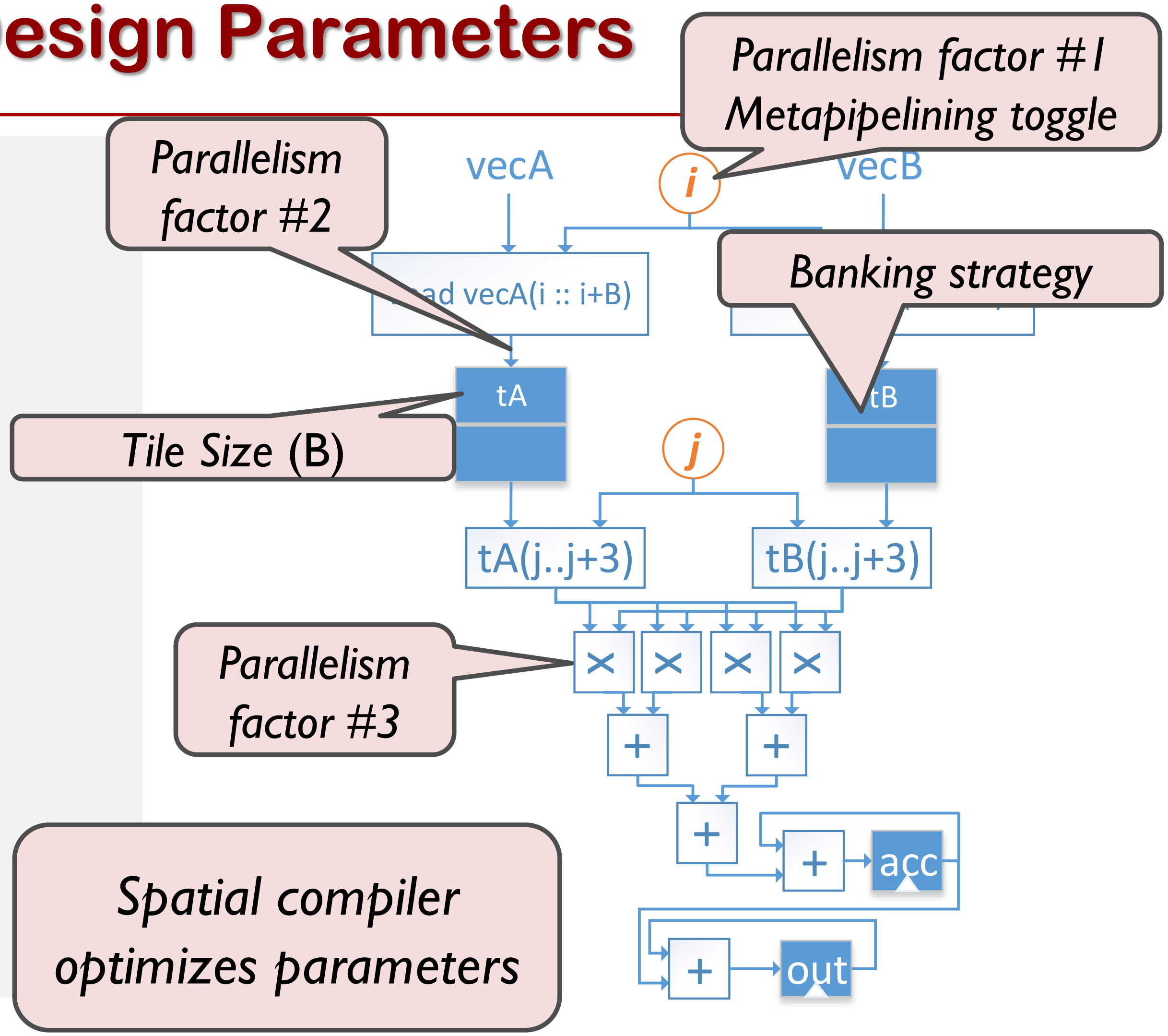
Tiled Dot Product

```
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}
```

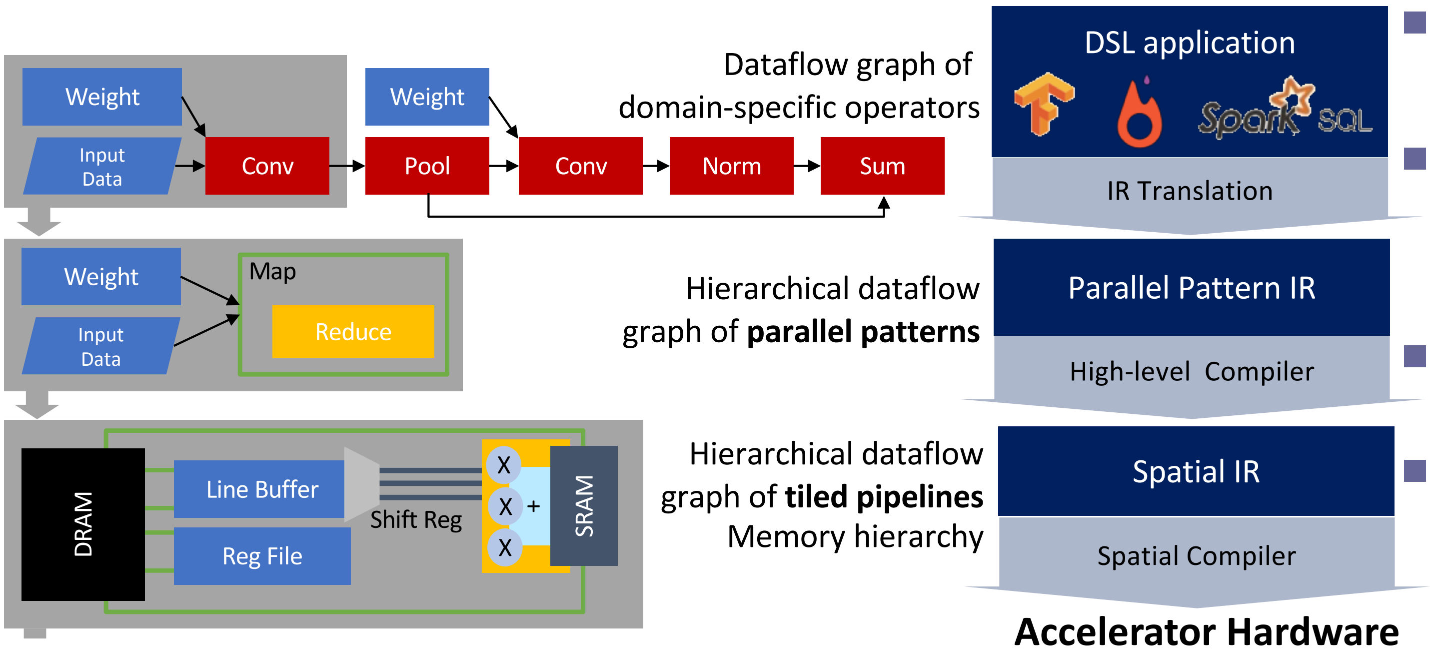


Tiled Dot Product Design Parameters

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

Reduce(N 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}
```



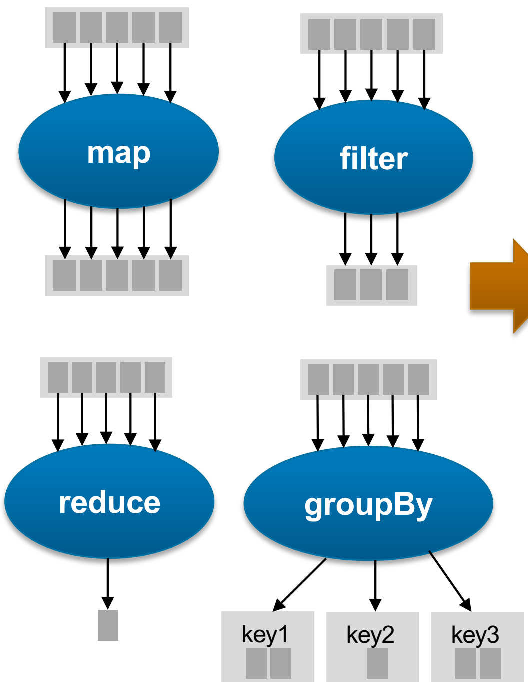
Compiler Architecture



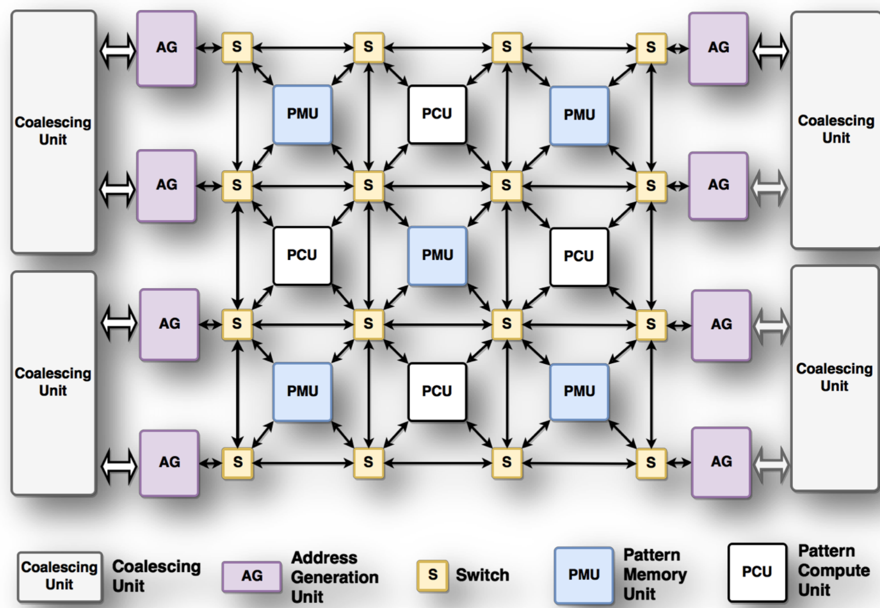
- Start from productive, high level DSLs
- Use a common parallel pattern representation across DSLs
- Tile and metapipeline
- **Spatial**: captures memory hierarchy, design parameters, arbitrarily nested pipelines
- Map to accelerator hardware

Plasticine: A Reconfigurable Architecture for Parallel Patterns

High-level Parallel Patterns (Spatial)



Plasticine Architecture



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

High Performance

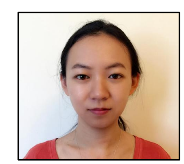
Energy Efficiency

Up to **95x** Performance

Up to **77x** Perf/W

vs. Stratix V FPGA

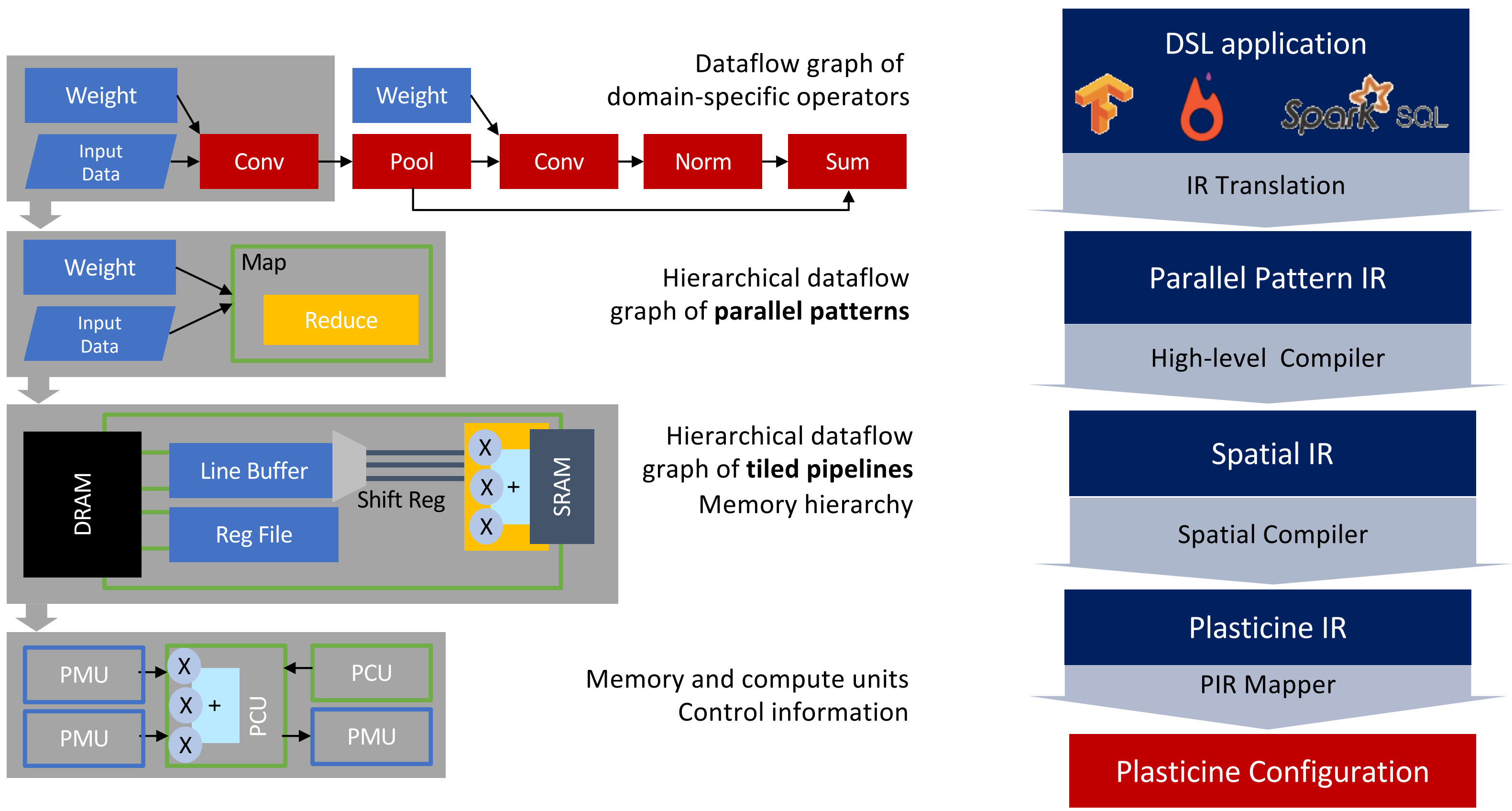
Raghu Prabhakar



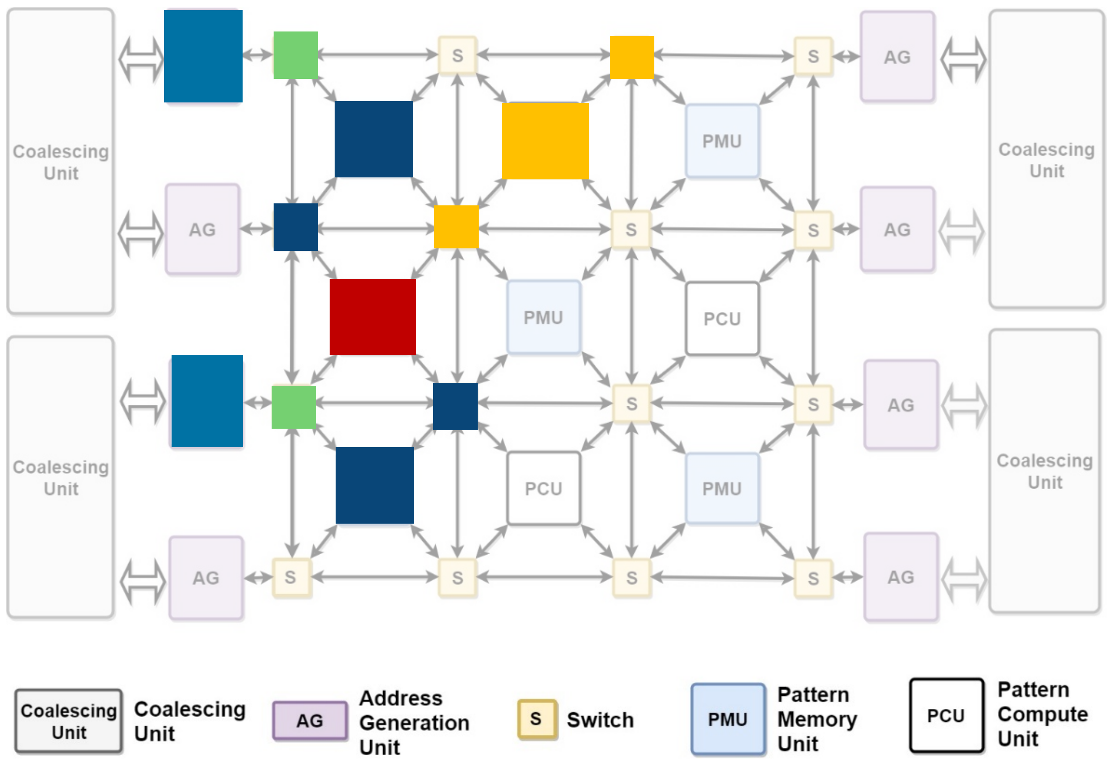
Yaqi Zhang

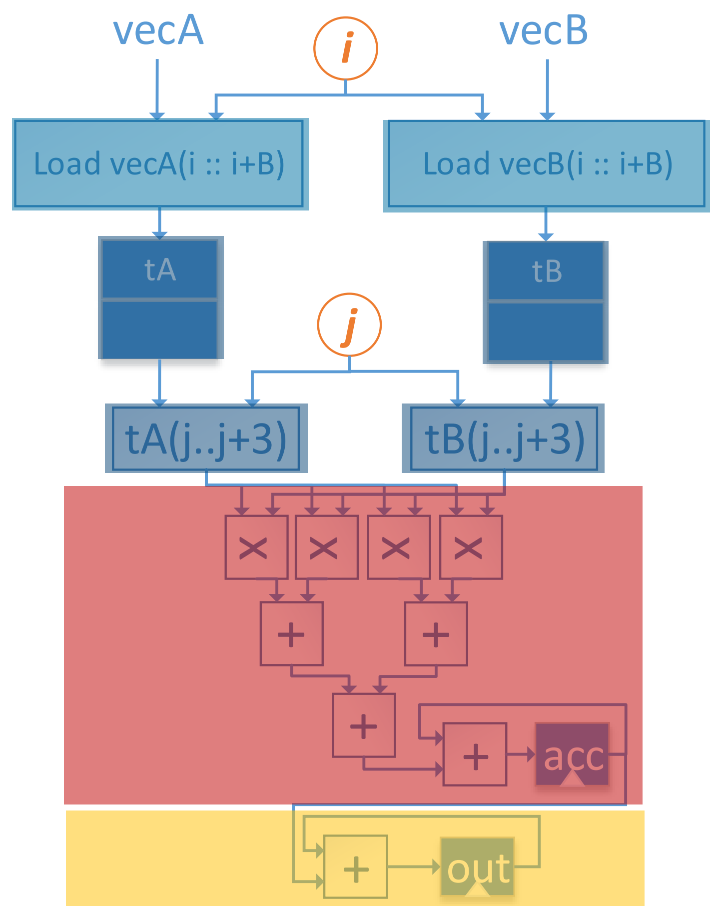
Prabhakar, Zhang, et. al. ISCA 2017

Compiler Architecture



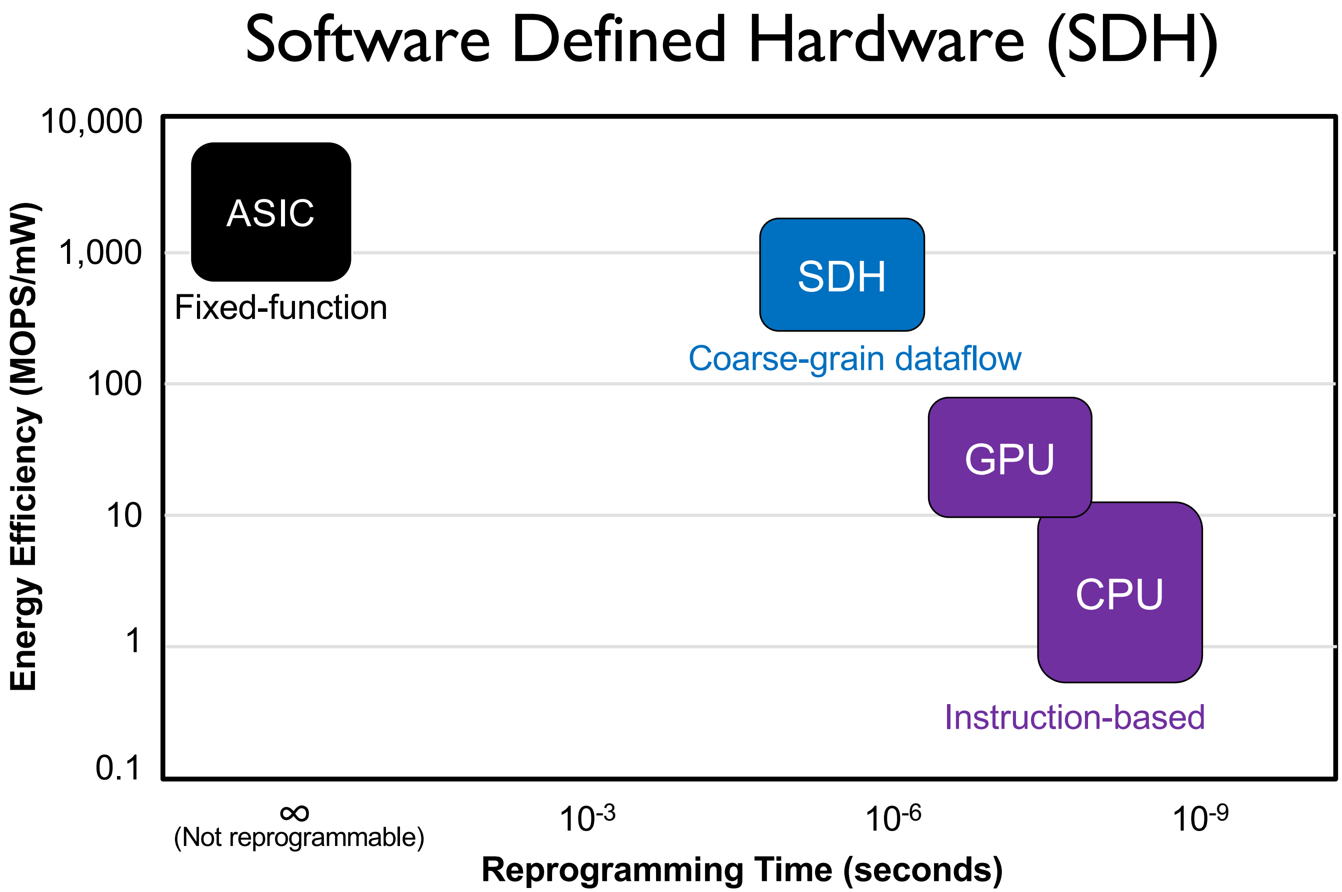
Mapping Spatial to Plasticine





Dot Product

Efficiency vs. Flexibility



We Can Have It All with Software 2.0!

- Productivity
- Power
- Performance
- Programmability
- Portability

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

ML Developer

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?